Cranfield University

Lorna Frewer

The Development of an Agent-Based Model to Investigate Possible Power Law Relationships in Peacekeeping Operations

Defence College of Management and Technology

PhD Thesis

Cranfield University

Defence College of Management and Technology

Engineering Systems Department

PhD Thesis

Academic Year 2006-2007

Lorna Frewer

The Development of an Agent-Based Model to Investigate Possible Power Law Relationships in Peacekeeping Operations

Supervisor : M R Bathe

February 2007

ABSTRACT

Modelling peace support operations is a growing area of research in the defence sector. Extensive development has been done in the area of combat models but they are not always sufficient when modelling operations other than war.

The DIAMOND model is a large scale peace support model capable of modelling entire countries. Taking an agent-based approach, we have created a model that has the potential to be used in conjunction with DIAMOND, providing the detail the larger model lacks. Improvements need to be made before this is possible but our model provides a strong starting point.

Self-organised criticality is an area of complexity theory that is, in part, identified by a fractal frequency-size 'avalanche' distributions. Previous research has shown a link between self-organised criticality and combat modelling. We looked for power-law behaviour in a variety of peacekeeping scenarios.

Using our agent-based model we devised a set of scenarios, each one more complex than the previous one. Taking the conflict between the peacekeepers and local insurgents, we used two different measures of such to represent the 'avalanches'. The results showed no real evidence of power law relationships but more experimentation and analysis is needed.

ACKNOWLEDGEMENTS

First I would like to thank my supervisor Mike Bathe whose support and guidance throughout the past four years has been invaluable. Thanks also go to my thesis committee, Venkat Sastry and Kathryn Wand. Special thanks to Venkat for all the programming advice.

Thank you to the Policy and Capability Studies Department at Dstl Farnborough who initiated the research. In particular I would like to thank Professor Jim Moffat for his extensive support on the project. Thanks also goes to Pete Bailey and Dan Edwards for their help in understanding the DIAMOND model.

My friends and fellow students in Heaviside postgraduate centre have made my time here enjoyable, thank you for the lunches and friendly chats. To the girls in the AMOR and ESD offices, thank you all for your assistance and encouragement.

And last, but certainly not least, a massive thanks to my family and friends outside Cranfield University who have supported me throughout my research.

Contents

List of Tables

List of Figures

GLOSSARY

Acronyms

ABM	Agent-Based Model
ANOVA	ANalysis Of VAriance
CA	Cellular Automata
DIAMOND	DIplomatic And Military Operations in a Non-warfighting Domain, a model for representing peace support operations
FFM	Forest Fire Model
ISAAC	Irreducible Semi-Autonomous Adaptive Combat, an agent-based combat model
MANA	Map-Aware Non-uniform Automata, an agent-based combat model
MATLAB	MATrix LABoratory, mathematics and engineering software
NGO	Non-Governmental Organisation
OOTW	Operations Other Than War
PSO	Peace Support Operations
SOC	Self-Organised Criticality
SSKP	Single Shot Kill Probability

Chapter 1

INTRODUCTION

Recent developments have resulted in the emphasis of many armed forces switching to peace support operations rather than the traditional warfighting. The current situations in Afghanistan and Iraq being two such examples.

There is an extensive array of models available for combat operations but the modelling of operations other than war is a relatively new research area. Dstl have developed a large scale model, DIAMOND, capable of modelling whole countries, but they do not have a model to provide the detail at town and village level. The first aim of our research is to develop a model that can complement DIAMOND by providing a representation of events at a selected area of the larger model.

Research has shown that data related to combat can exhibit a type of complex behaviour called self-organised criticality. The second aim of our research is to investigate this possibility in a peacekeeping scenario. For this we use our own agent-based model and design a range of scenarios.

The structure of the thesis is as follows. Chapter 2 provides a survey of relevant literature. We introduce the concept of agent-based modelling and two agent-based combat models are described in detail. Self-organised criticality is explained and three agent-based models that exhibit such behaviour are given. Finally we look at current methods of modelling peace support operations.

Initial experiments are described in Chapter 3. Here we describe experiments done with the models MANA and ISAAC in order to understand how these agent-based combat models work. We replicate MANA experiments that show it is unsuitable for modelling peacekeeping scenarios, hence the reason for developing our own model. Away from the agent-based models, we developed a method for calculating tension in the DIAMOND peace support model.

The development of our agent-based peacekeeping model is documented in Chapter 4. A full description is given along with verification and initial validation.

We devised four major scenarios, each a development of the previous one. The scenario details and results are given in Chapter 5 We look for evidence of power laws and self-organised criticality in these results.

We summarise our research and main findings in Chapter 6 before suggesting future directions for the project in Chapter 7.

Chapter 2

LITERATURE REVIEW

There are two main topics involved in our research: first we have the military modelling and then there is the self-organised criticality. The link between the two is provided by agent-based models, and in particular cellular automata.

We start by defining agent-based models in Section 2.1 and illustrate the concept with the example of Conway's Game of Life. Next in Section 2.2 we introduce the idea of self-organised criticality and give three examples of models that exhibit this behaviour, all of which are agent-based models. We then describe two more complex agent-based models that are used for military purposes, ISAAC and MANA, in Section 2.3. Finally, in Section 2.4, we look at current models used to represent peacekeeping operations.

2.1 Cellular Automata and Agent-Based Models

An agent-based model is a model such that the entities, or agents, are controlled by a finite set of behavioural rules that are implemented at each step rather than by pre-determined events. Cellular automata are a subclass of agent-based models. These models consist of an $n-$dimensional grid of cells, each of which

can take one of a number of states. The cells are the agents in the model and the evolution of these cells is governed by a set of rules.

2.1.1 Game of Life

An example of a two-dimensional cellular automata is Conway's Game of Life, given in [43]. This is a widely studied model with many interesting behaviours resulting from three very simple rules. The model is defined on a two-dimensional square grid where each cell is defined to be either alive or dead. The cells evolve according to the number of neighbouring alive and dead squares. For the purposes of this model the neighbouring cells are defined to be the eight surrounding squares. The following rules are followed:

1. If a live cell has less than two alive neighbours it becomes a dead cell at the next step;

2. If a live cell has four or more alive neighbours it will become a dead cell at the next step;

3. If a dead cell has exactly three alive neighbours it will become a live cell at the next step.

If none of these situations are applicable the cell remains unchanged. An example model evolution over four time steps is shown in Figure 2.1. Live cells are shown as black, dead cells are white.

The Game of Life has been used to symbolise simple population dynamics. Rule One represents populations dying out due to isolation or lack of food. Rule Two represents overcrowding and therefore too much competition for food. Rule Three gives ideal conditions for population growth. In our example in Figure 2.1 we can see an overall decline in the population.

When studying agent-based models we are looking for emergent behaviour. Examples of emergent behaviour for the Game of Life are stable patterns, blinkers

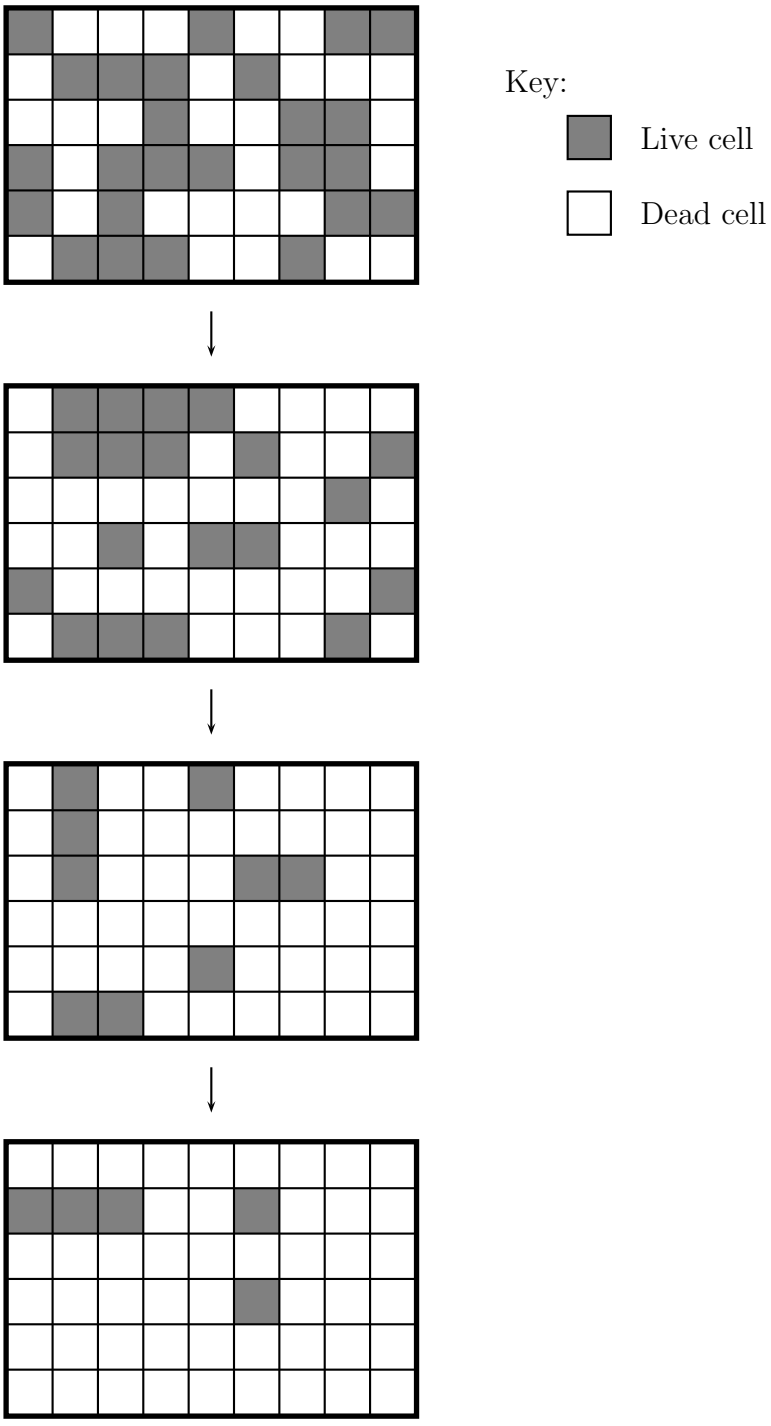

Figure 2.1: An Example Game of Life

and gliders. Stable patterns, as the name suggests, are configurations that do not change as the model evolves. An example would be a two-by-two square. A blinker is a pattern that repeats after a number of time steps, an example of a blinker of period two is shown in Figure 2.2 (a). Gliders are a special type of blinker. As well as having a cyclic pattern, the shape is also displaced diagonally by one square so it moves across the grid. An example glider of period four is shown in Figure 2.2 (b).

(a) Blinker

(b) Glider

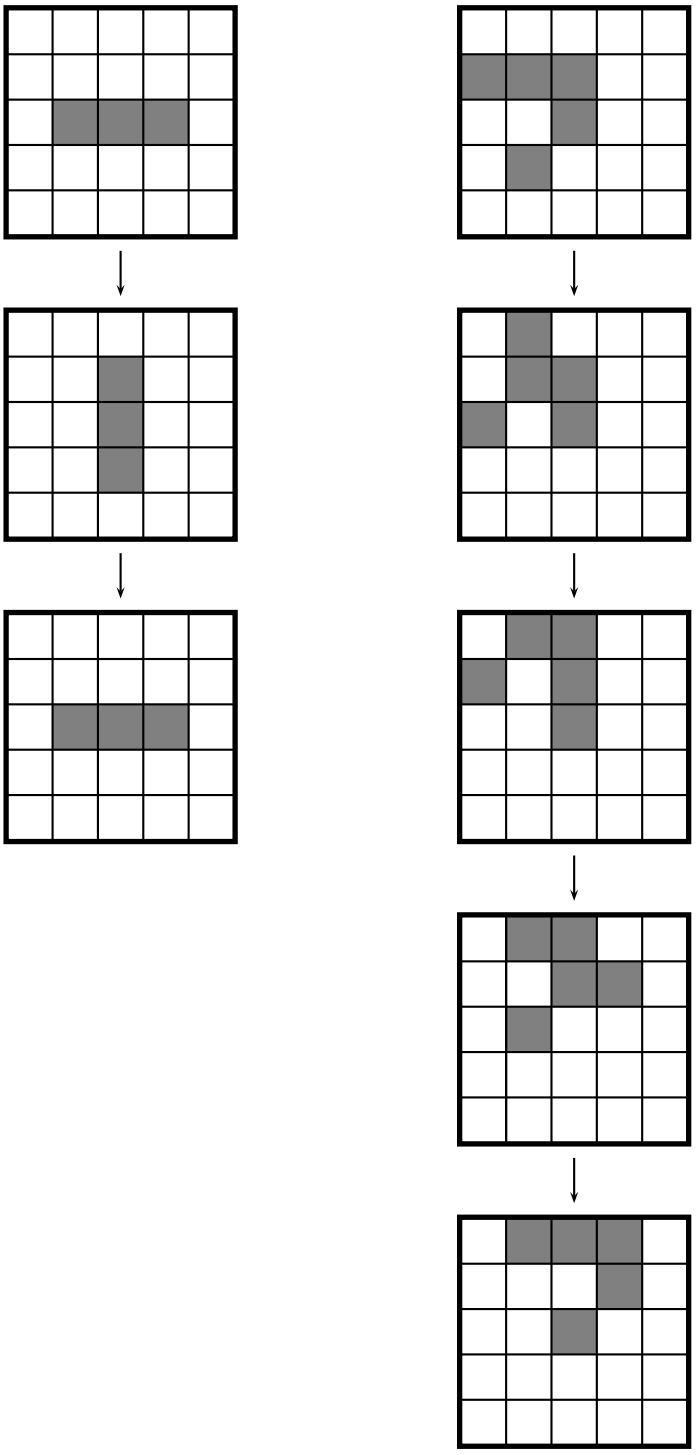

Figure 2.2: Example Game of Life Behaviours

2.2 Self-Organised Criticality

Self-organised criticality is related to the wider topic of complexity theory. Complexity theory is an evolving area of research concerned with finding simple behaviour in large systems with many rules. Related to this field is the study of fractals. A fractal is a geometric object that appears to be the same no matter how much it is magnified. The dimension of such an object is non-integer.

The idea of self-organised criticality was introduced by Bak, Tang and Wiesenfeld in their 1987 paper [3] as an explanation for the behaviour of a sandpile model they had developed; this model is described in Section 2.2.1. A general introduction to the subject is given in the key text by Bak [1], a more in depth analysis is given in the book by Jensen [18].

In this section we shall first define three simple models that have been shown to exhibit self-organised critical, (SOC), behaviour: the sandpile model, forest fire model and invasion percolation. We then move on to discuss the common features of SOC systems.

SOC may be used to explain the behaviour of naturally occurring systems. One of the models we describe below is the forest fire model. In their paper [24], Malamud, Morein and Turcotte compare results given by the model and data from actual forest fires. They conclude that the statistics associated with the computer model can be applied to real data. Barriere and Turcotte discuss the application of SOC to earthquakes in the paper [5].

It is hoped that we will be able to link the ideas from this theory and apply them in relation to military peace support operations. This has already been done for general warfighting scenarios. For example, in their report [33] Rowland, Keys and Stephens study irruption of forces. It is suggested in the paper [28] by Moffat and Witty that we could look at the fractal dimension of the troop formation and link this to the invasion percolation clustering. Invasion percolation is one of the models detailed later that has been shown to exhibit SOC behaviour; a

mathematical model is given in the paper [31] by Paczuski, Maslov and Bak. In their paper [32], Turcotte and Roberts find a link between battle casualty data and forest fire data. Both exhibit power law frequency size statistics that are a characteristic of SOC systems. They then try to relate the SOC forest fire model, described in Section 2.2.2, to the initial outbreak and spread of conflict.

2.2.1 Sandpile Models

The basic two-dimensional sandpile model works as follows. We start with an empty square grid, at each step a grain of sand is added to a random site on the grid. Once the count at a square is equal to four a toppling occurs, the four grains are distributed to the four nearest neighbour sites. This in turn may induce further topplings as some sites may now have grain counts of four. The process continues until all sites have a count of less than four. At some stage in the avalanche we may have a situation where the count at a site is greater than four, in this case only four grains are redistributed, the remainder stay at the site. If a site at the edge of the grid becomes unstable, four grains are lost from the site and only one is added to each nearest neighbour even though there are less than four nearest neighbours. The size of an avalanche is measured by the number of sites that become unstable. For example, the diagram in Figure 2.3 shows an avalanche of size three. At first one site is unstable, the resulting toppling leads to a further two sites becoming unstable then finally all sites are stable and we can resume the addition of sand grains.

It is also stated in the book by Turcotte [39] that instead of counting the number of sites that become critical, we can count the number of grains that are lost from the model. In that case Figure 2.3 shows an avalanche of size one. Both variations of the model exhibit self-organised criticality.

Turcotte's book [39] gives the method used to determine the presence of self-organised criticality. We look at the frequency-size distribution for the avalanches.

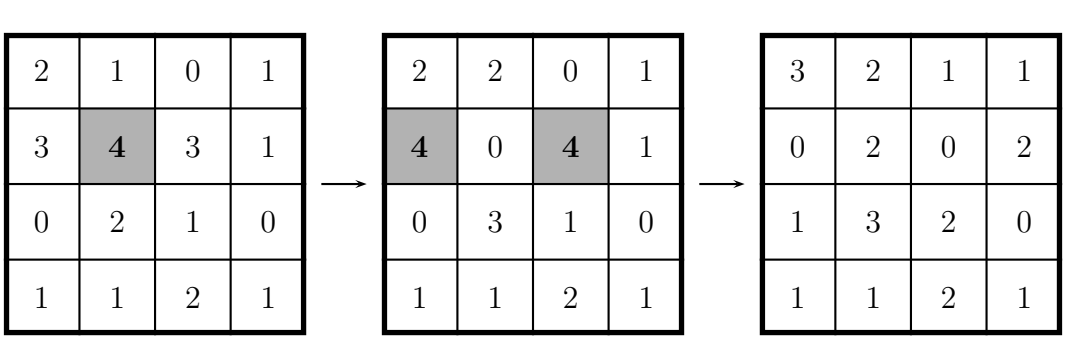

Figure 2.3: An Example of the Two Dimensional Sandpile Model

After the model has run for a certain time we can count how many avalanches, $N(A)$, there were of a certain size A. If we plot $\log(N(A))$ against $\log(A)$ and obtain a straight line then the frequency-size statistics are fractal which suggests that we have self-organised criticality.

2.2.2 Forest Fire Models

We give two alternative versions of the two-dimensional forest fire model, both of which have been shown to exhibit self-organised criticality.

Version One: The first version of the forest fire model is given in Turcotte and Roberts' paper [32]. We start with an empty square grid. We are given a sparking frequency f_s, this defines the frequency at which a match is dropped onto a randomly selected site. For example, if $f_s = \frac{1}{100}$ a match is dropped every 100 steps. For the remaining steps a tree is planted at a random site. Once a fire has been started it spreads to nearest neighbour sites, and then to their nearest neighbour sites, and so on. The number of trees set on fire is recorded then all burning sites become empty sites. If a match is dropped onto an empty site, nothing happens and the fire is recorded as being of size zero. If a site selected randomly for planting is already occupied we do nothing, we do not assume there are now two trees at that site. The diagram in Figure 2.4 shows an example of a fire of size seven.

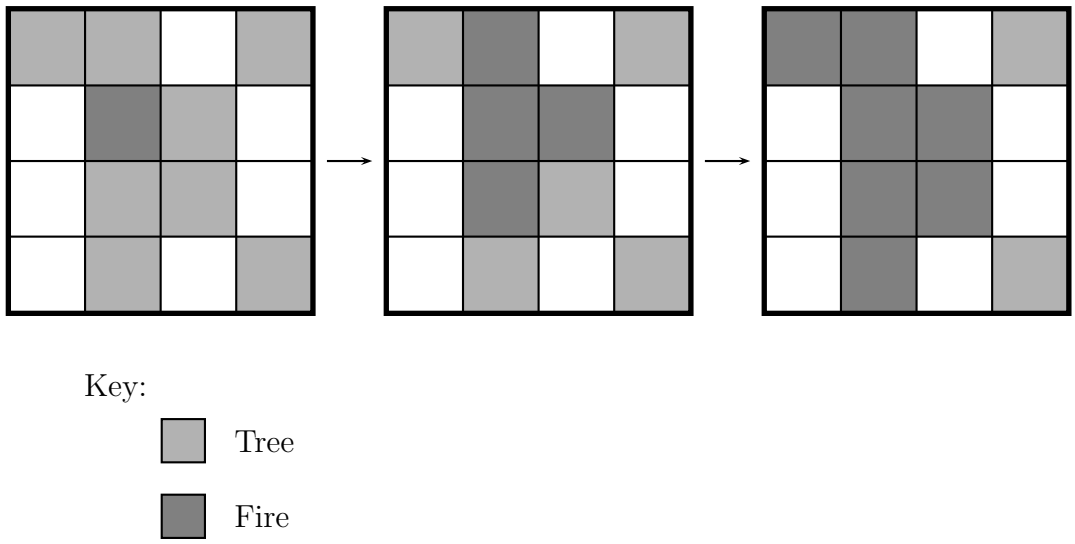

Figure 2.4: An Example of the Two Dimensional Forest Fire Model

In their paper [32], Turcotte and Roberts found a power law distribution for fire size. The model was run for N_S timesteps and the size of each fire was recorded. For each fire size A_F, the number of fires of that size were counted, this was denoted N_F. Plotting $\log(N_F/N_S)$ against $\log(A_F)$ gave straight line graphs for each of the three sparking frequencies used. This implied that

$$\frac{N_F}{N_S} \sim A_F^{-\alpha},$$

where α was found to be 1.02, 1.09 and 1.16 for sparking frequencies of $\frac{1}{125}$, $\frac{1}{500}$ and $\frac{1}{2000}$ respectively. The model was run on a square grid of size 128×128. We tried to reproduce the results from the paper but found that our program would have to run for months in order to cope with the values for N_S used by the authors. Either our program was too inefficient or Turcotte and Roberts made use of a supercomputer when conducting their experiments.

Version Two: The two-dimensional forest fire model can also be defined slightly differently. We again have a square grid and sparking frequency f_s but this time we also have a planting frequency f_p, this is the probability that a tree will be planted at an empty site at each time step. We start with either an empty

grid or a configuration of trees determined using a random number generator. At each time step each grid square is assigned a random number r_{xy} in the range $(0, 1)$. If the site (x, y) is empty and $r_{xy} < f_p$ then a tree is planted at the site. If there is a tree at site (x, y) and $r_{xy} < f_s$ then the tree is set alight. If the site was a burning tree at the previous time step then it changes to an empty site. If a tree is planted at a site and a nearest neighbour site was burning at the previous timestep, then the tree is set alight.

Much research has been conducted with both versions of the model as well as adaptations. For example, Strocka, Duarte and Schreckenberg add bushes to define a two level model of forest fires; this model is detailed in [36]. Instead of having just a tree at a site, there can be a bush as well. The two levels add to the behaviour rules. In addition to the planting probability for trees, there is also a second, higher, probability that a bush will be planted. Both burning trees and burning bushes disappear at the next timestep. A tree catches fire if it is at a site with a burning bush, nearest neighbour to a burning tree or if it is ignited according to its sparking frequency. A bush is set alight if it is at a site with a burning tree or nearest neighbour to a burning bush.

In the paper [30], Moßner, Drossel and Schwabl looked at computer simulations of the model in different dimensions. In her paper [11], Drossel concentrated on mathematical analysis of the one dimensional model. Drossel, Clar and Schwabl find exact results for the one-dimensional model in their paper [12] which are then confirmed by computer simulations.

2.2.3 Invasion Percolation

Another example of a model that can exhibit self-organised criticality is invasion percolation. Details of this model are given in the paper [31]. We start with a square grid with one invaded edge; all other sites on the grid are assigned a random number. Consider all the nearest neighbours to the invaded sites. At

each step the nearest neighbour site with the lowest associated random number is invaded. As an example consider the diagram in Figure 2.5. We start with a 4×4 grid with the top edge invaded. At each step all the relevant random numbers are shown. We continue until we have a cluster that spans the grid from top to bottom.

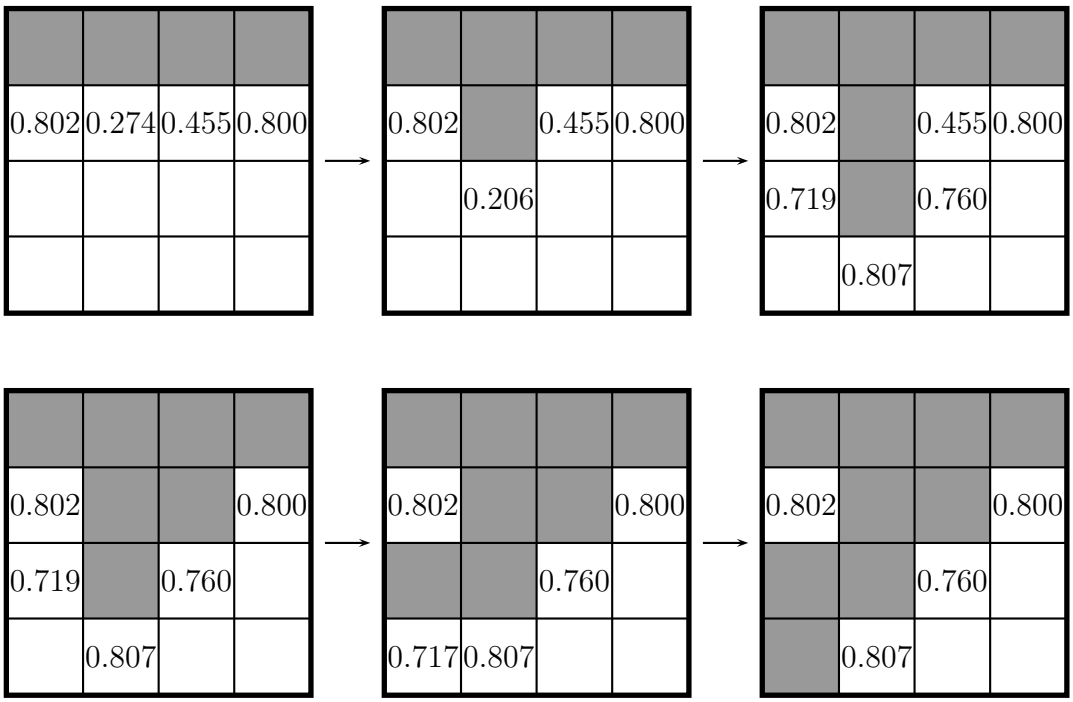

Figure 2.5: An Example of the Invasion Percolation Model

We can find the fractal dimension of the invasion percolation using the box counting method, detailed in [14]. Say we have a grid of size $2^n \times 2^n$. Then for $m = 1, 2, \ldots, (n-1)$ we split the grid into squares of side length 2^m and count how many of these squares the cluster is in, call this number $B(m)$. We then plot $\log(B(m))$ against $\log(2^m)$ and find the gradient of the best fit line, this is the box counting dimension.

2.2.4 Defining and Identifying a SOC System

As yet there is no formal mathematical definition of a SOC system, instead there have been many descriptions of the general features of such a system. In his lecture titled 'Self-Organised Criticality: A Holistic View of Nature', and written up in the book [10], Bak gives the description

" 'Self-Organised Criticality' (SOC) describes the tendency of large dynamical systems to drive themselves to a critical state with a wide range of length and time scales. "

In their paper [5], Barriere and Turcotte provide further detail.

"The concept of self-organised criticality is defined to be a natural system in a marginally stable state, evolving naturally back to the state of marginal stability when perturbed from that state. The input to the system is continuous but the loss is in a discrete set of events that satisfy fractal frequency-size statistics."

In her paper [11], Drossel gives the final important feature.

"In the stationary state the size distribution of dissipative events obeys a power law, irrespective of initial conditions and without the need to fine-tune parameters."

We now bring all these ideas together. We need a large dynamical system in order to negate any finite size effects. The attractor for the system should be a critical state, and if the system is driven away from this state it will naturally evolve back to it. By the attractor we mean the subset of the space on which the system is defined, to which the system will eventually evolve. The input to the system is continuous, for example the grains of sand are added steadily to the sandpile model. The output is discrete, for example the avalanches in the sandpile model do not occur at every timestep. After the model has run we can examine the frequency-size distribution, this should be a straight line when plotted on a log-log scale. The model should be run for a wide range of initial conditions and parameters, power law distributions should be found in each case.

We encounter several problems if we want to say for sure whether or not a model displays SOC behaviour. The definitions given in the papers only describe how to find out if the system is SOC after it has finished running and the frequency-size distribution for the output 'avalanches' can be examined, there is no way of determining whether a system exhibits SOC while it is running. Another problem is identifying the data that could be thought of as an 'avalanche'. By this we mean a measurable effect analogous to the avalanche of grains in the sandpile model or the fire in the forest fire model. This is a particular problem when looking at a complex model with a variety of parameters, such as a military model.

The main problem with identifying SOC behaviour is the fact that we are looking for a list of necessary, but not sufficient, conditions. The main analysis of model output comes with the frequency-size distribution of the 'avalanche' data. We are looking for a power law distribution, but even if we find evidence to suggest this is the case we cannot conclude for definite that the system is showing SOC behaviour since other processes can show the same power law property. Therefore, when we analysed our results we were looking for power law relationships rather than SOC specifically.

2.3 Agent-Based Combat Models

In this section we shall be looking at the two agent-based combat models ISAAC and MANA. We describe how the models work and the parameters involved. We pay particular attention to the laws governing the agents' movement in the models.

The acronym ISAAC stands for Irreducible Semi-Autonomous Adaptive Combat. The model was developed by Andrew Ilachinski for the Centre for Naval Analyses in the USA. MANA stands for Map Aware Non-uniform Automata and was developed by Michael Lauren and Roger Stephen for the New Zealand Defence Technology Agency. In order to understand the two models we used the userguides [17] and [22].

2.3.1 ISAAC

Model Parameters

The first parameters in ISAAC are the general battle parameters. These define the size of the battlefield, initial distribution of troops, fratricide and reconstitution settings. The battlefield can only be a square, and we input the side length. We define the initial positions of the agents giving the length, width and centre of the box in which the agents are initially randomly distributed; this is done for both sides. Then we state the position of each side's flag which is usually the goal location for the enemy. Finally there are options to define whether or not we allow fratricide and reconstitution. Fratricide is where an agent can be accidently shot by an agent from his own side. Reconstitution is where an agent who has been injured can be restored back to the full 'alive' status if he is not shot again in a user-defined period of time.

Each side has a set of defining parameters. First the number of agents is given, then the number of squads per side is defined followed by the number of

agents in each squad. ISAAC allows up to ten squads per side. The movement range defines the maximum distance of the sites it can consider moving to. For example if we set this to be one, the agents can only move to adjacent squares or stay where they are. Next the personality weights are defined for each squad, they are defined separately for alive and injured agents. These weights are for movement towards alive Red, alive Blue, injured Red, injured Blue, Red goal and Blue goal. All values are between zero and 100, with a higher value meaning the agents are more attracted to that entity. Next sensor and fire ranges are given as the number of squares over which enemy agents can be detected and weapons can be fired respectively. An example of the cell covered by a sensor range of two is given in Figure 2.6. The single shot kill probability is given as a percentage value between zero and one.

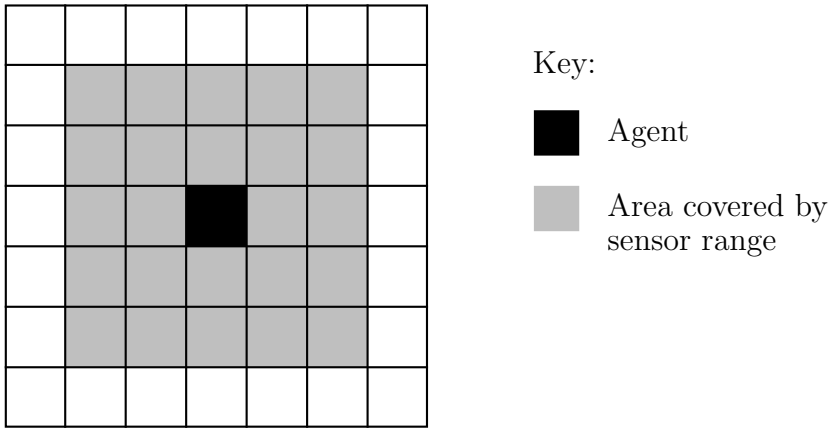

Figure 2.6: Example ISAAC Agent Sensor Range of Two

Next we have the movement constraints. Note that all parameters mentioned here are defined for each squad, and all, apart from the threshold range, are defined separately for alive and injured agents. The first parameter defines the threshold range of the agent, this is the radius around the agent within which

it can count the number of friendly and enemy agents. Note that this is not the same as the sensor range. This is used with the advance threshold number, cluster threshold number and combat threshold number. If the number of friendly agents within the threshold range is greater than or equal to the advance threshold number, the agents will continue to advance towards the goal. If the number of friendly agents within the threshold range is greater than the cluster threshold number, the agent will not move towards the other agents. If the advantage over enemy forces within the threshold range is less than the combat threshold number, the agent will retreat. We then have distance constraints, which are the minimum distances to friendly agents, enemy agents and the goal. All are given in terms of the number of squares, and hence must be a non-negative integer.

Finally we have terrain and statistics parameters. The terrain parameters can be used to define objects on the battlefield. The statistics parameters are set in order to define the data that is to be collected.

Movement of Agents

In order to determine where an agent should move, penalties are calculated for each square the agent could move to. In our experiments we set the parameters such that agents could only move to adjacent squares, so this is the scenario we shall consider here. Note that the number of possible moves is nine: the agent could move to any of the adjacent eight squares or stay where he is. It is also possible to program the agents to move up to two squares away. This just increases the number of calculations needed, the method stays the same.

We denote the personality weight vector $\mathbf{W}$, where the components are defined in Table 2.1. We denote the sensor ranges for Blue and Red by S_B and S_R respectively. Consider the area within sensor range of, for example, a Blue agent situated at position (x, y). In order to calculate the penalty for each possible move we consider all agents within sensor range. Denote the alive friends by

Weight	Entity
W_1	Alive friends
W_2	Alive enemy
W_3	Injured friends
W_4	Injured enemy
W_5	Own flag
W_6	Enemy flag

Table 2.1: Personality Weights

$B_1, B_2, \ldots, B_p$, and the alive enemies by $R_1, R_2, \ldots, R_q$. Similarly, denote the injured blues by $B_{p+1}, B_{p+2}, \ldots, B_n$, and the injured reds by $R_{q+1}, R_{q+2}, \ldots, R_m$. We calculate the distances from the Blue agent to all those within sensor range for each of the nine possible moves. We also calculate the distance to each of the flags from both the current square and the proposed position. If we call the square we are considering moving to (x_1, y_1), then we can calculate the penalty for this move using Equation (2.1). The notation $d[a, b]$ represents the distance between the points a and b.

$$P(x_1, y_1) = \frac{W_1}{p\sqrt{2}S_B} \sum_{i=1}^{p} d[(x_1, y_1), B_i] + \frac{W_2}{q\sqrt{2}S_B} \sum_{i=1}^{q} d[(x_1, y_1), R_i] + \frac{W_3}{(n-p)\sqrt{2}S_B} \sum_{i=p+1}^{n} d[(x_1, y_1), B_i] + \frac{W_4}{(m-q)\sqrt{2}S_B} \sum_{i=q+1}^{m} d[(x_1, y_1), R_i] + W_5 \frac{d[(x_1, y_1), B_{flag}]}{d[(x, y), B_{flag}]} + W_6 \frac{d[(x_1, y_1), R_{flag}]}{d[(x, y), R_{flag}]} \quad (2.1)$$

The movement constraints are taken into consideration by adapting the vector **W**. If the cluster threshold number is used, and the number of friendly forces within the threshold range is above this value then the weights W_1 and W_3 are taken to be zero. This means that other friendly agents will not be attracted to that cluster. If the number of friendly forces within the threshold range is less than the advance threshold number then W_6 is changed to $-W_6$. This ensures that the cluster does not move towards the enemy flag unless there is a sufficient

number of friendly agents in the cluster. If the advantage over enemy forces within the threshold range is less than the combat threshold number, then the weights W_2 and W_4 are set to $-W_2$ and $-W_4$ respectively. This means that the friendly agents will move away from enemy agents if they do not have enough support in the surrounding area. If the distance to friendly agents is less than the set minimum distance, the weight $-W_1$ is used rather than W_1. Similarly, if the distance to enemy agents or own flag are less than the set parameters, then $-W_2$ and $-W_5$ would be used respectively.

As an example we consider the configuration in Figure 2.7. Suppose that we have a Blue agent at position (x, y) in the centre of the grid with a sensor range of two. We want to calculate the penalty involved in moving to the square marked $*$. We assume that none of the movement constraints are in effect.

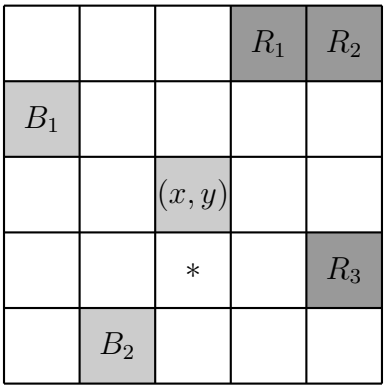

Figure 2.7: Example Agent Configuration

Of the agents within sensor range, B_1, B_2, R_1 and R_2 are alive, R_3 is injured. Let the personality vector be given by

$$\mathbf{W} = (10, 20, 0, 15, 0, 0).$$

The distance between two squares is taken to be the distance between the cen-

tres of those squares. Then, using Equation (2.1), we can calculate the penalty involved in moving to $*$,

$$\begin{aligned} P(*) &= \frac{10}{4\sqrt{2}}(2\sqrt{2}+\sqrt{2}) + \frac{20}{4\sqrt{2}}(\sqrt{10}+\sqrt{13}) + \frac{15}{2\sqrt{2}}(2) \\ &\quad + W_5\frac{d[*, B_{flag}]}{d[(x,y), B_{flag}]} + W_6\frac{d[*, R_{flag}]}{d[(x,y), R_{flag}]} \\ &= 42.0345 + W_5\frac{d[*, B_{flag}]}{d[(x,y), B_{flag}]} + W_6\frac{d[*, R_{flag}]}{d[(x,y), R_{flag}]}. \end{aligned}$$

We would calculate the penalty for the nine possible moves in this way, then the agent would move to the square with the highest penalty value. This suggests that the word 'penalty' is not really appropriate, it is more of an incentive function.

Cluster Analysis

In the papers [28] and [42], Moffat and Witty look at the clustering of agents in an ISAAC scenario. The scenario under consideration involves a red force of 100 agents and a smaller blue force of 16 agents. The red force has to negotiate its way around an obstacle before engaging in combat with the blue force. In all but one of their model runs the red force was successful. In these cases, when the largest cluster size for the red force was plotted, the three phases of the battle were clear from the graph. These phases were movement around the obstacle, combat and regroup. The equivalent graph for the blue force showed no pattern. In the case where blue were successful, there are only two phases clear on the red cluster size graph, this is because there is no regrouping of forces. The graph for the largest blue cluster shows a general upward trend and much larger clusters than in the other model runs. This work shows that the clustering of agents is an indicator of troop behaviour in cellular automata models and is something we could consider in our own research.

2.3.2 MANA

Model Parameters

The general parameters in MANA are used to set the size of the battlefield, the position of the flags and the size and initial location of the two sides. In order to program the goal for each side we use waypoints. These are specific coordinates that the agents aim towards in turn, giving them a route to their goal. We can also set alternative waypoints so that the agents have a choice of route. The exact initial positions of the agents varies with each run according to the random seed.

We now move on to the personality parameters. These are used to determine the movement of an agent. There are 13 parameters, each giving a weight towards a specific entity. These are the other agents divided into the categories alive friends, alive enemy, injured friends, distant friends and alive neutrals. There are also weights towards the next waypoint, alternate waypoints, easy terrain, cover, concealment and situational awareness threats one, two and three. The situational awareness threat level of an enemy agent is explained later in this section. These 13 values are set between -100 and 100. A positive value means the agent is attracted towards the entity, a negative value indicates the agent will move away from it. A value of zero means the agent is neither attracted nor repelled. The values themselves are not what is important, it is the relative sizes that matter. So, for example, if the weights for the agents all take the values -100, zero or 50 then we could change them to -10, zero and five respectively and still get the same results.

MANA also has range parameters. These put constraints on the movement of an agent as well as determining his effectiveness in battle. The combat parameters include the single shot kill probability, sensor range, firing range, shot radius and armour thickness. The single shot kill probability is given as a percentage value between zero and 100. The sensor range, firing range and shot radius have

integer values representing a number of squares. They give the distance at which the enemy can be detected, the maximum range of the weapon and the kill radius of the weapon respectively. This is the same as in ISAAC, so the example sensor range of two in Figure 2.6 holds for a MANA agent too. There are also parameters governing how close the agent can get to certain entities. These are minimum distance to friends, enemies, neutrals, next waypoint, enemy waypoint and easy going terrain. There are three parameters called cluster constraint, combat constraint and advance constraint. The cluster constraint gives the maximum cluster size, the combat constraint gives the numerical advantage needed for a group of agents to advance towards an enemy group, and the advance constraint gives the minimum number of friendly agents within sensor range needed to advance towards the goal. The final parameters are threat and stealth. The value for stealth should be between zero and 100; this determines the visibility of an agent when he is within sensor range of the opposition. The higher the value, the less visible the agent. The threat parameter gives the level of threat posed by the agent to the enemy; it can take the values one, two and three where a higher value means a higher threat.

The personality and range parameters are defined for each side and for each state the agents can be in. In our scenarios each side had only one default state, but they could, for example, be programmed to have different behaviour if they are injured.

Movement of Agents

As with ISAAC, MANA uses penalty calculations to determine where an agent should move to. We use the same notation as that for the ISAAC penalty function in Section 2.3.1. First, squares which are already occupied or include impassable terrain are excluded. All entities within sensor range are considered. The current distance to each entity is calculated as well as the distance from the proposed

square. The penalty for each type of entity, for example alive Red agents, is then calculated using the algorithm below.

- If the proposed move satisfies the distance constraints then set $Direction = 1$, if not put $Direction = -1$.
- Calculate the penalty term

$$Direction * \frac{d[(x_1, y_1), \text{entity}] + (100 - d[(x, y), \text{entity}])}{100}$$

- Sum the penalties for each of the entities of this type and divide by the number of such entities.

For example, the penalties for alive agents are then given by Equations (2.2) and (2.3).

$$P_{AliveBlue} = \frac{1}{p} \sum_{i=1}^{p} Direction_i \left\{ \frac{d[(x_1, y_1), B_i] + (100 - d[(x, y), B_i])}{100} \right\} \tag{2.2}$$

$$P_{AliveRed} = \frac{1}{q} \sum_{i=1}^{q} Direction_i \left\{ \frac{d[(x_1, y_1), R_i] + (100 - d[(x, y), R_i])}{100} \right\} \tag{2.3}$$

It is unclear in the manual [22] how these penalty calculations are used to determine where the agent moves to. As they have not taken into account the personality weights, we would suggest that the next step would be to multiply these penalties by the appropriate weights before seeing which move has the least penalty. Then other constraints, such as cluster size, would be considered to make sure the proposed move would satisfy the parameters.

Calculating the Effective Fractal Dimension of the Battlefield in MANA

Michael Lauren suggests a method for calculating the 'effective battlefield fractal dimension' in MANA in his paper [19]. To understand the method we tried to reproduce his results. In the paper both infantry assault and mobile battle scenarios are considered. We shall only describe the infantry assault ones here as

this will be sufficient to illustrate the method. We used the parameters given in the paper as far as possible but some of the values used by Lauren were not made clear, in particular no personality weights were stated. In these cases we used values that seemed appropriate. Note that after conducting the experiments we contacted Michael Lauren with our results. He confirmed that he had used different personality weights when designing his scenario, which explains the difference in results.

In these infantry assault scenarios the Blue defending force is stationary and the Red attacking force is mobile. The Blue force consists of 30 agents who are arranged in a line across the battlefield. The Red force varies in number, but is situated 14 squares away from the Blue force. All agents have a sensor range of nine, and a firing range of eight. The weapons have a single shot kill probability of 0.05. The Red agents can move a maximum of one square per timestep. The simulation is stopped when we have 50% Blue casualties or 100% Red casualties. The initial Red force size is varied, the model is run 100 times in order to obtain mean casualty figures.

The base case scenario includes a suppression effect on Red agents. When an agent is shot at, if he is not killed he remains stationary for three timesteps. While in this state the agents are still able to shoot. In this scenario we also have beaten zones, (BZs), for the weapons from both sides. Any agent within three squares from a target can be killed. The second scenario removes the suppression effect, the final scenario reinstates suppression but removes the beaten zones. Figure 2.8 shows the results for the three scenarios.

We found the lines of best fit for the graphs in Figure 2.8 and recorded the gradient. The results are given in Table 2.2. The graph for our base case scenario has a gradient of 0.882. In his report, Lauren compared his result of 0.63 to a value of 0.685 obtained by Thornton and Hall, Wright and Young from historical data, their reports [38] and [16] were cited by Lauren.

Lauren then considers fractional casualty levels in order to obtain graphs

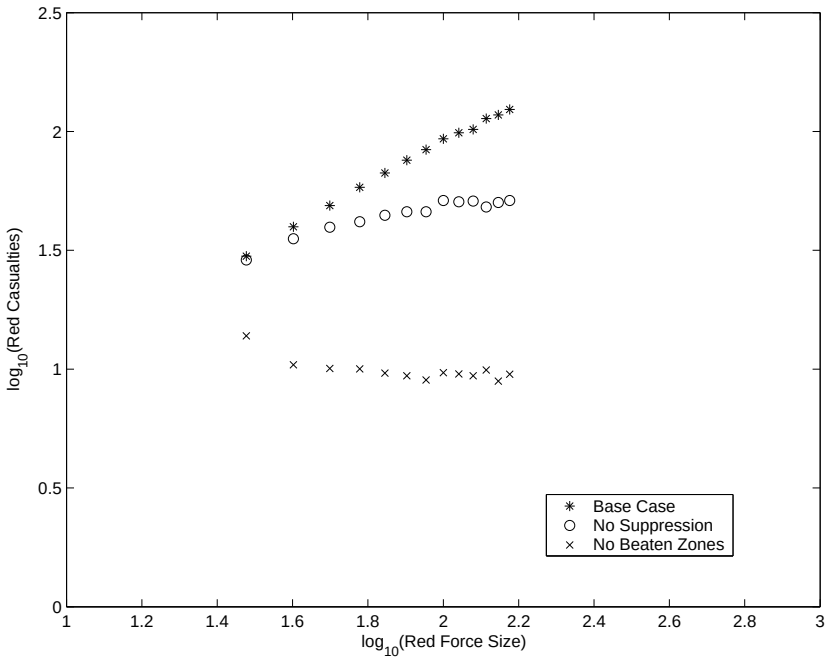

Figure 2.8: Red Casualties Incurred in Inflicting 50% Blue Casualties for Infantry Assaults

Scenario	Gradient of Graph
Base Case	0.882
No Suppression	0.325
No Beaten Zones	-0.166

Table 2.2: Gradients of Best-Fit Lines for Infantry Assault Scenarios

that approximate to a straight line. The fractional Red casualty size was plotted against initial force ratio. Using this analysis with our results gives the graphs in Figure 2.9.

These graphs form a better line than those for casualty numbers. Lauren found this to be the case in his analysis and suggested the equation

$$C_R = AC_B \left(\frac{r(0)}{b(0)} \right)^{-D_B}, \tag{2.4}$$

C_R and C_B represent the fractional Red and Blue casualty levels, $r(0)$ and $b(0)$ represent initial Red and Blue force sizes. The two parameters to be determined

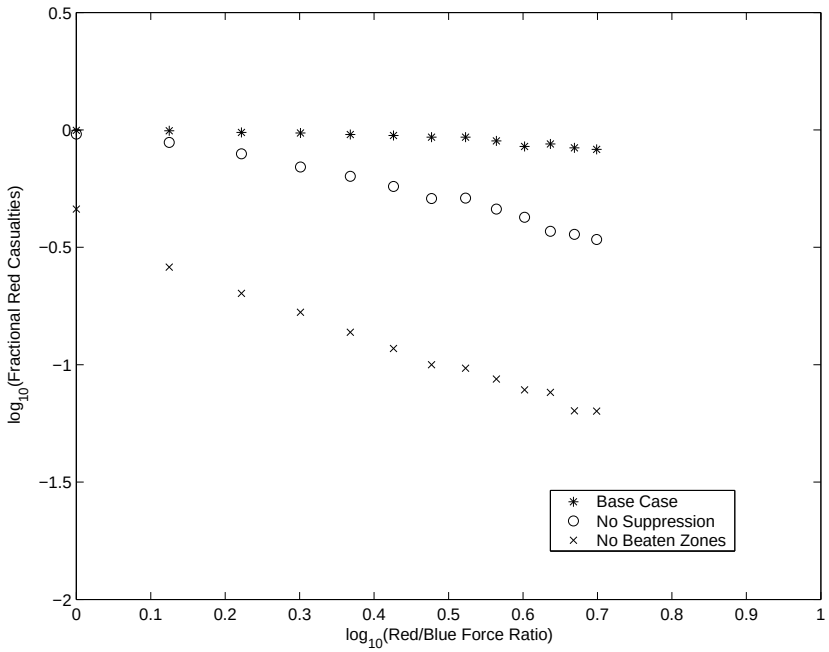

Figure 2.9: Fractional Red Casualties Incurred in Inflicting 50% Blue Casualties for Infantry Assaults

are the constant A and the 'effective battlefield fractal dimension' D_B. To find these values we draw the line of best fit on the fractional casualty level graph. The parameter D_B is the modulus of the gradient. The parameter A is calculated using the intercept as follows

$$\log(A) = intercept - \log(0.5), \tag{2.5}$$

this is since the Blue casualty level C_B is always 0.5. After drawing the best fit line through the graphs in Figure 2.9 we calculated A and D_B and the results are shown in Table 2.3. We also note the results from [19] in this table, they are headed A_L and D_{BL}.

Scenario	A	D_B	A_L	D_{BL}
Base Case	2.070	0.118	1.8	0.37
No Suppression	2.142	0.675	1.8	0.60
No Beaten Zones	0.777	1.166	1.3	1.5

Table 2.3: Calculated Parameter Values for Infantry Assault Scenarios

2.4 Modelling OOTW

We have looked at two models for representing peacekeeping operations, DIAMOND and PAX. The two models are very different in scale: DIAMOND can be used to model a whole country whereas PAX is used to model a basic local food distribution operation. There has been a great deal of research conducted regarding modelling combat operations, but the area of modelling operations other than war, OOTW, is relatively new.

2.4.1 DIAMOND

The acronym DIAMOND stands for DIplomatic And Military Operations in a Non-warfighting Domain. The model is used to represent operations other than war and usually involves scenarios that include military factions and civilian populations along with peacekeeping forces. An overview of the model is given in [37] and the functional specification is given in the reports [6], [7] and [8]. Whilst on placement in the Policy and Capability Studies department at Dstl Farnborough I was asked to work with output data from DIAMOND to develop a method to calculate tension away from civilians. The results appear later in Chapter 3.

DIAMOND works on an arc and node network. The nodes represent significant locations such as towns, villages or road junctions. The arcs are the routes between the nodes. The military forces in the model are represented by entities where one entity is of squadron to battlegroup size for ground troops, individual

to group for maritime forces and individual to package for air forces. There can be more than one military entity from each party present at a node. Civilian entities are different, they consist of all the civilians from a party at the node which means that they can vary greatly in size.

DIAMOND is mission based rather than agent based. This means that instead of each agent having pre-determined behavioural rules, the entities are given a mission and work towards completing it. Their mission can be changed by their superiors and relayed to them through the communications network.

2.4.2 PAX

The model PAX was created in Germany by EADS Dornier GmbH. In their paper [35], Schwarz and Bertsche describe the background to the model and how it was developed. They wished to model a simple food distribution scenario using an agent based model but found the combat model MANA to be inadequate. Since MANA is a combat model the civilians could not be modelled satisfactorily, they had to be defined as an opposing faction. It was also found that the escalation and de-escalation of a situation could not be modelled properly. Since MANA's purpose is to model combat, only a situation of high tension and conflict could be represented. We repeat these experiments in Chapter 3.

As a result a new model, PAX, was developed using the social science model PECS as a basis. This works on the same scale as MANA. It therefore seems that there is a large gap between the PAX and DIAMOND models in scale and it would be useful to have a model that fits between them in size. This is where our research fits in, we hoped to be able to model events at node level in DIAMOND to provide the detail that this model lacks.

Bibliography

[1] P. Bak, How Nature Works: The Science of Self-Organised Criticality (1997), Oxford University Press.

[2] P. Bak, K. Chen & C. Tang, A Forest Fire Model and Some Thoughts on Turbulence, Physics Letters A 147(5,6):297-300, (1990).

[3] P. Bak, C. Tang & K. Wiesenfeld, Self-Organised Criticality: An Explanation of 1/f Noise, Physical Review Letters 59(4):381-384, (1987).

[4] P. Bak, C. Tang & K, Wiesenfeld, Self-Organised Criticality, Physical Review A 38(1):364-374, (1988).

[5] B. Barriere & D. L. Turcotte, Seismicity and Self-Organised Criticality, Physical Review E 49(2):1151-1160, (1994).

[6] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume I : Entities, Terrain, Environment & Facilities (2000), DERA Report.

[7] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume II : Movement, Sensing, Communication, Local Picture, Perception, Relationships, Negotiation and Test Scenarios (2000), DERA Report.

[8] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume III : Missions, Command & Control, Operations, Logistics and Model Outputs (2000), DERA Report.

[9] S. Clar, B. Drossel, K. Schenk & F. Schwabl, Self-Organised Criticality in Forest Fire Models, Physica A 266:153-159, (1999).

[10] G. A. Cowan, D. Pines & D. Metzeler (Ed.), Complexity: Metaphors, Models and Reality (1994), Westview Press.

[11] B. Drossel, Self-Organised Criticality and Synchronisation in a Forest Fire Model, Physical Review Letters 76(6):936-939, (1996).

[12] B. Drossel, S. Clar & F. Schwabl, Exact results for the One-Dimensional Self-Organised Critical Forest Fire Model, Physical Review Letters 71(3):3739-3742, (1993).

[13] B. Drossel & F. Schwabl, Self-Organised Criticality in a Forest Fire Model, Physica A 191:47-50, (1992).

[14] J. Feder, Fractals (1988), Plenum Press.

[15] A. W. Gill & D. Grieger, Validation of Agent Based Distillation Movement Algorithms (2001), DSTO Report.

[16] A. Hall, W. Wright & M. Young, Simplified Model of Infantry Close Combat (SMICC) (User Guide and Technical Guide). Issue 2 (1997), Centre for Defence Analysis.

[17] A. Ilachinski, Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial-Life Approach to Land Warfare (1997).

[18] H. J. Jensen, Self-Organised Criticality: Emergent Complex Behaviour in Physical and Biological Systems (1998), Cambridge University Press.

[19] M. K. Lauren, A Conceptual Framework for Understanding the Power-Exponent Nature of Empirically Obtained Casualty Laws (2002), DTA Report.

[20] M. K. Lauren, Fractal Methods Applied to Describe Cellular Automaton Combat Models, Fractals 9(2):177-184, (2001).

[21] M. K. Lauren, Modelling combat Using Fractals and the Statistics of Scaling Systems, Military Operations Research 5(3):47-58, (2002).

[22] M. K. Lauren & R. T. Stephen, Map Aware Non-Uniform Automata, Version 1.0 Users Manual (2001).

[23] M. K. Lauren & R. T. Stephen, Map Aware Non-Uniform Automata (MANA) - A New Zealand Approach to Scenario Modelling, Journal of Battlefield Technology 5(1):27-31, (2002).

[24] B. D. Malamud, G. Morein & D. L. Turcotte, Forest Fires: An Example of Self-Organised Critical Behaviour, Science 281:1840-1842, (1998).

[25] J. Moffat, Command and Control in the Information Age: Representing its Impact (2002), The Stationery Office.

[26] J. Moffat, Complexity Theory and Network Centric Warfare (2003), CCRP.

[27] J. Moffat & M. Passman, Metamodels and Emergent Behaviour in Models of Conflict, Simulation Study Group Workshop (2002).

[28] J. Moffat & S. Witty, Experimental Validation for Intelligent Agents in Conflict, Submitted to IEEE (2002).

[29] J. Moffat & S. Witty, Metamodels for the Collaborative Behaviour of Intelligent Agents in Conflict, Submitted to IEEE (2002).

[30] W. K. Moßner, B. Drossel & F. Schwabl, Computer Simulations of the Forest Fire Model, Physica A 190:205-217, (1992).

[31] M. Paczuski, S. Maslov & P. Bak, Avalanche Dynamics in Evolution, Growth and Depinning Models, Physical Review E 53(1):414-443, (1996).

[32] D. C. Roberts & D. L. Turcotte, Fractality and Self-Organised Criticality of Wars, Fractals 6(4):351-357, (1998).

[33] D. Rowland, M. C. Keys & A. B. Stephens, Breakthrough and Manoeuvre Operations (Historical Analysis of the Conditions for Success), Interim Report Volume I - Main Discussion and Annexes A-C (1994), Defence Operational Analysis Centre DOAC R 9412.

[34] A. M. Saperstein, War and Chaos, American Scientist 83(6):548-577, (1995).

[35] G. Schwarz & K. A. Bertsche, Agent-Based Simulation of (De-)Escalation in Peace Support Operations: Application of the Model PAX, 20 ISMOR (2003).

[36] B. Strocka, J. A. M. S. Duarte & M. Schreckenberg, The Self-Organised Critical Forest Fire Model With Trees and Bushes, Journal de Physique I France 5:1233-1238, (1995).

[37] B. Taylor & A. Lane, Development of a Novel Family of Military Campaign Simulation Models, Journal of the Operational Research Society 55:333-339, (2004).

[38] R. C. Thornton, Historical Analysis of Infantry Defence in Woods. Volume I: Main Text (1993), Defence Operational Analysis Centre DOAC R 9321.

[39] D. L. Turcotte, Fractals and Chaos in Geology and Geophysics, 2nd Edition (1997), Cambridge University Press.

[40] D. L. Turcotte, Self-Organised Criticality: Does it Have Anything to do With Criticality and is it Useful?, Nonlinear Processes in Geophysics 8:193-196, (2001).

[41] S. Witty, Analysis of Agent Based Model Runs Carried Out at MHPCC, Dstl Report (2003).

[42] S. Witty & J. Moffat, Initial Findings on Global Behaviour of an Agent Based Combat Model: An Investigation of the ISAAC Model, Swarming: Network-Enabled C4ISR Workshop (2002).

[43] S. Wolfram. A New Kind of Science (2002), Wolfram Media, Inc.

Chapter 3

INITIAL EXPERIMENTS

In this chapter we will be describing the experiments we carried out to gain greater understanding of the combat models MANA and ISAAC and the peace support model DIAMOND. We carried out a basic investigation and comparison of the agent-based models MANA and ISAAC by looking at the personality parameters. We also worked with output data from the DIAMOND model to develop a measure for tension between civilians. Finally, we reproduced experiments carried out by Schwarz and Bertsche to try to model a peacekeeping scenario using the combat model MANA in order to find the areas in which the model is not suitable for this purpose. All of these experiments helped with the development of our own agent-based model for peace support operations.

3.1 A Comparison of the MANA and ISAAC Models

The first experiments we did were aimed at comparing the combat models ISAAC and MANA. By using the models we were able to gain an in-depth understanding of them, which helped when we were developing our own agent-based model. We developed a simple scenario which could be put into both models and then

changed some of the agent parameters in order to determine their effect on the behaviour of the model.

3.1.1 ISAAC Scenario

Our scenario consists of two opposing forces, red and blue, located at diagonally opposite corners of a grid. The aim of both squads is to reach the opposition flag. The battlefield is a 150×150 square grid. Each force is initially randomly distributed in a 20×20 square, the flags are located at the centres of these squares. Each squad consists of 30 ISAAC agents and we set up the scenario so that the two forces are equal in terms of their parameters. This initial set-up for the scenario is illustrated in Figure 3.1, note that the diagram is not drawn to scale. In each experiment we considered ten initial configurations of troops; these formations were the same for each experiment so we can directly compare the results.

We tried five variations of this scenario, denoted $exp1_I$ to $exp5_I$, and each variation was a development of the previous one. In our original scenario, $exp1_I$, we set the personality weights to be as follows: the weights towards friendly and enemy agents were 20, the weight towards the enemy flag was 50 and the minimum distance between friendly agents was two. The sensor range was 12 and the firing range was eight. Remember that both squads have equal parameters. In $exp2_I$ we set the three personality weights mentioned above to be equal with value 50, then in $exp3_I$ the weight towards the enemy goal was reduced to 20. We then changed the firing range so it was equal to the sensor range of 12, we denote this $exp4_I$. Finally in $exp5_I$ we reduced the minimum distance between friendly agents to zero. A summary of these values is given in Table 3.1.

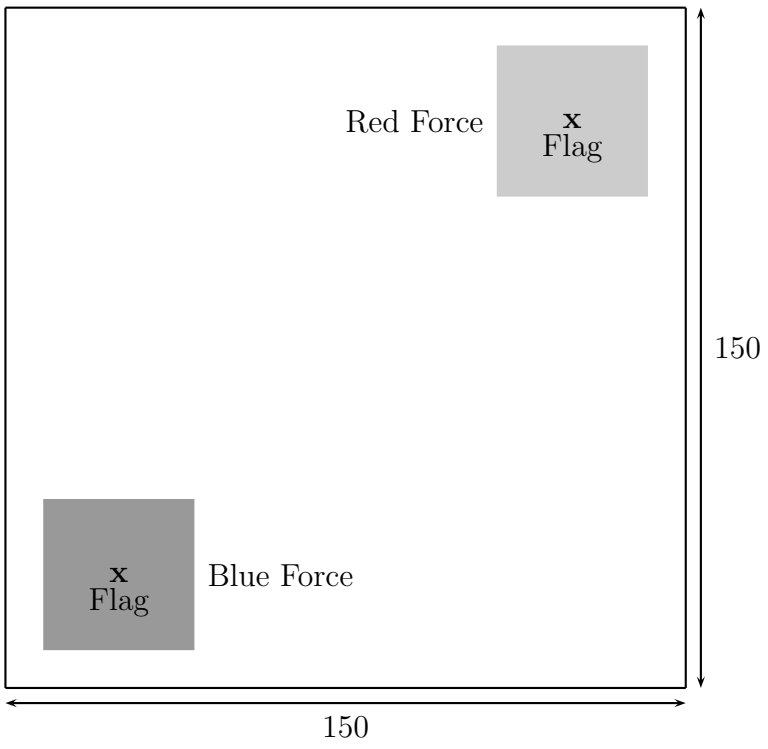

Figure 3.1: Initial Positions in the ISAAC Scenario

3.1.2 Results and Observations

When we looked at $exp1_I$, $exp2_I$ and $exp3_I$ we found very few casualties, in fact there were no casualties in $exp3_I$. In the first run of $exp1_I$ there was one Blue agent injury and one Red agent injury, all other runs were casualty free. In the second run of $exp2_I$ there was one Blue injury. The casualty numbers for $exp4_I$ and $exp5_I$ were much higher and are shown in Tables 3.2 and 3.3 respectively.

We also recorded the time of the first casualty, either an injury or a death, and the last casualty and then calculated the total combat time from these values. The results are shown in Tables 3.4, 3.5 and 3.6 respectively. Note that the scenario $exp3_I$ has not been included in the tables because no casualties were incurred.

The agents start by moving towards their goal in a close approximation to

Scenario	Weight to Alive Friends	Weight to Alive Enemy	Weight to Enemy Flag	Sensor Range	Firing Range	Minimum Distance to Friends
$exp1_I$	20	20	50	12	8	2
$exp2_I$	50	50	50	12	8	2
$exp3_I$	50	50	20	12	8	2
$exp4_I$	50	50	20	12	12	2
$exp5_I$	50	50	20	12	12	0

Table 3.1: ISAAC Experiment Parameters

	Run Number									
Survivors	1	2	3	4	5	6	7	8	9	10
Red Alive	8	13	9	13	7	13	6	11	11	7
Red Injured	10	9	9	13	9	9	10	7	11	13
Blue Alive	4	9	9	4	5	12	10	7	14	0
Blue Injured	3	10	7	6	4	9	7	8	7	0

Table 3.2: Results for $exp4_I$

their original formation. When the minimum distance between friendly agents is greater then zero the agents may move apart to satisfy this condition before moving towards the goal in formation. This suggests that if we were to consider either a smaller or larger battlefield, with the agents in the same initial configurations, we would see the same pattern of behaviour in the middle of the battlefield, though we may get different results near the corners due to the difference in distance travelled.

The formation of the troops changes when they detect the opposing force, in other words when the enemy agents are within sensor range. The squads then try to move around each other by, for example, forming lines and rotating so they are then able to get to their goals. From our experiments we saw that when the sensor range was greater than the firing range then there were very few, if any, casualties no matter what personality weights we chose. However, when the firing range was the same as the sensor range the amount of combat increased

Survivors	Run Number									
	1	2	3	4	5	6	7	8	9	10
Red Alive	9	3	9	9	7	6	1	5	8	6
Red Injured	9	5	11	7	6	8	9	4	13	10
Blue Alive	5	6	2	0	4	7	1	12	7	15
Blue Injured	8	2	5	1	4	9	3	7	7	6

Table 3.3: Results for $exp5_I$

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_I$	180	-	-	-	-	-	-	-	-	-
$exp2_I$	-	197	-	-	-	-	-	-	-	-
$exp4_I$	55	55	49	51	51	50	51	52	50	57
$exp5_I$	55	51	49	51	51	50	48	53	50	57

Table 3.4: Time First Casualty Occurs

and there were more significant casualties.

Changing the three personality weights mentioned above leads to differing levels of interaction in the middle of the battlefield when the agents first detect each other. The time to get to the opposition flag increased when we set the weights towards all other agents to be at the same level as that for the goal. Also, when we made this change the behaviour of the agents became more complex and we saw more elaborate formations.

The agents form three main types of configuration when they get within sensor range: lines, L-shapes and step shapes. The L-shapes and step shapes occur when part of the line pushes forward, forcing the opposition back. This usually happens when the agents are not evenly distributed along the line. There are often splits in the formations which happen when one side is stronger and exploits weak points in the opposition's formation. We also have situations where one side clusters more, forming a shorter line, and forces the other side back towards their own flag. When groups of agents are forced back they are often killed, assuming the firing range is equal to the sensor range. This happens because they are usually

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_I$	183	-	-	-	-	-	-	-	-	-
$exp2_I$	-	197	-	-	-	-	-	-	-	-
$exp4_I$	192	148	115	186	165	146	200	154	192	209
$exp5_I$	168	179	170	183	167	165	247	164	185	101

Table 3.5: Time Last Casualty Occurs

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_I$	4	0	0	0	0	0	0	0	0	0
$exp2_I$	0	1	0	0	0	0	0	0	0	0
$exp4_I$	138	94	67	136	115	97	150	153	143	153
$exp5_I$	114	129	122	133	117	116	200	112	136	45

Table 3.6: Total Combat Time

forced back into the corner while the opposition reach their goal so they remain within firing range.

We notice in Table 3.4 that there is very little variability in the time of the first casualty for the scenarios $exp4_I$ and $exp5_I$. We also notice that for seven of the runs the values did not change between the two scenarios. This shows that the agents headed straight for the opposition flag in both experiments and all of the runs. We see more variation in the times from Tables 3.5 and 3.6. This suggests that the combat in the middle of the battlefield does not follow the same pattern each time.

We notice from Tables 3.2 and 3.3 that the Red force usually ended up in a stronger position than the Blue. When casualties were incurred they were often heavier on the Blue side, and the Red force usually got their remaining agents to their goal before Blue. The situation was more balanced when the sensor range was greater than the firing range. As we only looked at ten different initial formations, these observations are most likely to be explained by coincidence and it may well be that these configurations favoured Red.

3.1.3 MANA Experimental Scenario

We tried to keep the scenarios in MANA as close to those we considered in ISAAC as possible. We again considered a 150×150 battlefield with two opposing forces of 30 agents at diagonally opposite corners. The first three MANA scenarios, denoted $exp1_M, exp2_M$ and $exp3_M$, are equivalent in parameter choice to the first three ISAAC scenarios. The remaining scenarios, $exp4_M$ and $exp5_M$, differ from the corresponding ISAAC scenarios. In $exp4_M$ we adapted the scenario $exp1_M$ so that the sensor range was increased to 20, in $exp5_M$ we increased it further to 30. This was done because we felt that increasing the firing range so that it was equal to the sensor range, as we had done in $exp4_I$, would not have been as constructive a change in MANA since casualties occurred in the previous scenarios. We reverted back to the personality weights from $exp1_M$ since they had given the most movement of agents. A summary of the parameters is given in Table 3.7. Note that unlike in ISAAC, the initial agent configurations given by MANA are not the same for each scenario. For example, Run One in scenario $exp1_M$ will not have the same initial distribution of agents as Run One in $exp2_M$. This is due to the different methods of random number generation used in the models.

Scenario	Weight to Alive Friends	Weight to Alive Enemy	Weight to Enemy Flag	Sensor Range
$exp1_M$	20	20	50	12
$exp2_M$	50	50	50	12
$exp3_M$	50	50	20	12
$exp4_M$	20	20	50	20
$exp5_M$	20	20	50	30

Table 3.7: MANA Experiment Parameters

3.1.4 Results and Observations

Table 3.8 shows the number of survivors in each scenario. Note that, unlike ISAAC, MANA does not record the state of the remaining agents, it just gives the number of deaths which we then use to calculate the number of survivors.

Survivors		Run Number									
		1	2	3	4	5	6	7	8	9	10
$exp1_M$	Red	20	20	25	18	28	24	18	24	7	26
	Blue	5	23	29	13	23	21	23	25	18	28
$exp2_M$	Red	30	30	30	30	28	30	29	29	30	30
	Blue	30	30	30	30	26	30	29	28	30	30
$exp3_M$	Red	30	30	30	30	30	30	30	30	30	30
	Blue	30	30	30	26	30	30	30	30	30	30
$exp4_M$	Red	14	18	13	6	15	18	19	20	15	17
	Blue	9	19	19	19	19	19	15	16	14	20
$exp5_M$	Red	13	18	16	12	19	19	15	17	21	22
	Blue	18	16	13	21	19	7	27	14	17	16

Table 3.8: Survivors

As with ISAAC, we recorded the times at which the combat began and ended and calculated the total combat time. These results are shown in Tables 3.9, 3.10 and 3.11. Note that we cannot compare these results with those for ISAAC since we have a different definition of a 'casualty': in ISAAC it was an injury or a death, in MANA it is a death.

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_M$	93	61	64	76	87	169	73	52	89	82
$exp2_M$	-	-	-	-	140	-	338	194		
$exp3_M$	-	-	-	755	-	-	-	-	-	-
$exp4_M$	58	51	55	54	54	56	55	53	53	55
$exp5_M$	53	54	55	56	54	51	54	52	54	52

Table 3.9: Time First Casualty Occurs

In the first three scenarios the agents immediately form clusters before mov-

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_M$	268	238	104	286	299	449	219	127	243	113
$exp2_M$	-	-	-	-	275	-	344	195	-	-
$exp3_M$	-	-	-	765	-	-	-	-	-	-
$exp4_M$	77	70	73	73	69	72	70	72	74	71
$exp5_M$	69	71	73	71	69	69	69	70	69	67

Table 3.10: Time Last Casualty Occurs

Scenario	Run Number									
	1	2	3	4	5	6	7	8	9	10
$exp1_M$	176	178	41	211	213	281	147	76	155	32
$exp2_M$	0	0	0	0	136	0	7	2	0	0
$exp3_M$	0	0	0	11	0	0	0	0	0	0
$exp4_M$	20	20	19	20	16	17	16	20	22	17
$exp5_M$	17	18	19	16	16	19	16	19	16	16

Table 3.11: Total Combat Time

ing towards the opposition flag. The movement towards the goal is very slow, usually only one or two clusters advance, the rest stay near their own flag. In the runs we considered none of the agents made it to their goal. All the agents who advanced towards the enemy flag were either killed when they got within range of the opposition, or stopped before they got to the flag. After the weight towards all other agents was increased in $exp2_M$ the amount of movement decreased, the agents usually formed clusters and then stayed close to their own flag. Occasionally one group would advance but they would be killed. When, in $exp3_M$, the weight towards the goal was reduced, there was only one instance where a group moved towards their goal. These results prompted us to change the parameters back to the settings from $exp1_M$ for the next experiment $exp4_M$.

The results for scenario $exp4_M$ show an earlier engagement time. This was due to the increase in the sensor range that meant that the agents were able to detect each other over a greater area and hence they did not cluster so much at the start. We observed that the agents head towards their goal straight away

and meet in the middle of the battlefield. This is shown in Table 3.9 where there is little variation in the time of the first casualty for this scenario. Unlike in ISAAC, there is no formation of agents and avoidance of the enemy, instead they pass through each other with no apparent attempt to avoid contact. Most of the surviving agents made it to the opposition flag, but some just stopped in the middle of the battlefield. After increasing the sensor range again in $exp5_M$ we observe the same behaviour, though in this case there was only one instance where a group of agents did not move from the middle of the battlefield. We notice from Table 3.11 that this further increase of sensor range led to an overall slight decrease in the total combat time, though this could be a coincidence. Note also that Table 3.11 shows a consistent battle time for $exp4_M$ and $exp5_M$. This suggests a similar pattern of combat in each run.

The graph in Figure 3.2 shows the loss ratios for the eighth run of $exp4_M$. The loss ratio at each time t is given by

$$\frac{30 - b(t)}{30 - r(t)},$$

where $b(t)$ and $r(t)$ are the number of alive Blue and Red agents at that time. This graph shows a pattern typical of the other runs from this experiment, and indeed to those from $exp5_M$. At the start there are no points on the graph as no casualties have been incurred, at this point the two forces are moving towards each other. Then, for a short time, there are a lot of points with a change of value at nearly every timestep. This portion of the graph relates to the period of combat in the middle of the battlefield starting with the first Red casualty. After this period the graph shows a constant value which is the final loss ratio. This represents the time when the agents have either got to their goal or have stopped in the middle of the battlefield. Note that the graph shown only shows the time up to 100 timesteps. This was done so that the combat period of the graph was clear; the graph stays constant at the final loss ratio for the remainder of the run time of 1000 timesteps. We can now compare this graph to an example from

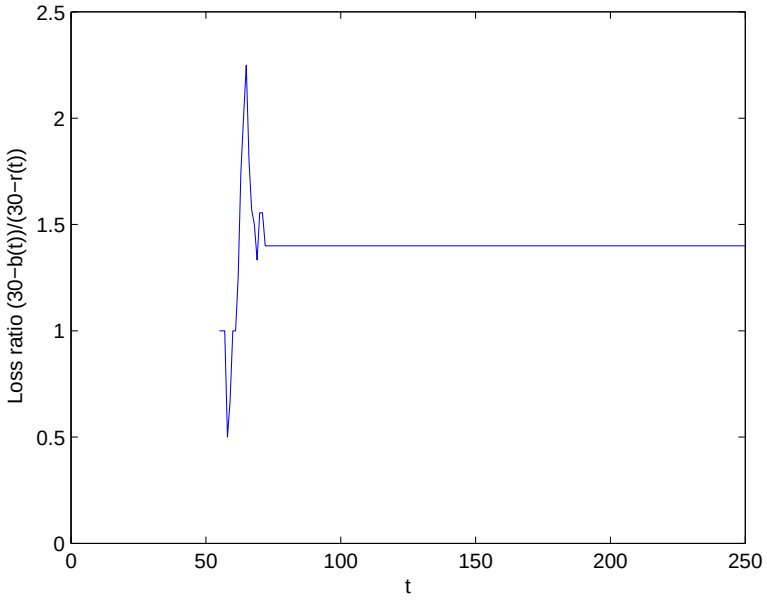

Figure 3.2: A Graph of the Loss Ratios for Run Eight of $exp4_M$

scenario $exp1_M$.

If we now look at Figure 3.3 we can see that this particular run of $exp1_M$ shows different behaviour from that in Figure 3.2. We notice that there is a greater amount of time before any casualties are recorded. This was because the agents formed clusters before advancing towards the goal and most of the agents stayed near their own flag rather than moving. We then have a greater period of combat than in $exp4_M$. There are times when the loss ratio does not change at each timestep showing that no casualties are incurred. This is due to the fact that in this particular run there were two clusters advancing, one Red and one Blue. They were moving quite slowly and hence it took longer for each agent to get within firing range. We can see that there was a period of combat where there were casualties at most timesteps, this occurred when the two forces were in close contact and therefore within firing range of each other. We observe a levelling of the graph after time 243, this shows the final loss ratio. For this particular run

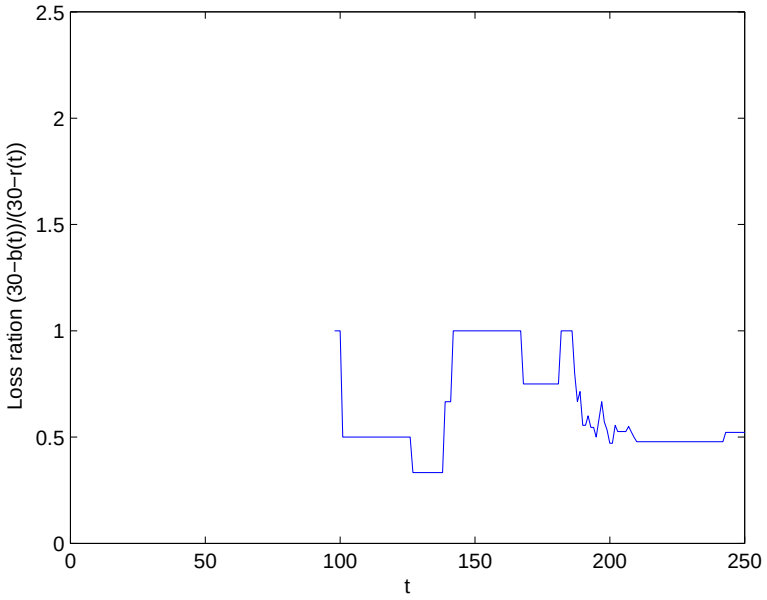

Figure 3.3: A Graph of the Loss Ratios for Run Nine of $exp1_M$

the value is 0.522 showing that Red incurred more casualties than Blue.

3.1.5 Comparing the Models

We chose to use the same parameters as far as possible so that we could compare the two combat models. Both models have features that are not included in the other, we set these values either to zero or to a sensible level so that we could focus on the effects of parameters that are common to both ISAAC and MANA.

We saw differences in our two initial scenarios $exp1_I$ and $exp1_M$. The agents in ISAAC immediately moved towards their goal in a close approximation of their initial distribution, whereas those in MANA grouped together in clusters and many of them did not advance towards their goal. We also saw a difference in combat. When ISAAC agents were within sensor range of their opposition they changed their formation and manoeuvred themselves around the enemy so they could get to their goal. MANA agents did not try to avoid opposition agents

once they were within sensor range, instead they kept moving towards their goal. This was the reason why there were more casualties in the MANA scenario even though there was less movement. We suggested in Section 3.1.2 that altering the size of the battlefield in ISAAC may not change the behaviour since the agents kept to a close approximation of their original formation until they came into contact with the enemy. The only foreseeable change could occur if the agents start within detection range of each other.

Changing the weights towards other agents in the second pair of scenarios, $exp2_I$ and $exp2_M$, led to less movement in MANA but more movement in ISAAC. This was because the behaviour of the agents in ISAAC changed so that there was more manoeuvring of troops once they came into contact with the enemy. In MANA the priority for the agents seemed to be to cluster together rather than move towards their goal.

We have seen how the two models produce different behaviour whilst having similar parameters. This can be at least partly explained by the difference in the movement algorithms. We also noticed behaviour in MANA that we could not explain, namely why clusters of agents sometimes stop in the middle of the battlefield when their path to the enemy flag is clear. This suggests that the movement algorithm in MANA may not be adequate in some situations.

These experiments were used to discover types of behaviour that can be seen in each scenario and to provide an overall understanding of the two agent-based combat models rather than an in-depth analysis. If we had wished to obtain more concrete results we would have to repeat the experiments more times.

3.2 Using MANA to Model Peacekeeping

In Chapter 2 we mentioned that Schwarz and Bertshe had used MANA to model a simple food distribution scenario but had found it inadequate. We decided to replicate their experiments as far as possible using the details from their paper [6] to see if we came to the same conclusions. After correspondence with the authors we found that the scenario we experimented with was a simplified version of their own, this was not an issue since our aim was not to directly compare results.

The scenario was set up as follows: there are 60 civilian agents split into three groups of 20 and one group of 15 military agents. The civilian agents start spread out over a relatively large area whereas the military agents are grouped together. The agents were initially randomly distributed in the areas indicated in Figure 3.4. The military agents move around the grid passing through the centre of each civilian group in turn. The civilians move towards the military agents when they arrive and then move away from them, this was to represent the handover of food. We set the rules of engagement such that the civilians had a low single shot kill probability (SSKP) and the military agents had a higher SSKP but only engaged in combat if fired at.

After running this scenario we found that we had very high casualty numbers, this led to some changes being implemented. Initially the military agents were programmed to pause at the centre of each civilian group, but this resulted in the majority of the agents being killed so it was decided that the military agents should not stop. This reduced the casualty numbers but they were still very high. Consequently we reduced the SSKP for both sets of agents. This succeeded in reducing the casualty numbers further but was not ideal since there was still the same amount of gunfire so the scenario was basically one of combat. This observation confirmed one of the conclusions Schwarz and Bertsche came to: they noted that they could either model situations with full combat or none at all, there was no representation of a situation with rising, or falling, levels of conflict. We

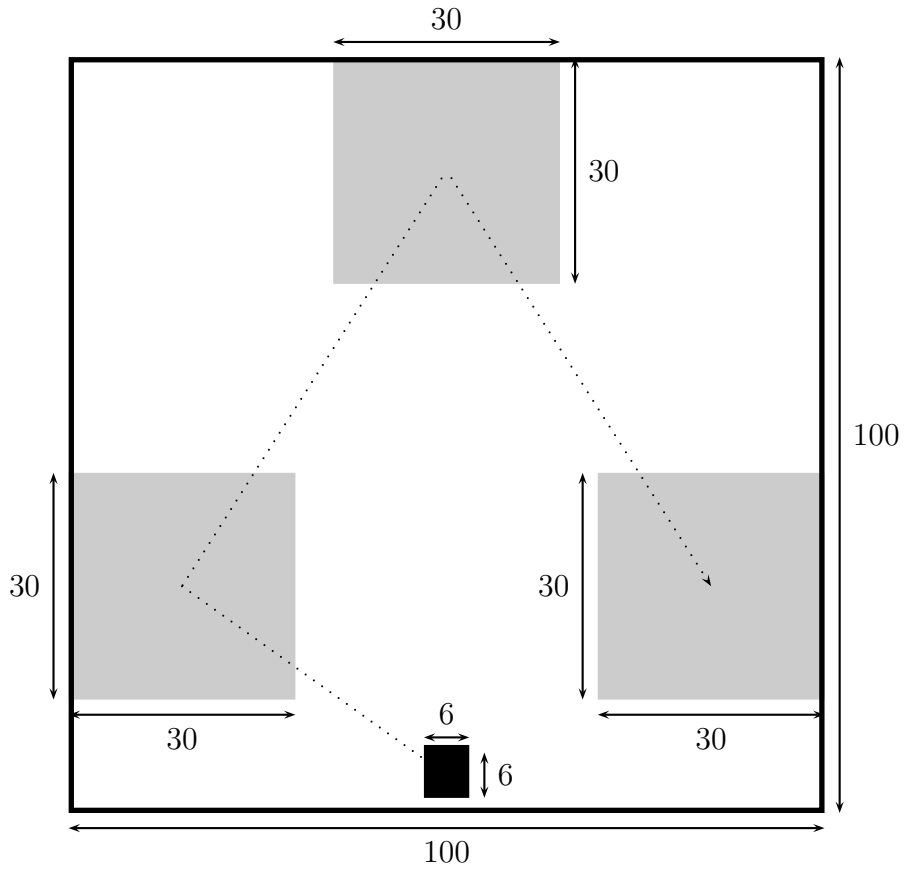

Figure 3.4: MANA Distribution Scenario

would suggest that this is in part because relationships between squads can only be hostile, neutral or friendly and they cannot be changed; in comparison the peace support model DIAMOND uses a set of five possible relationships which can change during the model run. Another limitation of MANA identified in [6] was the fact that civilians could only be modelled as a rival force. Parameters designed to model military agents are not sufficient when modelling civilians. Some are not relevant, such as the combat parameters, and the ones that are cannot accurately model civilians on their own.

These experiments confirmed that we could not use the MANA model for our research into the possible occurrence of self-organised criticality in peace support operations. This meant that the focus of our work changed to the development of a suitable model which is described later in Chapter 4.

3.3 Tension Calculations in DIAMOND

The following work was completed whilst on placement at Dstl Farnborough. It is not directly related to the power law research but was relevant to our work since we were able to gain a greater understanding of the DIAMOND model and peace support models in general. The task we were given was to model the tension relating to civilian groups at each node using the output from the peace support model DIAMOND. This would be useful in order to predict when any fighting between civilians could occur, and when to send in police forces to prevent any violence.

Civilian entities in DIAMOND are fairly benign. They are forced to move if the threat from military parties becomes too great, or if their life expectancy due to lack of food or shelter becomes too short. However they do not react to other civilian entities at the node. This is not true to life as we would expect there to be some civil unrest. For example, if there are two hostile civilian groups at a node, one of which is considerably greater in size, we would expect the smaller group to be targeted by the larger group and possibly forced out of the node.

3.3.1 Preprocessing the Data

We used the DIAMOND output files for Entity Status, Casualties, Mission and Refugees. Unfortunately some of the data we require needed to be converted to numeric form so that it could be used as input for a MATLAB program. The data set we were given by Dstl is output from a Bosnia scenario and covers a time period of 240 hours. There are seven parties involved, three military factions and their associated civilian populations and an international peacekeeping force.

First the node labels were changed. Most were initially in the form '*Node (x)*,' where x was an identifying number between one and 109. The obvious way to change these was to identify the node solely by the number x. The junction nodes have no associated number, so we put them in alphabetical order and numbered

them from 110 to 116. Some of the arcs also appear as location identifiers so we number them 117 to 131 though we will not be calculating tension along them.

Table 3.12 shows how we changed the party and entity names into numeric form. In the case of civilian entities, the final three digits of the entity ID number

Party	Party ID	Entity ID
BIH (MUSLIM)	10	$1 * **$
HVO (CROATS)	20	$2 * **$
VRS (SERB)	30	$3 * **$
IFOR	40	$4 * **$
Civilian-MUSLIM	11	$5 * **$
Civilian-CROATS	21	$6 * **$
Civilian-SERB	31	$7 * **$

Table 3.12: Numeric Identifiers for Parties and Entities

come from the node the entity is originally at. The military entities cannot be identified by this method because there is often more then one military entity from a party present at a node. Instead we look at the Entity Status output and number the entities from each military party according to the order they appear in this data.

Looking at the Mission data, we changed the mission type according to identifiers in Table 3.13. We then changed the entire 'Mission Name' column to

Mission Type	Mission ID
Defend Mission	1
Movement Mission	2
Presence Mission	3
Reserve Mission	4
Secure Mission	5
None	0

Table 3.13: Numeric Identifiers for Mission Types

represent the node the entity was moving to if it was on a movement mission. To do this we used both the Mission data which stated the name of the town

the entity was moving to, and the Entity Status data which gives the position of the entities at 24 hour intervals. The Entity Status data was used if it was unclear which was the target node. For example, if the Mission data stated the entity was moving to Sarajevo it could be moving to either Node 84, Sarajevo Centar, or Node 86, Novo Sarajevo; the Entity Status data gives us the full node identifier. If the entity was not on a movement mission we set the value to be that of the node it was stationary at.

All other text data in the four output files considered were removed from the file as they were not needed.

The data set we considered does not include any changes in relationship; if there had been any we would have had to use the Party Relationship Change output as well. Instead we only needed to use the input data that specified the relationships between the parties, specifically the civilian parties' relationships to the military parties. We specify a scale for representing relationships, the highest value being given to hostile relations. This is given in Table 3.14. In this case

Relationship	Weight
Hostile	10
Uncooperative	6
Neutral	3
Cooperative	1
Friendly	0

Table 3.14: Relationship Scale

all the civilian parties' relationship to their corresponding military party were, unsurprisingly, friendly, and their relationships to other military parties were uncooperative. The relationships to the peacekeeping force were not specified.

3.3.2 Development of the Method

Intuitively we would expect the level of tension at a node to increase if the situation at the node has degraded. Similarly, if the situation at the node has

improved we would expect the tension to decrease. If there has been no change in the situation the tension would be expected to remain approximately the same, there may be a slight increase or decrease depending upon the trend of the tension. We would also suggest that the tension would be higher if there is a noticeable difference in size between two civilian groups at a node.

From the above we saw that we needed some measure of the situation at each node. This should take into account both positive and negative factors such as amount of food and civilian deaths. We suggested a weighted sum where the values for the weights will depend on the number of military parties present at the node. As tension is related to change in situation, we will, in some cases, be considering changes in factors. For example, we will be considering the percentage change in military personnel at a node rather than the actual number that are there. In other cases, such as amount of food and friendly entities approaching the node, we will be looking at the actual numbers. This is done so that if the situation at the node remains the same we will see either no change or some slight change in the tension. For example, if there is no food and no entity is approaching the node there would be no change in the tension.

The Mission and Casualty data is recorded as events occur, the Entity Status and Refugee data is recorded at 24 hour intervals. As the scenario we are considering was only run for 240 hours, we have decided to calculate tension at hourly intervals rather than daily. This means that we will be relying on the Mission and Casualty output more than the Entity Status data.

We decided to identify four main negative and four main positive factors to include in the sum. If they did not give reasonable results we would increase the number of elements. All the factors should be in the range $[-1, 1]$. We identified the four most important negative factors as civilian deaths at the node, military deaths at the node, civilian deaths in the network and civilians without shelter at the node. The main positive factors were thought to be friendly military at the node, civilian entities approaching the node, military entities approaching the

node and food stored at the node.

We needed to ensure that the factors all fell into the range $[-1, 1]$. We give party deaths at the node as a percentage of the total number of personnel from that party at the node. Similarly civilian deaths in the network are given as a percentage of the total number of civilians in that party at the previous timestep. Change in military presence at the node is given as a percentage change from the previous timestep. As this value has to be restricted to the range $[-1, 1]$, if it is less than minus one we define it to be minus one, similarly if it is greater than one we define it to be one. This would be needed if, for example, a military entity is entering the node after there was previously no presence, or alternatively if all party entities left a node. The number of civilians without shelter is given as a percentage of those at the node. The amount of food is given as the number of weeks of food stored, but again restricted to $[-1, 1]$. This is calculated using the Entity Status data which gives the number of rations available, one ration can feed one person for a day. The decision to use weeks was taken because changes in the amount of food available beyond one week would not affect the tension to any great extent. The number of entities approaching the node is given as a percentage of the total number of entities for that party. The decision to use entity rather than personnel numbers was taken because civilian entities split before moving and it does not seem possible to tell how many are in the travelling part from the output data.

We define three sets of weights for the factors depending on the number of military parties present at the node. If there are two or more military parties present then there is the possibility of combat. If there is one military party present then a rival civilian entity at that node may be intimidated which would affect tension. If there are no military entities present then tension will mainly be affected by non combat factors such as food and shelter. Table 3.15 shows the weights we decided for each case, a higher value in modulus indicates a greater influence on tension.

Factor	Number of Military Parties Present		
	≥ 2	1	0
Civilian Deaths at Node (f_{1j})	$w_1 = 4$	$w_1 = 4$	$w_1 = 4$
Military Deaths at Node (f_{2j})	$w_2 = 3$	$w_2 = 1$	$w_2 = 1$
Civilians Without Shelter (f_{3j})	$w_3 = 2$	$w_3 = 3$	$w_3 = 3$
Civilian Deaths in Network (f_{4j})	$w_4 = 1$	$w_4 = 2$	$w_4 = 2$
Change in Military Presence (f_{5j})	$w_5 = -4$	$w_5 = -4$	$w_5 = -1$
Food Stored (f_{6j})	$w_6 = -3$	$w_6 = -2$	$w_6 = -4$
Military Entities Approaching (f_{7j})	$w_7 = -2$	$w_7 = -3$	$w_7 = -3$
Civilian Entities Approaching (f_{8j})	$w_8 = -1$	$w_8 = -1$	$w_8 = -2$

Table 3.15: Factor Weights w_i

We can now calculate the weighted sum for each civilian party at a node. The military factors refer to the military party friendly to the civilian party. To find a measure for the overall situation at the node we average the sums for the civilian parties present at the node, we call this value S_t where t denotes time. If there is only one civilian party present we need not compute the weighted sum as there will be no tension. As an example suppose there are N_t civilian parties present at a node, then S_t is calculated using Equation (3.1).

$$S_t = \frac{1}{N_t} \sum_{j=1}^{N_t} \sum_{i=1}^{8} f_{ij} w_i \tag{3.1}$$

As mentioned before, the tension will also depend upon the relative sizes of the civilian groups at the node and so we calculate the ratios between the civilian parties at the node. We use the value greater then one, rather than that in the unit interval, as we would expect the tension to increase as the difference between the sizes gets larger. If there are two civilian parties present at the node we just use this one ratio. If there are more than two parties present we use the average of all the relevant ratios. We denote the resulting value R_t. For example, suppose there are three civilian parties of sizes C_1, C_2 and C_3 at a node at time t, where

$$C_1 \geq C_2 \geq C_3.$$

Then we calculate R_t using Equation (3.2).

$$R_t = \frac{1}{3}\left(\frac{C_1}{C_2} + \frac{C_2}{C_3} + \frac{C_1}{C_3}\right) \tag{3.2}$$

We now consider the degree of hostility between the civilian parties. In the scenario we are using as an example all relationships between rival parties are uncooperative. If we have a situation where there are more than two civilian parties at a node, or if the relationships are asymmetric, we use the most hostile relationship. We denote this value H_0.

To calculate the tension we use the above values and the previous tension value. We start with a tension value equal to the relationship factor. At each subsequent timestep we add the product of the average weighted sum and the average civilian ratio to the previous tension value. We denote the tension values T_t, where T_0 is the initial value. We calculate the tension separately for each node. We call this Method One, the equation for this method is given in Equation (3.3).

$$T_t = \begin{cases} H_0 & \text{if } t = 0 \\ T_{t-1} + S_t R_t & \text{if } t > 0 \end{cases} \tag{3.3}$$

We can see the ranges for the tension calculated using this method in Tables 3.16 and 3.17. Note that all values recorded are given to four significant figures. Table 3.16 shows the nodes where significant tension was recorded, we define this to be a tension range greater than 0.1. Table 3.17 shows the nodes at which two or more civilian entities were present, but no significant amount of tension is registered. In this section we shall only look at the ranges for the tension calculation. Further analysis using graphs will be given later in Section 3.3.6.

3.3.3 Method Two

To improve on this first model we adapted the method as follows. First of all we decided to incorporate the relationship into the step-by-step calculation rather

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Velika - Kladusa (1)	6	7.916	Zenica (52)	6	105.2
			Vares (54)	-45.99	7883
Srbac (10)	6	296.5	Olovo (55)	6	69.70
Prnjavor (11)	6	6.155	Kladanj (56)	6	4154
Derventa (12)	6	2356	Vlasenica (58)	-28.30	7.553
Odzak (14)	-200.9	6	Kupres (62)	6	6.149
Orasje (16)	6	32.74	Vitez (66)	6	18.01
Gradacac (18)	6	45.15	Busovaca (67)	6	34.75
Brcko (19)	-1.636	7.988	Fojnica (68)	6	6.420
Bijeljina (20)	6	3720	Kiseljak (69)	5.368	6
Bosanski - Petrovac (21)	6	6.413	Visoko (70)	6	468.5
			Breza (71)	6	323.6
Banja - Luca (23)	6	1089	Ilijas (72)	6	7.597
			Sokolac (73)	6	28.51
Celinac (24)	6	6.219	Han - Pijesak (74)	6	6085
Doboj (25)	6	545.1			
Tesanj (26)	6	219.9	Tomislavgrad (75)	6	479.1
Srebrenik (30)	5.889	6.003	Jablanica (77)	6	4221
Tuzla (31)	6	70.60	Konjic (78)	3.937	6.142
Lopare (32)	6	1135	Ilidza (81)	6	18.2
Ugljevik (33)	6	6.838	Vogosca (83)	-5.348	7.910
Kljuc (35)	6	6.171	Sarajevo - Centar (84)	1.593	9.398
Mrkonjic - Grad (36)	6	139.9			
			Novo - Sarajevo (86)	-2.355	9.976
Skender - Vakuf (38)	6	9.297			
			Rogatica (89)	6	334.2
Banovici (43)	6	523.1	Visegrad (90)	6	6987
Zinivice (44)	6	237.6	Posusje (91)	6	326.5
Bosanski - Grahova (47)	6	6.157	Ljubuski (102)	6	6.204
			Capljina (104)	6	6.169
Sipovo (49)	6	6.412			

Table 3.16: Maximum and Minimum Values for the Nodes with Significant Tension Using Method One

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Cazin (2)	6	6.024	Zavidovici (42)	6	6.019
Bihac (3)	6	6.010	Zvornik (46)	5.960	6.003
Bosanska -	6	6.001	Glamoc (48)	6	6.031
Krupa (4)			Donji -	6	6.007
Bosanski -	6	6.056	Vakuf (50)		
Novi (5)			Travnik (51)	6	6.017
Bosanska -	6	6.052	Kakanj (53)	6	6.026
Dubica (6)			Livno (61)	6	6.016
Prijedor (7)	6	6.005	Bugojno (63)	6	6.036
Laktaci (9)	6	6.015	Gornji -	6	6.001
Bosanski -	6	6.002	Vakuf (64)		
Brod (13)			Novi -	6	6.022
Sanski -	6	6.015	Travnik (65)		
Most (22)			Prozor (76)	6	6.090
Maglaj (27)	6	6.026	Novi -	6	6.004
Gracanica (28)	6	6.073	Grad (82)		
Lukavac (29)	6	6.041	Stari -	6	6.016
Jajce (37)	6	6.018	Grad (85)		
Kotor -	6	6.028	Mostar (94)	6	6.006
Varas (39)			Stolac (106)	6	6.001
Zepce (41)	6	6.024			

Table 3.17: Maximum and Minimum Values for the Nodes with Minimal Tension Using Method One

then just at the beginning. This was done for two reasons: firstly, in case the relationships change throughout the scenario run and secondly, because the values for the tension got so large that any difference between start values would be negated by the increments, hence we could end up with two cooperative civilian groups with higher tension levels than two hostile groups. Instead we decided to start at the value zero and use the relationship factor as a multiplier to the weighted sum and civilian ratio product at each time step. We use H_t to represent the most hostile relationship at the node at time t. We then noticed that in many cases the dominant factor in the increments was the force ratio, as some groups are much larger than others. As a result of this we have decided to cap the ratio at the value five. This is since at this point the smaller group is greatly outnumbered, so any further opposition would not make a lot of difference. The new formula for R_t is given in Equation (3.4). Note that we are again using our example with three civilian parties at a node.

$$R_t = \max\left\{\frac{1}{3}\left(\frac{C_1}{C_2} + \frac{C_2}{C_3} + \frac{C_1}{C_3}\right), 5\right\} \tag{3.4}$$

The tension formula for Method Two is given in Equation (3.5).

$$T_t = \begin{cases} 0 & \text{if } t = 0 \\ T_{t-1} + S_t R_t H_t & \text{if } t > 0 \end{cases} \tag{3.5}$$

The resultant tension ranges are shown in Table 3.18. Here we have decided only to record those nodes with significant tension. We can see that some of the nodes that had significant readings using Method One do not appear here. This shows that the force ratio had been obscuring some of the results. For example, Table 3.16 shows that the node Velika Kladusa (1) has a tension range $[6, 7.916]$ using Method One. The corresponding range using Method Two is $[0, 0.02674]$ which does not even appear in Table 3.18 since the range is so small.

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	72.88	Kladanj (56)	0	392.4
Derventa (12)	0	563.9	Vlasenica (58)	-0.3120	0.01412
Odzak (14)	-122.5	0	Vitez (66)	0	20.79
Orasje (16)	0	160.5	Busovaca (67)	0	95.84
Gradacac (18)	0	36.29	Kiseljak (69)	-3.345	0
Brcko (19)	-45.82	11.93	Visoko (70)	0	222.7
Bijeljina (20)	0	263.4	Breza (71)	0	221.9
Banja -	0	448.1	Ilijas (72)	0	9.582
Luca (23)			Sokolac (73)	0	135.1
Doboj (25)	0	173.1	Han -	0	932.9
Tesanj (26)	0	356.9	Pijesak (74)		
Srebrenik (30)	-0.2152	0.006710	Tomislavgrad (75)	0	228.9
Tuzla (31)	0	196.7	Jablanica (77)	0	797.6
Lopare (32)	0	38.35	Konjic (78)	-3.014	0.02076
Ugljevik (33)	0	5.026	Ilidja (81)	0	73.19
Mrkonjic -	0	20.01	Vogosca (83)	-43.85	7.376
Grad (36)			Sarajevo -	-26.44	20.39
Banovici (43)	0	428.1	Centar (84)		
Zinivice (44)	0	371.2	Novo -	-50.13	23.85
Zvornik (46)	-0.2379	0.01538	Sarajevo (86)		
Zenica (52)	0	183.8	Rogatica (89)	0	30.02
Vares (54)	-0.7544	114.3	Visegrad (90)	0	131.4
Olovo (55)	0	15.25	Posusje (91)	0	52.74

Table 3.18: Maximum and Minimum Values for the Nodes with Significant Tension Using Method Two

3.3.4 Methods 3.1 and 3.2

We then decided to change the way the ratio was used. We reasoned that as the ratio between groups increased, the increase in the tension decreased. This would suggest the use of a logarithm function. We then decided that instead of using the ratio, or average, we would use the natural logarithm of the ratio and then add one. We add the one because the logarithm of one is zero which would give us a tension value of zero if the groups were of equal size. Initially we decided to remove the cap on the ratio to see if the dominance of the civilian ratio was negated by the logarithm function. Thus the ratio factor R_t would be calculated as in Equation (3.6).

$$R_t = \ln\left[\frac{1}{3}\left(\frac{C_1}{C_2} + \frac{C_2}{C_3} + \frac{C_1}{C_3}\right)\right] + 1 \tag{3.6}$$

We call this Method 3.1. The results are given in Table 3.19.

We then reinstated the cap on the ratio again at the value five. This was Method 3.2. The formula for R_t is given in Equation (3.7).

$$R_t = \ln\left[\max\left\{\frac{1}{3}\left(\frac{C_1}{C_2} + \frac{C_2}{C_3} + \frac{C_1}{C_3}\right), 5\right\}\right] + 1 \tag{3.7}$$

Note that the cap on the ratio is applied to the value before the logarithm transform. The results are given in Table 3.20. For both Methods 3.1 and 3.2 the formula for the tension calculations remains as in Equation (3.5).

3.3.5 Methods 4.1, 4.2 and 4.3

Next we decided to change the tension value used. Note that we are adapting Method 3.2 so we are still using Equation (3.6) for R_t. Instead of just using the previous value we used an average of previous values. This was because the current tension would not just be influenced by the tension an hour ago. In Method 4.1 we used an average of the previous five values. The formula for

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	84.31	Kladanj (56)	0	530.5
Derventa (12)	0	657.4	Vlasenica (58)	-0.5678	0.02570
Odzak (14)	-120.7	0	Vitez (66)	0	16.02
Orasje (16)	0	87.48	Busovaca (67)	0	61.28
Gradacac (18)	0	32.49	Kiseljak (69)	-1.829	0
Brcko (19)	-41.75	10.87	Visoko (70)	0	228.6
Bijeljina (20)	0	370.6	Breza (71)	0	210.7
Banja -	0	473.5	Ilijas (72)	0	5.334
Luca (23)		0	Sokolac (73)	0	120.0
Doboj (25)	0	0	Han -	0	1044
Tesanj (26)	0	277.6	Pijesak (74)		
Srebrenik (30)	-0.1609	0.005016	Tomislavgrad (75)	0	234.8
Tuzla (31)	0	129.3	Jablanica (77)	0	967.7
Lopare (32)	0	59.70	Konjic (78)	-2.424	0.167
Ugljevik (33)	0	4.255	Ilidja (81)	0	50.72
Mrkonjic -	0	25.22	Vogosca (83)	-26.74	4.500
Grad (36)			Sarajevo -	-15.39	11.86
Banovici (43)	0	393.0	Centar (84)		
Zinivice (44)	0	291.7	Novo -	-35.19	16.77
Zvornik (46)	-0.1459	0.009433	Sarajevo (86)		
Zenica (52)	0	139.1	Rogatica (89)	0	40.78
Vares (54)	-1.303	197.4	Visegrad (90)	0	218.0
Olovo (55)	0	17.78	Posusje (91)	0	65.46

Table 3.19: Maximum and Minimum Values for the Nodes with Significant Tension Using Method 3.1

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	38.04	Kladanj (56)	0	204.8
Derventa (12)	0	294.3	Vlasenica (58)	-0.1628	0.007369
Odzak (14)	-63.95	0	Vitez (66)	0	10.85
Orasje (16)	0	87.48	Busovaca (67)	0	50.02
Gradacac (18)	0	18.94	Kiseljak (69)	-1.746	0
Brcko (19)	-41.75	10.87	Visoko (70)	0	116.2
Bijeljina (20)	0	137.5	Breza (71)	0	115.8
Banja -	0	233.8	Ilijas (72)	0	5.334
Luca (23)			Sokolac (73)	0	120.0
Doboj (25)	0	90.32	Han -	0	486.9
Tesanj (26)	0	186.3	Pijesak (74)		
Srebrenik (30)	-0.1123	0.003502	Tomislavgrad (75)	0	119.5
Tuzla (31)	0	102.6	Jablanica (77)	0	416.3
Lopare (32)	0	20.01	Konjic (78)	-1.573	0.1084
Ugljevik (33)	0	4.255	Ilidja (81)	0	50.72
Mrkonjic -	0	10.44	Vogosca (83)	-22.88	3.849
Grad (36)			Sarajevo -	-15.39	11.86
Banovici (43)	0	223.4	Centar (84)		
Zinivice (44)	0	193.7	Novo -	-35.19	16.77
Zvornik (46)	-0.1459	0.009433	Sarajevo (86)		
Zenica (52)	0	95.91	Rogatica (89)	0	15.67
Vares (54)	-0.3937	59.66	Visegrad (90)	0	68.57
Olovo (55)	0	7.958	Posusje (91)	0	27.53

Table 3.20: Maximum and Minimum Values for the Nodes with Significant Tension Using Method 3.2

tension calculation now becomes Equation (3.8).

$$T_t = \begin{cases} 0 & \text{if } t = 0 \\ T_0 + S_1 R_1 H_1 & \text{if } t = 1 \\ \dfrac{T_1 + T_0}{2} + S_2 R_2 H_2 & \text{if } t = 2 \\ \dfrac{T_2 + T_1 + T_0}{3} + S_3 R_3 H_3 & \text{if } t = 3 \\ \dfrac{T_3 + T_2 + T_1 + T_0}{4} + S_4 R_4 H_4 & \text{if } t = 4 \\ \dfrac{T_{t-1} + T_{t-2} + T_{t-3} + T_{t-4} + T_{t-5}}{5} + S_t R_t H_t & \text{if } t \geq 5 \end{cases} \tag{3.8}$$

The tension ranges calculated are given in Table 3.21.

Next we reasoned that we should use a weighted average so that recent tension values were more influential than older ones. For this we used the exponential smoothing method. All previous tension values were included and the ratio between successive weights remains constant. For example, suppose we have four previous values and the most recent is given a weighting of one. Then if we define the ratio between the weights to be 0.5 say, the weights for the third, second and first values will be a half, a quarter and an eighth respectively. To calculate the averaged tension value we would calculate the weighted sum and divide by the sum of the weights. In our work we begin by using a weight ratio of 0.5, this was

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	11.14	Kladanj (56)	0	59.98
Derventa (12)	0	86.20	Vlasenica (58)	-0.2486	0
Odzak (14)	-42.19	0	Vitez (66)	0	10.84
Orasje (16)	0	49.08	Busovaca (67)	0	36.01
Gradacac (18)	0	15.59	Kiseljak (69)	-1.638	0
Brcko (19)	-21.52	11.43	Visoko (70)	0	47.51
Bijeljina (20)	0	39.04	Breza (71)	0	35.90
Banja -	0	67.85	Ilijas (72)	0	4.906
Luca (23)			Sokolac (73)	0	41.49
Doboj (25)	0	26.45	Han -	0	142.6
Tesanj (26)	0	54.55	Pijesak (74)		
Srebrenik (30)	-0.1147	0.001228	Tomislavgrad (75)	0	34.99
Tuzla (31)	0	30.06	Jablanica (77)	0	121.9
Lopare (32)	0	14.95	Konjic (78)	-1.419	0
Ugljevik (33)	0	4.251	Ilidja (81)	0	19.98
Mrkonjic -	0	10.44	Vogosca (83)	-25.42	2.064
Grad (36)			Sarajevo -	-13.00	10.91
Banovici (43)	0	65.43	Centar (84)		
Zinivice (44)	0	68.01	Novo -	-38.20	16.72
Zvornik (46)	-0.1522	0.003332	Sarajevo (86)		
Zenica (52)	0	28.09	Rogatica (89)	0	15.66
Vares (54)	-14.20	31.69	Visegrad (90)	0	31.82
Olovo (55)	0	5.116	Posusje (91)	0	8.059

Table 3.21: Maximum and Minimum Values for the Nodes with Significant Tension Using Method 4.1

Method 4.2. The equation for the method is given as Equation (3.9).

$$T_t = \begin{cases} 0 & \text{if } t = 0 \\ \dfrac{\sum_{i=1}^{t} (0.5)^{i-1} T_{t-i}}{\sum_{i=1}^{t} (0.5)^{i-1}} + S_t R_t H_t & \text{if } t > 0 \end{cases} \tag{3.9}$$

The ranges are given in Table 3.22

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	20.40	Kladanj (56)	0	109.9
Derventa (12)	0	157.9	Vlasenica (58)	-0.1239	0.003702
Odzak (14)	-42.41	0	Vitez (66)	0	10.84
Orasje (16)	0	59.75	Busovaca (67)	0	40.65
Gradacac (18)	0	16.43	Kiseljak (69)	-1.638	0
Brcko (19)	-26.86	11.41	Visoko (70)	0	69.71
Bijeljina (20)	0	70.33	Breza (71)	0	62.25
Banja -	0	124.3	Ilijas (72)	0	4.906
Luca (23)			Sokolac (73)	0	67.49
Doboj (25)	0	48.45	Han -	0	261.2
Tesanj (26)	0	99.92	Pijesak (74)		
Srebrenik (30)	-0.1141	0.001824	Tomislavgrad (75)	0	64.10
Tuzla (31)	0	55.06	Jablanica (77)	0	223.3
Lopare (32)	0	16.22	Konjic (78)	-0.9825	0.09832
Ugljevik (33)	0	4.252	Ilidja (81)	0	27.90
Mrkonjic -	0	10.44	Vogosca (83)	-24.46	2.647
Grad (36)			Sarajevo -	-13.47	10.91
Banovici (43)	0	119.8	Centar (84)		
Zinivice (44)	0	109.7	Novo -	-34.32	16.72
Zvornik (46)	-0.1506	0.004886	Sarajevo (86)		
Zenica (52)	0	51.45	Rogatica (89)	0	15.66
Vares (54)	-10.63	38.85	Visegrad (90)	0	44.00
Olovo (55)	0	5.118	Posusje (91)	0	14.76

Table 3.22: Maximum and Minimum Values for the Nodes with Significant Tension Using Method 4.2

We then changed the weight ratio to 0.75, this was Method 4.3. This method

is given in Equation (3.10).

$$T_t = \begin{cases} 0 & \text{if } t = 0 \\ \dfrac{\sum_{i=1}^{t} (0.75)^{i-1} T_{t-i}}{\sum_{i=1}^{t} (0.75)^{i-1}} + S_t R_t H_t & \text{if } t > 0 \end{cases} \tag{3.10}$$

It is the ratio between the weights that matters rather than the actual values so it did not matter what initial value we chose, we used $(0.75)^0$ which is equal to one. This was our final method. We computed the ranges for the nodes with minimal tension in addition to those with significant tension. The results are given in Tables 3.24 and 3.23 respectively.

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Srbac (10)	0	12.35	Kladanj (56)	0	66.51
Derventa (12)	0	95.58	Vlasenica (58)	-0.1045	0.001937
Odzak (14)	-42.08	0	Vitez (66)	0	10.84
Orasje (16)	0	49.87	Busovaca (67)	0	36.36
Gradacac (18)	0	15.17	Kiseljak (69)	-1.638	0
Brcko (19)	-19.41	11.69	Visoko (70)	0	48.16
Bijeljina (20)	0	40.81	Breza (71)	0	37.39
Banja -	0	75.23	Ilijas (72)	0	4.906
Luca (23)			Sokolac (73)	0	43.41
Doboj (25)	0	29.33	Han -	0	158.1
Tesanj (26)	0	60.49	Pijesak (74)		
Srebrenik (30)	-0.1150	0.0009846	Tomislavgrad (75)	0	38.80
Tuzla (31)	0	33.33	Jablanica (77)	0	135.2
Lopare (32)	0	14.32	Konjic (78)	-0.6873	0.09643
Ugljevik (33)	0	4.251	Ilidja (81)	0	19.75
Mrkonjic -	0	10.44	Vogosca (83)	-25.43	2.046
Grad (36)			Sarajevo -	-12.78	10.91
Banovici (43)	0	72.55	Centar (84)		
Zinivice (44)	0	71.13	Novo -	-33.89	16.72
Zvornik (46)	-0.1530	0.002613	Sarajevo (86)		
Zenica (52)	0	31.15	Rogatica (89)	0	15.66
Vares (54)	-15.75	28.44	Visegrad (90)	0	32.72
Olovo (55)	0	5.115	Posusje (91)	0	8.935

Table 3.23: Maximum and Minimum Values for the Nodes with Significant Tension Using Method 4.3

Node	Min. Tension	Max. Tension	Node	Min. Tension	Max. Tension
Velika - Kladusa (1)	0	0.003580	Zepce (41)	0	0.003581
			Zavidovici (42)	0	0.003581
Cazin (2)	0	0.001727	Bosanski - Grahova (47)	0	0.003874
Bihac (3)	0	0.001790			
Bosanska - Krupa (4)	0	0.001422	Glamoc (48)	0	0.003581
			Sipovo (49)	0	0.003581
Bosanski - Novi (5)	0	0.003580	Donji - Vakuf (50)	0	0.003581
Bosanska - Dubica (6)	0	0.003580	Travnik (51)	0	0.001790
			Kakanj (53)	0	0.003581
Prijedor (7)	0	0.001790	Livno (61)	0	0.001790
Laktaci (9)	0	0.003580	Kupres (62)	0	0.003454
Prnjavor (11)	0	0.003581	Bugojno (63)	0	0.003581
Bosanski - Brod (13)	0	0.002622	Gornji - Vakuf (64)	0	0.001135
Bosanski - Petrovac (21)	0	0.003581	Novi - Travnik (65)	0	0.001727
Sanski - Most (22)	0	0.001790	Fojnica (68)	0	0.003581
			Prozor (76)	0	0.003454
Celinac (24)	0	0.003581	Novi - Grad (82)	0	0.003580
Maglaj (27)	0	0.001790			
Gracanica (28)	0	0.001790	Stari - Grad (85)	0	0.003581
Lukavac (29)	0	0.003581			
Kljuc (35)	0	0.001790	Mostar (94)	0	0.001790
Jajce (37)	0	0.001790	Ljubuski (102)	0	0.003581
Skender - Vakuf (38)	0	0.003874	Capljina (104)	0	0.001790
			Stolac (106)	0	0.001372
Kotor - Varas (39)	0	0.003581			

Table 3.24: Maximum and Minimum Values for the Nodes with Minimal Tension Using Method 4.3

3.3.6 Comments on the Results

In this section we will be looking at the results of the tension calculations for each method in greater detail. We shall be comparing the methods to see what effect the changes have on the development of the tension. To do this we will be concentrating on two nodes, Orasje (16) and Bijeljina (20), and using the graphs from them to identify any changes. We will also be using values recorded in the tables in Section 3.3.2. The graphs for the final method, Method 4.3, are given in Appendix A for the additional nodes where the tension was significant.

Method One

Figure 3.5 shows the tension pattern for Orasje. To start with there is a steady increase in tension before it reaches a constant level. At around 50 hours there is a sharp increase followed by fluctuations. After approximately 125 hours the tension remains constant.

Figure 3.6 shows the results for Bijeljina. We see a steady increase in tension to start with before a period of constant tension. There is a change in situation at approximately time 75 which leads to the tension moving to a higher level. At around 100 hours there is an increase in tension until it levels out at the value 3720. Notice that the range for this node is much larger than that for Orasje.

Method Two

The graph in Figure 3.7 shows the tension calculated for Orasje using Method Two. We can compare it to that calculated using Method One shown in Figure 3.5. We notice that the two graphs show the same overall pattern, but have different ranges. As the range has increased we can assume that the civilian ratio at the node is below five. Indeed, if we look at the input data for the scenario we calculate the civilian ratio to initially be 4.655. Therefore the change has come from the multiplication by the relationship value, although the ratio is close to

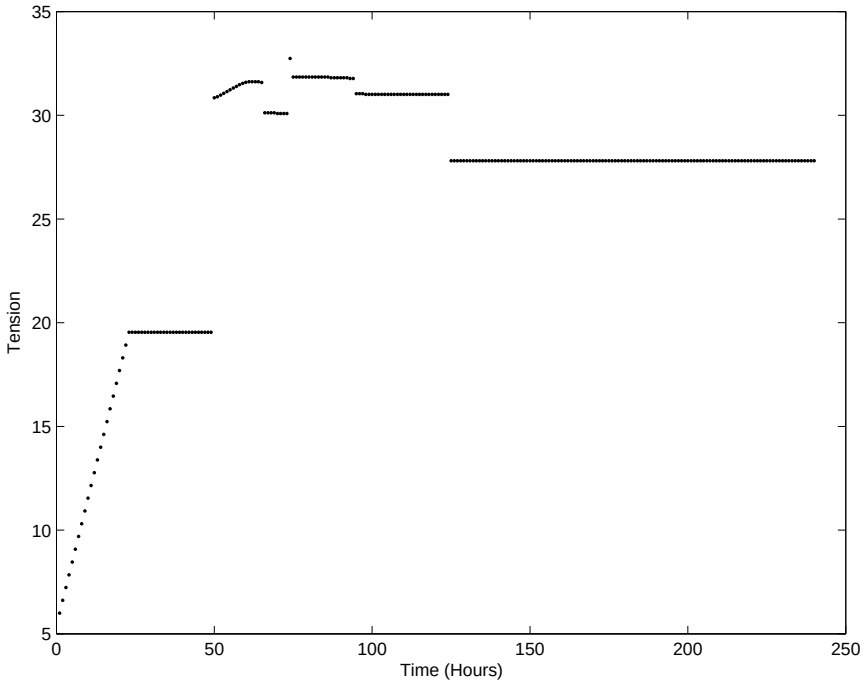

Figure 3.5: Tension at Orasje (16) Using Method One

five so it is feasible that it could have increased to over five and there may be some minimal effect from the capping of the ratio.

If we look at the initial number for the civilians at Bijeljina we calculate the initial average ratio to be 399.2. As this is much larger than five we can predict that the range will decrease as the cap in the ratio will affect the tension calculations. We can see that this is indeed the case from the results in Section 3.3.2. The range of values for Method One is $[6, 3720]$, whereas it is $[0, 263.4]$ for Method Two. The graph for Bijeljina is shown in Figure 3.8. We can see the reduction in range, but again the pattern of the graph is very similar to that in Figure 3.6.

Method 3.1

When we changed from Method Two to Method 3.1 we changed the force ratio factor again. We would therefore expect there to be a change in the ranges. This

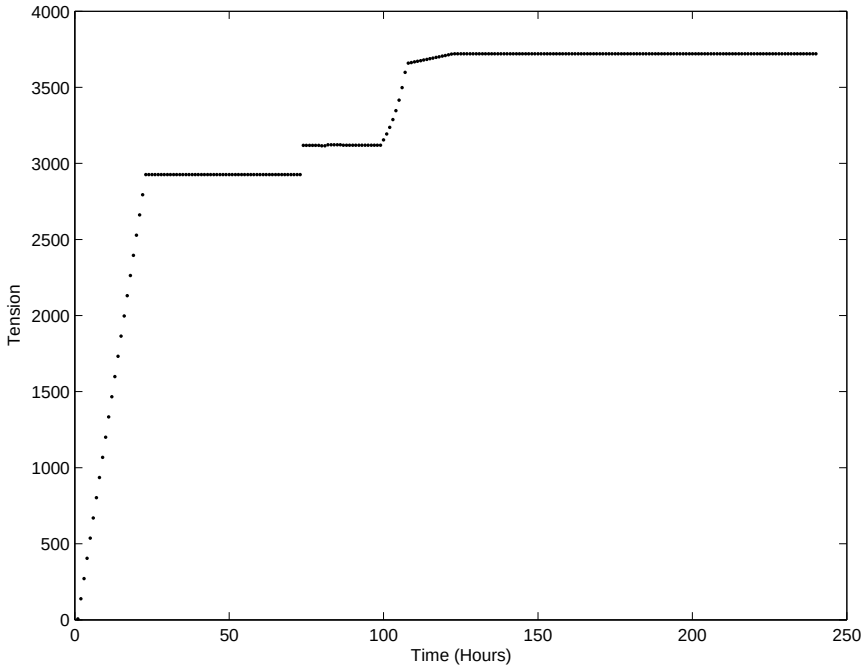

Figure 3.6: Tension at Bijeljina (20) Using Method One

was indeed the case as shown in Table 3.19. Recall that in Method 3.1 we use a ratio factor

$$\ln(AverageRatio) + 1$$

instead of the actual ratio but capped at a maximum of five used in Method Two. This suggests that any civilian ratios less than exp(4) will have a lower ratio factor in Method 3.1 than in Method Two. In which case we would expect the range for the tension to decrease. Similarly, if the civilian ratio is above exp(4) then

$$\ln(AverageRatio) + 1 > 5,$$

and so the ratio factor will have increased. This would lead to an increase in the range for the tension. As the only change in the method was the ratio factor we would not expect to see any change in the overall shape of the graph, just in the range.

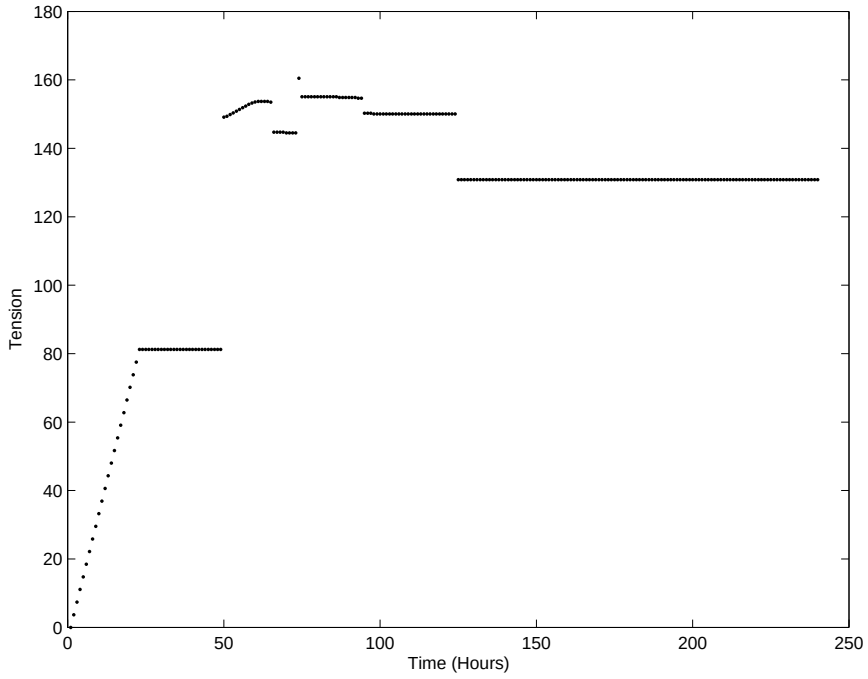

Figure 3.7: Tension at Orasje (16) Using Method Two

Our results show our predictions to be correct. Figure 3.9, showing the tension for Orasje, has a range $[0, 87.48]$ which is nearly half of that for Method Two. This was expected since the initial civilian ratio of 4.655 is less than exp(4).

However, the initial civilian ratio for Bijeljina was 399.2 which is greater than exp(4). Comparing Figure 3.10 to Figure 3.8 shows that, as predicted, the tension range for this node has increased.

Method 3.2

The difference between Methods 3.1 and 3.2 is the reintroduction of the capping of the ratio at value five. Recall that this cap is used on the actual ratio before the ratio factor is calculated. This means that the only changes in tension calculated with Methods 3.1 and 3.2 occur at the nodes where the civilian ratio is greater than five. The graph for the tension at the Orasje node is identical to that for

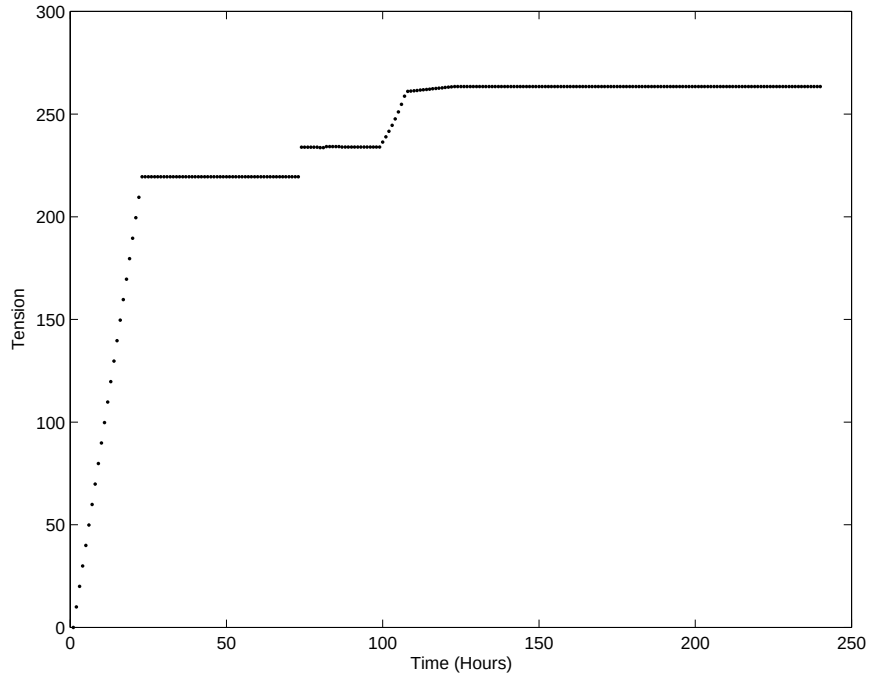

Figure 3.8: Tension at Bijeljina (20) Using Method Two

Method 3.1. This suggests that the civilian ratios at the node did not reach the value five. However, at the Bijeljina node the civilian ratio is initially 399.2 so the range for the tension will be affected. The results are shown in Figure 3.11. We can see from the graph that the tension range has decreased significantly.

Method 4.1

So far we have seen that changing the method used to calculate tension produces changes in the ranges of calculated values, but no real change in pattern. This is because we have mainly been changing the ratio factor which is a multiplier in the methods. We will now be looking at methods where we change the way we use previous tension values in the calculation of current tension. This is a more significant change as it should affect the overall shape of the graph as well as the range of values.

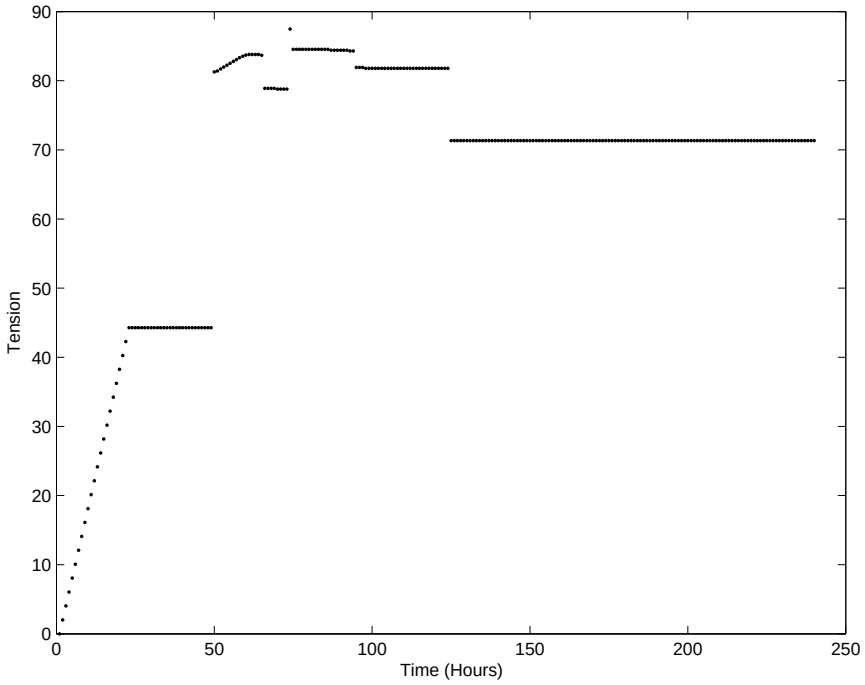

Figure 3.9: Tension at Orasje (16) Using Method 3.1

Figure 3.12 shows the tension calculations for Orasje using Method 4.1. Comparing the graph to Figure 3.9 we can see that we do indeed have a change in the shape of the graph. The fluctuations have less structure than those in the graph for Method 3.1. Instead of having points in obvious groupings there are a lot more points on their own, and one point, the maximum, that appears to be an outlier. Instead of jumping between levels there are more gradual increases and decreases. This would seem to be a more feasible representation of what would actually happen at the node. There is also a decrease in the range with most of the points lying in the interval $[0, 30]$.

The graph in Figure 3.13 shows the tension for Bijeljina. We see that for this node there has been less of a change. This is because there were gradual increases between some of the levels in the graph using previous methods. We do however notice that there is a considerable decrease in the range for the tension values.

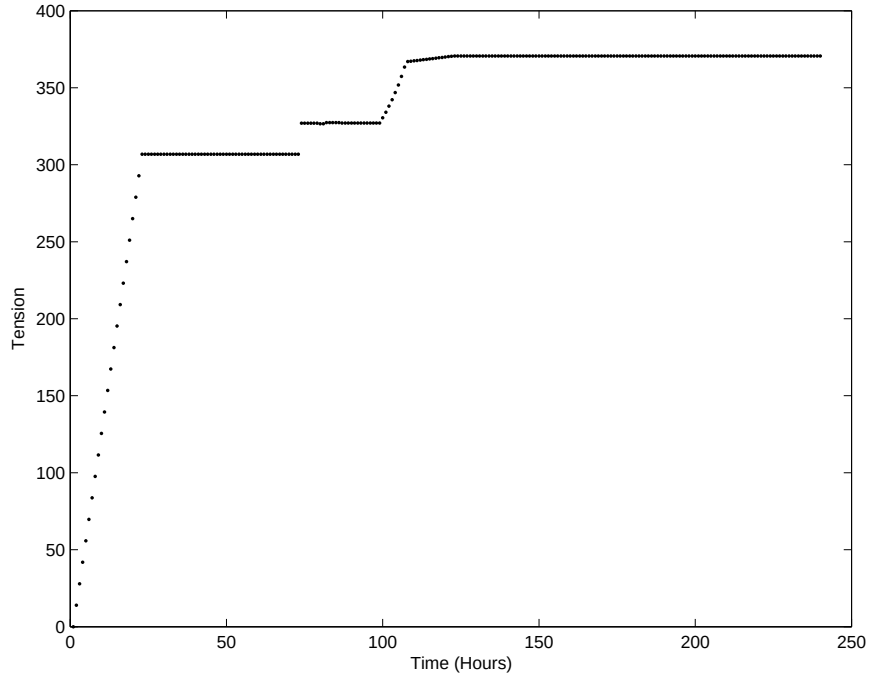

Figure 3.10: Tension at Bijeljina (20) Using Method 3.1

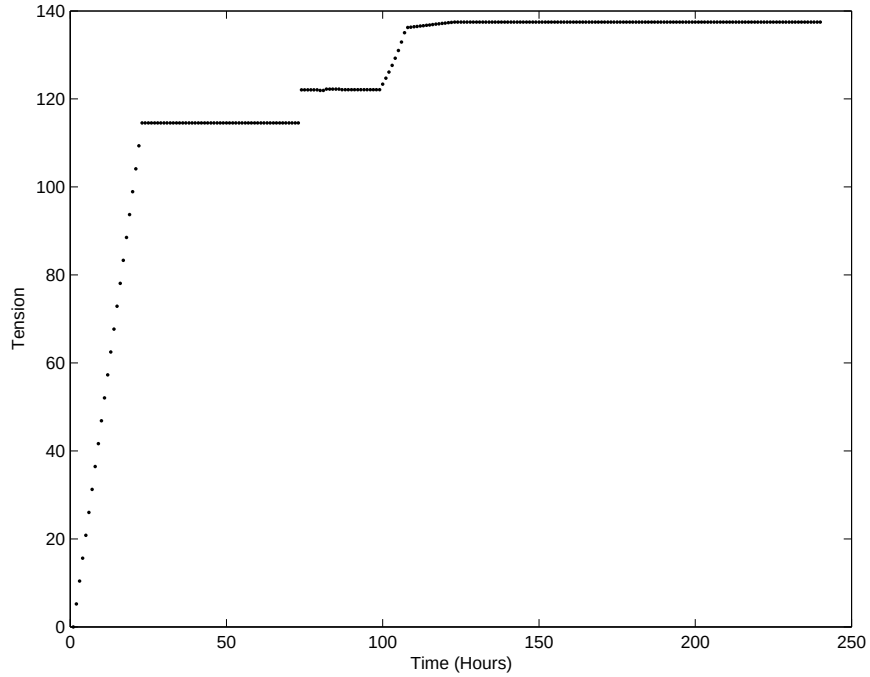

Figure 3.11: Tension at Bijeljina (20) Using Method 3.2

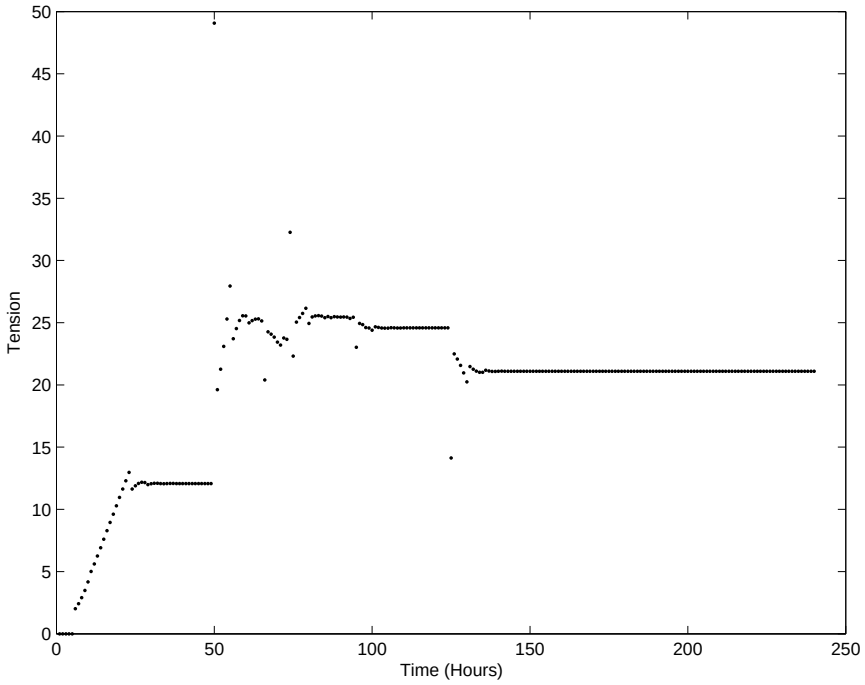

Figure 3.12: Tension at Orasje (16) Using Method 4.1

Method 4.2

Recall that in Method 4.2 we introduced exponential smoothing. This meant that all previous tension values were taken into account and were weighted according to how far in the past they were. Figure 3.14 shows the graph for tension at Orasje using this method. We can see that the levels of tension are now clear again. This is because we are giving more weight to recent tension values rather than just averaging the previous five values with equal weight. Though we would expect to see gradual increases and decreases rather than distinct levels, it seems more sensible to attach higher weights to more recent values. Therefore, although the graph in Figure 3.12 for Method 4.1 would appear to be truer to life, Method 4.2 would seem to be intuitively better. Notice that there has also been an increase in range.

The graph for the Bijeljina node is shown in Figure 3.15. Again this graph is

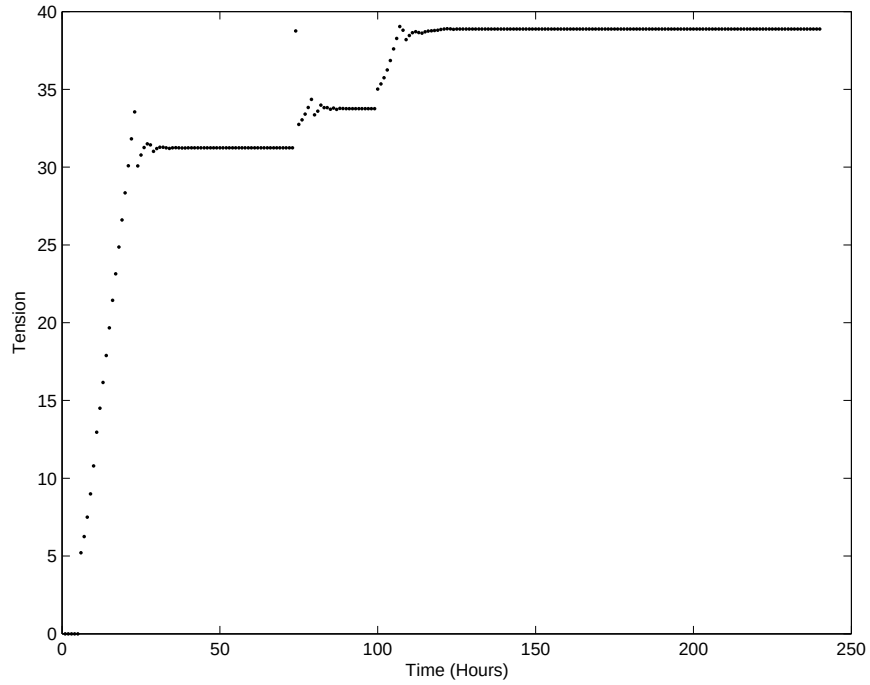

Figure 3.13: Tension at Bijeljina (20) Using Method 4.1

smoother than that for Method 4.1. In fact it is very similar in shape to those from Methods One, Two, 3.1 and 3.2.

Method 4.3

In order to try to reproduce plots with as much fluctuation as those from Method 4.1, but retaining the exponential smoothing from Method 4.2, we increased the weighting ratio from 0.5 to 0.75. This was our final method. If we look at the results for Orasje in Figure 3.16 we can see that this change has not made much difference to the appearance of the graph. We notice that the maximal point seems to be an outlier in both Figure 3.14 and Figure 3.16, but it is more obvious in the graph for Method 4.3.

The graph in Figure 3.17 shows the results for the Bijeljina node. We see that there is not as much fluctuation as in Figure 3.13 where we were using Method

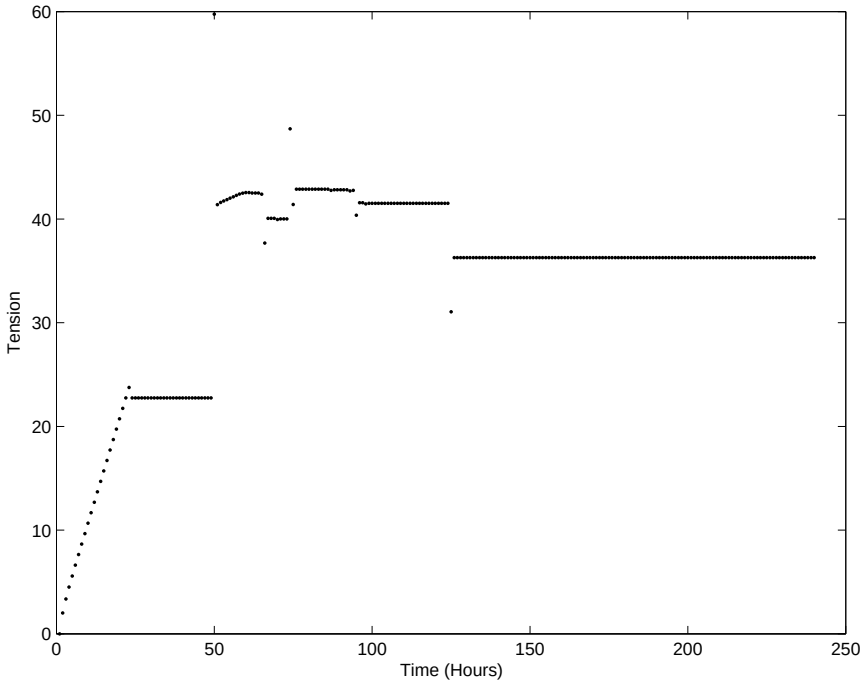

Figure 3.14: Tension at Orasje (16) Using Method 4.2

4.1, but we do see more than in Figure 3.15. We also notice that the range has again been decreased.

We decided Method 4.3 should be our final method because it seems to be the best intuitively. The exponential smoothing ensures that all previous tension values are taken into account, and to a lesser degree as the time increases. It also seems that increasing the weight ratio to 0.75 from 0.5 gives better results because the ranges for the results decreased. This suggests that the tension had a slower build up, which is what we would expect to happen at the nodes.

3.3.7 Conclusions and Suggested Improvements

The method detailed in this report is just one way of determining tension. It could be further improved and justified if more time were available. Unfortunately we do not know the details of the scenario we were testing the model on, therefore

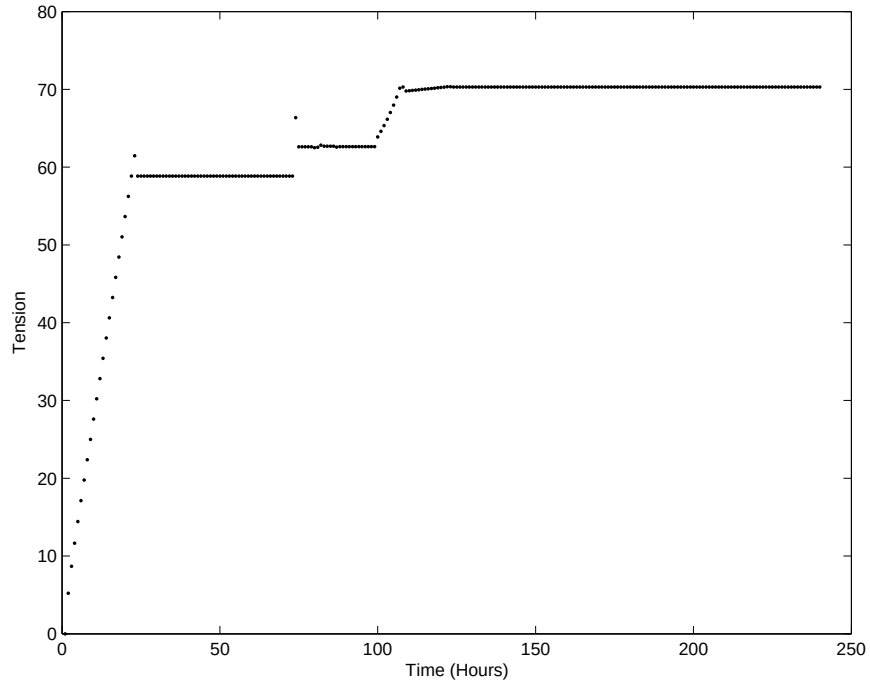

Figure 3.15: Tension at Bijeljina (20) Using Method 4.2

it is difficult for us to determine how viable our method is.

Our method does not take into account some of the factors that would affect tension between civilian groups. For example, the peacekeeping forces are not used in the model and we would expect their presence to affect the tension. They were not included because we wished to keep the number of factors to a minimum and we felt that the other variables included in the model were more important. There is also the question of how the peacekeepers would have affected the tension. We could argue that they would reduce it as they would discourage any violent action, alternatively we could say that they would add to it as they increase the military presence. We could also have distinguished between the causes of the deaths for the civilians. In the scenario we were considering this was not a problem as all deaths were due to combat attrition, but other scenarios would have death due to lack of food, shelter and medical facilities.

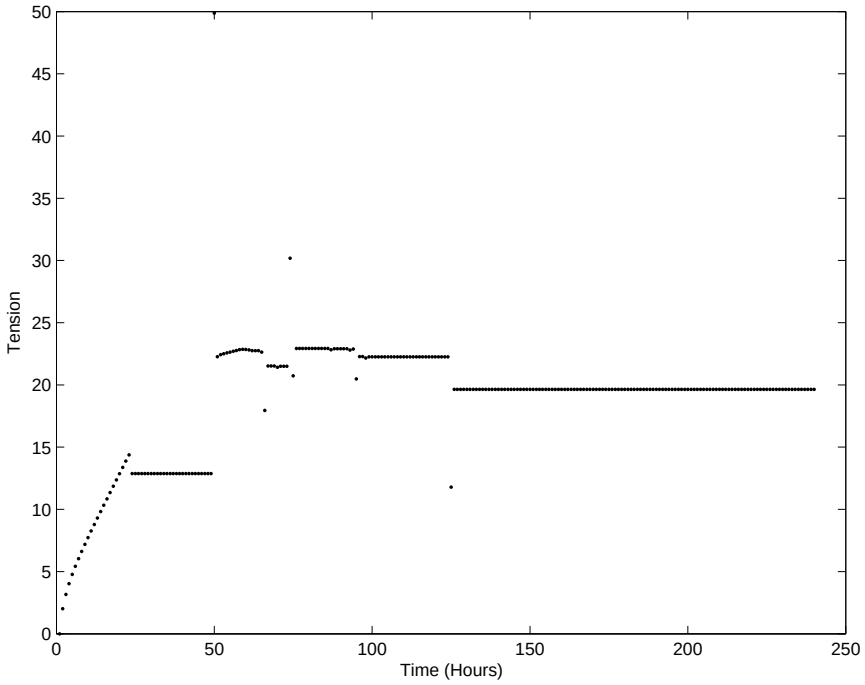

Figure 3.16: Tension at Orasje (16) Using Method 4.3

Other methods could have been used to approach the problem. For example, the use of systems dynamics methods and software was considered during our work. Instead we decided to use our current approach as we could incorporate averaging and exponential smoothing to take account of previous values. We were also familiar with the MATLAB application so it seemed better to use this rather than learning new software. An alternative method could perhaps be developed by someone who was more familiar with systems dynamics theory.

Our main problem with the work was determining the factors that would affect tension, and the subsequent ordering according to the situation. We guessed which factors would be the most important but someone with more knowledge of sociology would be better informed and would be able to predict the reaction of civilians to certain events.

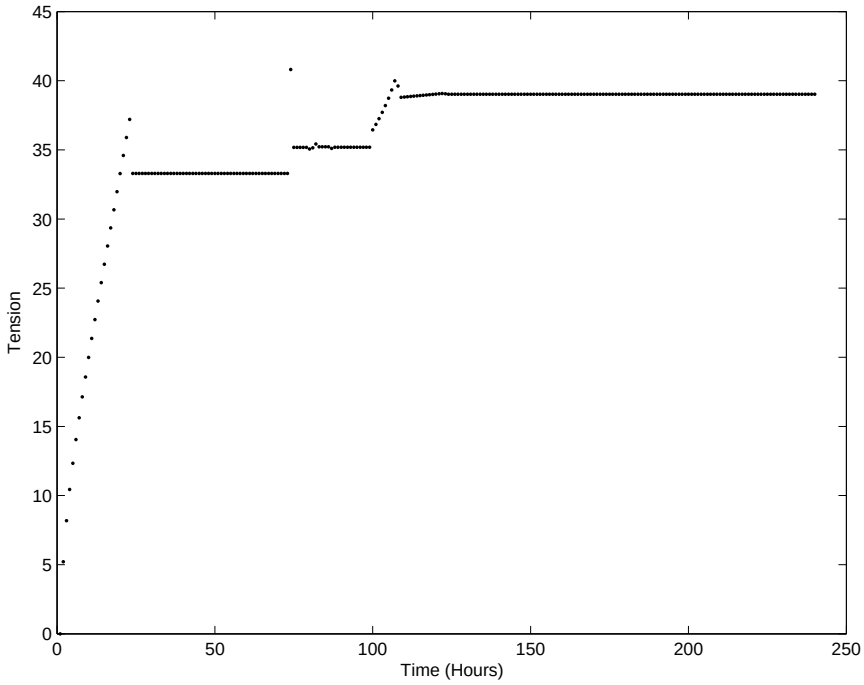

Figure 3.17: Tension at Bijeljina (20) Using Method 4.3

3.4 Conclusions

The work completed with the MANA and ISAAC models provided an excellent introduction to agent-based combat models, and understanding how they worked helped when we were developing our own agent-based model. The Schwarz and Bertsche experiments identified the limitations of MANA when used for modelling peacekeeping scenarios. We were able to address these limitations in the design process for our peace support model.

The work we did with the DIAMOND model proved very useful when we came to develop our own peace support model, described next in Chapter 4, since one of the aims for our work is that it can potentially be used in conjunction with the DIAMOND model. Also, our peace support model does not currently have a tension calculation incorporated into it, but this was something we considered during the design process and if time had allowed it would have been included.

The method described here for DIAMOND could be the starting point for a measure that could be used in a future development of our own model.

Bibliography

[1] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume I : Entities, Terrain, Environment & Facilities (2000), DERA Report.

[2] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume II : Movement, Sensing, Communication, Local Picture, Perception, Relationships, Negotiation and Test Scenarios (2000), DERA Report.

[3] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume III : Missions, Command & Control, Operations, Logistics and Model Outputs (2000), DERA Report.

[4] A. Ilachinski, Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial-Life Approach to Land Warfare (1997).

[5] M. K. Lauren & R. T. Stephen, Map Aware Non-Uniform Automata, Version 1.0 Users Manual (2001).

[6] G. Schwarz & K. A. Bertsche, Agent-Based Simulation of (De-)Escalation in Peace Support Operations: Application of the Model PAX, 20 ISMOR August 2003.

Chapter 4

MODEL DESIGN

After we had looked at previous research and current agent-based combat models we were able to use this knowledge to develop our own model. We had two main objectives when developing the model: one was to produce a model that would be a useful addition to the currently available peace support models. In particular, we aimed to develop a model that could be used in conjunction with DIAMOND, providing a low-level representation of events to complement the high-level view given by DIAMOND. Secondly, we wished to use this model to look for evidence of power law behaviour which may be indicative of self-organised criticality.

We have chosen to focus on scenarios where the Peacekeepers, and NGOs, are aiming to repair any failures to the water or electricity supplies on the grid. Thus one part of the analysis in Chapter 5 will be to look at how successful they were at this. We shall also be looking at the number of casualties each squad suffered. However, the main analysis will be looking for evidence of power-laws that may be an indicator of self-organised criticality (SOC) so we needed to identify behaviour that could be seen as analogous to the 'avalanches' described earlier in Chapter 2.

Here we provide a description of the agent-based model we have developed to represent peace support operations, along with a discussion of the design process. The structure of the chapter is as follows. Sections 4.1 and 4.2 describe the general

model details and parameters. We describe the four different types of agent objects in Section 4.3 before moving on to look at the cell objects in Section 4.4. The way the whole model is put together is discussed in Section 4.5 and the individual functions that are used throughout the program are described in more detail in Section 4.6. Section 4.7 describes the data generated by the model before Section 4.8 looks at how we can use some of this data to visualise the model runs with the aid of a simple MATLAB program. Section 4.9 gives our definition of two types of avalanche we can measure in the model. Finally we address the verification and validation of the model in Sections 4.10 and 4.11 respectively.

4.1 Overview

The basic structure of the model is a square grid that comprises a number of cells and agents move around this grid in accordance with certain rules. We took some of the concepts from the agent-based combat models MANA and ISAAC as a basis then added our own ideas so that our model concentrates less on combat and more on the needs of the civilians. For example, all the combat that occurs is caused by local insurgents and we can alter the parameters in the model so that they do not have to be included at all. Also, unlike in MANA and ISAAC, the agents do not have a goal as such, all their moves are determined by what is happening around them.

The ultimate aim for the scale of the model is such that it would be able to represent a town or a group of villages in detail. This is so that there is the potential for use in conjunction with the DIAMOND model. We have been able to model grid sizes up to 200×200 cells. This may be useful for modelling a town if we take an agent object to represent more than one person, but this would not provide the detail hoped for. Therefore, in the experiments we have carried out we decided to model one agent to be one person and have looked at a smaller area. We hope that this development model can be expanded in future work so that larger areas can be modelled. A full discussion of future model developments is given later in Chapter 7.

The model has been written using the object-oriented programming language C++. This seemed the obvious choice since this is the programming language Dstl use for all of their models. This means that we have been able to create agent and cell objects to store various parameters.

The C++ source code consists of ten separate files that combine to give the model. There are files containing the constructors for the agents and cells, function files for the combat, repair, initial positions and combat functions along with the main file. In addition, there are header files for the agent and cell object

definitions along with one for the main model. The full C++ code for the model is given in Appendix B.

4.2 General Parameters

The header file *agent.h* contains the $NOOFSQUADS$ definition, as the name suggests this set the number of squads that will be used in the model. A squad is a group of agents of the same type with the same personality parameters. Only the agent-specific parameters, such as location, will differ. At the moment it is only possible to have at most one squad per agent type so we have $NUMBEROFSQUADS \leq 4$. The squad sizes and capabilities are also defined in this file. The squad size is the number of agents in the squad. The squad capability is a measure of the amount the squad can do relative to the other squads. This should take into account relative movement speed and weapon specification. For example, a squad of Peacekeepers travelling in a vehicle would have a higher capability than Civilians moving on foot. This measure is used to determine how many actions the agents should do at each timestep, and is described further in Section 4.5. Finally, we have the two constants $SHOTMEMORY$ and $BOMBMEMORY$. These apply to all the agents and define how long the agents remember that there has been a shot fired to or from a cell, and how long they remember a bomb attack. Both are given as a number of timesteps and are used to determine the cell combat indicator functions given later in Section 4.4.

The *cell.h* header file contains the definitions $GRIDSIZE$, $SECTOR$, $WATERFAIL$ and $ELECFAIL$. The factor $GRIDSIZE$ defines the side length of the grid, for example $GRIDSIZE$ with value 100 gives a 100×100 grid. Similarly the $SECTOR$ factor gives the number of sectors each side is divided into. For example if we have a 100×100 grid with $SECTOR$ value two, then the whole grid is divided into four 50×50 sectors. There is a check included in the program to ensure that the value given for $GRIDSIZE$ is divisible by the $SECTOR$ value. The final two factors $WATERFAIL$ and $ELECFAIL$ are the probabilities the water and electricity supplies will fail, these values are given as a number between zero and one.

In terms of the program structure, the cell and agent parameters are set in the constructor functions. The four different agent constructors are in the file *agent.cpp*. The cell constructor function is in the file *cell.cpp*. The cell parameters are also set in the main program *main.cpp* due to a problem with this constructor function.

As it stands the program has to be compiled every time any of the parameters are changed. This is clearly not ideal and one future improvement should be that the user will input the data rather than having to change the actual source code.

4.3 Agent Objects

The agent objects are the personnel in the model. There are four different types of agent: Peacekeeper, Non-Governmental Organisation (NGO), Insurgent and Civilian. As the name suggests, the Peacekeepers are the outside military personnel who are there to keep the peace and protect the Civilians and NGOs. The NGOs represent the mainly outside personnel who are there to help the local population, for example this could be agencies such as the Red Cross. The Insurgents are the members of the local population who maybe disagree with the military presence and who have combat capability, both with guns and bombs. Any combat that occurs in the model will have been started by an Insurgent agent. Finally, the Civilians are the non-violent local population.

4.3.1 Parameters

Each of the four different types of agent object has a set of parameters. These differ slightly but the general structure is the same. Despite the four different types of agent having parameters in common, we decided that we would define four completely separate object types. An overview of the agent object is given here, a full list of the parameters is given for reference in Appendix C.

First we have the basic properties. There is the agent type which has an integer value between one and four: one represents Peacekeeper, two is NGO, three is Insurgent and finally four is Civilian. We then have the number of the squad the agent belongs to. General parameters for the initial positions of the squad are given too, these are the x and y coordinates of the home location, and a radius which determines how spread out the agents are. These values are used in the initial position function that is given later on in Section 4.6.1. The current location of the agent is given by the coordinates $xPos$ and $yPos$, the previous location for the agent is also noted with the coordinates $xPrev$ and $yPrev$. Finally we have the *alive* indicator that is changed if an agent is killed,

this could either be by gunfire or a bomb.

Next there are the ranges and constraints. The ranges define a distance away from the agent in terms of the number of cells. This concept is illustrated in Figure 4.1: the example shows cells that are one cell away, two cells away and those that are three cells away from an agent. All the agents have a sensor range which defines the square of cells an agent can detect. For example, if an agent has a sensor range of one it can only see the eight cells that surround its current location. In addition the Peacekeeper and Insurgent agents have ranges relating to gunfire. There is the single shot kill probability, SSKP, and the firing range which is the maximum number of cells away a target agent can be.

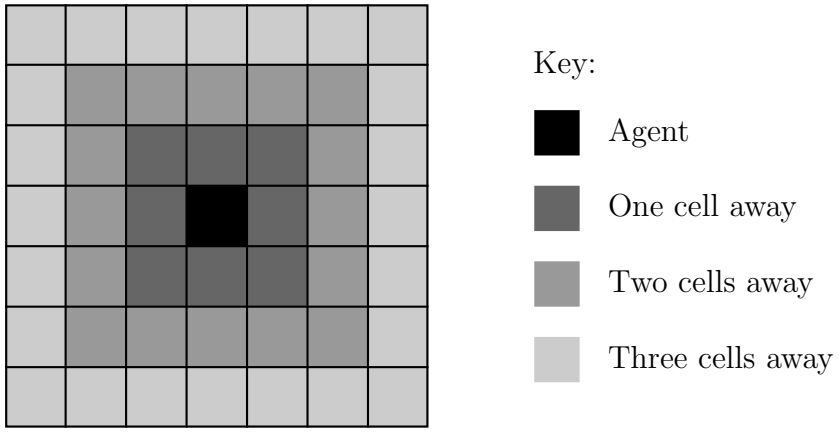

Figure 4.1: Example Cell Distances for Ranges

So far we seem to have many of the parameter types from the combat models MANA and ISAAC, but the next set of parameters differ from these models. The Insurgent agents have a probability of unprovoked fire and a probability they will set off a suicide bomb. There is also a radius of damage for the bombs, this specifies how much damage a bomb will do if set off. For example, if there is a bomb radius of two, all occupants of cells at most two cells away from the bomb

site will be killed. The Peacekeepers and NGOs also have probabilities related to their ability to fix the water and electricity supplies, these are used in the repair functions detailed in Section 4.6.4.

Next there are the relationships, these affect the movement and combat in the model. This idea is used in the DIAMOND model, as detailed in [1], and as such it was appropriate to include it here since we wish to make the two models compatible. Each squad has a relationship to each of the other squads, these remain constant throughout the model run although it is hoped that in the future the model can be developed such that they become dynamic. These relationships can be either friendly, cooperative, neutral, uncooperative or hostile; they have values one to five respectively.

Finally we have the personality weights, these are used to determine movement as will be shown later in Section 4.6.2. This is an idea that is used in the MANA and ISAAC models as shown in [2] and [3]. The ISAAC and MANA weights would not have been sufficient for our use, since we are not producing a purely combat model, therefore we had to modify the method. These weights are listed in Table 4.1.

Weight	Factor	Weight	Factor
W_1	Friendly Peacekeepers	W_{14}	Uncooperative Insurgents
W_2	Cooperative Peacekeepers	W_{15}	Hostile Insurgents
W_3	Neutral Peacekeepers	W_{16}	Friendly Civilians
W_4	Uncooperative Peacekeepers	W_{17}	Cooperative Civilians
W_5	Hostile Peacekeepers	W_{18}	Neutral Civilians
W_6	Friendly NGOs	W_{19}	Uncooperative Civilians
W_7	Cooperative NGOs	W_{20}	Hostile Civilians
W_8	Neutral NGOs	W_{21}	Tension
W_9	Uncooperative NGOs	W_{22}	Civilians in need
W_{10}	Hostile NGOs	W_{23}	No water
W_{11}	Friendly Insurgents	W_{24}	No electricity
W_{12}	Cooperative Insurgents	W_{25}	Combat
W_{13}	Neutral Insurgents		

Table 4.1: Personality Weights

Each weight has an integer value between -100 and 100, the higher the value, the more the agent wants to move towards that factor. The weights to the different types of agent, W_1 to W_{20}, are defined for all four types of agent. We have a set of weights covering all relationships to each agent type to allow for changing relations; clearly this is not needed at present since the relationships remain constant but it was best to leave it in to allow for further development. The weights towards cells without electricity or water and the weight towards civilians in need are only given for Peacekeepers and NGOs. The tension weight, W_{21}, is only given for the Civilians and Insurgents. The combat weight, W_{25}, is only defined for Insurgents and Peacekeepers.

4.4 Cell Objects

Here we give an overview of the cell objects ; a full list of the parameters is given in Appendix C.

The model is run on a square grid comprised of cells, each of which holds a variety of data. There are three constant values that each cell has: $x-$ and $y-$coordinates and the sector the cell is located in. The coordinates are sequenced such that x increases from left to right, and y increases from top to bottom, this is illustrated below in Figure 4.2.

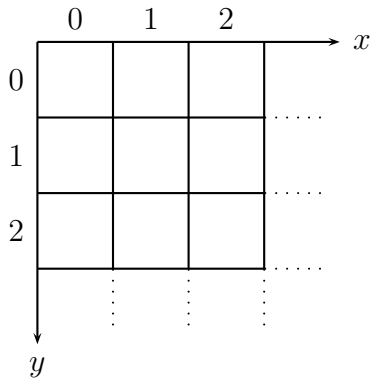

Figure 4.2: Coordinates

Next we have the occupant information. This comprises variables that give the squad the occupying agent is from, and the squad the previous occupant was from. Then there is the type of agent that is occupying the cell, and the type of agent that previously occupied the cell. Next there is the action type variable that gives the action the occupying agent will perform at the current sub-timestep; a value of zero indicates no action, one is combat, two is bomb, three is repairs to water or electricity supply and four is movement.

There are also general indicator variables for the water and electricity supply at the cell. If the water or electricity has failed in the relevant sector then there

are indicators that show this failure. If the supply does fail in a sector then a cell is chosen to represent the source of the failure; this cell is where repairs are needed and is flagged using the $fixWater$ or $fixElec$ indicator.

Next there are the combat parameters. First of these is the general combat parameter *combat*; this is an indicator function that shows whether there have been any shots fired to or from the cell in the past $SHOTMEMORY$ timesteps, or any bombs affecting the cell in the past $BOMBMEMORY$ timesteps. The bomb indicator *bombBlast* shows whether a bomb has affected the cell in the previous $BOMBMEMORY$ timesteps. The shot indicator *shotInd* is flagged if there have been any shots fired to the cell in the previous $SHOTMEMORY$ timesteps. We also have a counter for the number of shots fired to the cell.

Finally there are the psychological factors related to the cells. First of these is an indicator function, this indicates whether there are civilians in need at the cell. This is flagged if the occupying agent is a civilian and at least one of the following is true: the combat indicator is flagged, the water or electricity supply has failed or there is no food. We had hoped to add a second factor relating to tension at the cell, continuing on from the DIAMOND work detailed in Chapter 3, but we decided against this in the first instance to keep the model as simple as possible. This could be another possible future development. Of course this also means that the tension weight W_{21} becomes void.

4.5 Model Structure

Here we describe how we put the whole model together. To begin with all the required agents are constructed with default values that have been specified by the user. A square grid of cell objects is also constructed with default initial values. This grid is split into a specified number of equally sized square sectors, these are used to indicate the regions covered by a certain supply of water and electricity. When either supply fails in a sector it fails at all the individual cells in that region. The initial positions for the agents are then calculated using a random number generator in conjunction with the relevant squad home position and radius specified in the agent object.

We then move to the main part of the model, that is the timesteps. The number of timesteps has been specified in the main program. We then split these timesteps up into sub-steps according to the capabilities of the squads. The maximum of all the squad capabilities is the total number of sub-steps per timestep. The squad capabilities are then used to determine at which of these sub-steps an agent can perform an action. For example, say that we have four squads: a Peacekeeper squad with capability eight, an NGO squad with capability five, an Insurgent squad with capability four and a Civilian squad with capability three. Then each timestep would be split into eight sub-steps. The Peacekeepers would perform actions at each sub-step, the NGOs at the first five sub-steps, the Insurgent agents at every other sub-step and the Civilians also at every other sub-step, but only until they had performed three actions. This example is illustrated in Figure 4.3. If all the squad capabilities are equal there is no need for the sub-steps.

At each timestep we go through the sub-steps in turn. At each sub-step we go through the grid cell by cell and determine what action should be taken by the occupying agent, if indeed there is one. If there is no agent at a cell the action is clearly set to zero. If there is an agent at the cell we first determine whether or

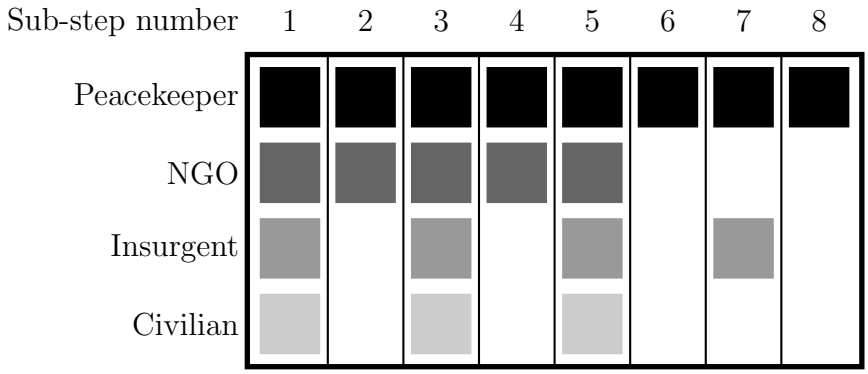

Figure 4.3: A Diagram to Indicate the Sub-Steps the Agents are in Action During One Complete Timestep

not they should be performing an action according to their capability. If not, the action is set to zero. If so, we determine what that action should be according to a priority list, this varies with agent type.

Peacekeeper: The Peacekeepers' first priority should be to protect themselves and so if they are under attack they should look to defend themselves. First we check to see if there have been any shots fired to the cell in the last timestep and if so we set the action to 'combat'. Next, if there have been any bomb blasts within sensor range in the last timestep we again set the action to 'combat'. Their next priority is to help the Civilian population. If repairs to the water or electricity supply are needed at the cell then set the action to 'repair', if not then set the action to 'move' so the Peacekeeper can head to where he may be needed. This process is illustrated in the flow diagram in Figure 4.4.

NGO: In the first instance the NGOs would want to move away from danger since they are unarmed, so if there have been any shots fired to the cell in the last timestep the action is set to 'move'. Next, if repairs to the water or electricity supply are needed at the cell then the action is set to 'repair'. Otherwise the agent is set to 'move' so he can best help the Civilians. This decision process is

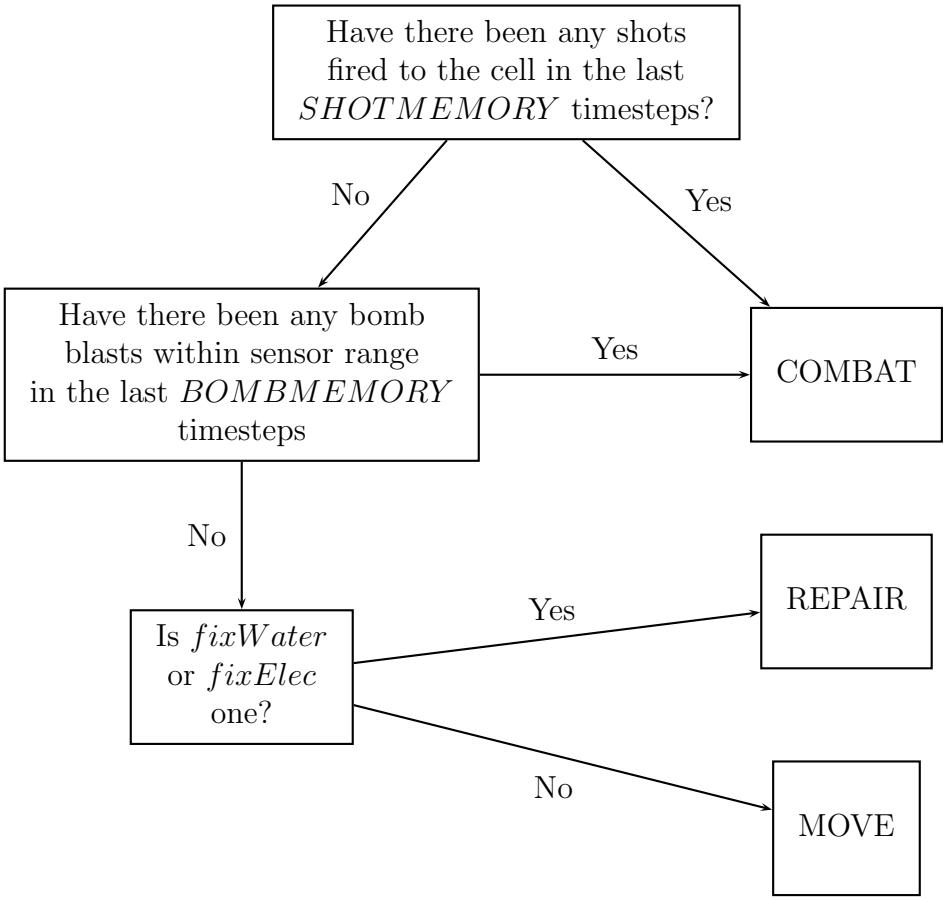

Figure 4.4: Peacekeeper Action Decision Process

shown in Figure 4.5.

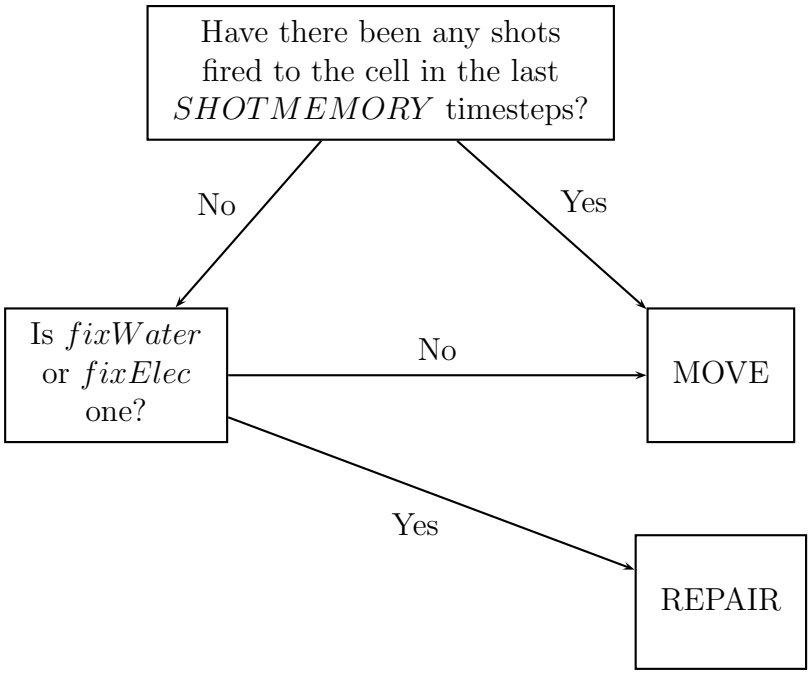

Figure 4.5: NGO Action Decision Process

Insurgent: First the Insurgents would want to defend themselves against any enemy fire so if there have been any shots fired to the cell in the last timestep then set the action to 'combat'. If they are not in immediate danger they would decide whether or not to start conflict according to their bomb and shot probabilities. Generate a random number to determine whether or not the agent will set off a suicide bomb, if so set action to 'bomb.' Generate a random number to determine whether the agent fires without provocation, if so set action to 'combat'. If the Insurgent is not involved in combat his action will be set to 'move'. Figure 4.6 shows this process.

Civilian: Since the Civilians do not carry out repairs and are unarmed, their action type is always set to 'move'.

Once the actions of all the agents have been determined we go through the

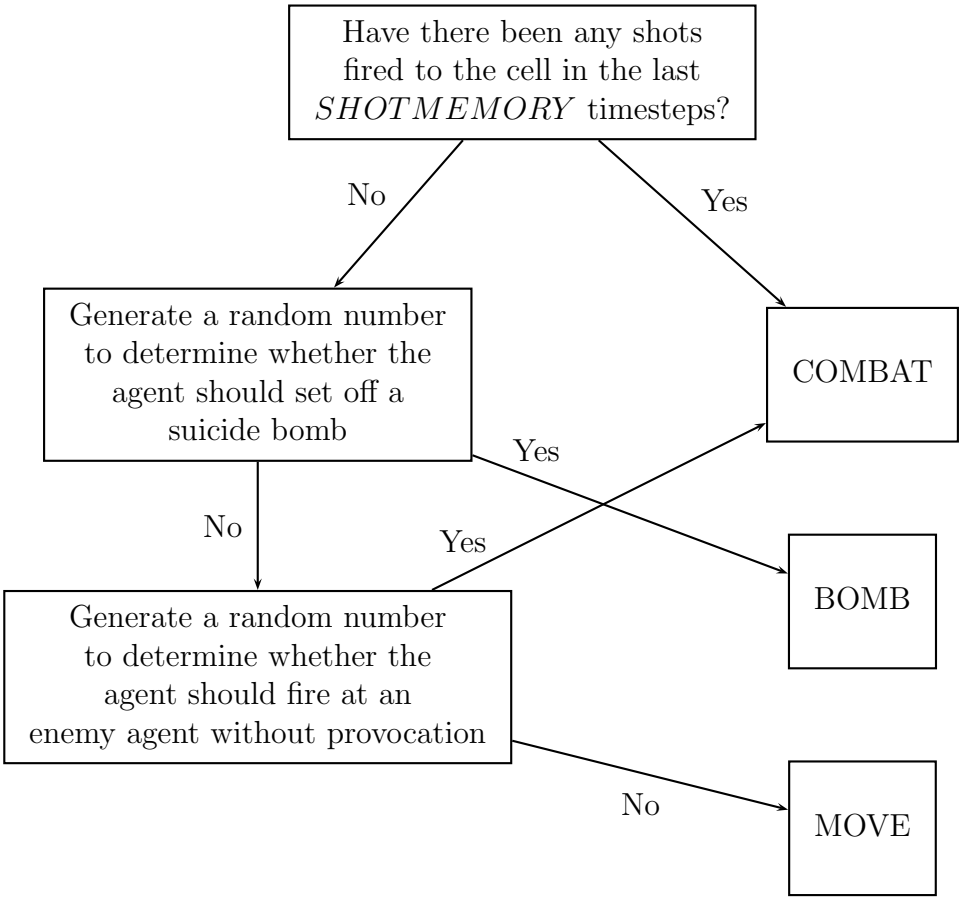

Figure 4.6: Insurgent Action Decision Process

grid again to initiate all the combat and bombings that will take place at this sub-step. When we go through the grid, if we find an agent who has an action set to 'combat' or 'bomb' we call the appropriate combat function. This part of the model is shown in the flow diagram in Figure 4.7. The individual combat functions are explained in detail later in Section 4.6.3.

After all the combat has taken place, we go through the grid for a third time in order to determine all the repairs and movements that occur at this timestep. If an agent has been flagged to repair or move, the appropriate function will be called. This part of the model is shown in Figure 4.8. The movement and repair functions are described later in Sections 4.6.4 and 4.6.2 respectively.

At the end of each sub-step some of the cell parameters are updated. The $shotInd$, $bombBlast$ and $combat$ indicators are revised. To do this we check whether any shots have been fired to the cell in the last $SHOTMEMORY$ timesteps, this is used for the $shotInd$ and $combat$ indicators. We also see if any shots have been fired from the cell in the previous $SHOTMEMORY$ timesteps, this is relevant to the $combat$ indicator only. In addition we look for any bombs affecting the cell in the last $BOMBMEMORY$ timesteps, this is used for the $bombBlast$ and $combat$ indicators. Next the $civInNeed$ indicator is updated at all the cells occupied by a Civilian; here we are looking for combat, no water, no electricity or no food.

We also have to reset all the 'previous' parameters for both the cells and the agents. These include the parameters for the previous occupying squad and agent type at the cell and previous coordinates for the agents.

At the end of each full timestep we determine whether the water or electricity supply in each sector will fail. To do this we use a random number generator and the probabilities $WATERFAIL$ and $ELECFAIL$.

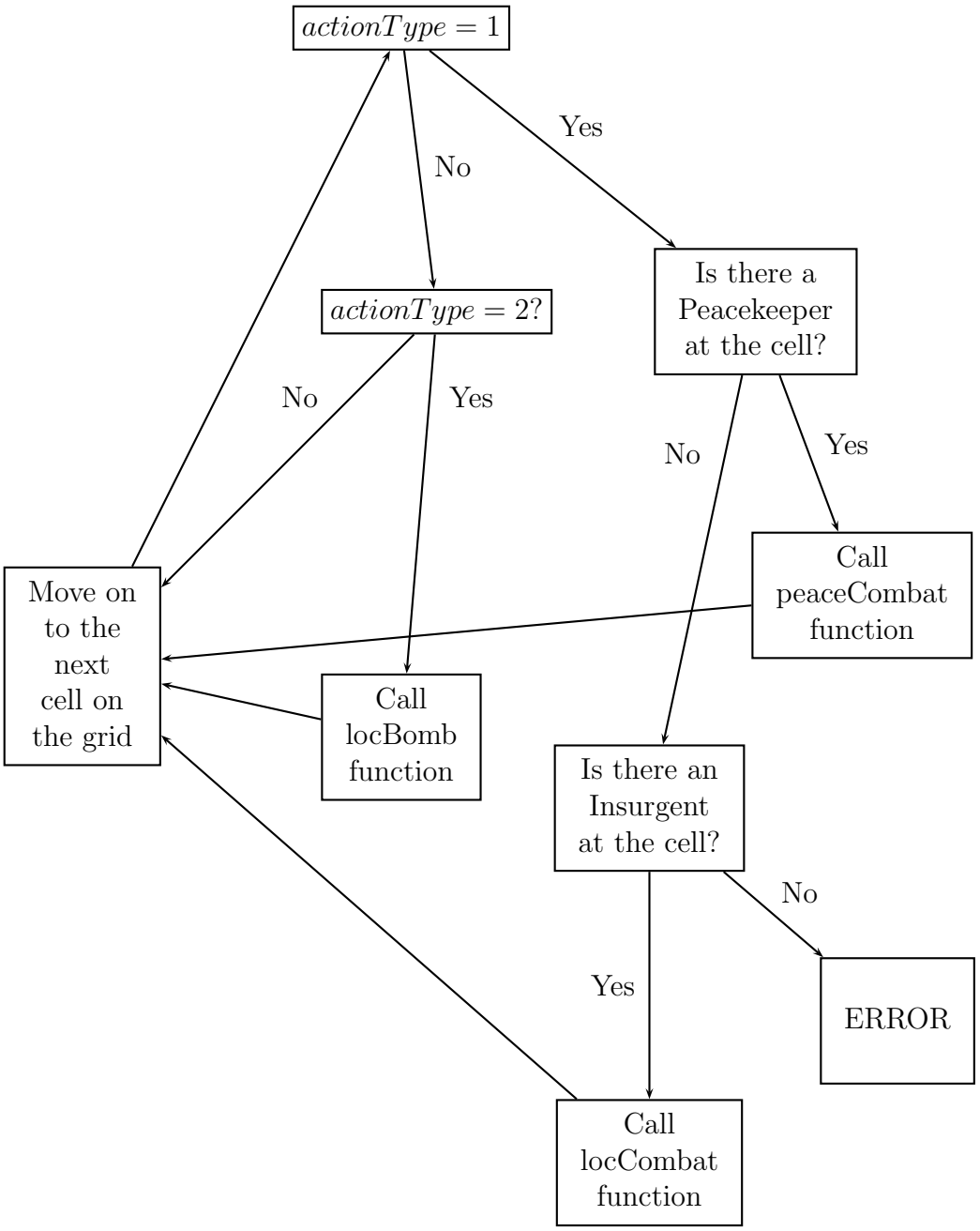

Figure 4.7: Calls to the Combat Functions

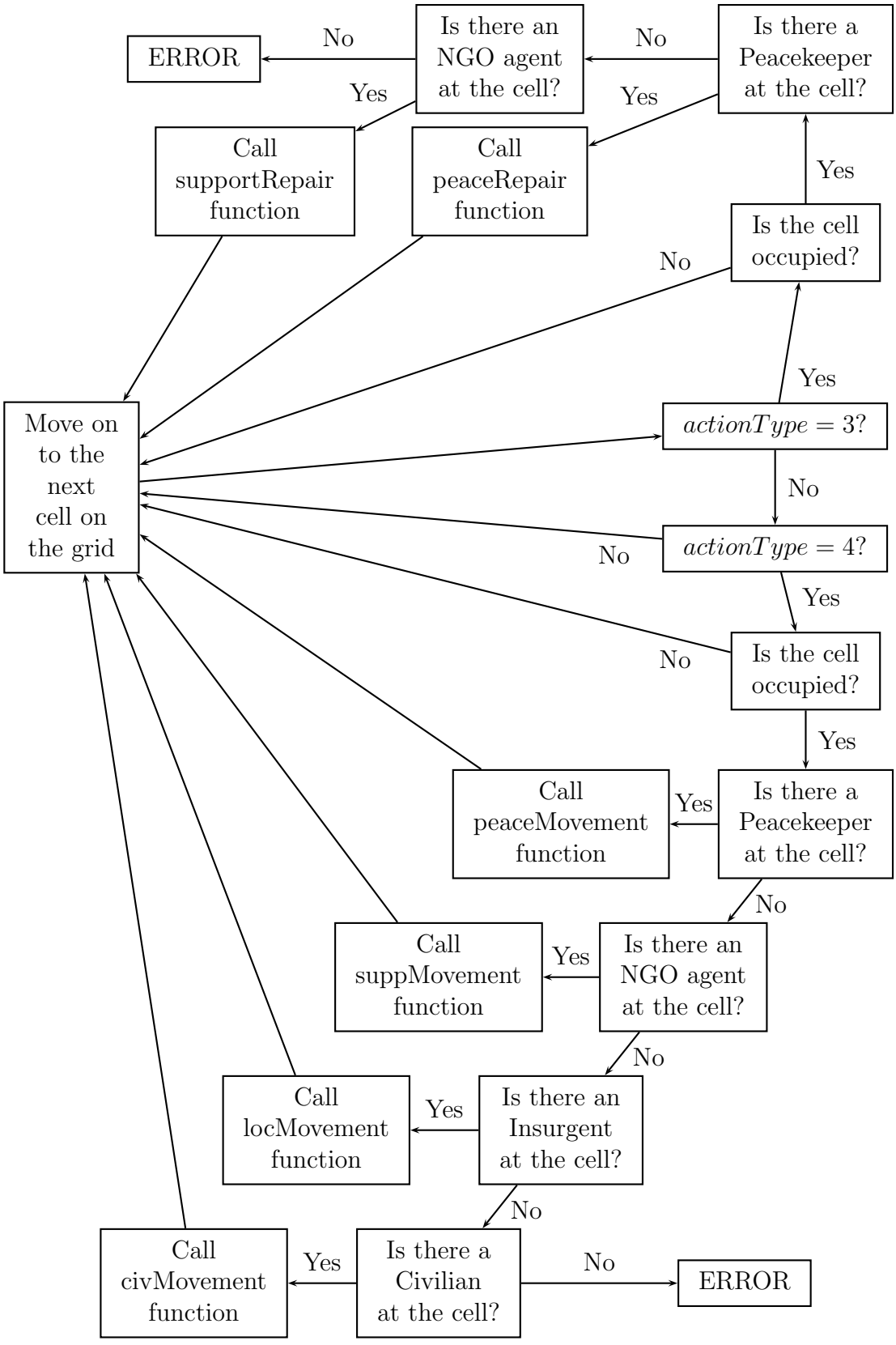

Figure 4.8: Calls to Repair and Movement Functions

4.6 Functions

4.6.1 Initial Positions

The initial positions for the agents are determined using four functions, one for each type of agent. They use the squad parameters for the x and y coordinates of the home location and the radius which determines a square of cells throughout which the agents are distributed. A random number generator is used to determine each agent's position in this square. As an example, suppose a squad has a home location (x_{home}, y_{home}) and a radius of two. Then the initial positions of the agents will all satisfy the conditions $(x_{home} - 2) \leq x \leq (x_{home} + 2)$ and $(y_{home} - 2) \leq y \leq (y_{home} + 2)$. This is shown in Figure 4.9 where the shaded cells are the possible locations for the agents, of course this also includes the centre home location itself.

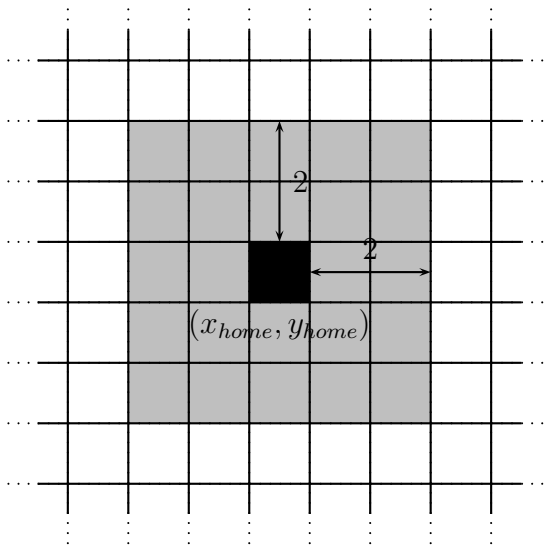

Figure 4.9: An Example Initial Squad Distribution Area

There may be a situation where more than one agent gets allocated the same starting location. To get around this we check that no agent is already at the

relevant cell, and if it is already occupied then another set of coordinates is generated. This means that priority is given to the first agents to be placed on the grid. Since the functions are called squad by squad the order of these squads should be considered before the model is run and changed accordingly.

4.6.2 Movement

We have used a method taken from the MANA and ISAAC combat models for the movement. This involves the use of an incentive function. This function is used on all the cells the agent can move to, including its current one, and then the cell with the highest incentive is the one the agent moves to. The cells considered are the eight surrounding cells and the current location. The incentive function we initially used in the model is given below in Equation (4.1).

$$I_{new} = \sum_{a=1}^{a=20} W_a * \left(\sum_{b=1}^{N_a} \frac{D_{b,old} - D_{b,new}}{D_{b,old}} \right) + W_{21} * \frac{tension_{new} - tension_{old}}{max_{tension}} + W_{22} * civInNeed_{new} + W_{23} * fixWater_{new} + W_{24} * fixElec_{new} + W_{25} * combat_{new} \quad (4.1)$$

Here N_a is the number of agents of the type indicated by the weight W_a within sensor range. The values $D_{b,new}$ and $D_{b,old}$ represent the distances to the specified agent from the proposed and current locations respectively. These distances are given as the distance between the centres of the relevant cells, where one cell is defined to have a side length of one. For example, if we look at the situation in Figure 4.10 the current distance to the other agent shown on the grid is

$$D_{old} = \sqrt{4^2 + 4^2} = 4\sqrt{2},$$

and the distance from the proposed cell is

$$D_{new} = \sqrt{5^2 + 3^2} = \sqrt{34}.$$

The factors $tension_{old}$ and $tension_{new}$ give the tension values at the current and

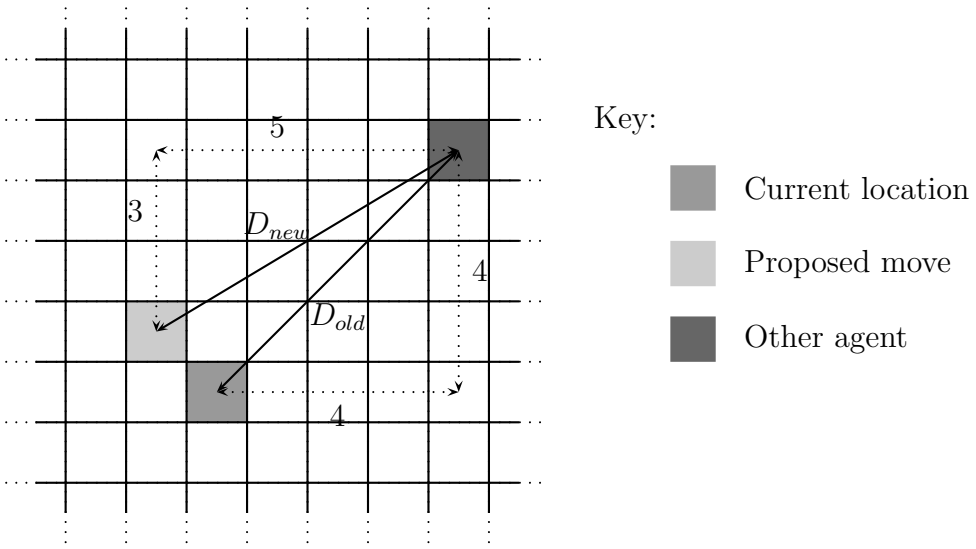

Figure 4.10: Example Distance Calculations

proposed locations respectively and $max_{tension}$ is the maximum value the tension variable can take. The remaining factors, $civInNeed_{new}$, $fixWater_{new}$, $fixElec_{new}$ and $combat_{new}$, give the values for the civilians in need, water failure, electricity failure and combat indicator functions at the proposed cell respectively.

There are some exceptions to using the incentive function. If there is already an agent occupying the relevant cell then the agent cannot move to it, hence a value of -50000 is given for that cell. Civilian agents should not move to cells where there is combat, no water, no food or no electricity. If any of these are the case then the values $-49000, -48000, -47000$ and -46000 are given respectively. NGOs should not move to cells with combat, therefore any cell with the combat indicator flagged should be given the value -49000. Insurgent agents should not move to a cell with no water supply, any cell with the $noWater$ indicator flagged should be given the value -49000. If none of these restrictions are applicable then the incentive function is used. In the case of equal incentive one of the cells with the highest value is chosen at random.

After conducting some initial experiments with the model we found that this movement algorithm did not give the behaviour we had hoped for the Peacekeepers, we needed to incorporate a change in distance factor for cells in need of repairs. Further explanation is given in Section 4.10.1 where we describe these experiments. As a result the incentive function was changed to that in Equation (4.2). Note that the tension factor has been taken out of the function since it does not now appear in the model. In Equation (4.2), $\Delta D_{i,j}$ is the fractional change in distance between the current agent cell to the cell (i, j) and the proposed location to (i, j), so

$$\Delta D_{(i,j)} = \frac{D_{(i,j),old} - D_{(i,j),new}}{D_{(i,j),old}}.$$

The coordinates x and y refer to the current agent location, and s represents the sensor range of the agent. The factors $civInNeed_{(i,j)}$ and $combat_{(i,j)}$ are the $civInNeed$ and $combat$ values at the cell (i, j) respectively. The values $waterFailure[sector_{(i,j)}]$ and $elecFailure[sector_{(i,j)}]$ give the values for the water and electricity failure indicators in the sector that the cell (i, j) falls in.

$$\begin{aligned} I_{new} = \sum_{a=1}^{a=20} W_a * \left(\sum_{b=1}^{N_a} \frac{D_{b,old} - D_{b,new}}{D_{b,old}} \right) + \\ W_{22} * \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{(i,j)} * civInNeed_{(i,j)} + \\ W_{23} * \left(FixWater_{new} + \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{(i,j)} * waterFailure[sector_{(i,j)}] \right) \\ W_{24} * \left(FixElec_{new} + \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{(i,j)} * elecFailure[sector_{(i,j)}] \right) \\ W_{25} * \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{(i,j)} * combat_{(i,j)} \end{aligned} \quad (4.2)$$

4.6.3 Combat

There are three types of combat function: the firing functions for the Peacekeepers and Insurgents, and the bomb function for the Insurgents.

Peacekeeper firing function: First we count the number of shots fired to the Peacekeeper's cell in the last timestep. If this count is zero then a bomb must have been exploded by an Insurgent agent otherwise combat wouldn't have been initiated for the Peacekeeper. In this case valid Insurgent targets within firing range are sought; if there are none then the agent's action is changed to 'move'. If there are valid targets one is chosen at random and fired at, and the survival of the target agent is then determined by the SSKP of the Peacekeeper. If the shot count is greater than zero then a cell from which a shot was fired is chosen at random. If the occupying agent is a valid target then a shot is fired, if not the Peacekeeper's action will change to 'move'. This is not ideal; one future improvement to the model could be to choose another target rather than automatically changing the action to movement. If a shot is fired then a random number is generated to determine if the target is killed according to the SSKP of the Peacekeeper.

Insurgent firing function: The number of shots fired to the cell in the last timestep is counted. If the count is zero then the agent must have been chosen to fire at a random enemy agent within firing range so valid targets are sought and recorded. If there are no possible targets then the agent's action is changed to 'move', if there are enemy agents within firing range then a target is chosen at random and fired at. The survival of this target agent is determined by the SSKP of the Insurgent agent. If the shot count is greater than zero then a cell from which a shot was fired is chosen at random; if the occupying agent is a valid target then a shot is fired, if not the Insurgent agent's action will change to 'move'. Again this is not ideal and could be changed in future developments of the model. If a shot has been fired then a random number and the SSKP for the

Insurgent are used to determine whether the target agent is killed.

A flow diagram illustrating the firing function is given in Figure 4.11. This applies to both Peacekeepers and Insurgents.

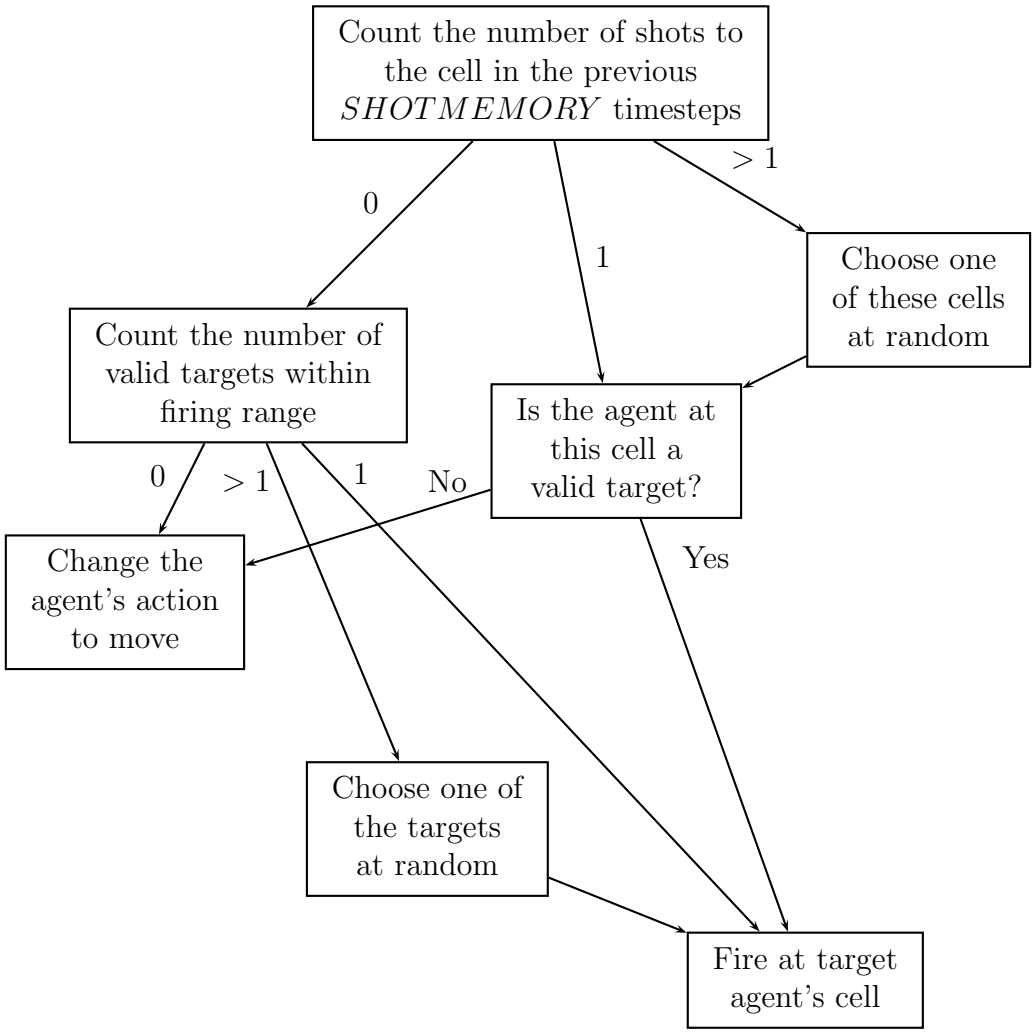

Figure 4.11: Firing Function

Insurgent bomb function: First we count the number of valid target agents within bomb range. If there are none then the agent's action is changed to 'move', otherwise a bomb is set off. The Insurgent agent is killed, as are all other agents within the bomb range. A diagram of this function is given in Figure 4.12.

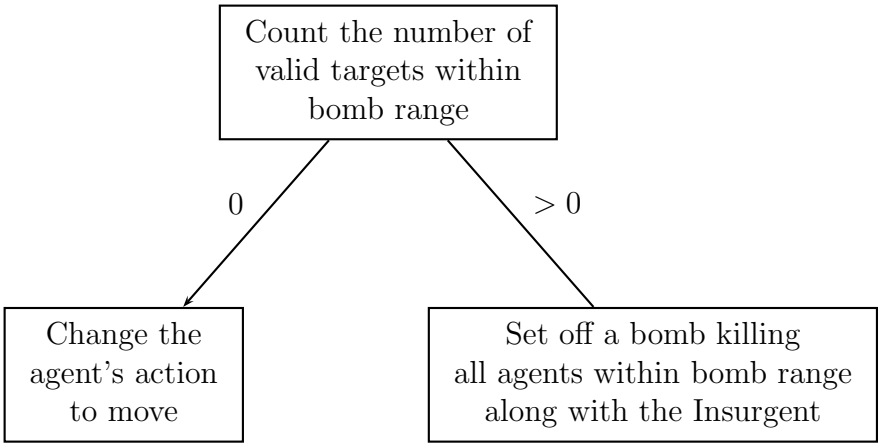

Figure 4.12: Bomb Function

4.6.4 Repairs

The repair functions are used for Peacekeepers and NGOs when they are at a cell at which the water or electricity, or both, have failed and the fault in the supply is to be repaired at that location. If the repair function is called there is no prior indicator to say whether it is the water or electricity that has failed, thus we first check to see if the $fixWater$ indicator at the cell has been flagged. We do this first because if both supplies need fixing then water takes priority over electricity. If $fixWater$ is zero then we check that $fixElec$ has been flagged to make sure the repair function has not been called in error. In both cases we then generate a random number and use the agent probability for fixing the relevant service to determine whether or not the supply is restored. A diagram of the repair function is shown in Figure 4.13.

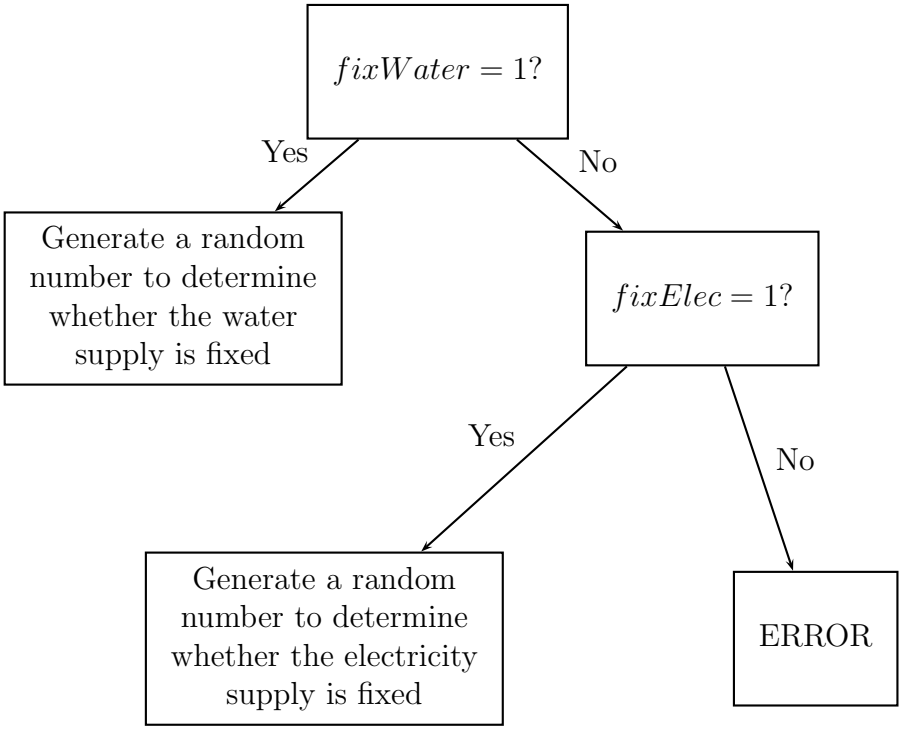

Figure 4.13: Repair Function

4.7 Data Output

As the model is running we record some of the variables for analysis. The most important output is the agent positions, the grid is written to file at the end of each complete timestep. This file is called 'gridoutput.txt' and will be used in the analysis detailed in Chapter 5. Since the model contains no graphics we also use this file in conjunction with a MATLAB program to show the model run, this is described further in Section 4.8.

Other variables recorded are the two indicators for Civilians in need and combat, in the files 'civinneed.txt' and 'combat.txt' respectively. Again the values for the entire grid are written to file at the end of every complete timestep. At the end of a timestep we also record the vector of water and electricity failures in the files 'waterfailind.txt' and 'elecfailind.txt'. We do not need to record the whole grid here, just the values for each sector.

Casualty details are recorded as they occur, this data is written to 'casualties.txt'. The agent, location and time are recorded. At the end of the model run the two arrays containing the details of every shot fired and every suicide bomb are written to the files 'shotoutput.txt' and 'bomboutput.txt'. The shot array contains the timestep, sub-time, agent location, agent type, agent squad, target location, target agent type and target agent squad for each shot. The bomb array contains the timestep, sub-time, Insurgent location, bomb radius and Insurgent squad number for each suicide bomb attack. The two files 'shotoutput.txt' and 'bomboutput.txt' have also been used in much of the analysis we have conducted that has been written up in Chapter 5. The remaining files have not been used because of the limited time available but additional analysis could be carried out on them.

4.8 Visualisation of the Model

Instead of programming a graphics element into the model we decided instead to use data output and a MATLAB script to show the agents' progression after the model had been run. We used the grid positions output, 'gridoutput.txt', and produced a series of plots with a slight delay between each, showing the whole grid at each timestep, which gives the impression of an animation of the moving agents. An example screenshot of a model run is shown in Figure 4.14. Here the Peacekeepers are shown in blue, the Insurgents are red, the NGOs are green and the Civilians are black. This visualisation only shows the movement of the agents, there is nothing to indicate shots fired or bomb blasts apart from the agents disappearing from the grid if they are killed.

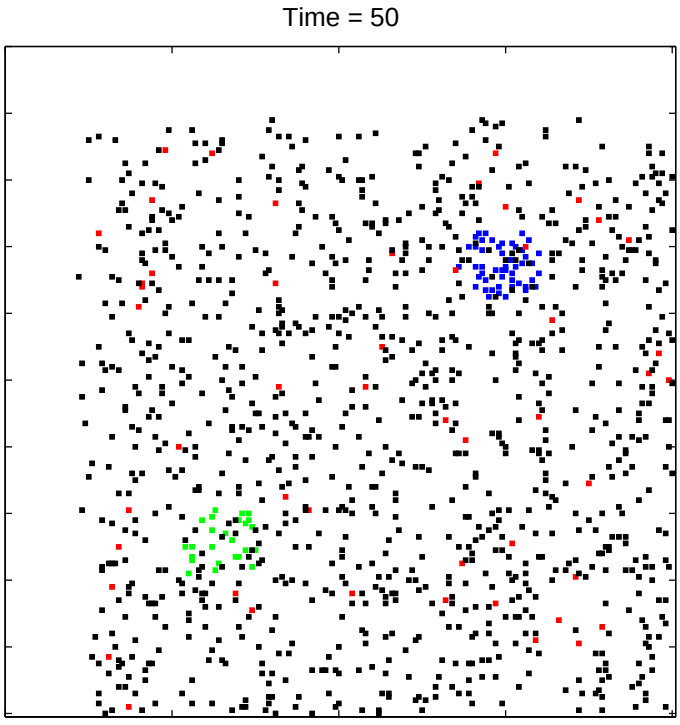

Figure 4.14: Screenshot of Model Visualisation

4.9 Avalanches

The original aim of the research was to look at the possibility that peacekeeping operations could show some form of self-organised criticality (SOC). Due to time constraints we could not investigate this fully, but we were able to look at one potential source of SOC behaviour.

We needed to find behaviour analogous to the avalanche in the sand pile model or the fire in the forest fire model. The skirmishes started by the Insurgents seem to fit the bill: they do not occur at all timesteps, they are started by a single shot or bomb and can result in anything from no response to an exchange of fire involving the whole Peacekeeper squad. The question then became, what aspect of these skirmishes do we measure? There are several possibilities, including casualty numbers or number of cells, but we decided to look at the number of shots and bombs and then a second measure of the length of the conflict in timesteps. These measures were chosen mainly because they were the easiest to obtain from the data and time was an issue.

Later in Chapter 5 we describe the experiments that were carried out. We ran each variation of each scenario 50 times so we pooled the data from the 50 runs together. We were able to do this since the skirmishes occur at random throughout the model runs, they are not linked to a particular stage of the run so they can be seen as independent from each other and from the particular model run. Once we have determined the 'avalanche' sizes we can plot the frequency-size distribution on a log-log scale. If this graph approximates a straight line we have evidence of a power law which may in turn indicate self-organised criticality.

For reference the MATLAB programs used for the avalanche analysis are given in Appendix E.2.

4.10 Verification of the Model

The object of model verification is to make sure that it works as it is supposed to. Here we concentrate on the movement of the agents. We need to check that not only is the movement algorithm working as it should, but also that it produces expected patterns of behaviour under certain situations.

First we take a very simple scenario to check that the equation is working properly. Set up a 5×5 grid with one agent of each type: the Peacekeeper at $(1, 1)$, the NGO agent at $(3, 1)$, the Insurgent at $(1, 3)$ and the Civilian at $(3, 3)$. This is shown in the left-hand grid in Figure 4.15. Each agent has a weight of -10 towards the other three types of agent and a sensor range of four. There is no water or electricity failure or combat so the movement will only be dictated by the other agents and the incentive function is simply,

$$I_{new} = \sum_{a=1}^{a=20} W_a * \left(\sum_{b=1}^{N_a} \frac{D_{b,old} - D_{b,new}}{D_{b,old}} \right).$$

Taking the Peacekeeper as an example we shall work through the nine possible moves to find the incentive value for each one. The three current distances to the agents are

$$D_{NGO,old} = D_{Ins,old} = 2$$

and

$$D_{Civ,old} = \sqrt{2^2 + 2^2} = 2\sqrt{2}.$$

The incentive value for the current cell is going to be zero since the old and new distances to the agents will be the same and there are no other factors involved. The incentive for the cell $(0, 0)$ is calculated as follows:

$$I_{(0,0)} = -10 * \left(\frac{2 - \sqrt{3^2 + 1^2}}{2} \right) - 10 * \left(\frac{2 - \sqrt{3^2 + 1^2}}{2} \right) - 10 * \left(\frac{2\sqrt{2} - \sqrt{3^2 + 3^2}}{2\sqrt{2}} \right)$$

$$\simeq 16.62$$

The incentives for all the possible moves are,

$$I_{(0,0)} \simeq 16.62 \qquad I_{(1,0)} \simeq 8.93 \qquad I_{(2,0)} \simeq 4.06$$
$$I_{(0,1)} \simeq 8.93 \qquad I_{(1,1)} = 0 \qquad I_{(2,1)} \simeq -5.91$$
$$I_{(0,2)} \simeq 4.06 \qquad I_{(1,2)} \simeq -5.91 \qquad I_{(2,2)} \simeq -10.86$$

From these results we see that the best move for the Peacekeeper will be to the cell $(0, 0)$. Similar calculations give the best move for the NGO agent to be to $(4, 0)$, the Insurgent to $(0, 4)$ and the Civilian to $(4, 4)$. This is the grid shown on the right in Figure 4.15 and is indeed the result we got when we ran this model.

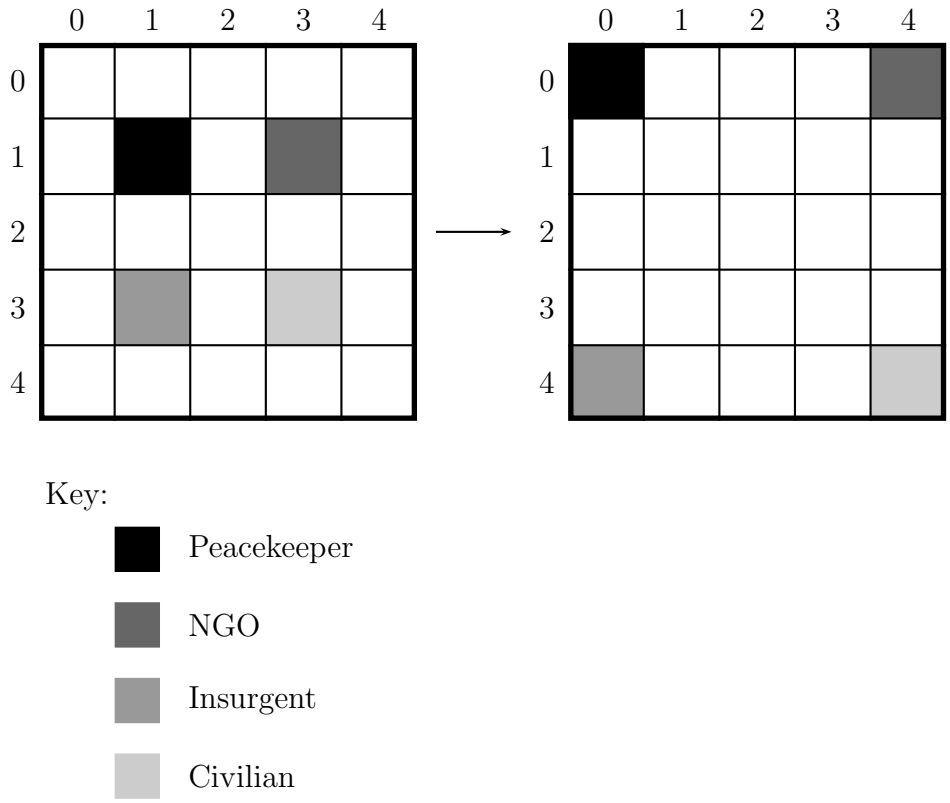

Figure 4.15: Basic Movement Test

Next we use a slightly more complicated scenario to test patterns of behaviour.

We use a 15×15 grid and four squads which each consist of ten agents. We set up the scenario so that the agents are uniformly distributed over the whole grid to start with, so we would expect them to be mixed together rather than in squad clusters. The weights are set such that the agents are attracted towards their own type with a weight of ten but all other weights are set to zero. All the agents have a sensor range of ten. Intuitively we would expect the agents to cluster with other agents of their type, we ran this scenario for 20 timesteps to give them a chance to do this. After running the model ten times we found this to be the case in each model run. An example of the start and finish grids is shown in Figure 4.16. the results from the other nine runs are given in Appendix D.1 for completeness.

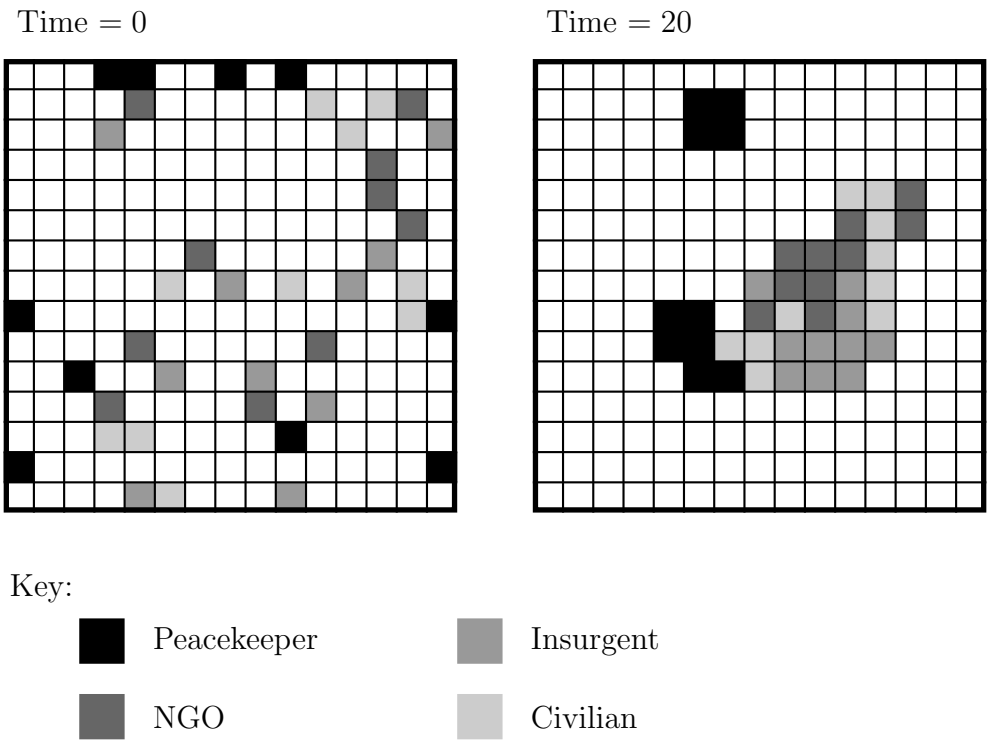

Figure 4.16: Model Verification A1

We then adapted this scenario so that in addition to the positive weight of

ten towards their own type, the agents have a negative weight of −10 towards all the other agent types. Here we would again expect the agents to cluster together with their other squad member, but they would also be moving away from the other types of agents. Again we ran the scenario ten times, each for 20 timesteps. The example in Figure 4.17 confirms this behaviour. The results for the other nine runs can be found in Appendix D.2.

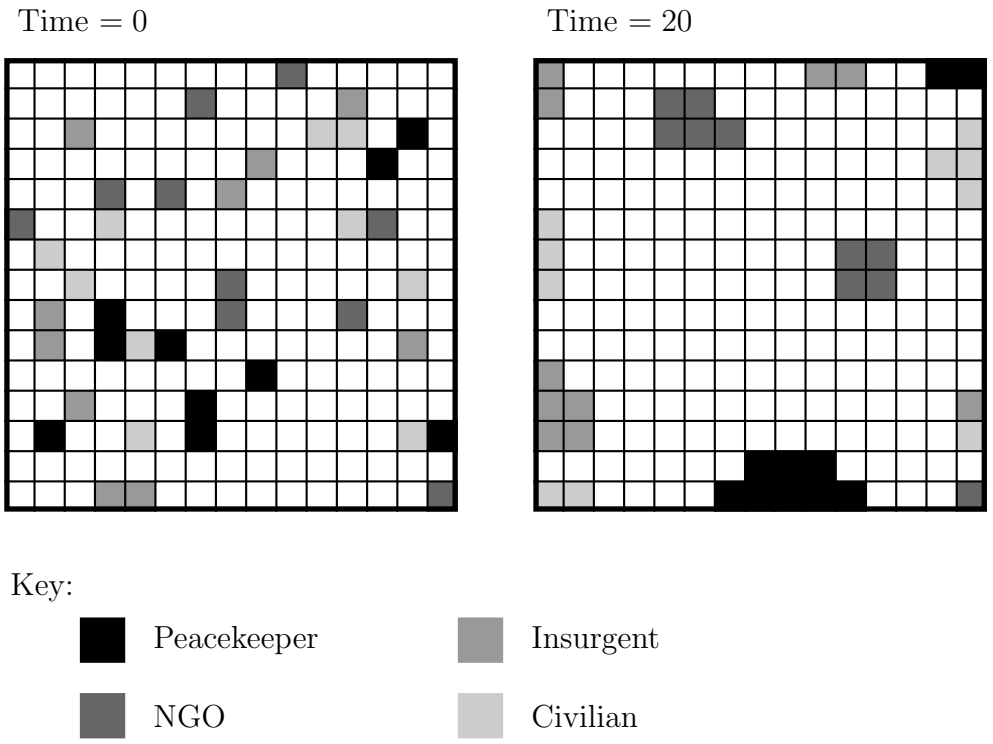

Figure 4.17: Model Verification AR1

4.10.1 Scenario One

When we planned the scenarios we would be running in the model, we started with one where the Insurgents are spread amongst the Civilian population and the Peacekeepers are together in a group in a corner of the 150×150 grid. The grid is split into nine 30×30 sectors. The probabilities for the water and electricity

failing at a sector are both set to 0.01 and the probability an Insurgent will set off a suicide bomb is also 0.01. The initial configuration is shown in Figure 4.18.

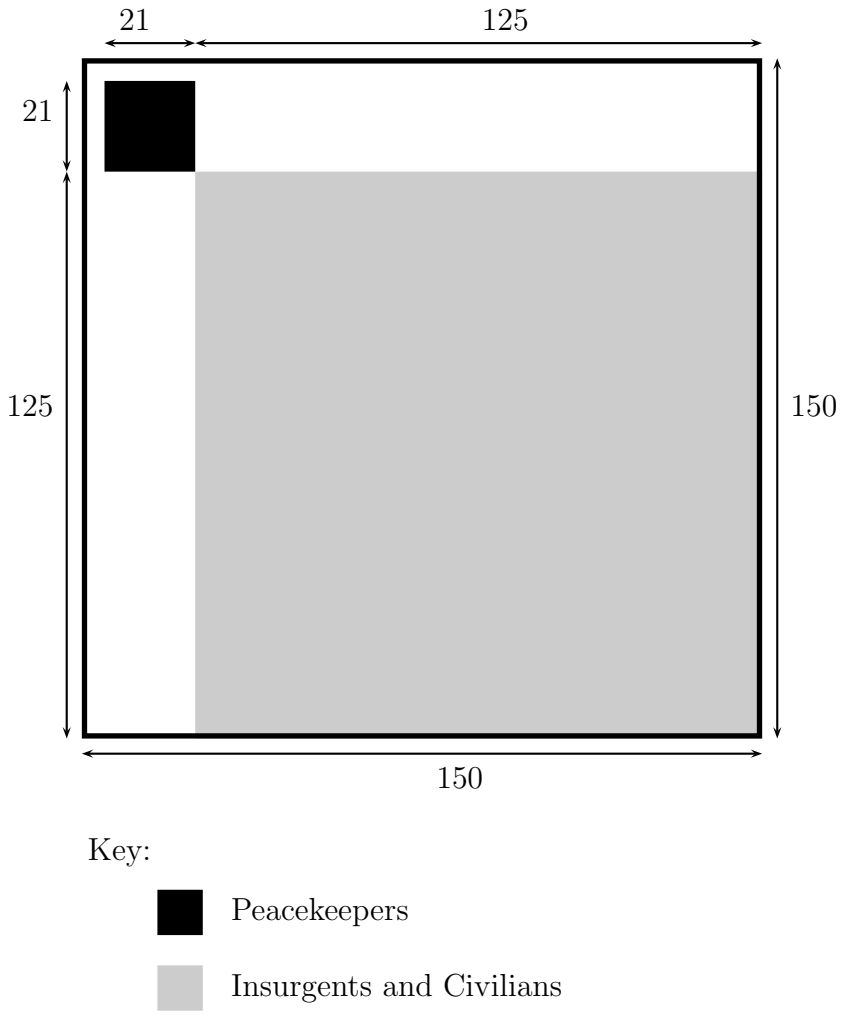

Figure 4.18: Initial Grid for Scenario One

The aim of the scenario was to see the effect of varying the size of the Peacekeeper squad; the Insurgent and Civilian squads were kept constant at 100 and 1000 agents respectively. There was no NGO squad. The Peacekeepers had a positive weight of ten towards cells where the water or electricity needed fixing and to cells where the civilian in need indicator was flagged.

The model did not provide any meaningful results but did show up some unforeseen problems with the movement algorithm. At this stage we were using the first incentive function shown as Equation (4.1) in Section 4.6.2. We found that the Peacekeepers were not moving towards the sectors where they were needed, they were just moving away from the Insurgents. Despite the fact that the Peacekeepers were able to 'see' cells where the water or electricity had failed within sensor range, this did not affect their movement unless it was at one of the neighbouring cells. This was because if we look at the incentive function we have these factors,

$$\cdots + W_{22} * civInNeed_{new} + W_{23} * fixWater_{new} + W_{24} * fixElec_{new} + \ldots$$

Unlike the part of the equation dealing with the agents we do not take into account any change in distance, just the situation at the proposed move location. Clearly we need to add in a similar change in distance factor so that the water and electricity failures they could detect but weren't in their immediate vicinity would also influence their movement. The factors

$$\cdots + W_{23} * fixWater_{new} + W_{24} * fixElec_{new} + \ldots$$

would also be kept in since they indicate the exact cell the repairs are needed at. Thus we end up with the final incentive function

$$\begin{aligned} I_{new} = \sum_{a=1}^{a=20} W_a * \left(\sum_{b=1}^{N_a} \frac{D_{b,old} - D_{b,new}}{D_{b,old}} \right) + \\ W_{22} * \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{i,j} * civInNeed_{i,j} + \\ W_{23} * \left(FixWater_{new} + \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{i,j} * waterFailure[sector_{i,j}] \right) \\ W_{24} * \left(FixElec_{new} + \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{i,j} * elecFailure[sector_{i,j}] \right) \\ W_{25} * \sum_{i=x-s}^{x+s} \sum_{j=y-s}^{y+s} \Delta D_{i,j} * combat_{i,j}. \end{aligned}$$

4.11 Validation of the Model

Military models are generally validated using a combination of peer review by experts and comparison of the results to historical data. In our timescale we were only able to conduct an initial peer review; obtaining data related to peacekeeping operations would have been very difficult.

A peer review was conducted by a team of experts from Dstl, [4]. Overall they seemed satisfied with the work but suggested a range of improvements. I have managed to incorporate some of these although further improvement is still necessary. At the time we had a working model but memory issues were a problem so we could only have very small grids and this was commented on. We have since managed to improve the model so that it can handle a grid of size 200×200. The suggested grid size to adequately model a DIAMOND node was 1000×1000 but major changes to the model code would need to be made to achieve this. It was also said that multiple squads of each agent type would be necessary to model many peacekeeping scenarios. Although this feature could not be included in the limited time available, it is discussed as one of the suggestions for future improvements in Chapter 7.

Bibliography

[1] A. Caldwell, R. Hayes, G. Mitchell, G. Maguire, S. Weston, C. Weir, B. Cope, R. Rycroft, P. Merret, P. Albano, D. Frankis, S. Colby & A. Brown, DIAMOND Functional Specification Phase Two Proposal, Volume II : Movement, Sensing, Communication, Local Picture, Perception, Relationships, Negotiation and Test Scenarios (2000), DERA Report.

[2] A. Ilachinski, Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial-Life Approach to Land Warfare (1997).

[3] M. K. Lauren & R. T. Stephen, Map Aware Non-Uniform Automata, Version 1.0 Users Manual (2001).

[4] Personal communication with Paul Glover, Mark Taylor and Major Carl Benfield, 1st November 2005.

Chapter 5

EXPERIMENTS AND RESULTS

Now we have a working agent-based model for peacekeeping operations we are able to look at some more complex scenarios than those in Chapter 4 used for model verification. We shall be starting with a basic scenario then adding more factors in so that it becomes more complex. This makes sense since we are looking for complex behaviour, the self-organised criticality, and if we do find it we would want to know at what point the model becomes sufficiently complex to exhibit this behaviour.

We ran four major scenarios, each of which had several variations. In this chapter we display the results from these experiments and draw some conclusions. We look at the two different types of 'avalanche' we described earlier in Section 4.9: the avalanche that measures the number of shots and bombs per skirmish, and the second time avalanche that measures the time of the conflict. We also look at the effectiveness of the Peacekeepers and NGOs at fixing failures in the water and electricity supplies along with the number of casualties suffered. Final discussion and conclusions are provided later in Chapter 6.

For reference we reproduce a table of the relevant weights and the factor they

refer to in Table 5.1.

Weight	Factor	Weight	Factor
W_1	Friendly Peacekeepers	W_{13}	Neutral Insurgents
W_2	Cooperative Peacekeepers	W_{14}	Uncooperative Insurgents
W_3	Neutral Peacekeepers	W_{15}	Hostile Insurgents
W_4	Uncooperative Peacekeepers	W_{16}	Friendly Civilians
W_5	Hostile Peacekeepers	W_{17}	Cooperative Civilians
W_6	Friendly NGOs	W_{18}	Neutral Civilians
W_7	Cooperative NGOs	W_{19}	Uncooperative Civilians
W_8	Neutral NGOs	W_{20}	Hostile Civilians
W_9	Uncooperative NGOs	W_{22}	Civilians in need
W_{10}	Hostile NGOs	W_{23}	No water
W_{11}	Friendly Insurgents	W_{24}	No electricity
W_{12}	Cooperative Insurgents	W_{25}	Combat

Table 5.1: Personality Weights

5.1 Scenario Two

Our first major scenario provides some structure for the Peacekeepers. We give them an incentive to move by programming in a fault in the water supply at the opposite end of the grid to their initial location. In order to reach this location they have to travel through an area populated by Insurgents and Civilians. This initial set up is illustrated in Figure 5.1. Note that the distances shown on this diagram refer to the number of cells, the model is run on a 200×200 grid. We have decided to leave out the NGOs; they will be introduced in later scenarios to add complexity.

The general model parameters are given in Table 5.2. Note that we have set the Civilian capability to zero so they are unable to move around the grid. This was done so that we can concentrate on the Peacekeepers' reaction to the Insurgents. In later scenarios the Civilians will be able to move, which will add complexity to the model. The size of the Peacekeeper squad will be varied throughout the different model trials. There will be no suicide bombings in this scenario, again this will be added in later scenarios to increase the complexity. The water and electricity failure probabilities are also set to zero so that the Peacekeepers have one clear goal, that being to fix the water supply in the final sector.

The agent parameters for the scenario are shown in Table 5.3. Here the only parameter that will change will be the weight to hostile Insurgents, W_{15}, for the Peacekeepers. Notice that we have set the Peacekeeper sensor range to 200, this is so that they are able to see the whole grid and can therefore react to everything that is happening. This reflects the fact that they would have intelligence relating to the general situation, such as the fault with the water supply, and would be in contact with other members of the squad. So in effect we are trying to compensate for not directly modelling communications and command and control.

We ran this scenario in seven different configurations, changing both the num-

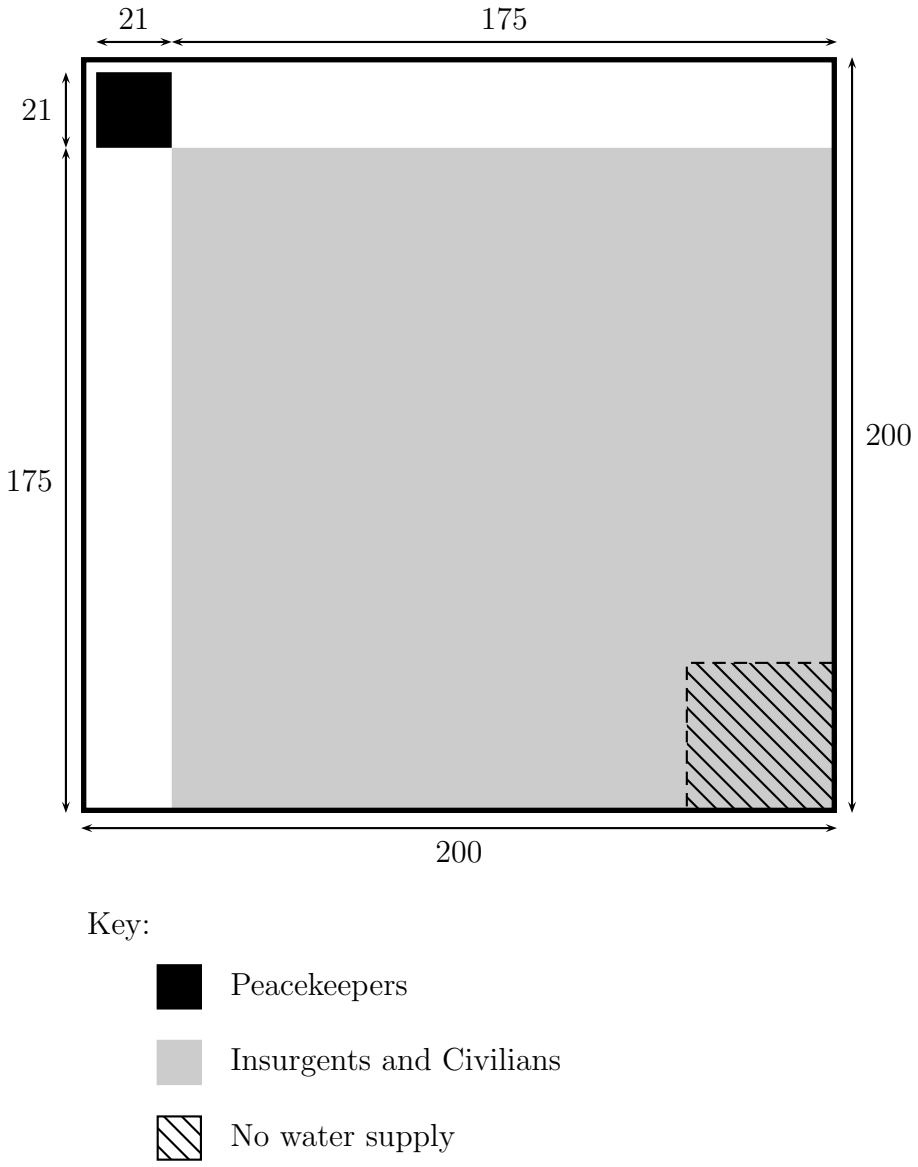

Figure 5.1: Initial Grid for Scenario Two

Constant	Value
$RUNTIME$	500
$GRIDSIZE$	200
$SECTOR$	5
$NOOFSQUADS$	3
$PEACENO$	Varies
$SUPPORTNO$	0
$LOCALNO$	100
$CIVNO$	1500
$PEACECAP$	1
$SUPPORTCAP$	0
$LOCALCAP$	1
$CIVCAP$	0
$BOMBMEMORY$	20
$SHOTMEMORY$	10
$LMBOMBPROB$	0.00
$LMFIREPROB$	0.01
$WATERFAIL$	0.00
$ELECFAIL$	0.00
$MAXSHOTS$	5000
$MAXBOMB$	100

Table 5.2: Scenario Two: General Parameters

Parameter	Peace.	NGO	Ins.	Civ.
$squadNo$	1	-	2	3
$xHome$	14	-	112	112
$yHome$	14	-	112	112
$homeRadius$	10	-	87	87
$sensorRange$	200	-	25	25
$fireRange$	20	-	15	-
$bombRadius$	-	-	-	-
$sSKP$	0.10	-	0.05	-
Relationship to Peacekeepers	F	-	H	C
Relationship to NGOs	-	-	-	-
Relationship to Insurgents	H	-	F	F
Relationship to Civilians	F	-	F	F
W_1	0	-	0	0
W_2	0	-	0	0
W_3	0	-	0	0
W_4	0	-	0	0
W_5	0	-	0	0
W_6	-	-	-	-
W_7	-	-	-	-
W_8	-	-	-	-
W_9	-	-	-	-
W_{10}	-	-	-	-
W_{11}	0	-	0	0
W_{12}	0	-	0	0
W_{13}	0	-	0	0
W_{14}	0	-	0	0
W_{15}	Varies	-	0	0
W_{16}	0	-	0	0
W_{17}	0	-	0	0
W_{18}	0	-	0	0
W_{19}	0	-	0	0
W_{20}	0	-	0	0
W_{22}	10	-	-	-
W_{23}	10	-	-	-
W_{24}	10	-	-	-
W_{25}	0	-	0	-
$probFixWater$	1.00	-	-	-
$probFixElec$	1.00	-	-	-

Table 5.3: Scenario Two: Agent Parameters

ber of Peacekeepers and the weight towards hostile Insurgents for the Peacekeepers. Each of these scenario trials was then run 50 times. The results are given in Sections 5.1.1 to 5.1.7.

5.1.1 Trial One

The additional parameters for this scenario are given in Table 5.4. For the first set of runs we have 50 Peacekeepers with a weight of -50 towards hostile Insurgents.

Parameter	Value
$PEACENO$	50
Peacekeeper W_{15}	-50

Table 5.4: Scenario Two, Trial One: Parameters

Figures 5.2 and 5.3 show the avalanche and time avalanche distributions respectively, both plotted on a log-log scale. Recall that the avalanche data measures the number of shots and bombs per conflict, the time avalanche data measures the time of each conflict. The best fit straight lines are also plotted and the equations of the lines are shown on the graphs.

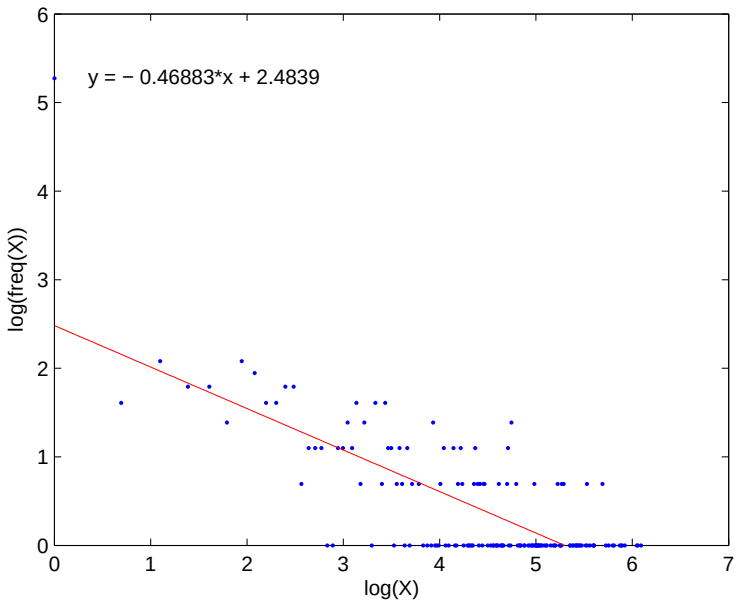

Figure 5.2: Scenario Two, Trial One: Avalanche Frequency-Size Distribution

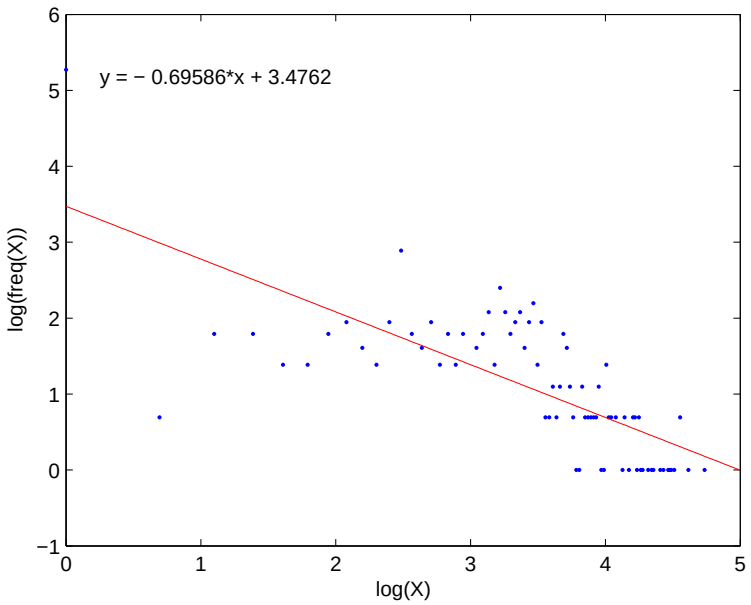

Figure 5.3: Scenario Two, Trial One: Time Avalanche Frequency-Size Distribution

We can see from the plots that the points follow a general downward trend but they are not close enough to the best fit line for us to conclude that we have evidence of a power law.

We also note that the point corresponding to $log(X) = 0$ has a much higher value than the subsequent points on both graphs. This point corresponds to an avalanche or time avalanche of size one; this is a situation where an Insurgent will have fired at an enemy, or set off a suicide bomb, without any response from the Peacekeepers. This is a feature of all the avalanche and time avalanche plots throughout this chapter.

The casualty numbers for the set of model runs are given in Table 5.5. The three values recorded are the minimum number of deaths, the maximum number and the mean number per model trial. Note that there are no Civilian casualties since there are no bomb blasts and Civilians are not valid targets for either the Insurgents or Peacekeepers so they cannot be shot at.

	Minimum	Maximum	Mean
Peacekeeper	1 (2%)	19 (38%)	8.84 (17.68%)
Insurgent	11 (11%)	48 (48%)	32.38 (32.38%)

Table 5.5: Scenario Two, Trial One: Casualty Numbers

To see how effective the Peacekeepers were at fixing the water and electricity supplies, we recorded the total number of sector failures and the number that were fixed over the model trial, then calculated the mean values per run. For this particular scenario there was only the initial water failure, so the total number of failures and mean value are 50 and one respectively by definition. The values are given in Table 5.6.

Full discussion of the results and comparison to the other scenario two variations will be given in Section 5.1.8.

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	9	0.18

Table 5.6: Scenario Two, Trial One: Utility Failure Numbers

5.1.2 Trial Two

The parameters specific to this trial are shown in Table 5.7. We have increased the negative weight towards the Insurgents for the Peacekeepers to -100 to see if this affects their ability to fix the water supply and whether or not it results in fewer casualties.

Parameter	Value
$PEACENO$	50
Peacekeeper W_{15}	-100

Table 5.7: Scenario Two, Trial Two: Parameters

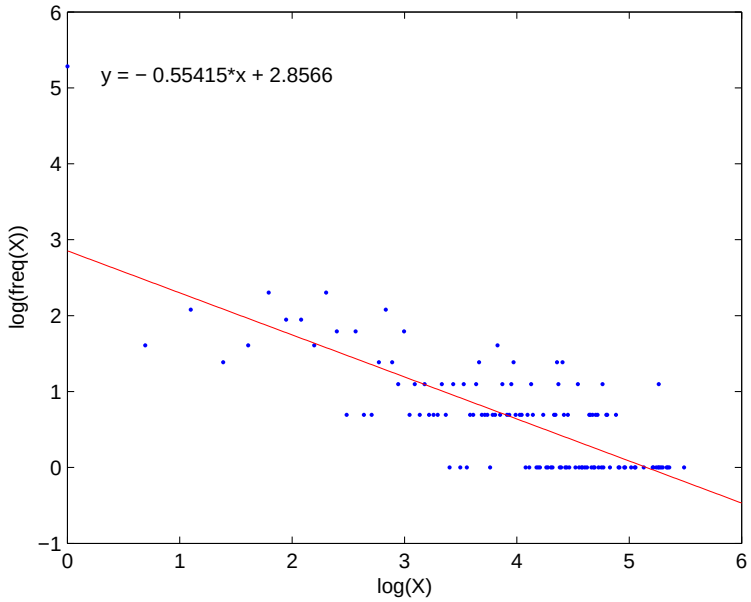

Figure 5.4: Scenario Two, Trial Two: Avalanche Frequency-Size Distribution

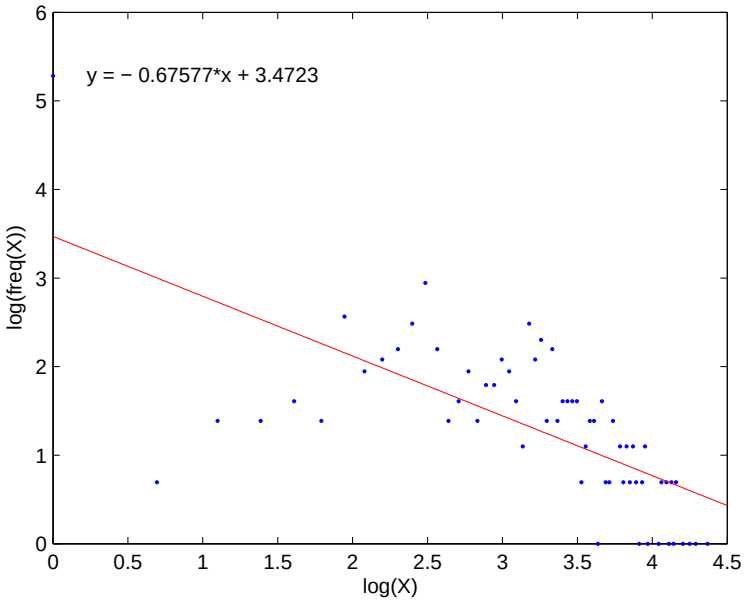

Figure 5.5: Scenario Two, Trial Two: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	17 (34%)	5.58 (11.16%)
Insurgent	0 (0%)	40 (40%)	19.78 (19.78%)

Table 5.8: Scenario Two, Trial Two: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	8	0.16

Table 5.9: Scenario Two, Trial Two: Utility Failure Numbers

5.1.3 Trial Three

For our next trial we reduce the number of Peacekeepers to see how squad size affects their success rate regarding fixing the water supply. The parameters are shown in Table 5.10.

Parameter	Value
$PEACENO$	25
Peacekeeper W_{15}	-100

Table 5.10: Scenario Two, Trial Three: Parameters

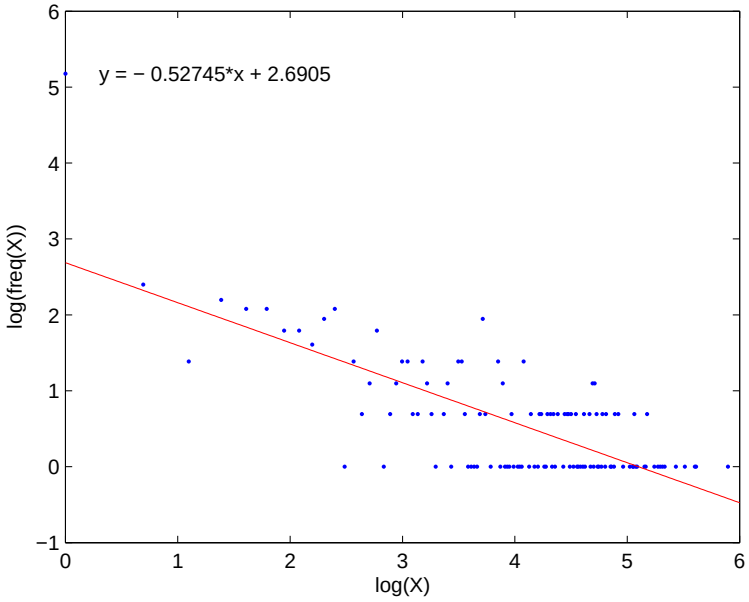

Figure 5.6: Scenario Two, Trial Three: Avalanche Frequency-Size Distribution

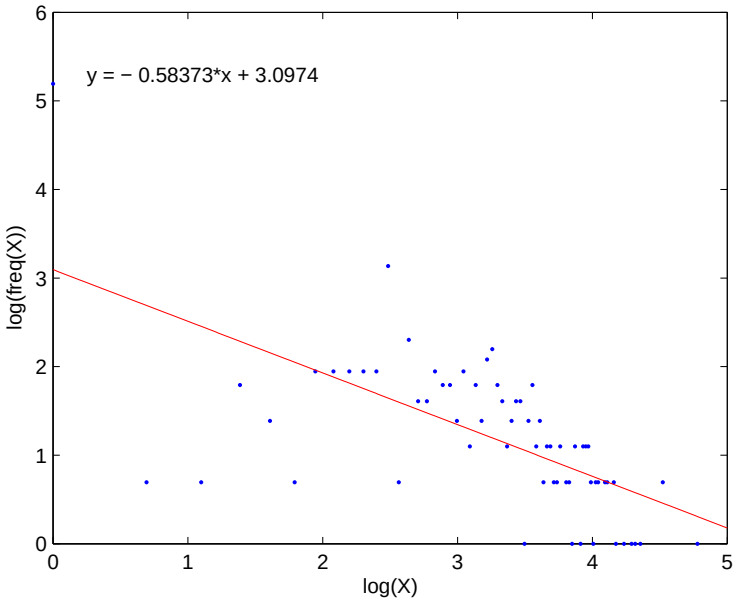

Figure 5.7: Scenario Two, Trial Three: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	16 (64%)	5.54 (22.16%)
Insurgent	0 (0%)	36 (36%)	17.40 (17.40%)

Table 5.11: Scenario Two, Trial Three: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	1	0.02

Table 5.12: Scenario Two, Trial Three: Utility Failure Numbers

5.1.4 Trial Four

We now reduce the negative weight to the hostile Insurgents for the Peacekeepers so it is back to its initial value. The parameters are given in Table 5.13.

Parameter	Value
$PEACENO$	25
Peacekeeper W_{15}	-50

Table 5.13: Scenario Two, Trial Four: Parameters

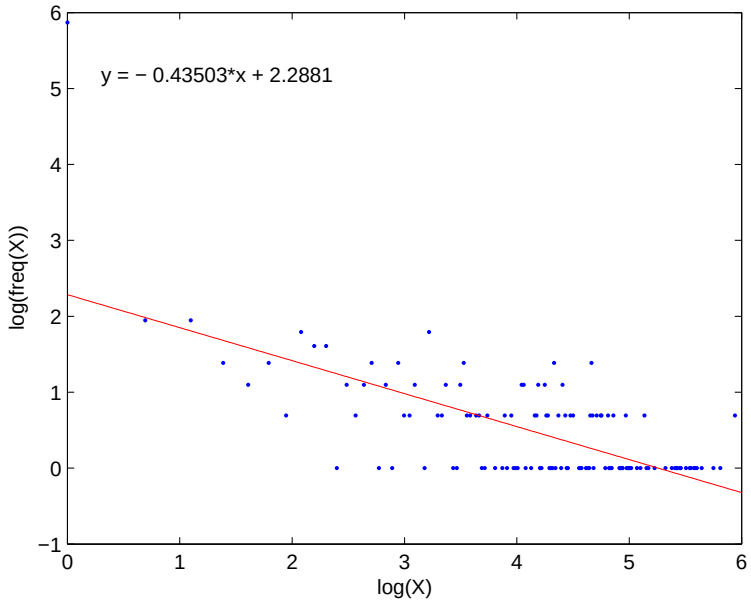

Figure 5.8: Scenario Two, Trial Four: Avalanche Frequency-Size Distribution

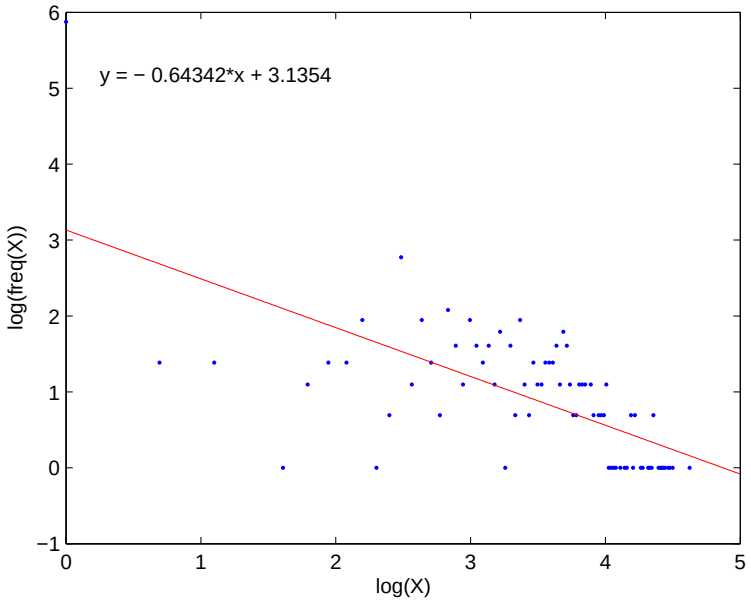

Figure 5.9: Scenario Two, Trial Four: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	20 (80%)	10.94 (43.76%)
Insurgent	5 (5%)	30 (30%)	18.52 (18.52%)

Table 5.14: Scenario Two, Trial Four: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	10	0.20

Table 5.15: Scenario Two, Trial Four: Utility Failure Numbers

5.1.5 Trial Five

Next we change the Peacekeepers' weight to hostile Insurgents so that it is halfway between the two previous values of -50 and -100. The parameters for this set of simulations are given in Table 5.16.

Parameter	Value
$PEACENO$	25
Peacekeeper W_{15}	-75

Table 5.16: Scenario Two, Trial Five: Parameters

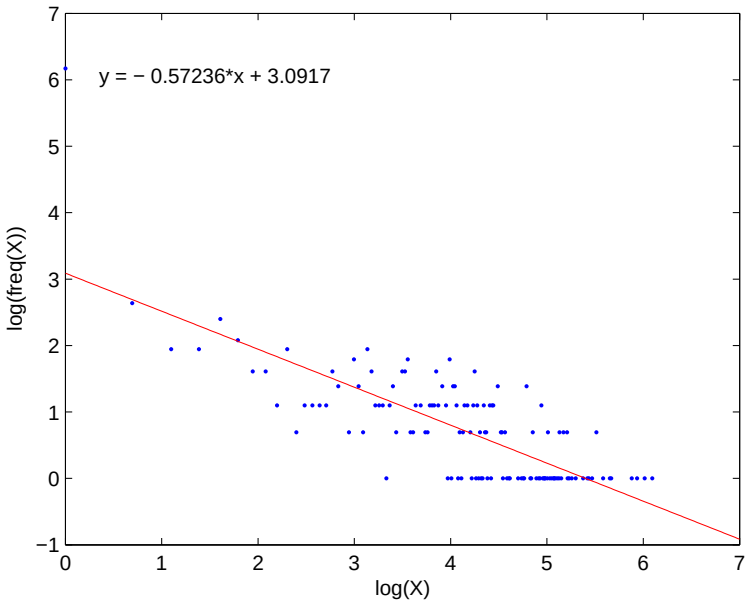

Figure 5.10: Scenario Two, Trial Five: Avalanche Frequency-Size Distribution

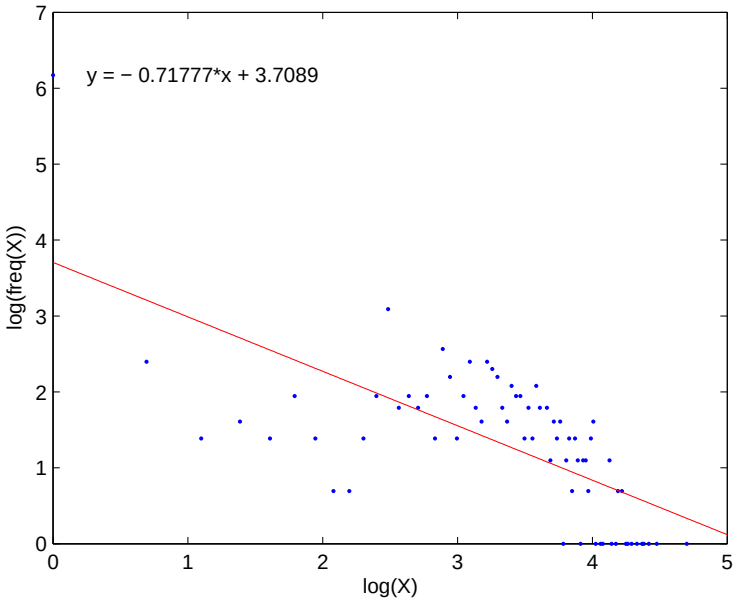

Figure 5.11: Scenario Two, Trial Five: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	1 (4%)	20 (80%)	9.84 (39.36%)
Insurgent	7 (7%)	38 (38%)	23.50 (23.50%)

Table 5.17: Scenario Two, Trial Five: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	5	0.10

Table 5.18: Scenario Two, Trial Five: Utility Failure Numbers

5.1.6 Trial Six

The parameter values for this trial are given in Table 5.19. We try increasing the number of Peacekeepers to see if this improves their ability to fix the water supply or if adding agents just increases the casualty numbers. The weight to hostile Insurgents is set back to -50.

Parameter	Value
$PEACENO$	100
Peacekeeper W_{15}	-50

Table 5.19: Scenario Two, Trial Six: Parameters

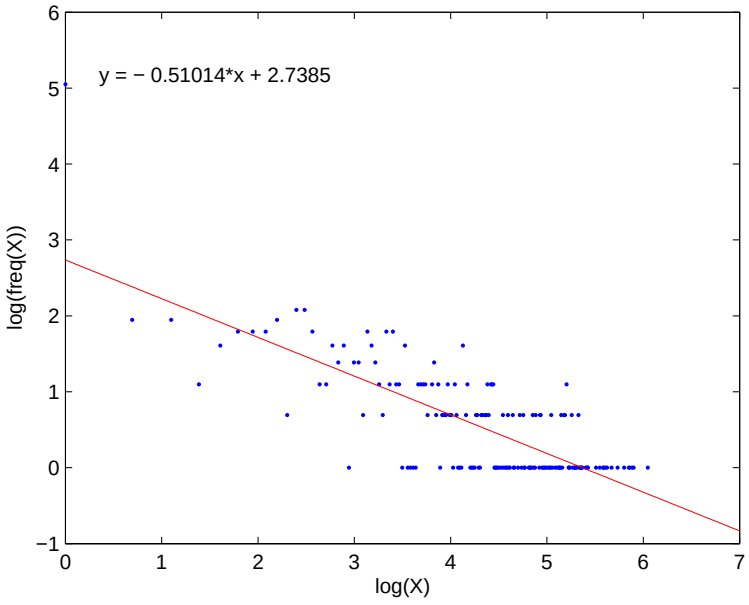

Figure 5.12: Scenario Two, Trial Six: Avalanche Frequency-Size Distribution

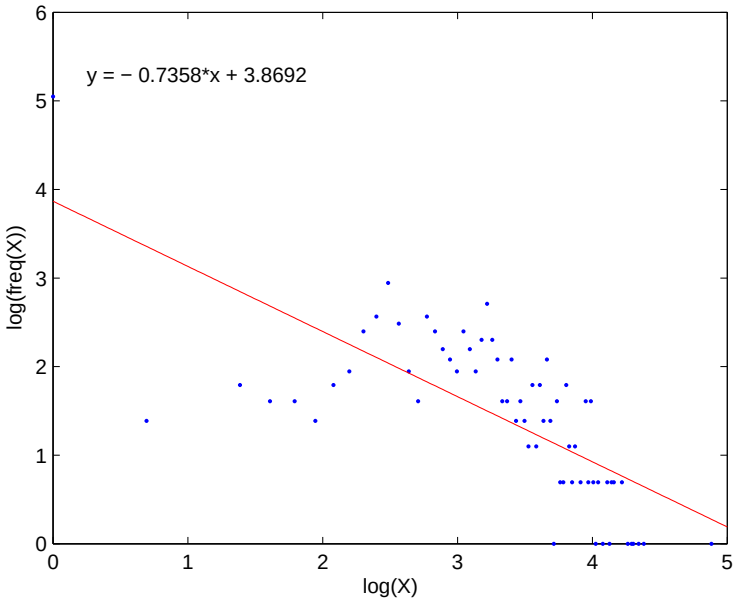

Figure 5.13: Scenario Two, Trial Six: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	2 (2%)	15 (15%)	7.74 (7.74%)
Insurgent	10 (10%)	56 (56%)	38.22 (38.22%)

Table 5.20: Scenario Two, Trial Six: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	8	0.16

Table 5.21: Scenario Two, Trial Six: Utility Failure Numbers

5.1.7 Trial Seven

Our final set of runs for this scenario takes the increased number of Peacekeepers and increases the negative weight towards hostile Insurgents to −100. The parameters are given in Table 5.22.

Parameter	Value
$PEACENO$	100
Peacekeeper W_{15}	-100

Table 5.22: Scenario Two, Trial Seven: Parameters

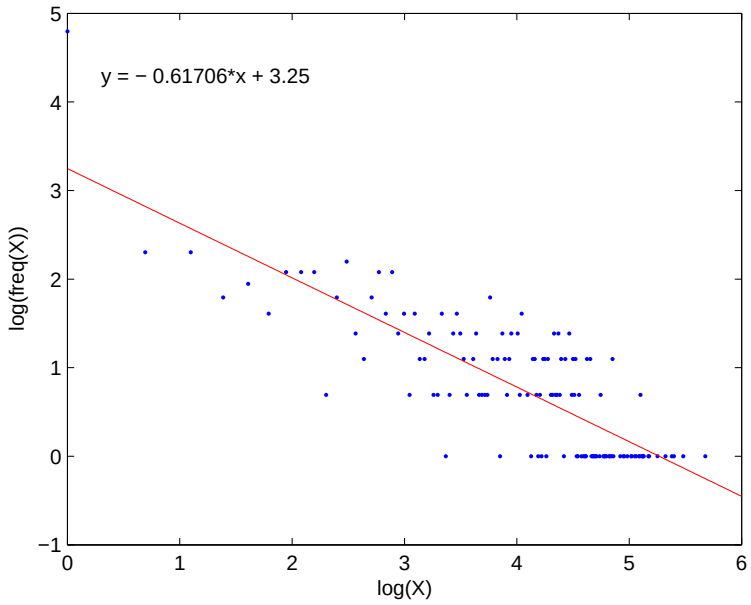

Figure 5.14: Scenario Two, Trial Seven: Avalanche Frequency-Size Distribution

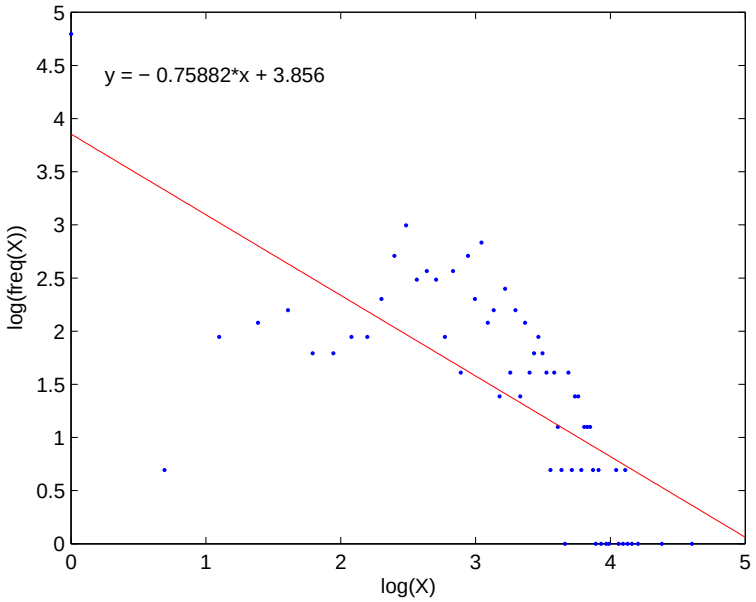

Figure 5.15: Scenario Two, Trial Seven: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	1 (1%)	14 (14%)	6.22 (6.22%)
Insurgent	12 (12%)	54 (54%)	26.52 (26.52%)

Table 5.23: Scenario Two, Trial Seven: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	6	0.12

Table 5.24: Scenario Two, Trial Seven: Utility Failure Numbers

5.1.8 Conclusions

There are two considerations for these experiments: first is the search for possible self-organised criticality, and secondly we have the general question of how successful the Peacekeepers were at carrying out repairs. Looking first at the question of whether we can find any evidence of SOC, our conclusion would have to be no from the results shown here. All the avalanche and time avalanche graphs show a general downward trend but none of them are close enough to a straight line. We can also see from the graphs that there are a wider range of sizes for the avalanches as opposed to the time avalanches, so the avalanche plots appear to be more stepped; that is we have lines of points on the avalanche graphs. More factors will be added in to the next scenario, Scenario Three, so this added complexity may produce SOC behaviour.

We shall now focus on the effectiveness of the Peacekeepers. We can see that in Trial Three we had a low Peacekeeper squad size, 25, and a high weight to avoid the Insurgents, -100, and this resulted in only one occasion where the water supply was fixed over the 50 model runs. When we increased the weight towards the Insurgents in Trial Four this figure increased to ten sectors fixed out of 50. Unfortunately this also came with an doubling of the mean casualties for the Peacekeepers. The lowest casualties occurred in Trial Two where we had a mean figure of 5.58 Peacekeeper deaths, this was from a squad size of 50 with the maximum negative weight of -100 towards the Insurgents. In this set of runs we also had eight instances of the water supply being fixed out of fifty so this seems to be the optimum set-up for this particular scenario.

5.2 Scenario Three

We now move on to Scenario Three. For this scenario we have added complexity to the model by introducing a squad of NGOs and the possibility that the water and electricity supply can fail at any sector on the grid. We still have the initial water supply failure in the final sector but now we will have the two squads of outside agents, the Peacekeepers and NGOs, who are aiming to get to the sector to fix the supply. The initial set up for the scenario is shown in Figure 5.16.

The general parameters for this scenario are shown in Table 5.25. Notice that many are the same as those in Scenario Two, in particular the Civilian agents still have a capability of zero so they are unable to move around the grid.

Constant	Value
$RUNTIME$	500
$GRIDSIZE$	200
$SECTOR$	5
$NOOFSQUADS$	4
$PEACENO$	Varies
$SUPPORTNO$	Varies
$LOCALNO$	100
$CIVNO$	1500
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$CIVCAP$	0
$BOMBMEMORY$	20
$SHOTMEMORY$	10
$LMBOMBPROB$	0.00
$LMFIREPROB$	0.01
$WATERFAIL$	Varies
$ELECFAIL$	Varies
$MAXSHOTS$	5000
$MAXBOMB$	100

Table 5.25: Scenario Three: General Parameters

The agent parameters for the scenario are given in Table 5.26. Notice that we

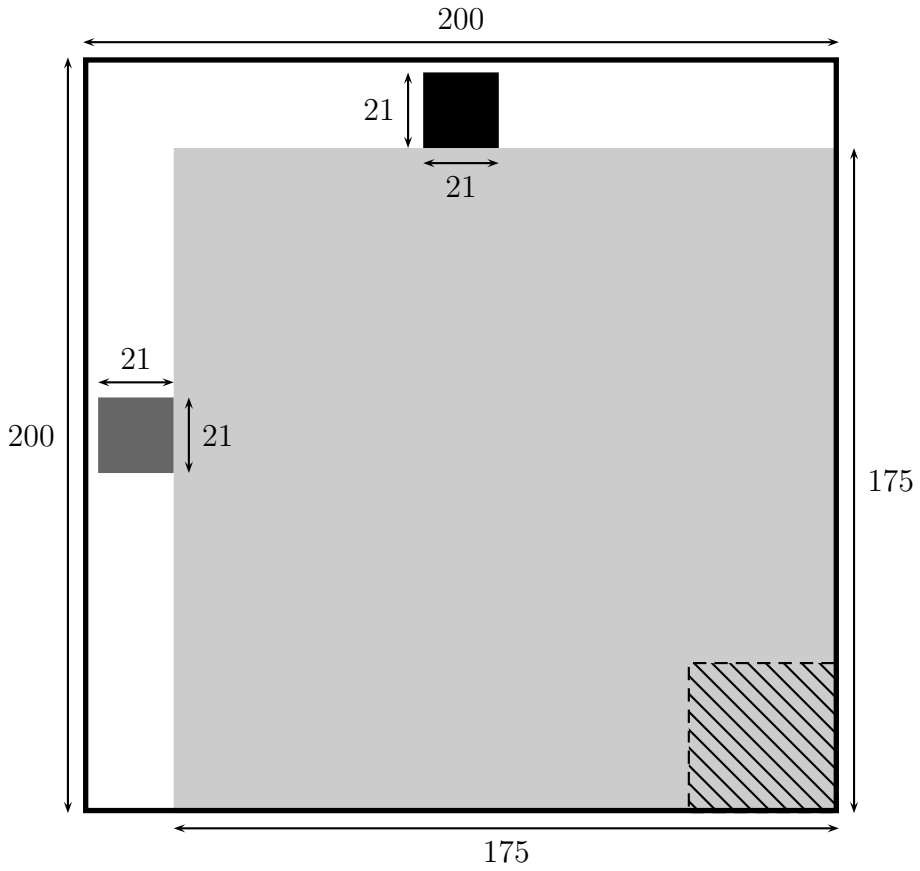

Figure 5.16: Initial Grid for Scenario Three

have set the NGOs' weight to uncooperative Insurgents at -100 and the Peacekeepers' weight to hostile Insurgents at -50. This is since the NGOs are unarmed and so they will be more likely to avoid the Insurgents than the Peacekeepers. Note also that as with the Peacekeepers, the NGO sensor range has been set to 200 to take account of intelligence and communications. All the parameters shown in this table will stay constant throughout the different trials.

There are nine variations of this scenario, the results for these trials are given in Sections 5.2.1 to 5.2.9.

Parameter	Peace.	NGO	Ins.	Civ.
$squadNo$	1	2	3	4
$xHome$	99	14	112	112
$yHome$	14	99	112	112
$homeRadius$	10	10	87	87
$sensorRange$	200	200	25	25
$fireRange$	20	-	15	-
$bombRadius$	-	-	-	-
$sSKP$	0.10	-	0.05	-
Relationship to Peacekeepers	F	F	H	C
Relationship to NGOs	F	F	F	F
Relationship to Insurgents	H	U	F	F
Relationship to Civilians	F	F	F	F
W_1	0	0	0	0
W_2	0	0	0	0
W_3	0	0	0	0
W_4	0	0	0	0
W_5	0	0	0	0
W_6	0	0	0	0
W_7	0	0	0	0
W_8	0	0	0	0
W_9	0	0	0	0
W_{10}	0	0	0	0
W_{11}	0	0	0	0
W_{12}	0	0	0	0
W_{13}	0	0	0	0
W_{14}	0	-100	0	0
W_{15}	-50	0	0	0
W_{16}	0	0	0	0
W_{17}	0	0	0	0
W_{18}	0	0	0	0
W_{19}	0	0	0	0
W_{20}	0	0	0	0
W_{22}	10	10	-	-
W_{23}	10	10	-	-
W_{24}	10	10	-	-
W_{25}	0	-	0	-
$probFixWater$	1.00	1.00	-	-
$probFixElec$	1.00	1.00	-	-

Table 5.26: Scenario Three: Agent Parameters

5.2.1 Trial One

We start with a basic scenario with no possibility of water and electricity failures other than the initial fault with the water supply. The Peacekeeper and NGO squads are of equal size, both consisting of 25 agents, hence there are 50 agents who will be attempting to repair the water supply. The parameters are given in Table 5.27.

Parameter	Value
$PEACENO$	25
$SUPPORTNO$	25
$WATERFAIL$	0.00
$ELECFAIL$	0.00

Table 5.27: Scenario Three, Trial One: Parameters

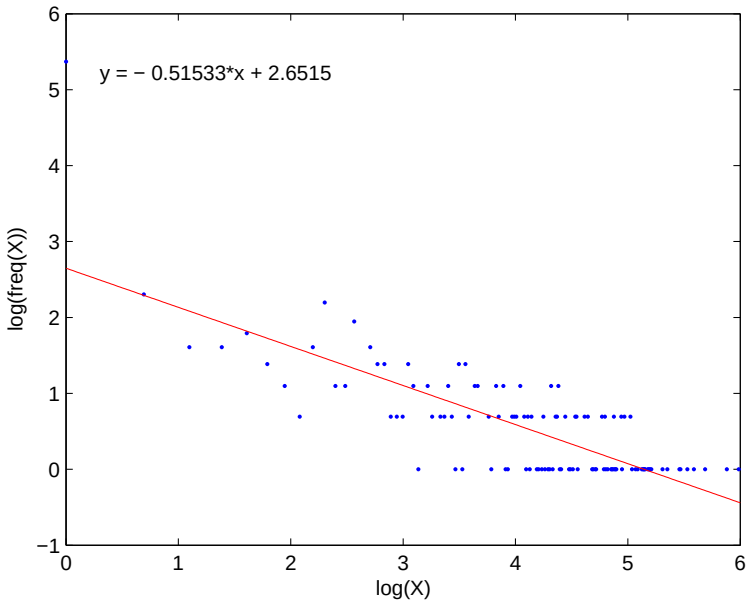

Figure 5.17: Scenario Three, Trial One: Avalanche Frequency-Size Distribution

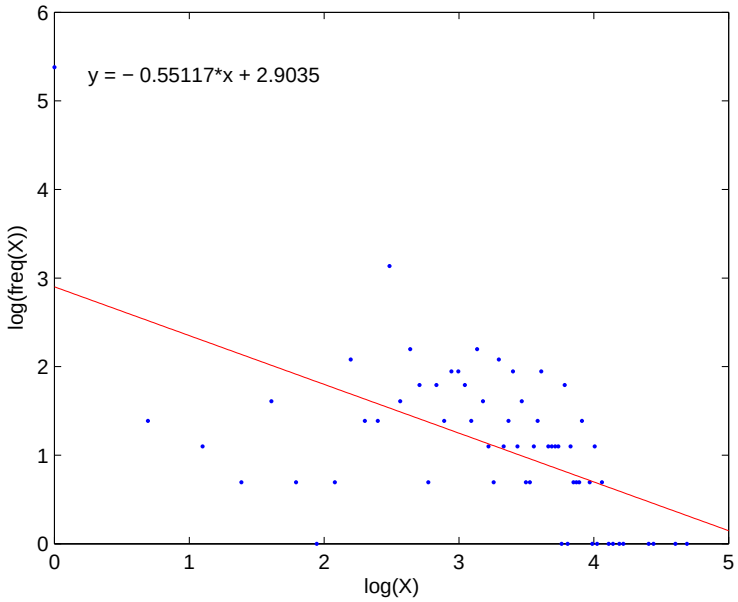

Figure 5.18: Scenario Three, Trial One: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	18 (72%)	6.12 (24.48%)
Insurgent	2 (2%)	30 (30%)	18.88 (18.88%)

Table 5.28: Scenario Three, Trial One: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	8	0.16

Table 5.29: Scenario Three, Trial One: Utility Failure Numbers

5.2.2 Trial Two

For this trial we increase the number of Peacekeepers and NGOs so there are now 50 agents in each squad. The parameters are shown in Table 5.30.

Parameter	Value
$PEACENO$	50
$SUPPORTNO$	50
$WATERFAIL$	0.00
$ELECFAIL$	0.00

Table 5.30: Scenario Three, Trial Two: Parameters

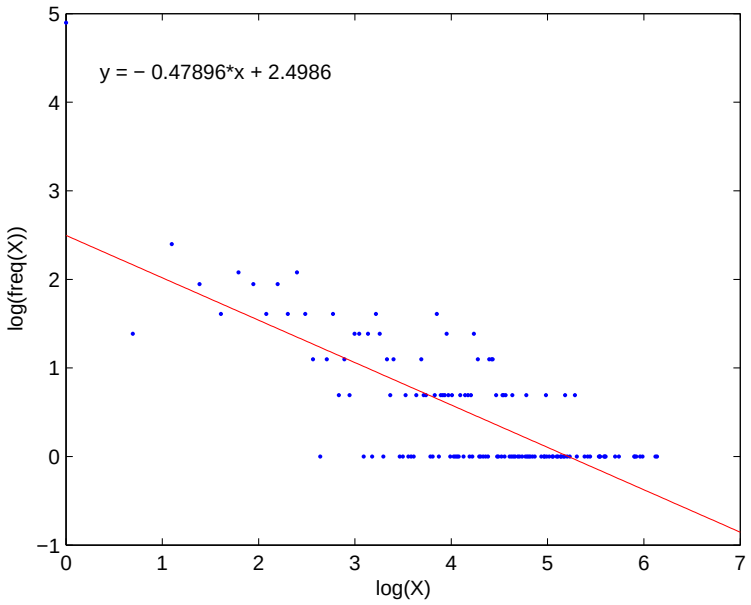

Figure 5.19: Scenario Three, Trial Two: Avalanche Frequency-Size Distribution

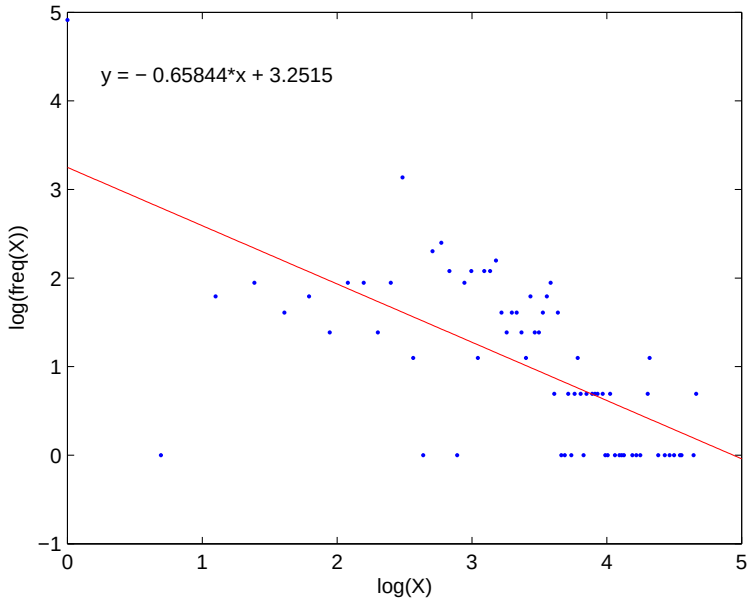

Figure 5.20: Scenario Three, Trial Two: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	2 (4%)	15 (30%)	6.46 (12.92%)
Insurgent	4 (4%)	46 (46%)	24.80 (24.80%)

Table 5.31: Scenario Three, Trial Two: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	9	0.18

Table 5.32: Scenario Three, Trial Two: Utility Failure Numbers

5.2.3 Trial Three

The parameters for this variation on Scenario Three are shown in Table 5.33. We now introduce the possibility that the water and electricity supply can fail at any sector on the grid as the model is running. We set this probability at 0.01, so for every sector at every timestep this is the chance that each utility will fail. The number of Peacekeepers and NGOs is put back to 25 per squad.

Parameter	Value
$PEACENO$	25
$SUPPORTNO$	25
$WATERFAIL$	0.01
$ELECFAIL$	0.01

Table 5.33: Scenario Three, Trial Three: Parameters

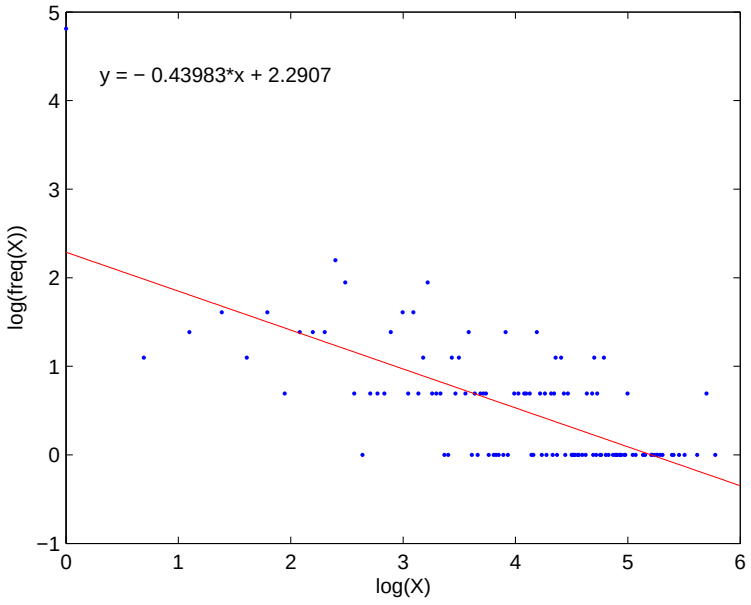

Figure 5.21: Scenario Three, Trial Three: Avalanche Frequency-Size Distribution

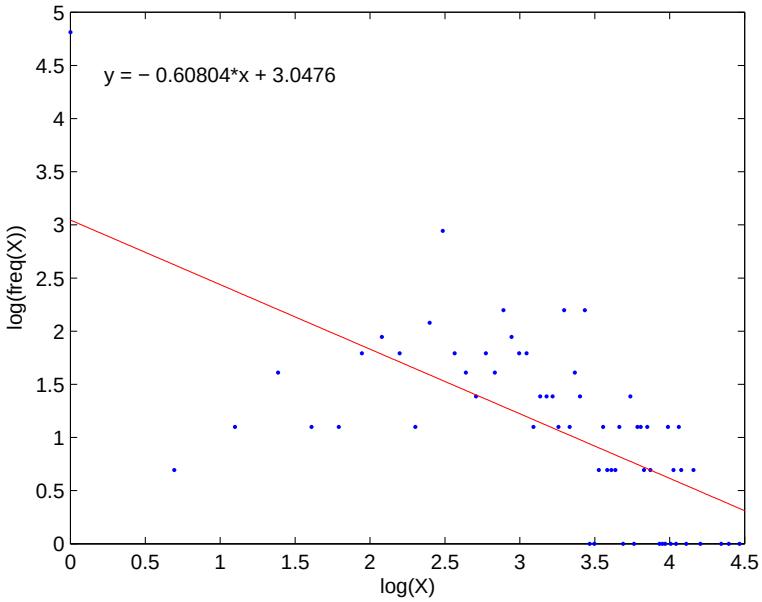

Figure 5.22: Scenario Three, Trial Three: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	4.80 (19.20%)
Insurgent	4 (4%)	33 (33%)	17.78 (17.78%)

Table 5.34: Scenario Three, Trial Three: Casualty Numbers

	Total	Mean per Run
Number of Failures	2581	51.62
Number Fixed	100	2

Table 5.35: Scenario Three, Trial Three: Utility Failure Numbers

5.2.4 Trial Four

Since we found that the probabilities for the utility failures had been set too high we reduce them to 0.001 for this set of model runs. The parameters for this trial are given in Table 5.36.

Parameter	Value
$PEACENO$	25
$SUPPORTNO$	25
$WATERFAIL$	0.001
$ELECFAIL$	0.001

Table 5.36: Scenario Three, Trial Four: Parameters

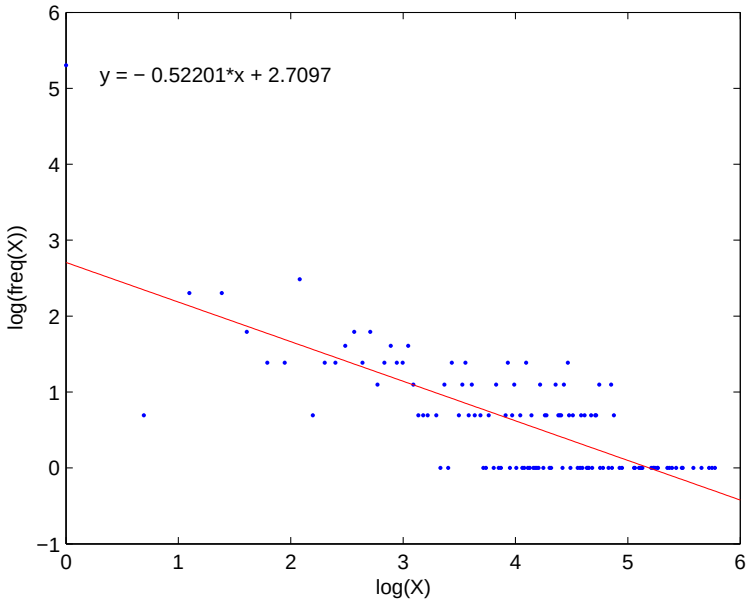

Figure 5.23: Scenario Three, Trial Four: Avalanche Frequency-Size Distribution

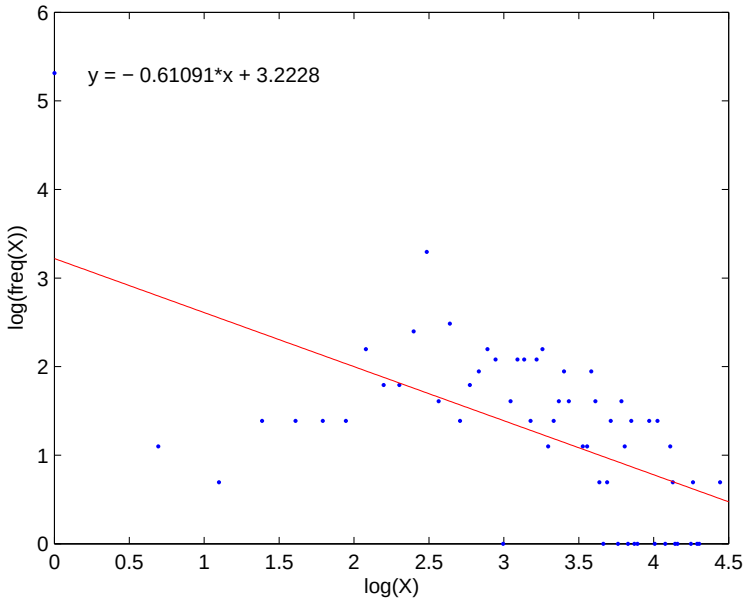

Figure 5.24: Scenario Three, Trial Four: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	5.98 (23.92%)
Insurgent	4 (4%)	39 (39%)	20.40 (20.40%)

Table 5.37: Scenario Three, Trial Four: Casualty Numbers

	Total	Mean per Run
Number of Failures	1008	20.16
Number Fixed	46	0.92

Table 5.38: Scenario Three, Trial Four: Utility Failure Numbers

5.2.5 Trial Five

The parameters for Scenario Three, Trial Five are shown in Table 5.39. Again we reduce the probabilities for water and electricity failure, this time to 0.0001.

Parameter	Value
$PEACENO$	25
$SUPPORTNO$	25
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.39: Scenario Three, Trial Five: Parameters

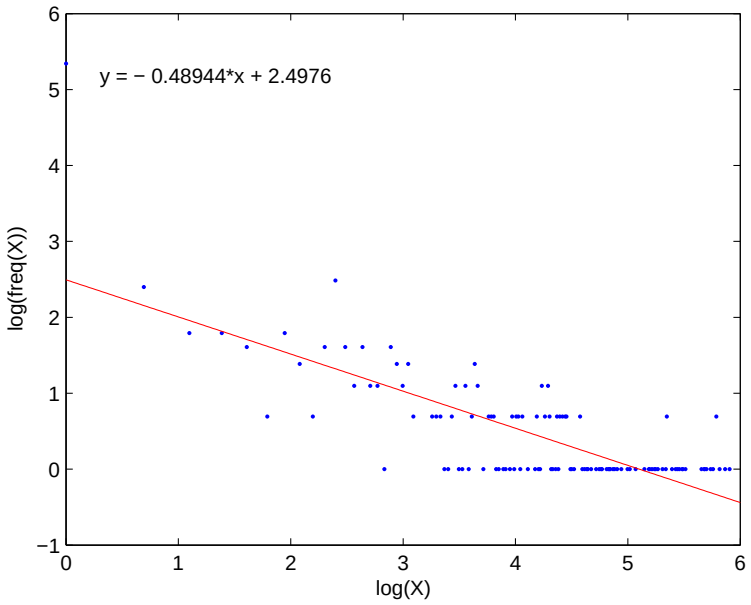

Figure 5.25: Scenario Three, Trial Five: Avalanche Frequency-Size Distribution

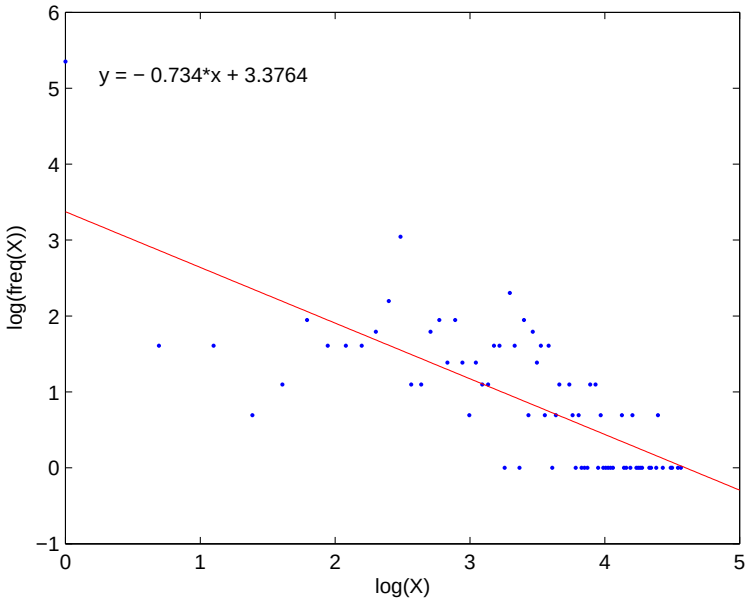

Figure 5.26: Scenario Three, Trial Five: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	5.98 (23.92%)
Insurgent	3 (3%)	39 (39%)	19.52 (19.52%)

Table 5.40: Scenario Three, Trial Five: Casualty Numbers

	Total	Mean per Run
Number of Failures	202	4.04
Number Fixed	6	0.12

Table 5.41: Scenario Three, Trial Five: Utility Failure Numbers

5.2.6 Trial Six

Now the utility failure probabilities have been set to a sensible level we can vary the squad sizes to see what effect this has. We double the Peacekeeper and NGO squads so that they each contain 50 agents. The scenario parameters are shown in Table 5.42.

Parameter	Value
$PEACENO$	50
$SUPPORTNO$	50
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.42: Scenario Three, Trial Six: Parameters

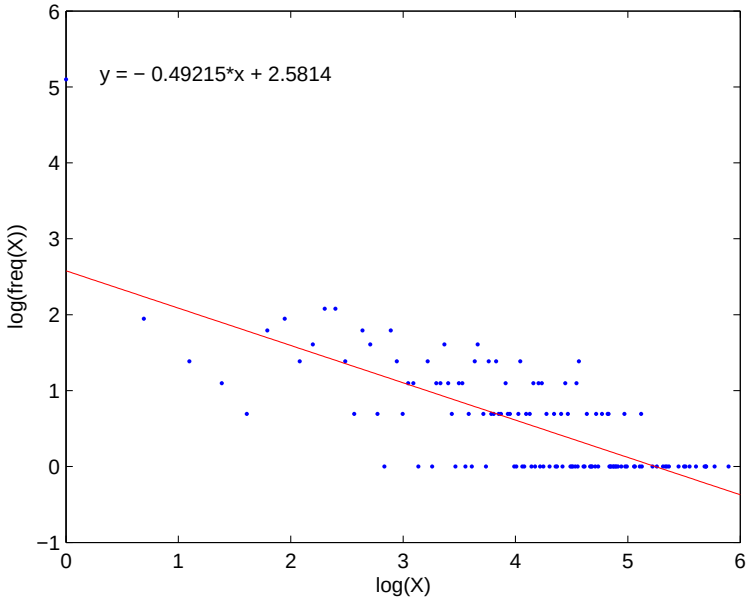

Figure 5.27: Scenario Three, Trial Six: Avalanche Frequency-Size Distribution

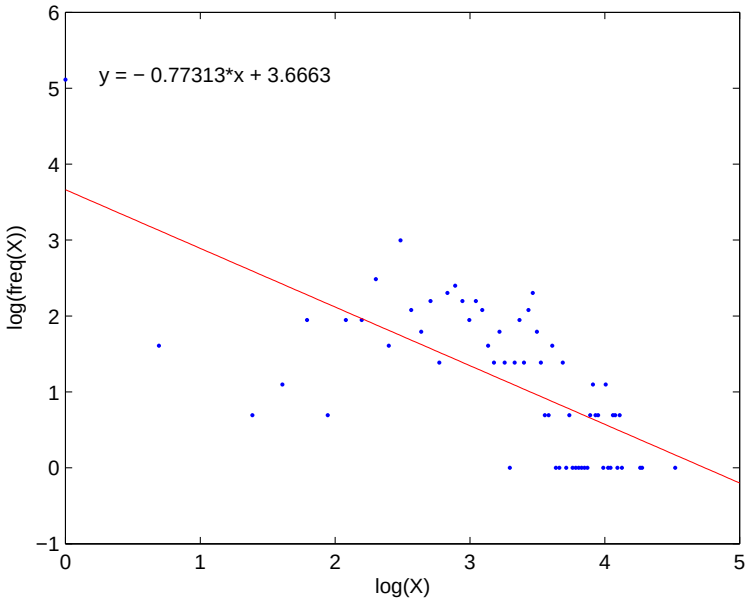

Figure 5.28: Scenario Three, Trial Six: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	20 (40%)	6.08 (12.16%)
Insurgent	2 (2%)	53 (53%)	23.02 (23.02%)

Table 5.43: Scenario Three, Trial Six: Casualty Numbers

	Total	Mean per Run
Number of Failures	201	4.02
Number Fixed	18	0.36

Table 5.44: Scenario Three, Trial Six: Utility Failure Numbers

5.2.7 Trial Seven

For this trial we increase the water and electricity failure probabilities to 0.0002 and reduce the NGO and Peacekeeper squads to 25 agents. The parameters are recorded in Table 5.45.

Parameter	Value
$PEACENO$	25
$SUPPORTNO$	25
$WATERFAIL$	0.0002
$ELECFAIL$	0.0002

Table 5.45: Scenario Three, Trial Seven: Parameters

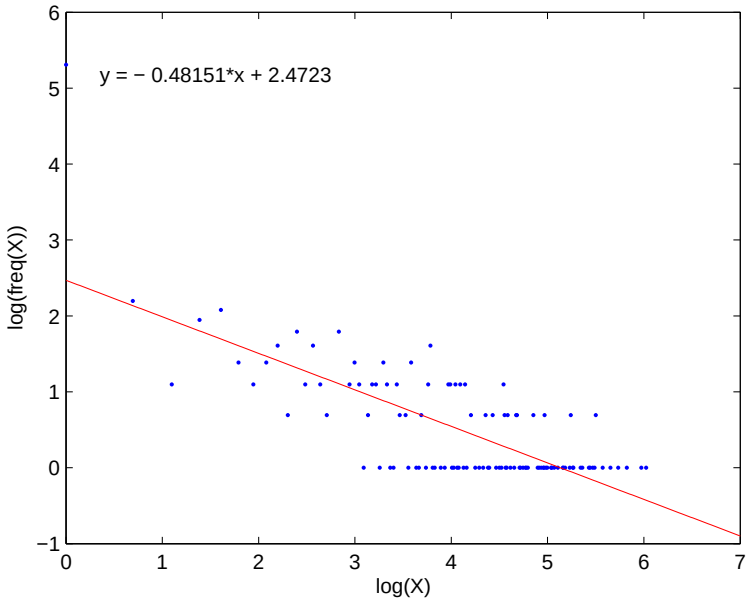

Figure 5.29: Scenario Three, Trial Seven: Avalanche Frequency-Size Distribution

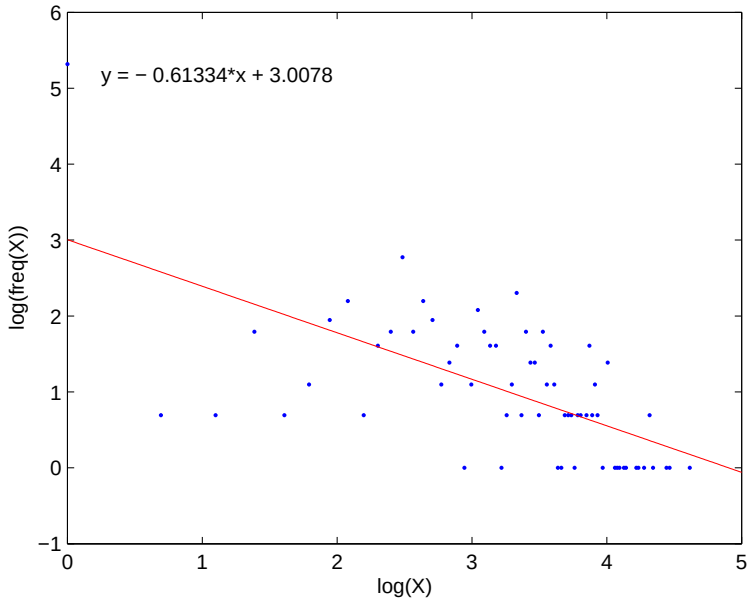

Figure 5.30: Scenario Three, Trial Seven: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	21 (84%)	6.30 (25.20%)
Insurgent	0 (0%)	37 (37%)	18.70 (18.70%)

Table 5.46: Scenario Three, Trial Seven: Casualty Numbers

	Total	Mean per Run
Number of Failures	296	5.92
Number Fixed	10	0.20

Table 5.47: Scenario Three, Trial Seven: Utility Failure Numbers

5.2.8 Trial Eight

The parameters for this trial are given in Table 5.48. We have increased the Peacekeeper and NGO squad sizes to 50.

Parameter	Value
$PEACENO$	50
$SUPPORTNO$	50
$WATERFAIL$	0.0002
$ELECFAIL$	0.0002

Table 5.48: Scenario Three, Trial Eight: Parameters

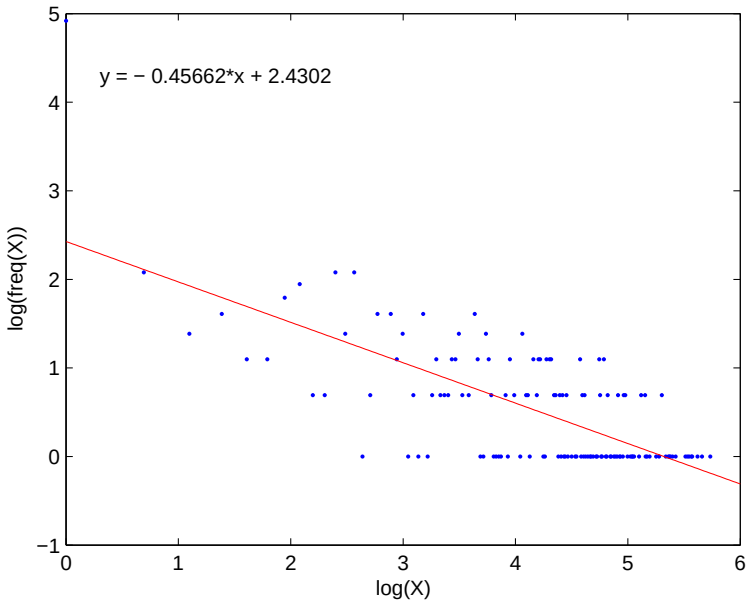

Figure 5.31: Scenario Three, Trial Eight: Avalanche Frequency-Size Distribution

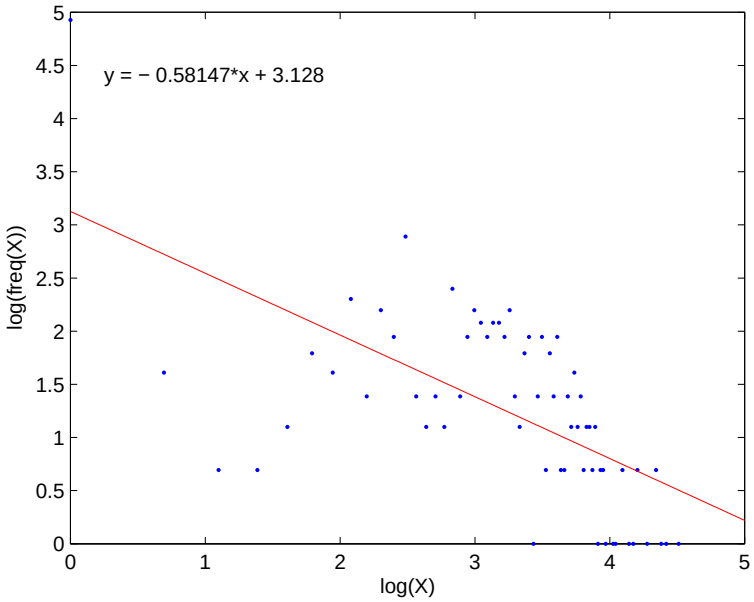

Figure 5.32: Scenario Three, Trial Eight: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	17 (34%)	6.24 (12.48%)
Insurgent	4 (4%)	51 (51%)	24.70 (24.70%)

Table 5.49: Scenario Three, Trial Eight: Casualty Numbers

	Total	Mean per Run
Number of Failures	304	6.08
Number Fixed	19	0.38

Table 5.50: Scenario Three, Trial Eight: Utility Failure Numbers

5.2.9 Trial Nine

For our final variation of Scenario Three we again increase the number of agents in the Peacekeeper and NGO squads. This time they are each of size 75. The parameters are shown in Table 5.51.

Parameter	Value
$PEACENO$	75
$SUPPORTNO$	75
$WATERFAIL$	0.0002
$ELECFAIL$	0.0002

Table 5.51: Scenario Three, Trial Nine: Parameters

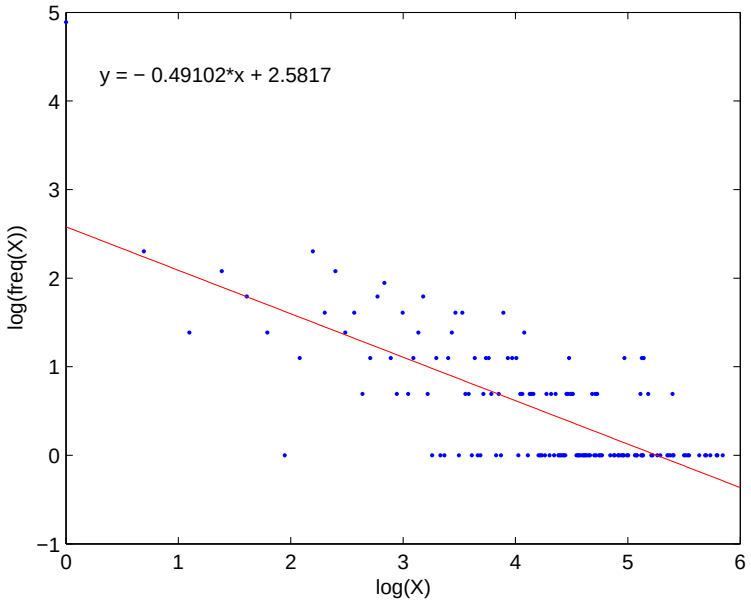

Figure 5.33: Scenario Three, Trial Nine: Avalanche Frequency-Size Distribution

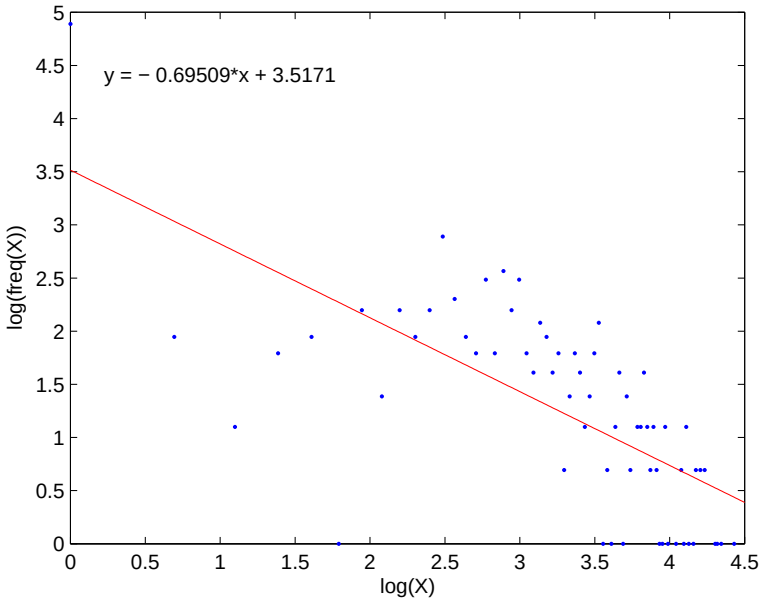

Figure 5.34: Scenario Three, Trial Nine: Time Avalanche Frequency-Size Distribution

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	18 (24%)	7.30 (9.73%)
Insurgent	3 (3%)	59 (59%)	27.80 (27.80%)

Table 5.52: Scenario Three, Trial Nine: Casualty Numbers

	Total	Mean per Run
Number of Failures	311	6.22
Number Fixed	23	0.46

Table 5.53: Scenario Three, Trial Nine: Utility Failure Numbers

5.2.10 Conclusions

Looking at the plots for the time avalanche distributions, it appears that the points are even more spread out around the line of best fit than those for Scenario Two, despite the added complexity of Scenario Three. We definitely could not say that these plots show evidence of self-organised criticality. The avalanche plots seem similar on general shape to those for Scenario Two so again we cannot say they show evidence of SOC.

In this scenario we have two squads of agents looking to help the local population, but we also have added the probability of multiple supply failures. In Trial Three and Trial Four we set the water and electricity failure probabilities too high so these results are not particularly meaningful. Doubling the number of Peacekeepers and NGOs to 50 of each from Trial One to Trial Two did not seem to have much effect. These two trials had similar mean Peacekeeper casualty numbers and number of sectors fixed. However when we had multiple sector supply failures in Trials Five and Six, doubling the number of Peacekeepers and NGOs to 50 agents per squad tripled the number of fixed sectors whereas the mean Peacekeeper casualty figures stayed fairly constant. Similarly, doubling the squad sizes from Trial Seven to Trial Eight nearly doubled the number of sectors fixed but the casualty figures for the Peacekeepers did not change significantly. Increasing the squad sizes again to 75 agents for Trial Nine again increased the number of sectors fixed but not by a significant amount, there was also an increase in the mean casualty figure for the Peacekeepers.

5.3 Scenario Four

Scenario Four has the same initial layout as Scenario Three in terms of the squad distributions and the initial failure of the water supply in the final sector. This set-up is shown again in Figure 5.35.

The general model parameters are shown in Table 5.54. Our main aim for this scenario is to show what happens when Insurgents use suicide bombs as their initial method of attack, hence the probability that they will fire at random is zero and their bomb probability is varied throughout the different model runs. We also give the Civilians the ability to move around the grid since they have a non-zero capability for all but one of the scenario variations. Notice that we also reduced the number of Civilians and Insurgents in the model to make the grid less crowded and to reduce the time taken for each simulation. The number of Peacekeepers and NGOs is also fixed at 25 agents per squad.

The agent parameters for the scenario are given in Table 5.55. These values all stay the same throughout the different variations of the scenario.

When we were running this scenario we decided that it would not be useful when looking for SOC behaviour since there are relatively few avalanches. Therefore we only ran each set of runs 15 times, or 16 in the case of Run One, to get a general idea of what was happening so we could adapt the scenario further as Scenario Five.

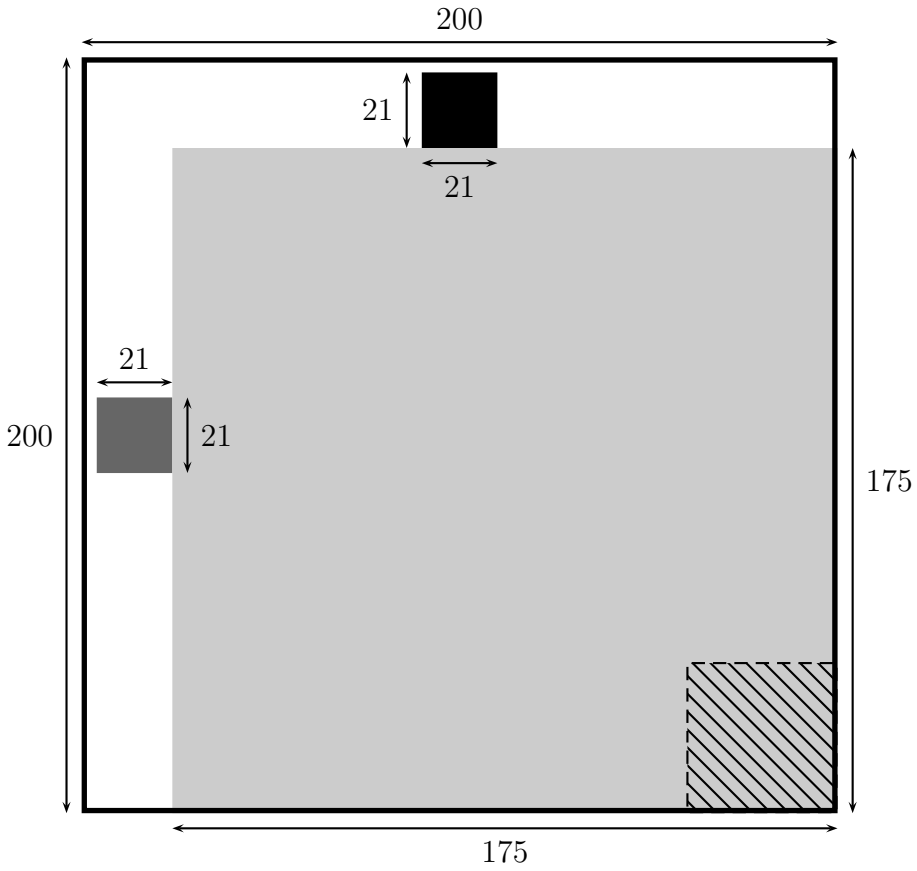

Figure 5.35: Initial Grid for Scenario Four

Constant	Value
$RUNTIME$	Varies
$GRIDSIZE$	200
$SECTOR$	5
$NOOFSQUADS$	4
$PEACENO$	25
$SUPPORTNO$	25
$LOCALNO$	50
$CIVNO$	1000
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$CIVCAP$	Varies
$BOMBMEMORY$	20
$SHOTMEMORY$	10
$LMBOMBPROB$	Varies
$LMFIREPROB$	0.00
$WATERFAIL$	Varies
$ELECFAIL$	Varies
$MAXSHOTS$	5000
$MAXBOMB$	100

Table 5.54: Scenario Four: General Parameters

Parameter	Peace.	NGO	Ins.	Civ.
$squadNo$	1	2	3	4
$xHome$	99	14	112	112
$yHome$	14	99	112	112
$homeRadius$	10	10	87	87
$sensorRange$	200	200	50	50
$fireRange$	20	-	15	-
$bombRadius$	-	-	5	-
$sSKP$	0.10	-	0.05	-
Relationship to Peacekeepers	F	F	H	C
Relationship to NGOs	F	F	U	F
Relationship to Insurgents	H	U	F	U
Relationship to Civilians	F	F	F	F
W_1	0	0	0	0
W_2	0	0	0	0
W_3	0	0	0	0
W_4	0	0	0	0
W_5	0	0	0	0
W_6	0	0	0	0
W_7	0	0	0	0
W_8	0	0	0	0
W_9	0	0	0	0
W_{10}	0	0	0	0
W_{11}	0	0	0	0
W_{12}	0	0	0	0
W_{13}	0	0	0	0
W_{14}	0	-100	0	0
W_{15}	-50	0	0	0
W_{16}	0	0	0	0
W_{17}	0	0	0	0
W_{18}	0	0	0	0
W_{19}	0	0	0	0
W_{20}	0	0	0	0
W_{22}	10	10	-	-
W_{23}	10	10	-	-
W_{24}	10	10	-	-
W_{25}	0	-	0	-
$probFixWater$	1.00	1.00	-	-
$probFixElec$	1.00	1.00	-	-

Table 5.55: Scenario Four: Agent Parameters

5.3.1 Trial One

We start with a bomb probability of 0.001 and to keep the scenario simple we set the water and electricity failure probabilities to be zero. The Civilian capability is one so the agents are able to move. The list of parameter values is shown in Table 5.56.

Parameter	Value
$RUNTIME$	500
$CIVCAP$	1
$LMBOMBPROB$	0.001
$WATERFAIL$	0.00
$ELECFAIL$	0.00

Table 5.56: Scenario Four, Trial One: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	1.25 (5%)
NGO	0 (0%)	23 (92%)	3.56 (14.25%)
Insurgent	0 (0%)	3 (6%)	0.56 (1.13%)
Civilian	0 (0%)	8 (0.80%)	0.94 (0.09%)

Table 5.57: Scenario Four, Trial One: Casualty Numbers

	Total	Mean per Run
Number of Failures	16	1
Number Fixed	1	0.06

Table 5.58: Scenario Four, Trial One: Utility Failure Numbers

5.3.2 Trial Two

The parameter values for this scenario are shown in Table 5.59. Here we introduce more complexity to the model by setting the water and electricity failure probabilities to be 0.0001.

Parameter	Value
$RUNTIME$	500
$CIVCAP$	1
$LMBOMBPROB$	0.001
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.59: Scenario Four, Trial Two: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	0 (0%)	0 (0%)
NGO	0 (0%)	7 (28%)	1.40 (5.60%)
Insurgent	0 (0%)	2 (4%)	0.40 (0.80%)
Civilian	0 (0%)	6 (0.60%)	1.33 (0.13%)

Table 5.60: Scenario Four, Trial Two: Casualty Numbers

	Total	Mean per Run
Number of Failures	67	4.47
Number Fixed	3	0.20

Table 5.61: Scenario Four, Trial Two: Utility Failure Numbers

5.3.3 Trial Three

Now we increase the water and electricity failure probabilities to 0.0002. We also set the Civilian capability back to zero to see what effect the Civilians being able to move had on the model. The parameters are given in Table 5.62.

Parameter	Value
$RUNTIME$	500
$CIVCAP$	0
$LMBOMBPROB$	0.001
$WATERFAIL$	0.0002
$ELECFAIL$	0.0002

Table 5.62: Scenario Four, Trial Three: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	1.73 (6.93%)
NGO	0 (0%)	10 (40%)	0.73 (2.93%)
Insurgent	0 (0%)	5 (10%)	0.80 (1.60%)
Civilian	0 (0%)	5 (0.50%)	0.80 (0.08%)

Table 5.63: Scenario Four, Trial Three: Casualty Numbers

	Total	Mean per Run
Number of Failures	91	6.07
Number Fixed	4	0.27

Table 5.64: Scenario Four, Trial Three: Utility Failure Numbers

5.3.4 Trial Four

The additional parameter values for this scenario are shown in Table 5.65. For this set of runs we set the Civilian capability back to one, the water and electricity failure probabilities back to 0.0001 and increase the Insurgent bomb probability to 0.002.

Parameter	Value
$RUNTIME$	500
$CIVCAP$	1
$LMBOMBPROB$	0.002
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.65: Scenario Four, Trial Four: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	11 (44%)	1.67 (6.67%)
NGO	0 (0%)	8 (32%)	1.73 (6.93%)
Insurgent	0 (0%)	6 (12%)	1.07 (2.13%)
Civilian	0 (0%)	7 (0.70%)	1.87 (0.19%)

Table 5.66: Scenario Four, Trial Four: Casualty Numbers

	Total	Mean per Run
Number of Failures	53	3.53
Number Fixed	3	0.20

Table 5.67: Scenario Four, Trial Four: Utility Failure Numbers

5.3.5 Trial Five

For this trial we increase the Insurgent bomb probability to 0.005. The parameters are shown in Table 5.68.

Parameter	Value
$RUNTIME$	500
$CIVCAP$	1
$LMBOMBPROB$	0.005
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.68: Scenario Four, Trial Five: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	5.20 (20.80%)
NGO	0 (0%)	25 (100%)	5 (20%)
Insurgent	0 (0%)	10 (20%)	4.33 (8.67%)
Civilian	0 (0%)	18 (1.80%)	6 (0.60%)

Table 5.69: Scenario Four, Trial Five: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	3.33
Number Fixed	7	0.47

Table 5.70: Scenario Four, Trial Five: Utility Failure Numbers

5.3.6 Trial Six

This variation of Scenario Four is the same as the previous trial but we run the model for 1000 timesteps instead of 500. The parameters are given in Table 5.71.

Parameter	Value
$RUNTIME$	1000
$CIVCAP$	1
$LMBOMBPROB$	0.005
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.71: Scenario Four, Trial Six: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	24 (96%)	7.13 (28.53%)
NGO	0 (0%)	19 (76%)	5.73 (22.93%)
Insurgent	0 (0%)	12 (24%)	5.80 (11.60%)
Civilian	0 (0%)	16 (1.60%)	7.67 (0.77%)

Table 5.72: Scenario Four, Trial Six: Casualty Numbers

	Total	Mean per Run
Number of Failures	242	16.13
Number Fixed	12	0.80

Table 5.73: Scenario Four, Trial Six: Utility Failure Numbers

5.3.7 Conclusions

We have not conducted any avalanche analysis for this scenario, there were so few data points that it would have been meaningless. Since the only combat comes as a result of suicide bomb attack, which we set to be fairly rare, we would either have to run the model for a very long time, or repeat the experiment a large number of times to get significant data. The model runs took a few hours each

so we did not have time to do this, instead we decided to abandon these scenario experiments and add random firing by the Insurgents. This became Scenario Five.

Looking at casualty numbers and the success the Peacekeepers and NGOs had repairing water and electricity supplies, again we cannot draw any meaningful conclusions since there is not a large enough sample size, but the results are given for completeness.

5.4 Scenario Five

Finally we have Scenario Five. For this set of trials we again have the same initial set-up as we had for Scenarios Three and Four, this is illustrated in Figure 5.36.

The general parameters for the model are given in Table 5.74. With this being the most complex set of simulations it follows that we would have the most variation of parameters, hence at various stages we change the agent numbers and capabilities as well as the probabilities related to the water and electricity failure. The Insurgents still use suicide bombs as a method of attack but here the bomb probability is fixed at 0.002 and firing at random targets is also used as an initial attack method with varying probability.

Constant	Value
$RUNTIME$	Varies
$GRIDSIZE$	200
$SECTOR$	5
$NOOFSQUADS$	4
$PEACENO$	Varies
$SUPPORTNO$	Varies
$LOCALNO$	50
$CIVNO$	1000
$PEACECAP$	Varies
$SUPPORTCAP$	Varies
$LOCALCAP$	Varies
$CIVCAP$	1
$BOMBMEMORY$	20
$SHOTMEMORY$	10
$LMBOMBPROB$	0.002
$LMFIREPROB$	Varies
$WATERFAIL$	Varies
$ELECFAIL$	Varies
$MAXSHOTS$	5000
$MAXBOMB$	100

Table 5.74: Scenario Five: General Parameters

The agent parameters stay constant throughout the different scenario varia-

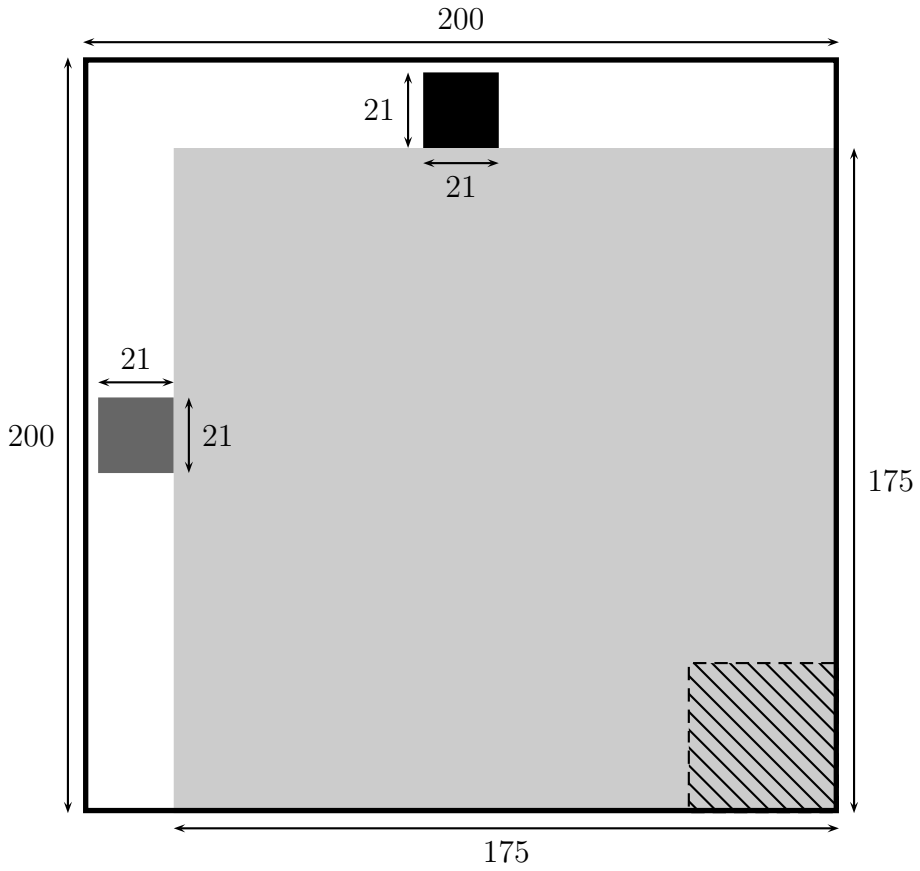

Figure 5.36: Initial Grid for Scenario Five

tions, they are shown in Table 5.75.

Parameter	Peace.	NGO	Ins.	Civ.
$squadNo$	1	2	3	4
$xHome$	99	14	112	112
$yHome$	14	99	112	112
$homeRadius$	10	10	87	87
$sensorRange$	200	200	50	50
$fireRange$	20	-	15	-
$bombRadius$	-	-	5	-
$sSKP$	0.10	-	0.05	-
Relationship to Peacekeepers	F	F	H	C
Relationship to NGOs	F	F	U	F
Relationship to Insurgents	H	U	F	U
Relationship to Civilians	F	F	F	F
W_1	0	0	0	0
W_2	0	0	0	0
W_3	0	0	0	0
W_4	0	0	0	0
W_5	0	0	0	0
W_6	0	0	0	0
W_7	0	0	0	0
W_8	0	0	0	0
W_9	0	0	0	0
W_{10}	0	0	0	0
W_{11}	0	0	0	0
W_{12}	0	0	0	0
W_{13}	0	0	0	0
W_{14}	0	-100	0	0
W_{15}	-50	0	0	0
W_{16}	0	0	0	0
W_{17}	0	0	0	0
W_{18}	0	0	0	0
W_{19}	0	0	0	0
W_{20}	0	0	0	0
W_{22}	10	10	-	-
W_{23}	10	10	-	-
W_{24}	10	10	-	-
W_{25}	0	-	0	-
$probFixWater$	1.00	1.00	-	-
$probFixElec$	1.00	1.00	-	-

Table 5.75: Scenario Five: Agent Parameters

5.4.1 Trial One

We start with a squad size of 25 for both the Peacekeepers and the NGOs and all the agents have the same capability of one. We set the Insurgents' firing probability at 0.01. Since this is higher than their bomb probability we would expect that most skirmishes would be started by a shot rather than a bomb. Since we are starting with a basic scenario there will be no water or electricity failures apart from the initial problem with the water supply in the final sector. The list of parameter values is given in Table 5.76.

Parameter	Value
$RUNTIME$	500
$PEACENO$	25
$SUPPORTNO$	25
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.01
$WATERFAIL$	0.00
$ELECFAIL$	0.00

Table 5.76: Scenario Five, Trial One: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	16 (64%)	3.12 (12.48%)
NGO	0 (0%)	11 (44%)	1.42 (5.68%)
Insurgent	0 (0%)	19 (38%)	7.56 (15.12%)
Civilian	0 (0%)	11 (1.10%)	2.40 (0.24%)

Table 5.77: Scenario Five, Trial One: Casualty Numbers

	Total	Mean per Run
Number of Failures	50	1
Number Fixed	10	0.20

Table 5.78: Scenario Five, Trial One: Utility Failure Numbers

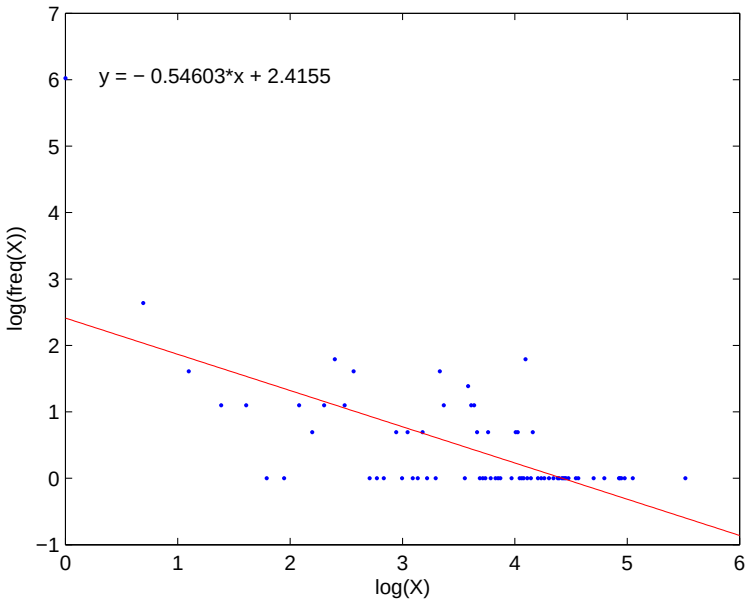

Figure 5.37: Scenario Five, Trial One: Avalanche Frequency-Size Distribution

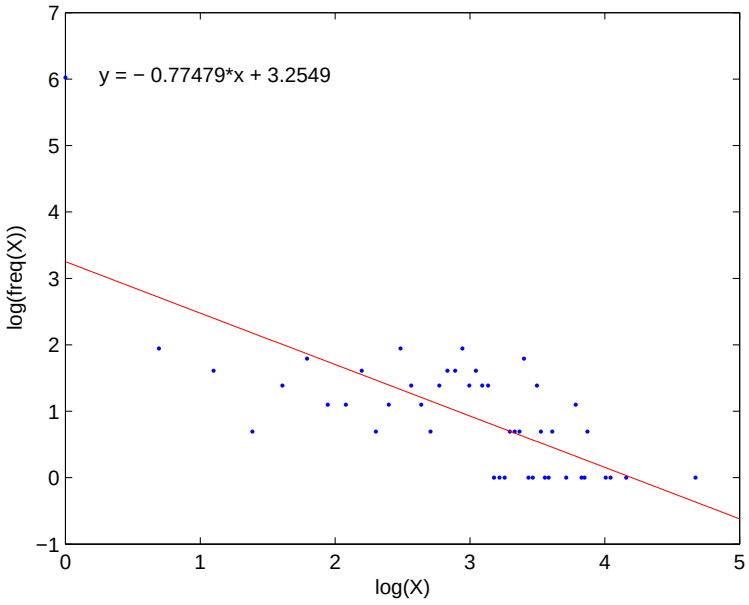

Figure 5.38: Scenario Five, Trial One: Time Avalanche Frequency-Size Distribution

5.4.2 Trial Two

We now add some extra complexity to the previous set of simulations by introducing the possibility of the water and electricity failing in each of the sectors. All the other parameters remain as they were for Scenario Five, Trial One and are given in Table 5.79.

Parameter	Value
$RUNTIME$	500
$PEACENO$	25
$SUPPORTNO$	25
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.01
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.79: Scenario Five, Trial Two: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	2.86 (11.44%)
NGO	0 (0%)	13 (52%)	1.98 (7.92%)
Insurgent	0 (0%)	19 (38%)	8.96 (17.92%)
Civilian	0 (0%)	14 (1.40%)	2.44 (0.24%)

Table 5.80: Scenario Five, Trial Two: Casualty Numbers

	Total	Mean per Run
Number of Failures	187	3.74
Number Fixed	14	0.28

Table 5.81: Scenario Five, Trial Two: Utility Failure Numbers

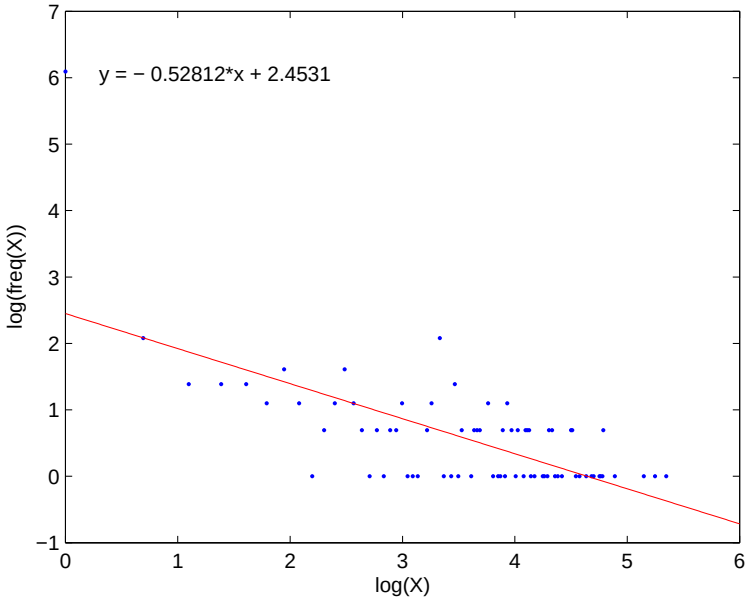

Figure 5.39: Scenario Five, Trial Two: Avalanche Frequency-Size Distribution

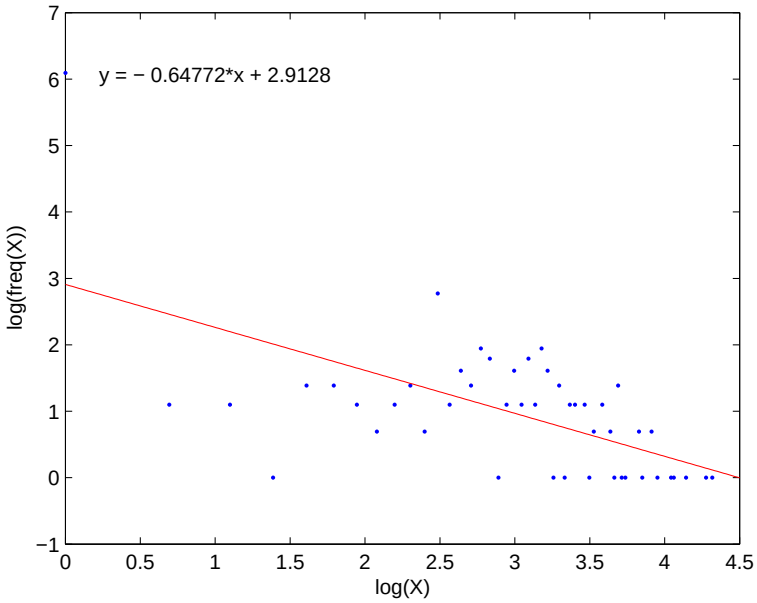

Figure 5.40: Scenario Five, Trial Two: Time Avalanche Frequency-Size Distribution

5.4.3 Trial Three

We now increase the level of conflict by increasing the shooting probability for the Insurgents. The other parameters remain as they were in the previous trial and are shown in Table 5.82.

Parameter	Value
$RUNTIME$	500
$PEACENO$	25
$SUPPORTNO$	25
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.82: Scenario Five, Trial Three: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	15 (60%)	4.30 (17.20%)
NGO	0 (0%)	14 (56%)	2.66 (10.64%)
Insurgent	1 (2%)	19 (38%)	8.56 (17.12%)
Civilian	0 (0%)	13 (1.30%)	2.74 (0.27%)

Table 5.83: Scenario Five, Trial Three: Casualty Numbers

	Total	Mean per Run
Number of Failures	172	3.44
Number Fixed	9	0.18

Table 5.84: Scenario Five, Trial Three: Utility Failure Numbers

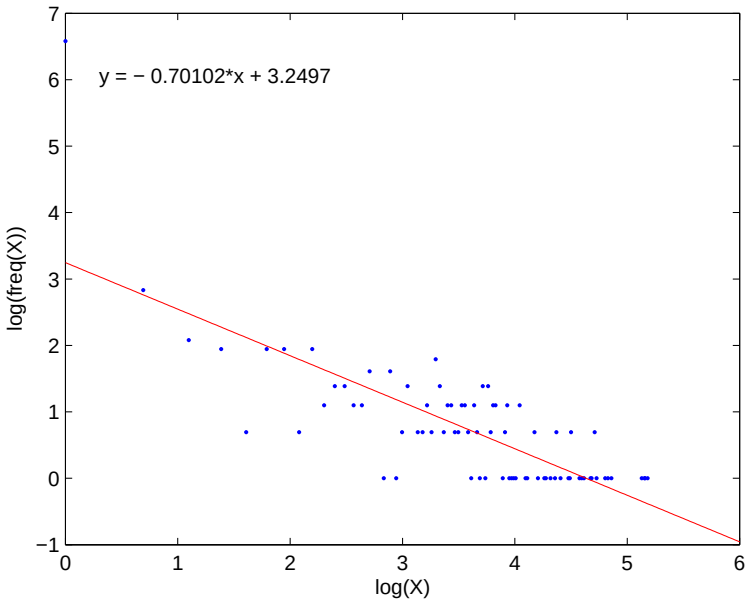

Figure 5.41: Scenario Five, Trial Three: Avalanche Frequency-Size Distribution

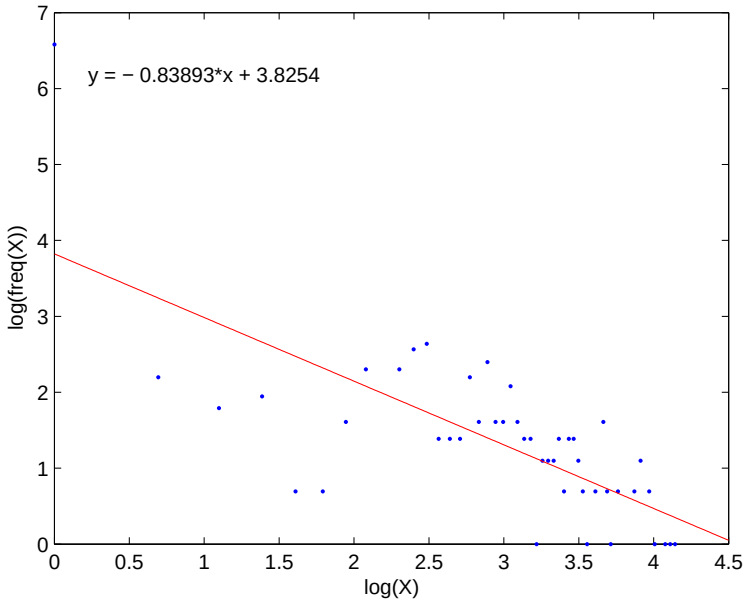

Figure 5.42: Scenario Five, Trial Three: Time Avalanche Frequency-Size Distribution

5.4.4 Trial Four

We now change the squad size for the Peacekeepers to see if having more Peacekeepers than NGOs keeps the casualty numbers down or increases the level of conflict and number of deaths. The parameters are given in Table 5.85.

Parameter	Value
$RUNTIME$	500
$PEACENO$	50
$SUPPORTNO$	25
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.85: Scenario Five, Trial Four: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	17 (38%)	4.84 (9.68%)
NGO	0 (0%)	25 (100%)	1.96 (7.84%)
Insurgent	0 (0%)	28 (56%)	12.84 (25.68%)
Civilian	0 (0%)	15 (0.15%)	2.18 (0.22%)

Table 5.86: Scenario Five, Trial Four: Casualty Numbers

	Total	Mean per Run
Number of Failures	189	3.78
Number Fixed	23	0.46

Table 5.87: Scenario Five, Trial Four: Utility Failure Numbers

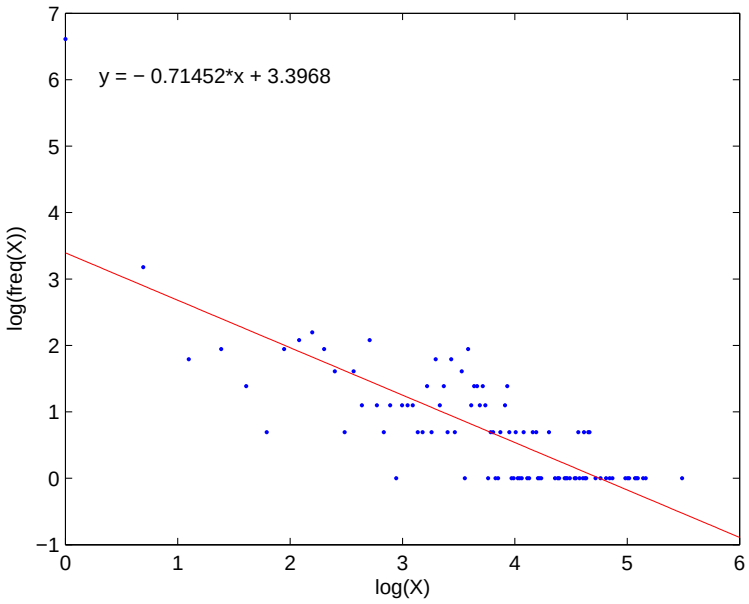

Figure 5.43: Scenario Five, Trial Four: Avalanche Frequency-Size Distribution

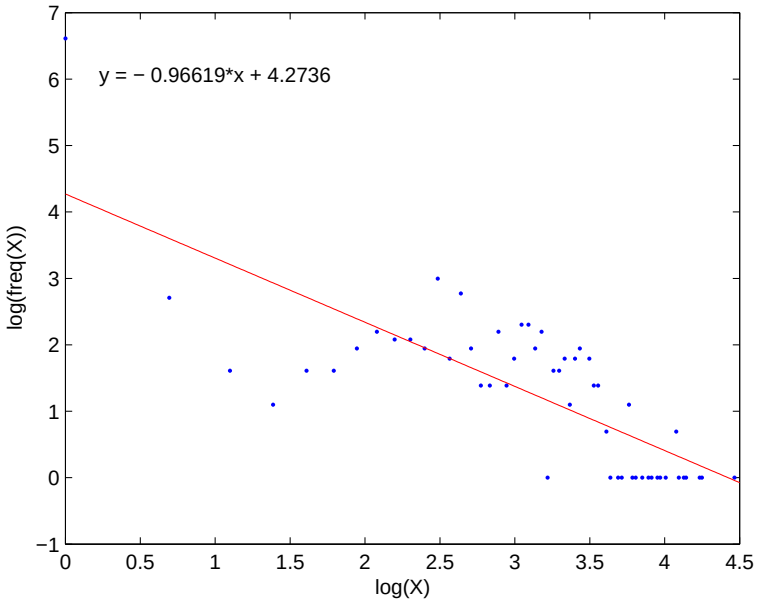

Figure 5.44: Scenario Five, Trial Four: Time Avalanche Frequency-Size Distribution

5.4.5 Trial Five

For this scenario variation we increase the number of NGOs so that the Peacekeeper and NGO squads are back to being of equal, but now increased, size. Again we are looking to see whether increasing the number of outside agents is a help or hindrance. The parameters are listed in Table 5.88.

Parameter	Value
$RUNTIME$	500
$PEACENO$	50
$SUPPORTNO$	50
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.88: Scenario Five, Trial Five: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	24 (48%)	5.96 (11.92%)
NGO	0 (0%)	25 (50%)	4.40 (8.80%)
Insurgent	4 (8%)	26 (52%)	12.08 (24.16%)
Civilian	0 (0%)	13 (1.30%)	3.20 (0.32%)

Table 5.89: Scenario Five, Trial Five: Casualty Numbers

	Total	Mean per Run
Number of Failures	189	3.78
Number Fixed	16	0.32

Table 5.90: Scenario Five, Trial Five: Utility Failure Numbers

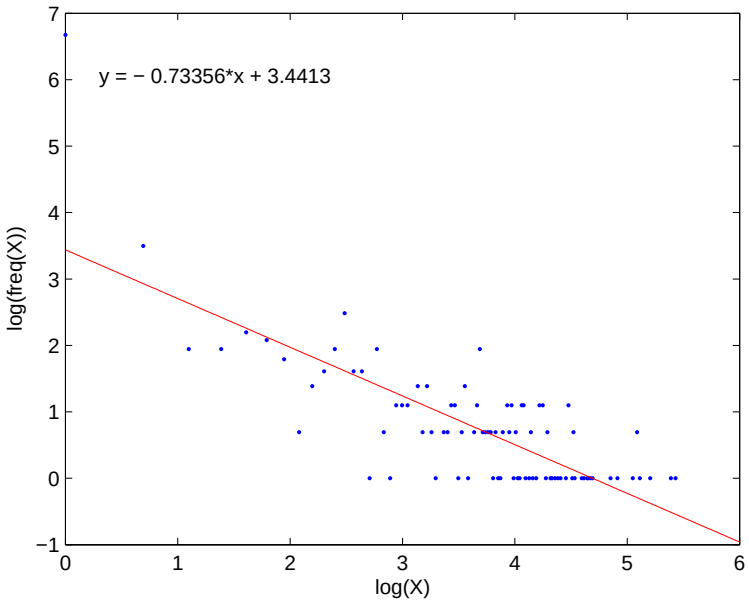

Figure 5.45: Scenario Five, Trial Five: Avalanche Frequency-Size Distribution

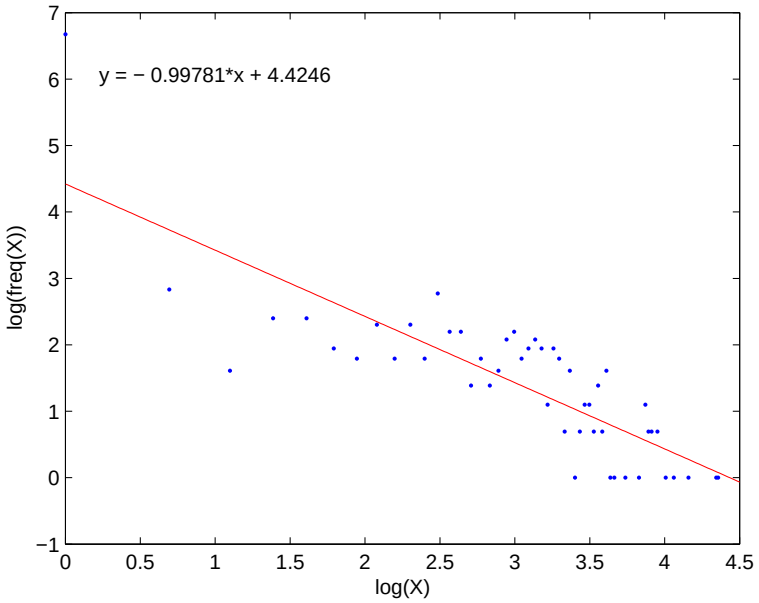

Figure 5.46: Scenario Five, Trial Five: Time Avalanche Frequency-Size Distribution

5.4.6 Trial Six

The parameter values for this trial are listed in Table 5.91. For this set of simulations we try another increase in the Peacekeeper force so that there are now more Peacekeepers than Insurgents in the model.

Parameter	Value
$RUNTIME$	500
$PEACENO$	100
$SUPPORTNO$	50
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.91: Scenario Five, Trial Six: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	31 (31%)	5.52 (5.52%)
NGO	0 (0%)	25 (50%)	2.96 (5.92%)
Insurgent	5 (10%)	25 (50%)	14.66 (29.32%)
Civilian	0 (0%)	12 (1.20%)	1.94 (0.19%)

Table 5.92: Scenario Five, Trial Six: Casualty Numbers

	Total	Mean per Run
Number of Failures	204	4.08
Number Fixed	17	0.34

Table 5.93: Scenario Five, Trial Six: Utility Failure Numbers

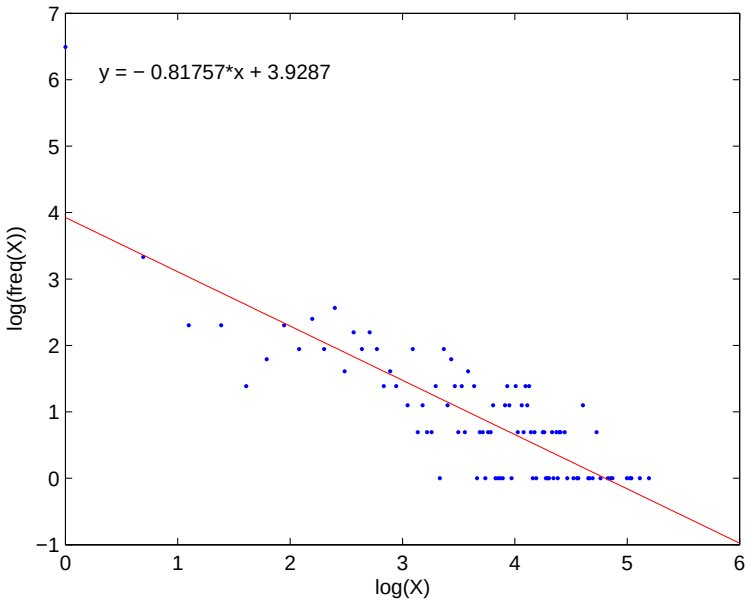

Figure 5.47: Scenario Five, Trial Six: Avalanche Frequency-Size Distribution

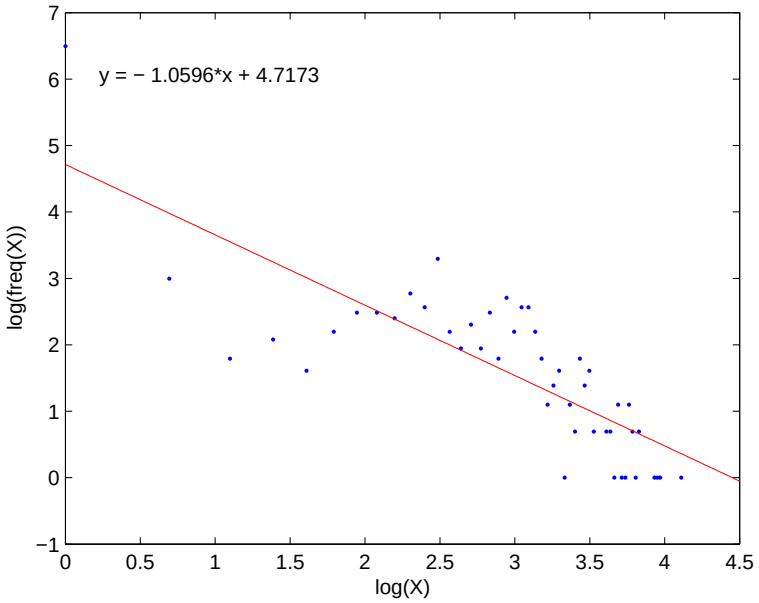

Figure 5.48: Scenario Five, Trial Six: Time Avalanche Frequency-Size Distribution

5.4.7 Trial Seven

Trial Seven is the same as Trial Five but we have increased the model run time to 1000 timesteps. This is to see if we are missing any crucial behaviour by only running the model for 500 timesteps for the majority of the scenarios. The parameters are shown in Table 5.94.

Parameter	Value
$RUNTIME$	1000
$PEACENO$	50
$SUPPORTNO$	50
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.94: Scenario Five, Trial Seven: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	21 (42%)	6.92 (13.84%)
NGO	0 (0%)	29 (58%)	4.40 (8.80%)
Insurgent	4 (8%)	32 (64%)	14.08 (28.16%)
Civilian	0 (0%)	13 (1.30%)	2.88 (0.29%)

Table 5.95: Scenario Five, Trial Seven: Casualty Numbers

	Total	Mean per Run
Number of Failures	324	6.48
Number Fixed	20	0.40

Table 5.96: Scenario Five, Trial Seven: Utility Failure Numbers

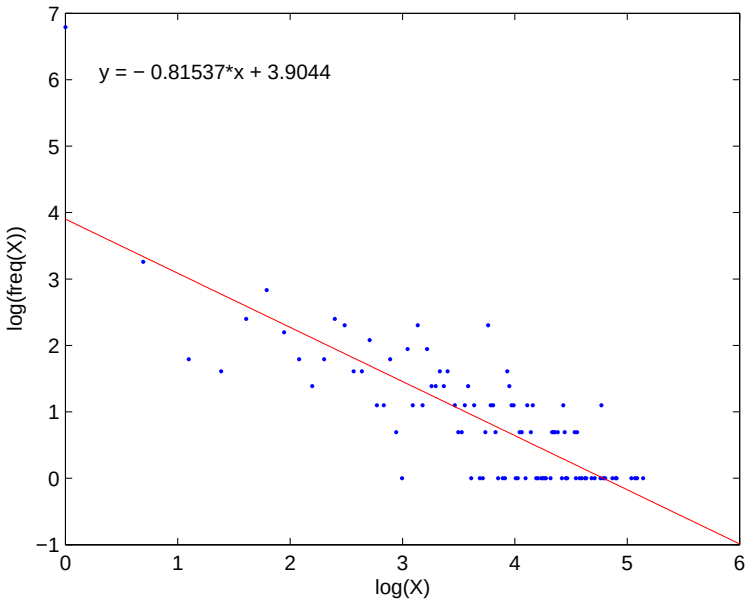

Figure 5.49: Scenario Five, Trial Seven: Avalanche Frequency-Size Distribution

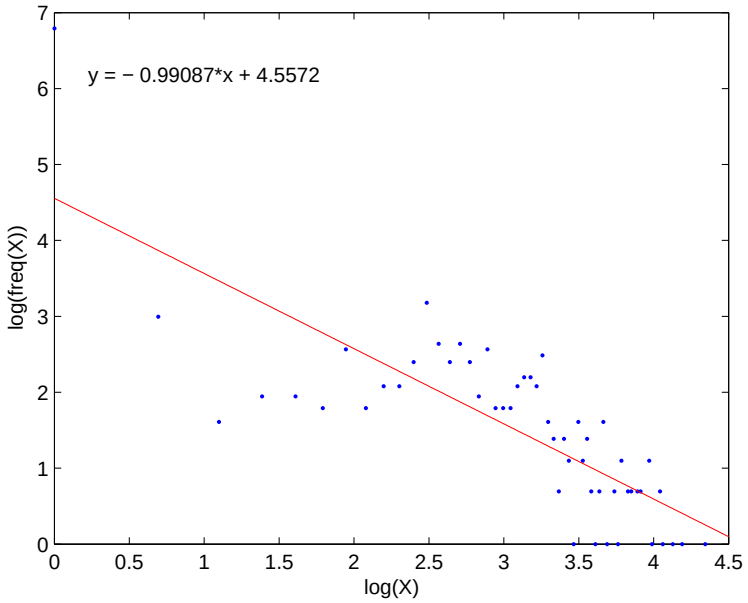

Figure 5.50: Scenario Five, Trial Seven: Time Avalanche Frequency-Size Distribution

5.4.8 Trial Eight

The parameters for this batch of runs are shown in Table 5.97. Again we take a previous trial, this time Trial Three, and increase the model run time to 1000 timesteps to see if our usual run time of 500 timesteps is adequate.

Parameter	Value
$RUNTIME$	1000
$PEACENO$	25
$SUPPORTNO$	25
$PEACECAP$	1
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.97: Scenario Five, Trial Eight: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	14 (56%)	4.72 (18.88%)
NGO	0 (0%)	17 (68%)	3.60 (14.40%)
Insurgent	3 (6%)	25 (50%)	12.20 (24.40%)
Civilian	0 (0%)	21 (2.10%)	2.84 (0.28%)

Table 5.98: Scenario Five, Trial Eight: Casualty Numbers

	Total	Mean per Run
Number of Failures	315	6.30
Number Fixed	16	0.32

Table 5.99: Scenario Five, Trial Eight: Utility Failure Numbers

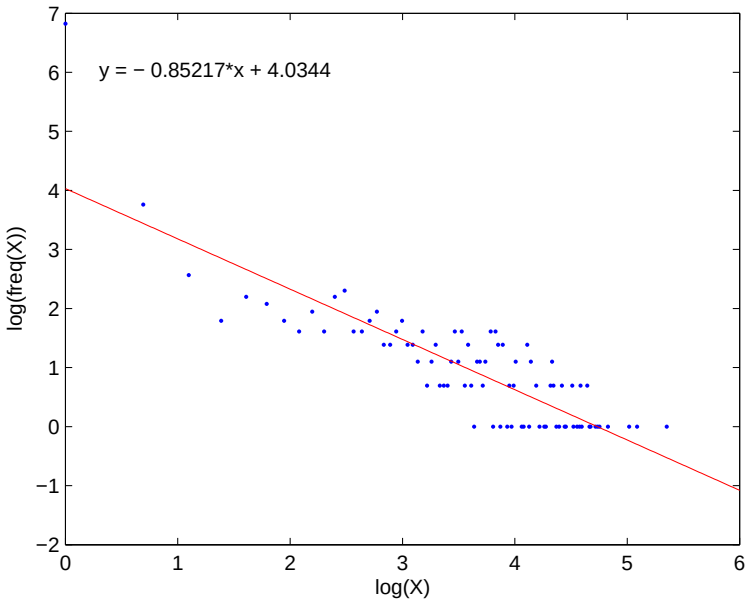

Figure 5.51: Scenario Five, Trial Eight: Avalanche Frequency-Size Distribution

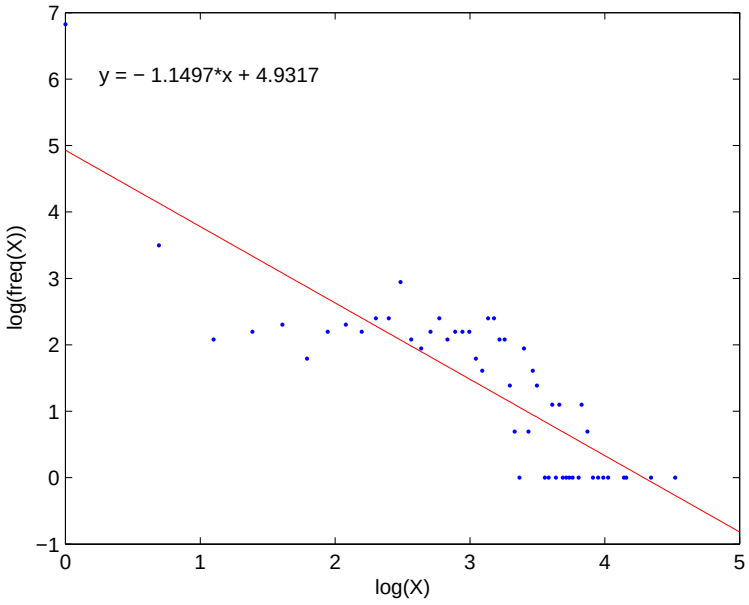

Figure 5.52: Scenario Five, Trial Eight: Time Avalanche Frequency-Size Distribution

5.4.9 Trial Nine

Now we look at what happens when we alter the agent capabilities. For this trial we give the Peacekeepers a capability of two with all the other agents remaining at capability one. We go back to our usual model run time of 500 timesteps and also set the NGO and Peacekeeper squads back to having 25 agents. The parameters are listed in Table 5.100.

Parameter	Value
$RUNTIME$	500
$PEACENO$	25
$SUPPORTNO$	25
$PEACECAP$	2
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.100: Scenario Five, Trial Nine: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	9 (36%)	2.20 (8.80%)
NGO	0 (0%)	13 (52%)	2.34 (9.36%)
Insurgent	0 (0%)	19 (38%)	7.62 (15.24%)
Civilian	0 (0%)	13 (1.30%)	2.30 (0.23%)

Table 5.101: Scenario Five, Trial Nine: Casualty Numbers

	Total	Mean per Run
Number of Failures	185	3.70
Number Fixed	7	0.14

Table 5.102: Scenario Five, Trial Nine: Utility Failure Numbers

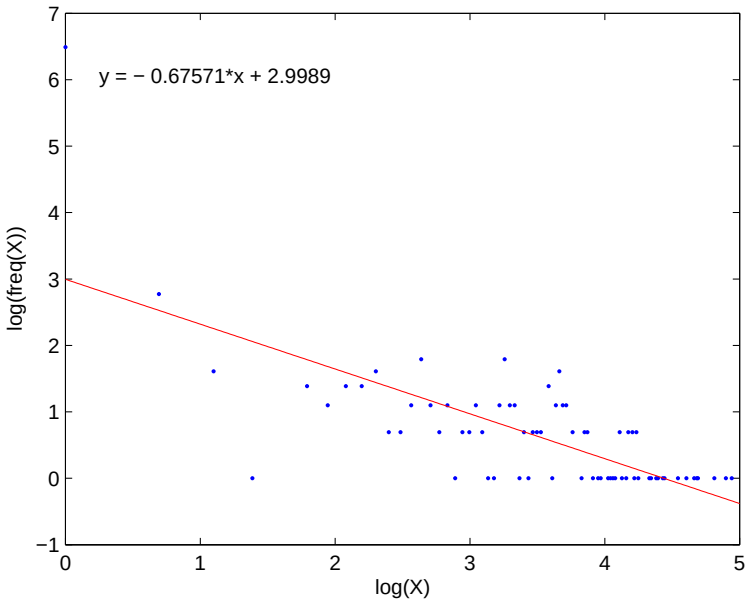

Figure 5.53: Scenario Five, Trial Nine: Avalanche Frequency-Size Distribution

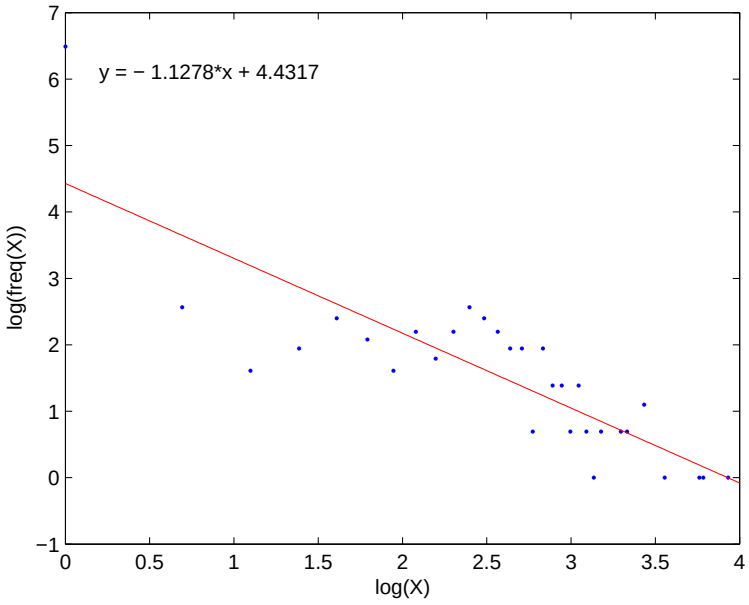

Figure 5.54: Scenario Five, Trial Nine: Time Avalanche Frequency-Size Distribution

5.4.10 Trial Ten

We now stay with the increased Peacekeeper capability and increase the squad size so that we have 50 agents. The parameters are shown in Table 5.103.

Parameter	Value
$RUNTIME$	500
$PEACENO$	50
$SUPPORTNO$	25
$PEACECAP$	2
$SUPPORTCAP$	1
$LOCALCAP$	1
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.103: Scenario Five, Trial Ten: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	35 (70%)	4.68 (9.36%)
NGO	0 (0%)	10 (40%)	1.76 (7.04%)
Insurgent	1 (2%)	21 (42%)	10.62 (21.24%)
Civilian	0 (0%)	7 (0.70%)	0.96 (0.10%)

Table 5.104: Scenario Five, Trial Ten: Casualty Numbers

	Total	Mean per Run
Number of Failures	214	4.28
Number Fixed	19	0.38

Table 5.105: Scenario Five, Trial Ten: Utility Failure Numbers

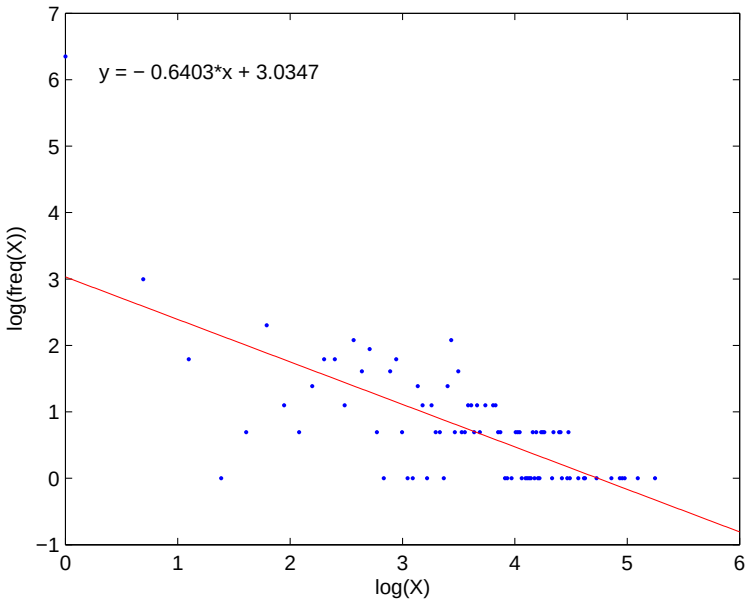

Figure 5.55: Scenario Five, Trial Ten: Avalanche Frequency-Size Distribution

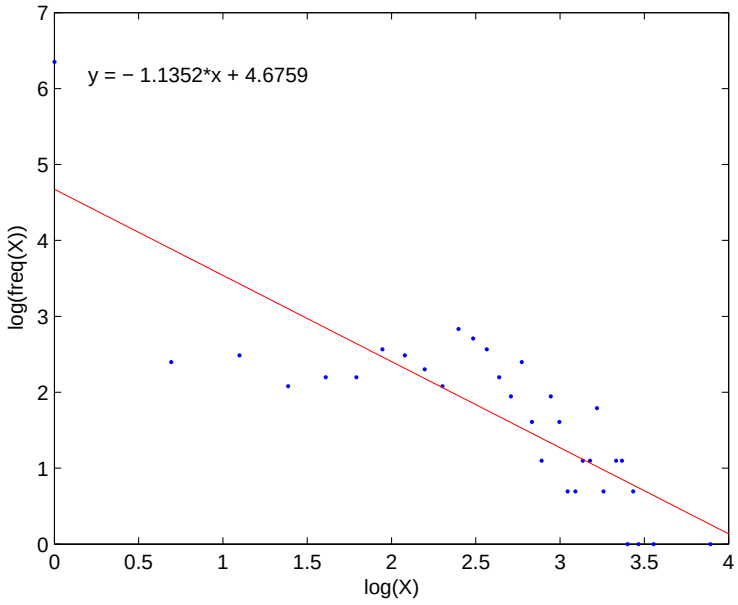

Figure 5.56: Scenario Five, Trial Ten: Time Avalanche Frequency-Size Distribution

5.4.11 Trial Eleven

For our final set of simulations we alter the Peacekeeper capability to four, the Insurgent and NGO capability to two and leave the Civilian capability at one. The parameters are given in Table 5.106.

Parameter	Value
$RUNTIME$	500
$PEACENO$	50
$SUPPORTNO$	25
$PEACECAP$	4
$SUPPORTCAP$	2
$LOCALCAP$	2
$LMFIREPROB$	0.02
$WATERFAIL$	0.0001
$ELECFAIL$	0.0001

Table 5.106: Scenario Five, Trial Eleven: Parameters

	Minimum	Maximum	Mean
Peacekeeper	0 (0%)	17 (34%)	3.72 (7.44%)
NGO	0 (0%)	11 (44%)	1.56 (6.24%)
Insurgent	1 (2%)	17 (34%)	8.58 (17.16%)
Civilian	0 (0%)	15 (1.50%)	2.08 (0.21%)

Table 5.107: Scenario Five, Trial Eleven: Casualty Numbers

	Total	Mean per Run
Number of Failures	198	3.96
Number Fixed	8	0.16

Table 5.108: Scenario Five, Trial Eleven: Utility Failure Numbers

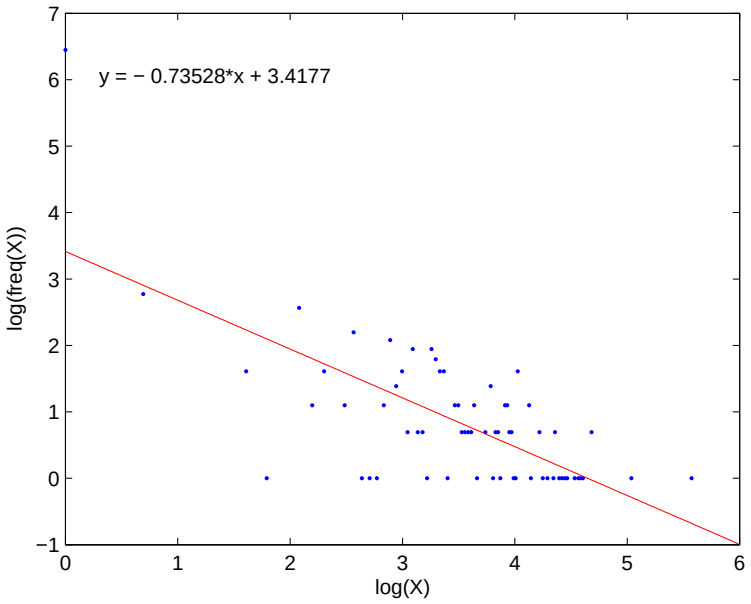

Figure 5.57: Scenario Five, Trial Eleven: Avalanche Frequency-Size Distribution

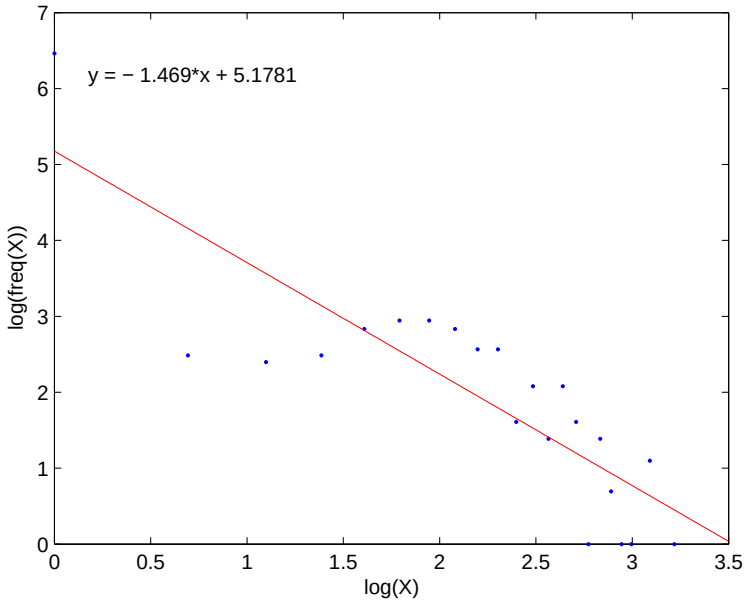

Figure 5.58: Scenario Five, Trial Eleven: Time Avalanche Frequency-Size Distribution

5.4.12 Conclusions

First we look at the graphs for the avalanche and time avalanche distributions. With this set of scenarios being the most complicated it gives the greatest opportunity to find evidence of SOC. In general, the plots do appear to be closer to the best fit line than those for Scenarios Two and Three but they still cannot be said for sure to approximate a line. It also appears that the most complicated scenarios, Trials Nine, Ten and Eleven where we changed the squad capabilities, have some of the worst fit avalanche plots. This suggests that varying agent capability further would not help if we want to find SOC using this particular measure; we would have to add more complexity to the actual scenario, for example by adding in additional Insurgent squads. In contrast, when we look at the time avalanche plots for Trials Nine, Ten and Eleven we see a much better fit to the straight line. In particular, Figure 5.58 shows a fair approximation to the line. This suggests that this measure may be more appropriate than the avalanche one when looking for SOC behaviour. In Section 5.5 we explore this further by looking at ANOVA analysis of the time avalanche plots for Trials Nine, Ten and Eleven.

We now look at how successful the Peacekeepers and NGOs were at fixing any faults with the water and electricity supply. Comparing Trial Three and Trial Four we can see that simply doubling the Peacekeeper squad size, and leaving the NGO squad constant, leads to the number of sectors fixed more than doubling. The mean casualty figure for the Peacekeeper squad stays fairly constant. However, when we then doubled the NGO squad in Trial Five so that it was equal to the Peacekeeper one, we ended up with fewer repairs and higher casualty numbers for both the NGOs and the Peacekeepers. Doubling the Peacekeeper squad again in Trial Six reduced the mean NGO casualty number, it did not significantly affect the mean Peacekeeper casualty figure or the number of repairs. Doubling the run time for the model in Trials Seven and Eight does not seem to significantly affect the number of repairs made or the casualty numbers. This suggests that

the shorter run time was adequate to examine the full behaviour of the model. Altering the various squad capabilities does not seem to have much effect in Trial Ten, with similar repair and casualty numbers to previous runs, but reduces both the number of repairs and the casualty numbers in Trials Nine and Eleven. This seems strange since we would expect the Peacekeepers to be able to conduct more repairs with a higher capability.

5.5 ANOVA Analysis

Throughout this chapter we have plotted the avalanche and time avalanche data along with a straight line of best fit, and for the majority of cases it was obvious that the data did not approximate a straight line. After completing this analysis we looked at ANOVA results, for powers up to cubic, in those graphs that showed a possibility of power law behaviour. We used a spreadsheet polynomial regression model that used orthogonal polynomials for our analysis; this had been given to us by Professor Russell Cheng from Southampton University. The results we were looking at were those for time avalanches from Scenario Five Trials Nine, Ten and Eleven. The results obtained are given below.

5.5.1 Scenario Five, Trial Nine

Table 5.109 gives the ANOVA results. The R square value for the model was calculated to be 0.93, this along with the low $p-$value shows that the model is significant.

	Sum of Squares	Degrees of Freedom	Mean Square	F	p
Mean	72.16	1	72.16	235.38	$7.45x10^{-15}$
Regression	38.53	3	12.84	41.89	$2.71x10^{-10}$
Residual	8.28	27	0.31		
Total	118.67	31			

Table 5.109: ANOVA Table for Scenario Five, Trial Nine

Table 5.110 gives the regression results that show the significant terms in the model. Here X_0 relates to the constant term, X_1 the linear, X_2 the quadratic and X_3 the cubic.

We can see from the low $p-$value for the cubic coefficient that we cannot dismiss this term. We therefore cannot say that the data approximates a straight line.

Coefficient	β	Standard Error of β	t	p
X_0	8.49	0.55	15.34	$7.45x10^{-15}$
X_1	-5.69	0.55	-10.27	$8.01x10^{-11}$
X_2	0.61	0.55	1.10	0.28
X_3	-2.42	0.55	-4.36	$1.69x10^{-4}$

Table 5.110: Regression Coefficients for Scenario Five, Trial Nine

5.5.2 Scenario Five, Trial Ten

The ANOVA and regression results for Trial Ten are given in Tables 5.111 and 5.112 respectively.

	Sum of Squares	Degrees of Freedom	Mean Square	F	p
Mean	98.48	1	98.48	372.28	$4.34x10^{-18}$
Regression	38.90	3	12.97	49.01	$1.78x10^{-11}$
Residual	7.67	29	0.26		
Total	145.05	33			

Table 5.111: ANOVA Table for Scenario Five, Trial Ten

Coefficient	β	Standard Error of β	t	p
X_0	9.92	0.51	19.29	$4.34x10^{-18}$
X_1	-5.66	0.51	-11.01	$7.11x10^{-12}$
X_2	-0.06	0.51	-0.11	0.92
X_3	-2.61	0.51	-5.07	$2.04x10^{-5}$

Table 5.112: Regression Coefficients for Scenario Five, Trial Ten

The R square value for the ANOVA was 0.95. Since the $p-$value was very small this indicates that the model is significant. Looking at the regression results we again see that we have a low $p-$values for the cubic term so we cannot discount this factor.

5.5.3 Scenario Five, Trial Eleven

The ANOVA results are given in Table 5.113. The R square value was calculated to be 0.95. Since the $p-$value is low we can conclude that the model is significant.

	Sum of Squares	Degrees of Freedom	Mean Square	F	p
Mean	81.94	1	81.94	230.41	$1.06x10^{-11}$
Regression	37.44	3	12.48	35.09	$9.90x10^{-8}$
Residual	6.40	18	0.36		
Total	125.78	22			

Table 5.113: ANOVA Table for Scenario Five, Trial Eleven

The regression results are given in Table 5.114. As with the previous two Trials, the results show that we cannot dismiss the cubic factor and therefore cannot say that the data fits a straight line.

Coefficient	β	Standard Error of β	t	p
X_0	9.05	0.60	15.18	$1.06x10^{-11}$
X_1	-5.60	0.60	-9.39	$2.35x10^{-8}$
X_2	-0.11	0.60	-0.19	0.86
X_3	-2.47	0.60	-4.14	$6.13x10^{-4}$

Table 5.114: Regression Coefficients for Scenario Five, Trial Eleven

5.5.4 Conclusions

In all three cases we have found that we cannot say the data approximates a straight line since regression results show that we cannot discount the cubic factor. This means that we cannot say that we have any evidence of a power law relationship. Since this is a necessary condition for self-organised criticality, the time avalanche data cannot be said to exhibit this behaviour.

Further discussion of all the results from the experiments is given in Chapter 6.

Chapter 6

CONCLUSIONS

We have successfully developed a working agent-based model for the representation of peace support operations. We have devised scenarios to show the effects of failing water and electricity supplies and the escalating violence resulting from attacks by Insurgents. The aim for the model is that it can be developed further so that it can be used in conjunction with the DIAMOND model and can provide the detail that DIAMOND lacks. Suggested improvements to help achieve this aim are given in Chapter 7 along with further analysis and experiments that could be undertaken.

There is insufficient evidence to say for sure whether or not the model is capable of exhibiting power law behaviour, and further to that self-organised criticality, (SOC). For the majority of the 'avalanche' and 'time avalanche' plots we can conclude that there is no evidence of a power law relationship, and hence no SOC. The 'time avalanche' plot for Scenario Five, Trial Eleven, reproduced here in Figure 6.1, was the closest approximation to a straight line but ANOVA results showed we could not discount a cubic model.

It may be that the scenarios we used were not complex enough to show SOC behaviour. We started in Scenario Two with a simple set-up then increased the complexity as we progressed through the different experiments. In Scenario Three

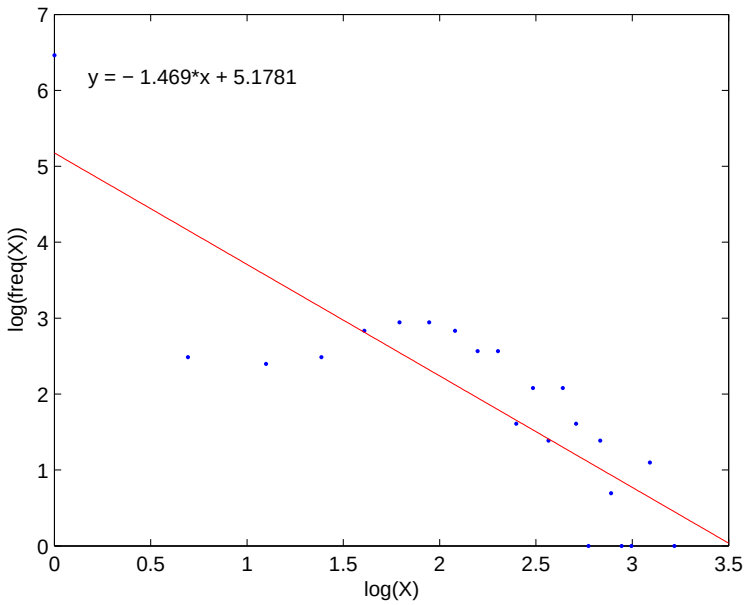

Figure 6.1: Scenario Five, Trial Eleven: Time Avalanche Frequency-Size Distribution

we introduced NGOs, which had not been present in Scenario Two. Scenario Four saw the introduction of Civilian movement and Insurgent bomb attacks. Finally in Scenario Five we reinstated Insurgent attacks by gunfire as well as suicide bomb attacks, in the final runs of this scenario we also altered agent capabilities so they were not necessarily equal. The 'avalanche' plots became more spread out once we introduced the change in agent capabilities so this may not be the best way to find SOC using this factor. In contrast, the 'time avalanche' plot in Figure 6.1 resulted from the most complex scenario in terms of agent capability, Scenario Five, Trial Eleven, so altering this further could lead to evidence of power law behaviour and SOC.

If the model could indeed show SOC behaviour it would suggest that the Peacekeepers do not have control of the grid. Self-organised criticality represents a situation where, in effect, anything could happen at a given timestep. If we relate this to our 'time avalanches' this means that the next skirmish started by an Insurgent could be anything from just one shot to a conflict that lasts until the end of the model run.

Aside from the question of SOC, we also looked at how the squad sizes for the Peacekeepers and NGOs affected their ability to fix water and electricity supplies,a long with casualty numbers. Surprisingly, it was not always better to have a larger squad size. For example, in Scenario Three when there was only the initial water failure, doubling the squad sizes to a size of 50 had little effect. However, this changed when there were multiple failures. Here increasing the squad sizes from 25 to 50 significantly increased the fix rate without having much affect on the casualty numbers. Further increase to a squad size of 75 did not have so much effect on the fix rate, but the casualty numbers did rise. This suggests that there is an optimum squad size after which any squad increases do not increase the effectiveness of the Peacekeepers and NGOs.

Chapter 7

FURTHER WORK

In this final chapter we shall be suggesting future directions for the research. There are two main strands to this: further developments to the model and additional experiments and analysis.

7.1 Suggested Model Improvements

When describing the design process for our agent-based model in Chapter 4, we noted some factors we were unable to include due to the limited time available, or in one case due to memory limitations. These possible developments are listed here along with the reasons why they would improve the model.

- The most important improvement would be to increase the possible grid size. At the moment we are limited to around 200×200 cells. If we wish to use the model in conjunction with DIAMOND we would want a grid size of at least 1000×1000 cell to model events at one node adequately. An experienced programmer should be able to carry out this improvement by identifying and correcting parts of the code that are inefficient.

- The Insurgents in the model are easily identified by the Peacekeepers and NGOs as such. It would be more realistic to assume that the Insurgents

would blend in with the Civilians until they identified themselves by engaging in combat. This could be programmed into the model so that the Insurgents appeared as Civilians until they fired at enemy agents. Their parameters would remain unchanged throughout the model run but the way the other agents react to them should change.

- At present the model can only be programmed to include one squad of each agent type. In many peacekeeping scenarios we would expect there to be rival sets of local militia and civilians. This is another possible source of conflict and could add extra complexity to the scenarios.

- We mentioned psychological factors in Chapter 4. We had planned to include a cell measure for tension, a squad measure for acceptance of the peacekeeping force by the local agents, and an agent measure of fear for the Civilians. Acceptance of the peacekeeping force could take into account, for example, the amount of violence, the effectiveness of the Peacekeepers and NGOs regarding the fixing of essential supplies and the availability of food. This factor could then influence the relationships between the Peacekeepers and the Insurgent and Civilian squads, and would therefore affect the amount of conflict in the model. The method we developed to measure tension at a DIAMOND node could be adapted to fit this model and then modified as appropriate. The Civilian fear factor could be, for example, affected by the violence within their sensor range, weighted by how long ago it occurred, along with the overall number of Civilian casualties. The tension and fear factors can be recorded as the model run progresses and analysed to discover how well the Peacekeepers are controlling the area.

- At present the relationships between the squads are set at the beginning of the model run and remain constant throughout. This is fine for small timescale scenarios, such as the ones we experimented with, but if we wish

to model a long period of time we would expect there to be some changes in relationship.

- Another consideration we would have to make if we wished to model larger timescales is representing civilian routines. That is, in a peacekeeping scenario we would expect the civilians to be able to carry on with their normal daily lives, so they would be travelling to work in the morning then back home in the evening.

- The agents can only be killed by gun fire or suicide bombs. If long time periods are to be modelled we would expect the availability of food and water to become a more important factor, and if the Civilians were without either for a set time then they could die of starvation.

- We would expect the combat functions, and possibly the movement functions, to be developed further. We noted in Chapter 4 that improvements could be made to the firing functions. If the Peacekeeper or Insurgent finds that the target cell they have chosen at random does not contain a valid target agent, then they should go back to the possible cells and pick another one at random rather than changing their action to 'move'. If none of the possible target cells contain a valid target agent then they should change their action to 'move'. This change is illustrated in the adapted firing function in Figure 7.1.

- At present, the agents in the model have only two states: alive or dead. An injured state could also be introduced which could lead to modified behaviour for the agents.

- For ease of use, it would be helpful if the user could input all the data to file which could then be read into the model. This would be better than the current situation where the source files have to be edited and the

whole program re-compiled when any changes are made to the scenario parameters.

- The graphics could be improved so that they were programmed within the model and could show the model run as it happens. The simple MATLAB script we programmed only shows the agent positions. It would be helpful if the shots and bomb attacks were shown along with an indicator for when the water or electricity fails or for when the food runs out.

- The final improvement would be to link the model to DIAMOND when it is capable of modelling events at a node, data could then be shared between the two models. This was the ultimate aim of our research and it is hoped that we have provided a good initial development model.

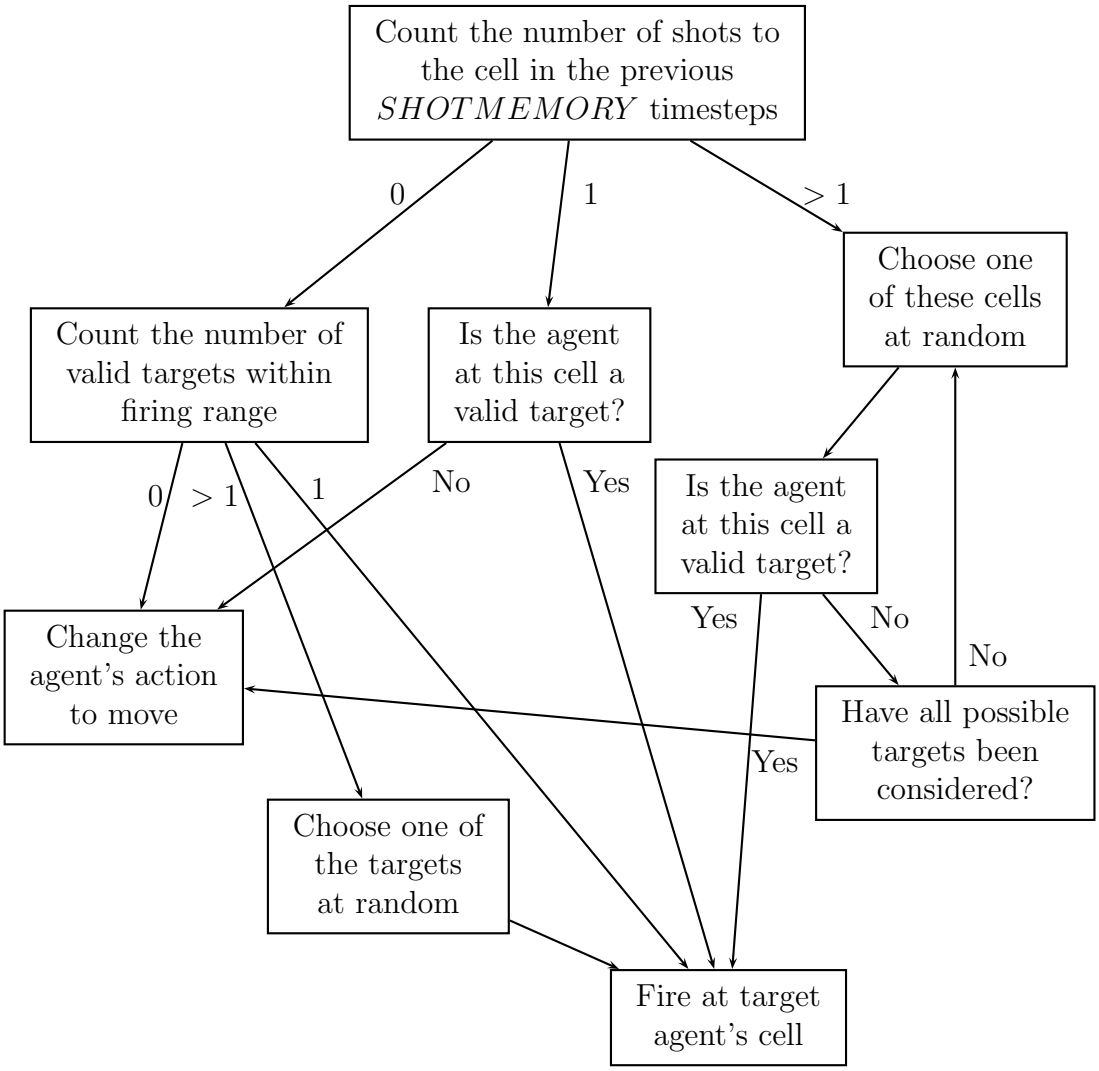

Figure 7.1: Adapted Firing Function

7.2 Additional Experiments and Analysis

Here we describe some additional analysis that could be done using the data we have already generated, along with some suggested further experiments.

A large amount of data is generated by the model, we did not have time to fully analyse all the output from our experiments. In particular, we did not use the civilians in need and combat indicator outputs at all. If more time had been available we would have analysed these files to see how the portion of the grid where these indicators were flagged developed as the model runs progressed. In a peace support scenario we would want the grid to be under control by the Peacekeepers. If the percentages had stayed constant, or decreased, this objective would have been met. An increase in the percentages would suggest the peacekeepers are unable to control the Insurgents, and that the Peacekeepers and NGOs are unable to help the Civilians adequately.

The clustering of agents has been shown to be an important indicator of complex behaviour. We wrote a MATLAB script to calculate the box-counting dimension of the cluster of Peacekeeper agents at each timestep using the grid positions output giving a time series of length $RUNTIME + 1$ for each model run. The box-counting dimension is an approximation to fractal dimension. This MATLAB program is given in Appendix E.3. Due to the limited time available, and the large number of model runs that were completed, we were unable to analyse the results.

The 'avalanches' we looked at in the model were related to the spread of conflict, this seemed the obvious choice. It would be useful to identify other factors that could be thought of as 'avalanches' that are not related to combat and could be applied in a non-violent scenario. This could be, for example, Civilians that are displaced due to lack of water.

The scenarios we looked at featured a fairly high level of conflict, although the aim for the Peacekeepers and NGOs was to fix the water and electricity supplies.

Further experiments could focus more on the peacekeeping aspects of the mode by setting lower probabilities for Insurgent violence.

Appendix A

TENSION CALCULATIONS IN DIAMOND: FINAL METHOD

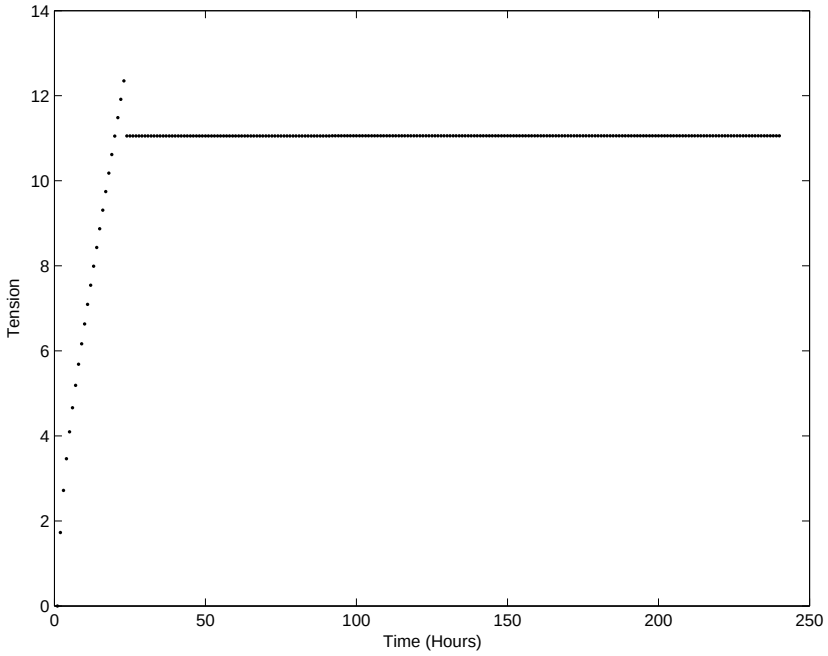

Figure A.1: Tension at Srbac (10) Using Method 4.3

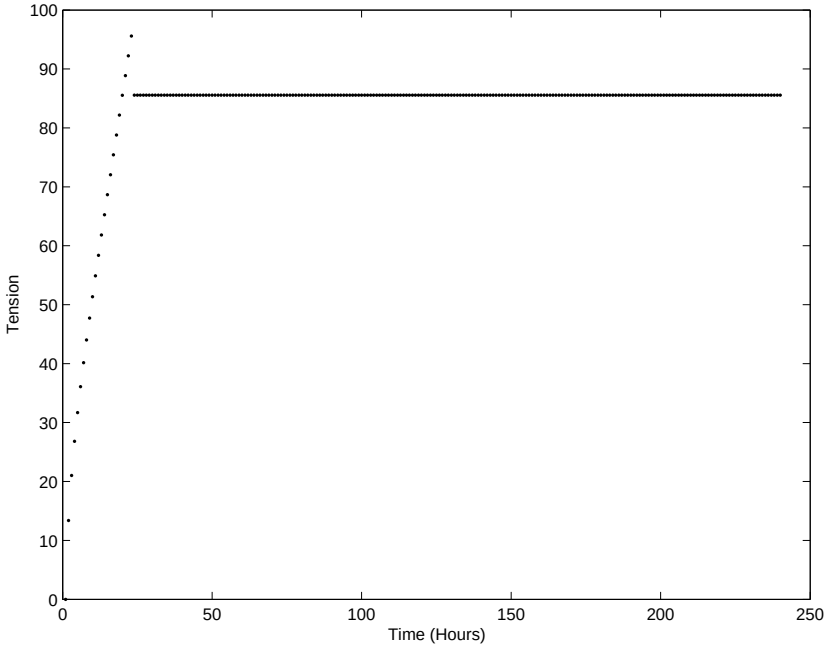

Figure A.2: Tension at Derventa (12) Using Method 4.3

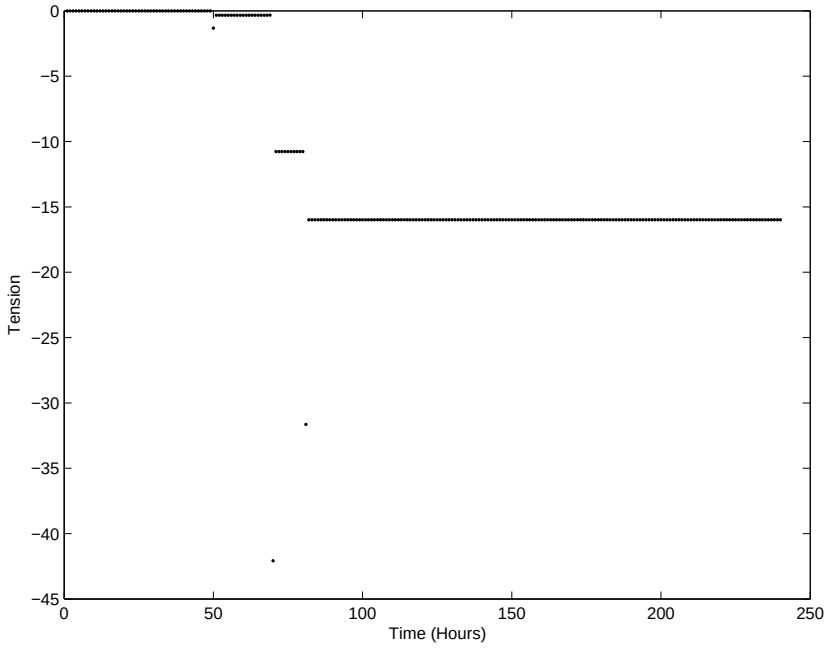

Figure A.3: Tension at Odzak (14) Using Method 4.3

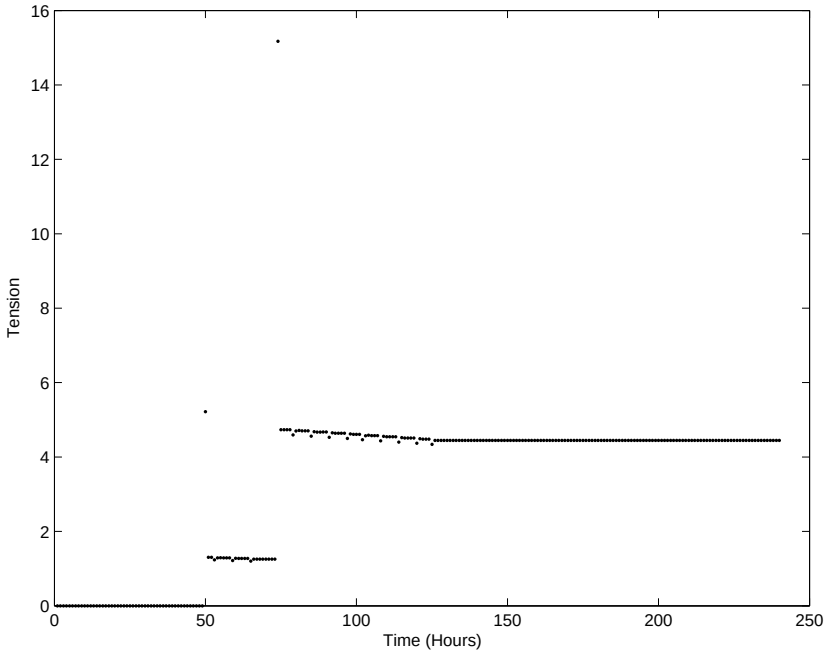

Figure A.4: Tension at Gradacac (18) Using Method 4.3

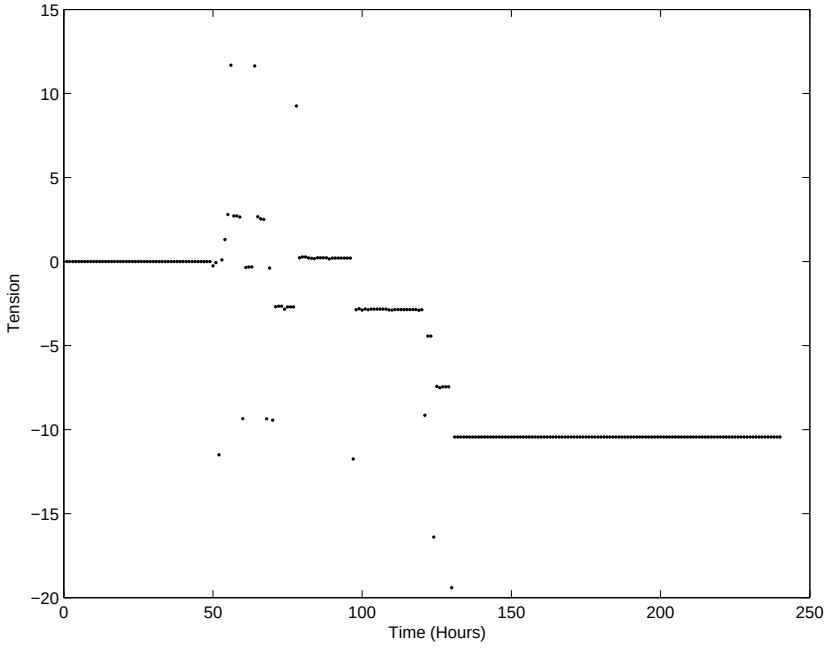

Figure A.5: Tension at Brcko (19) Using Method 4.3

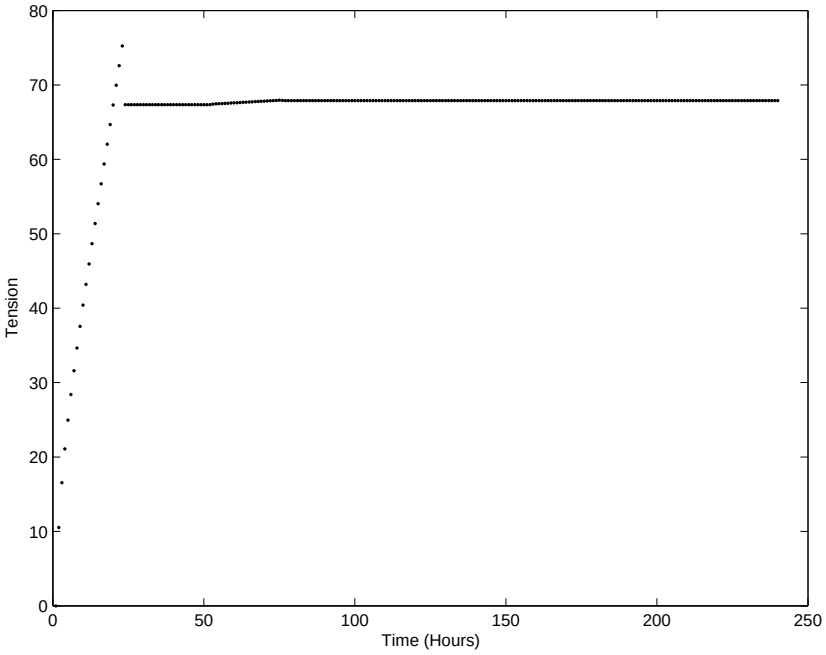

Figure A.6: Tension at Banja Luca (23) Using Method 4.3

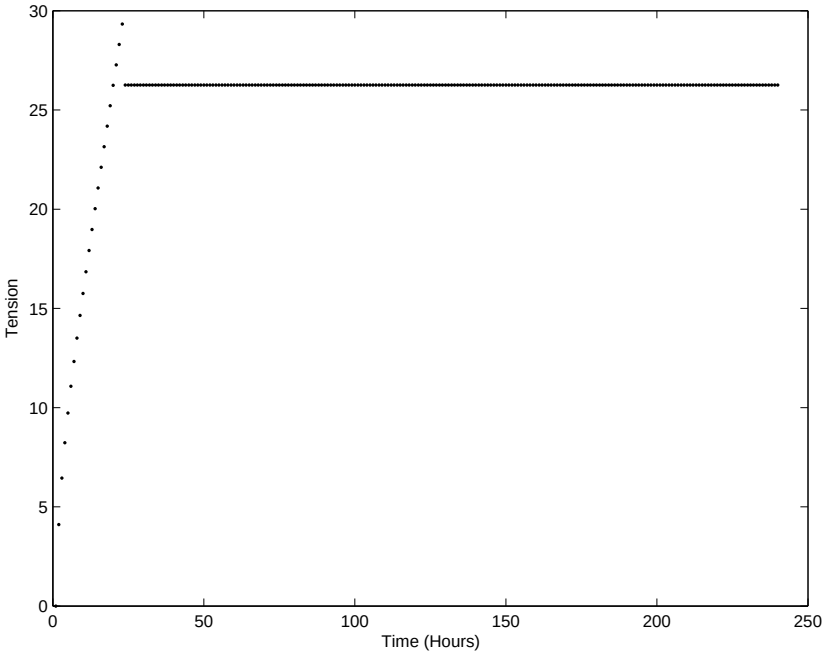

Figure A.7: Tension at Doboj (25) Using Method 4.3

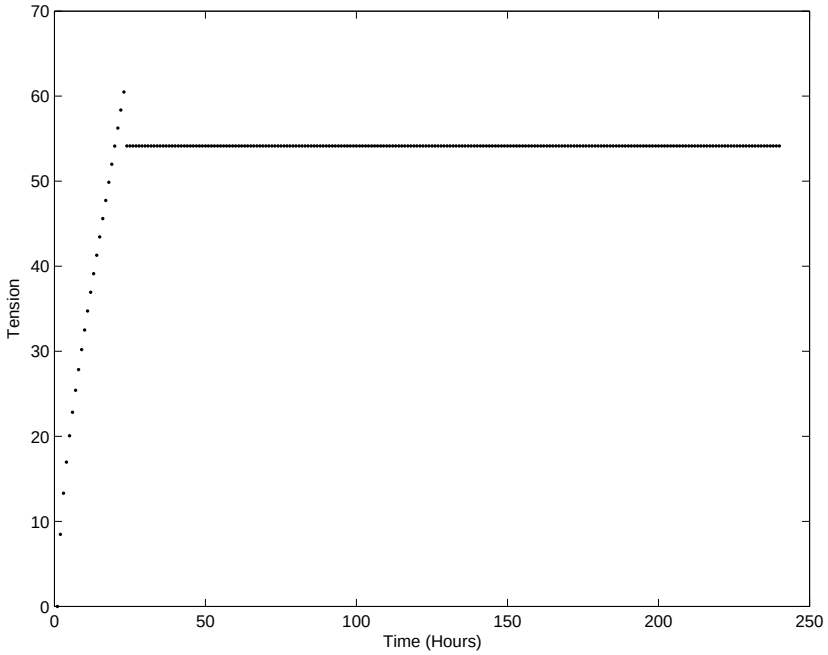

Figure A.8: Tension at Tesanj (26) Using Method 4.3

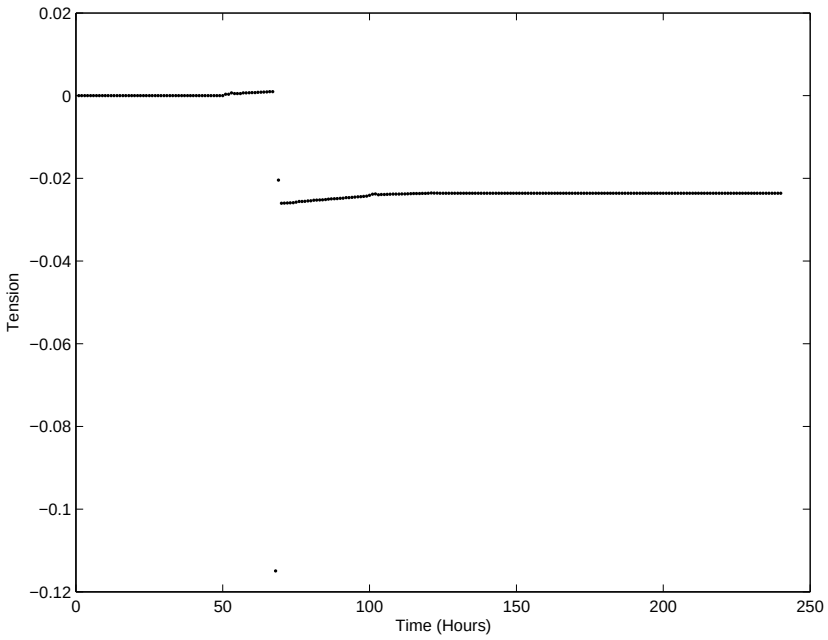

Figure A.9: Tension at Srebrenik (30) Using Method 4.3

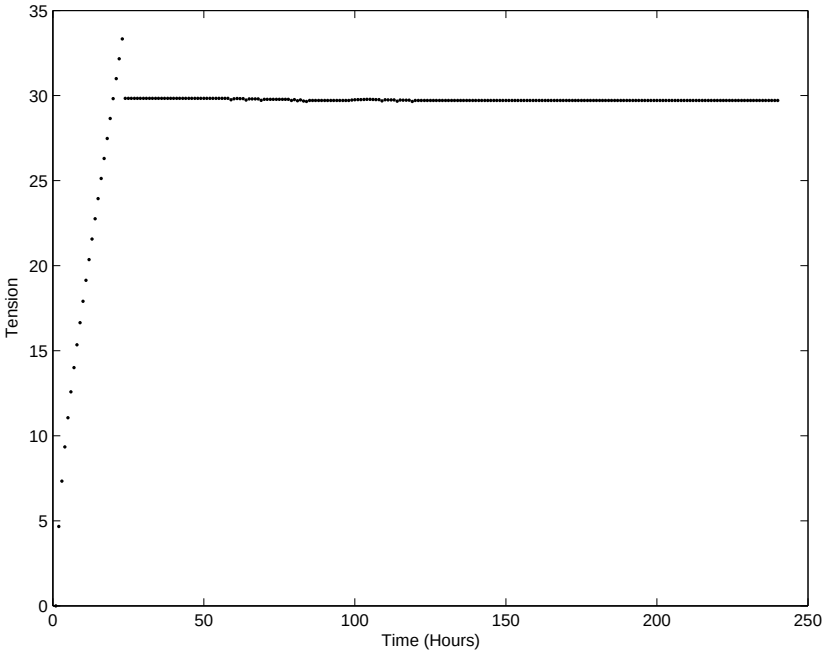

Figure A.10: Tension at Tuzla (31) Using Method 4.3

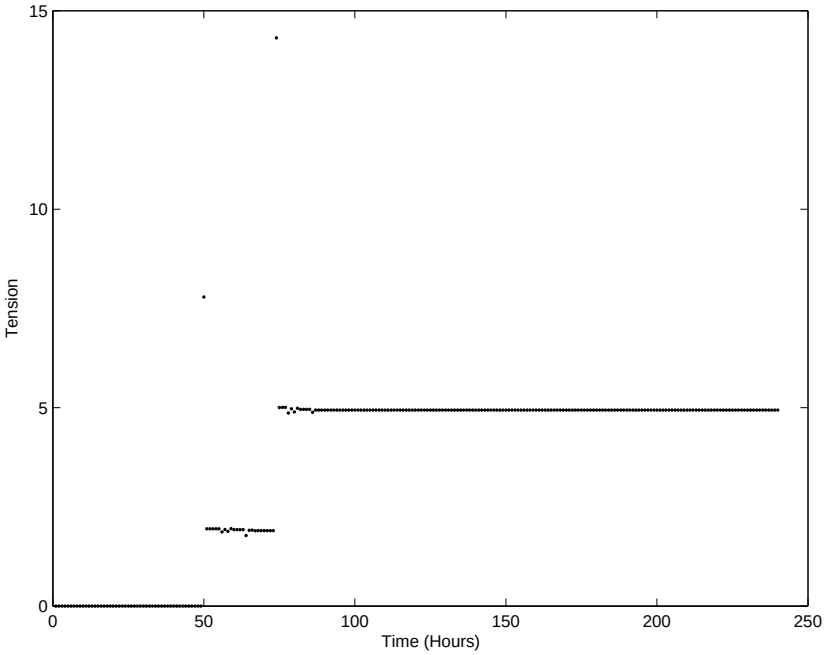

Figure A.11: Tension at Lopare (32) Using Method 4.3

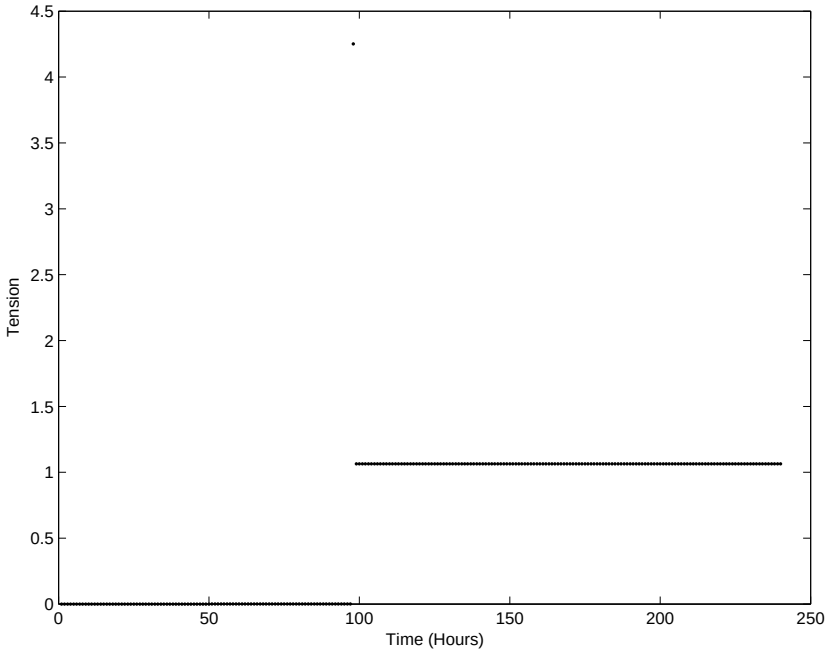

Figure A.12: Tension at Ugljevik (33) Using Method 4.3

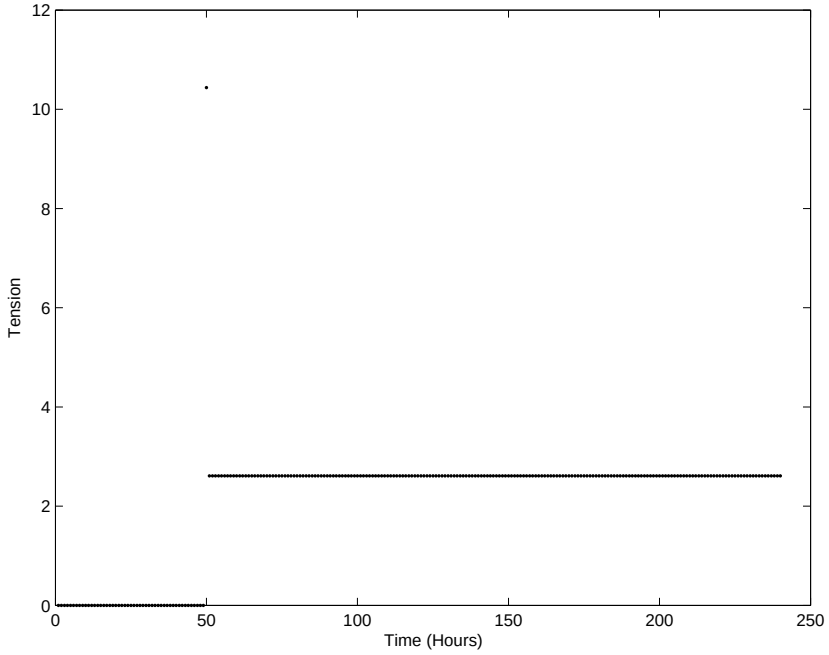

Figure A.13: Tension at Mrkonjic Grad (36) Using Method 4.3

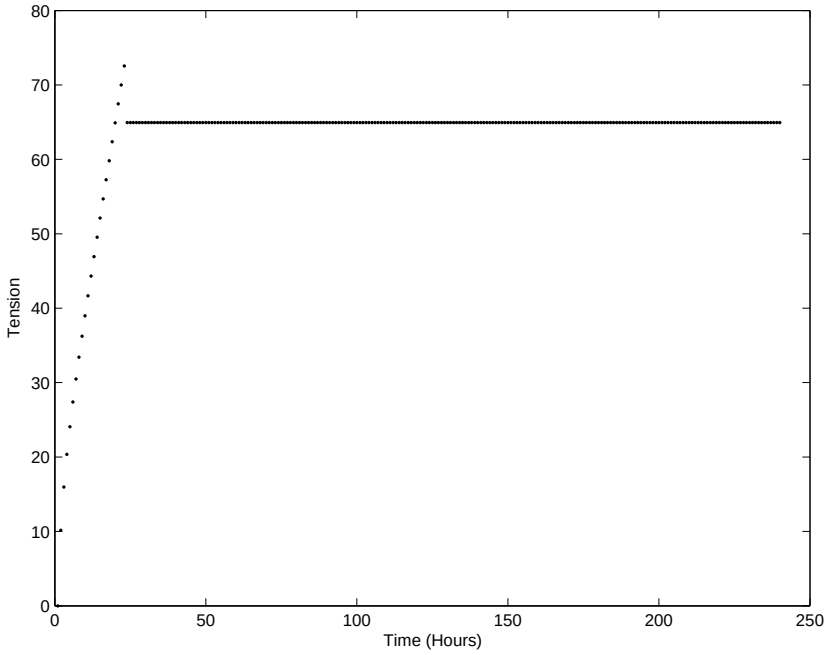

Figure A.14: Tension at Banovici (43) Using Method 4.3

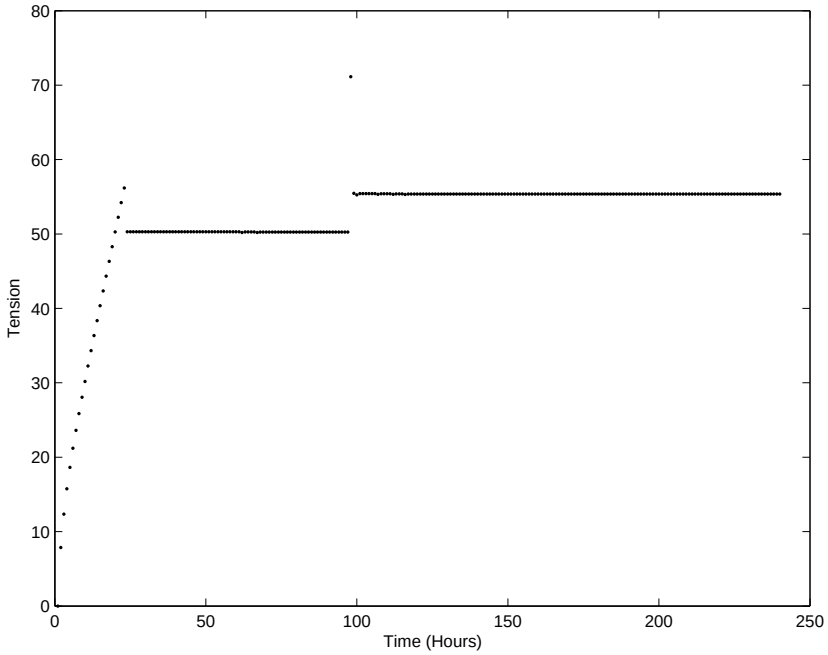

Figure A.15: Tension at Zinivice (44) Using Method 4.3

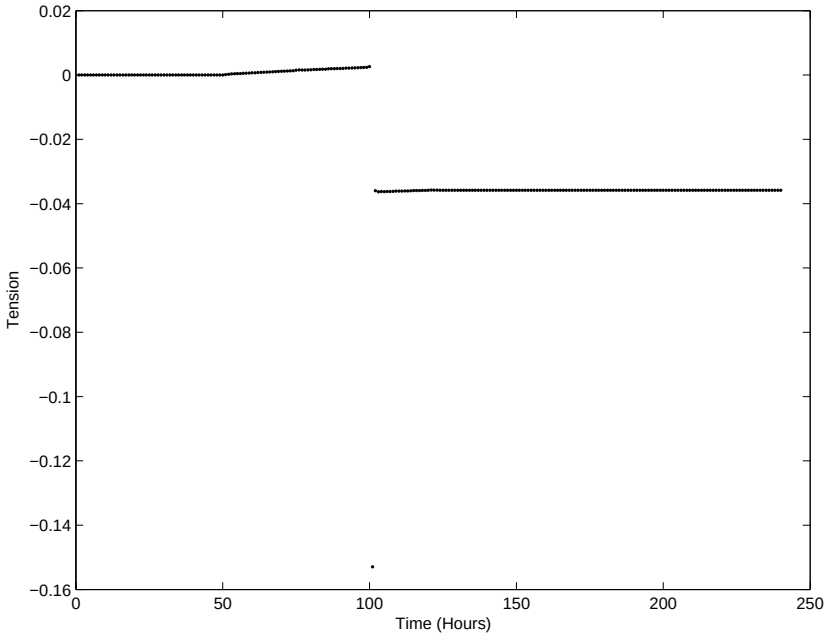

Figure A.16: Tension at Zvornik (46) Using Method 4.3

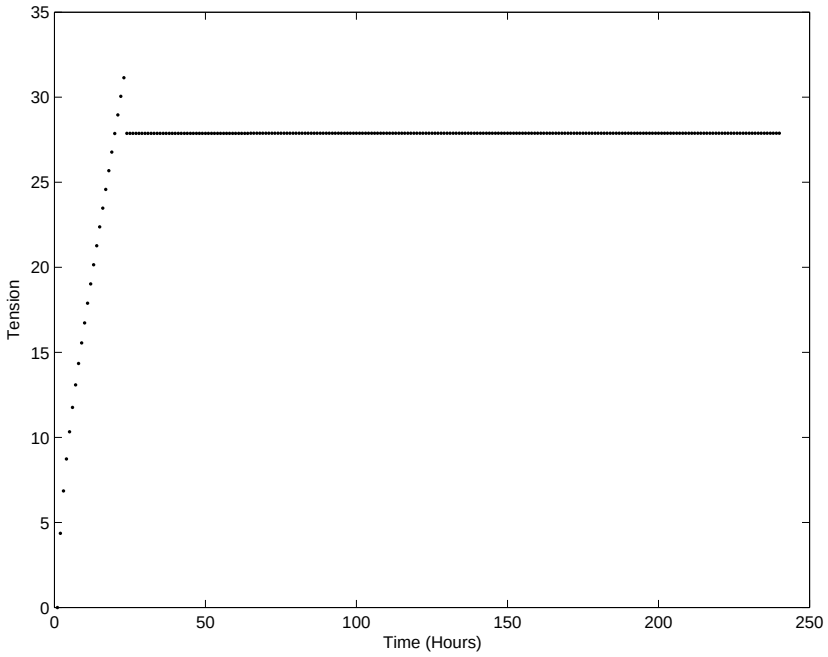

Figure A.17: Tension at Zenica (52) Using Method 4.3

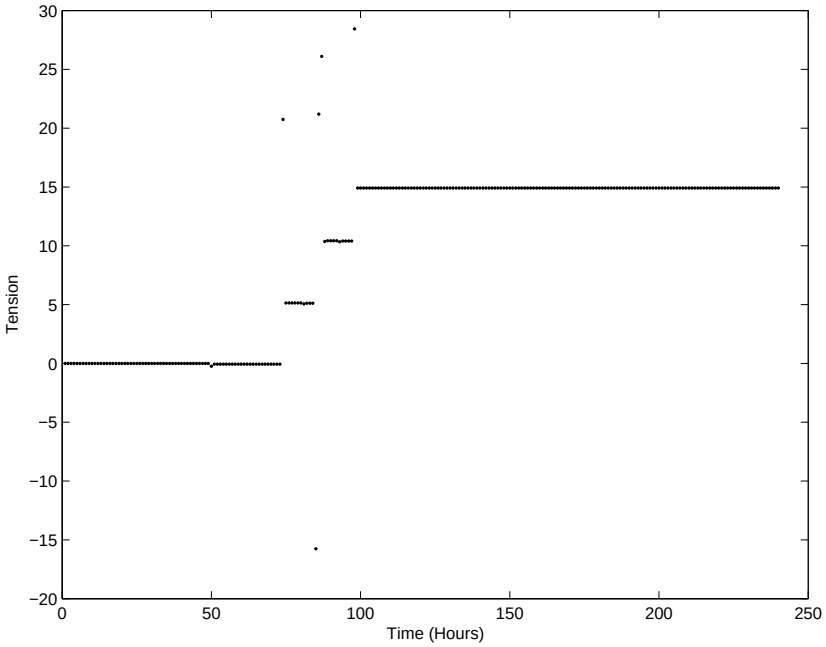

Figure A.18: Tension at Vares (54) Using Method 4.3

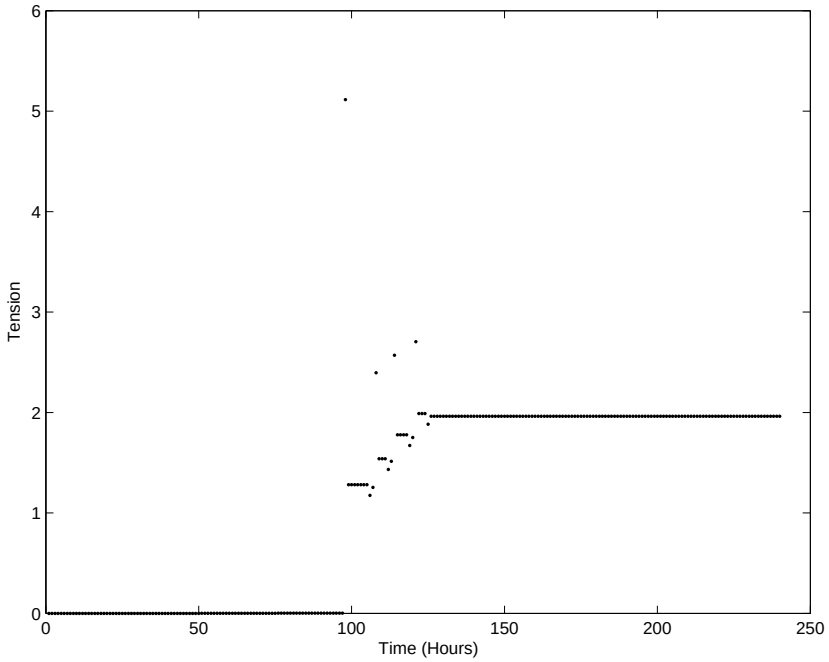

Figure A.19: Tension at Olovo (55) Using Method 4.3

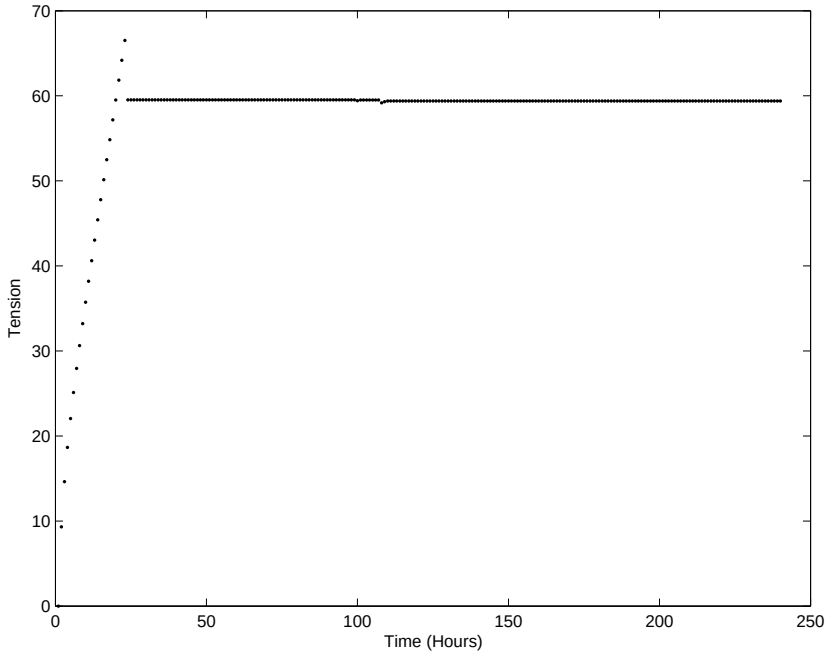

Figure A.20: Tension at Kladanj (56) Using Method 4.3

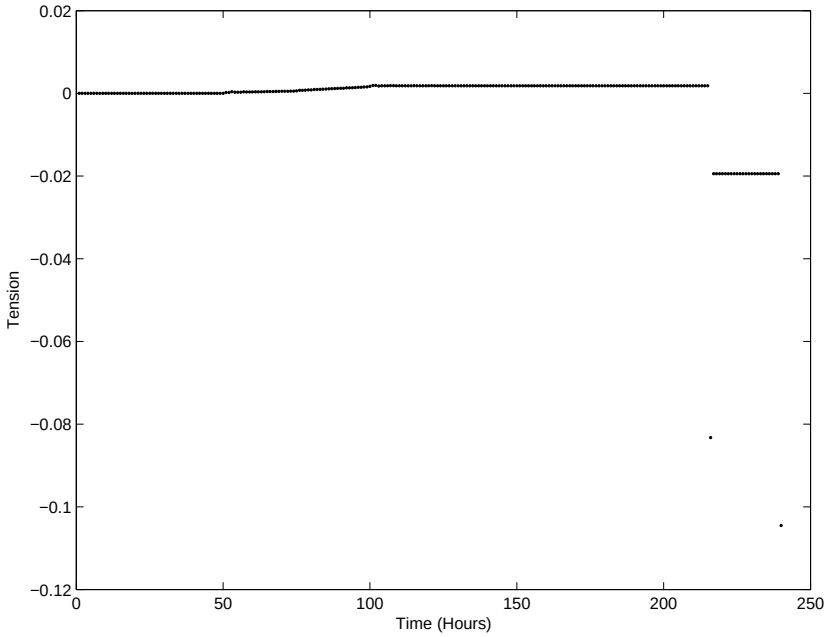

Figure A.21: Tension at Vlasenica (58) Using Method 4.3

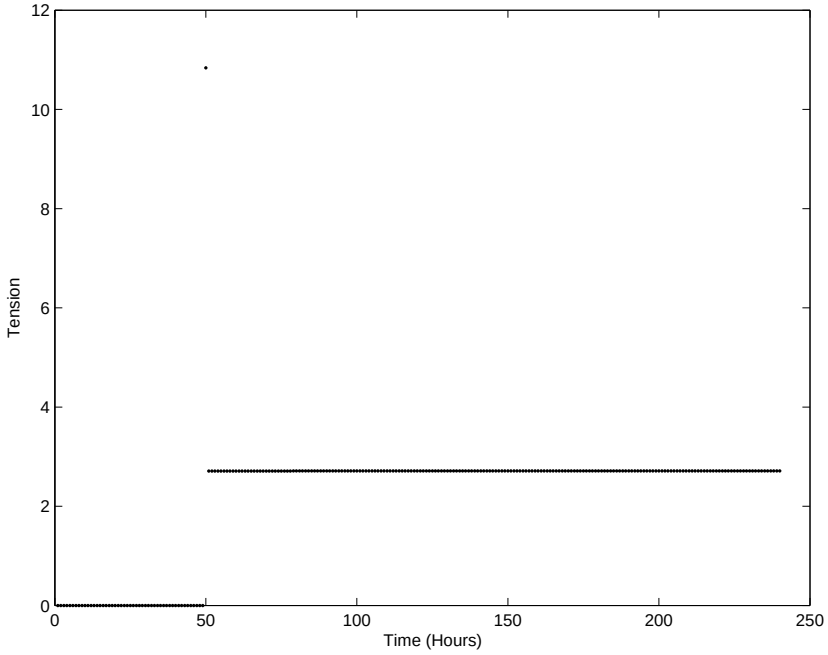

Figure A.22: Tension at Vitez (66) Using Method 4.3

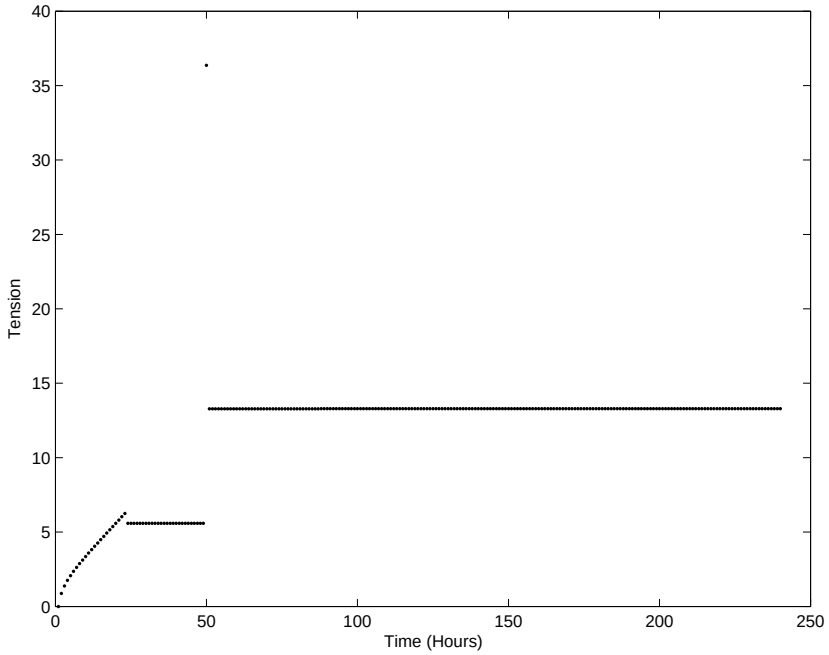

Figure A.23: Tension at Busovaca (67) Using Method 4.3

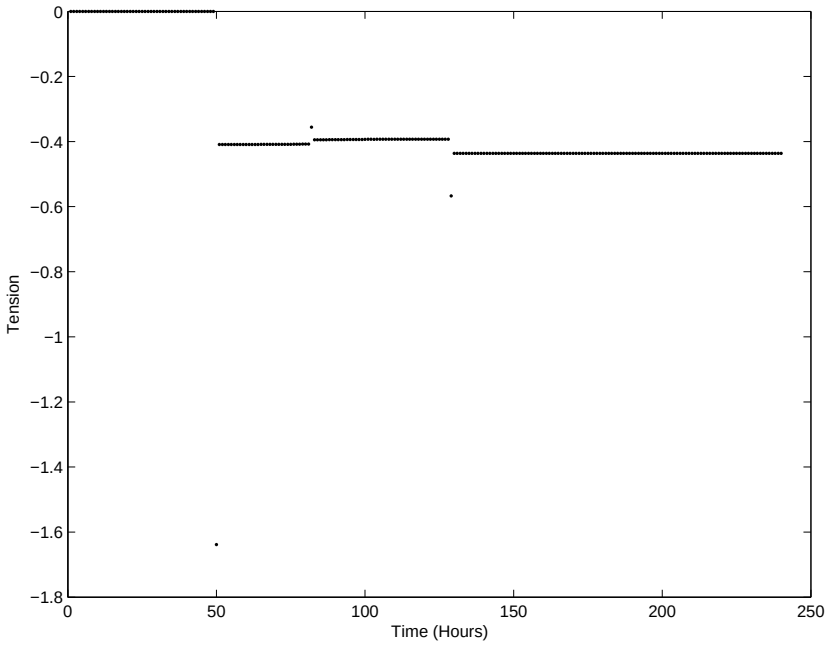

Figure A.24: Tension at Kiseljak (69) Using Method 4.3

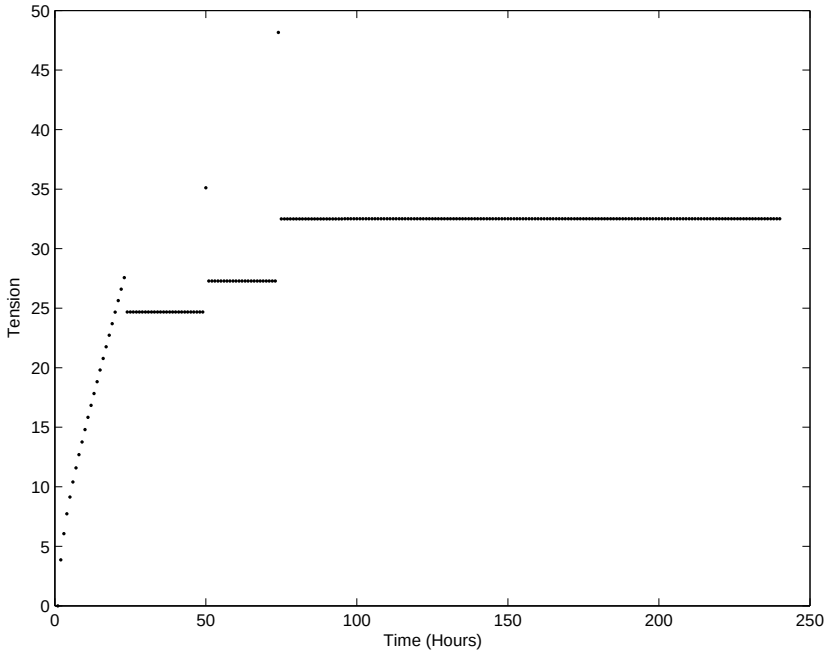

Figure A.25: Tension at Visoko (70) Using Method 4.3

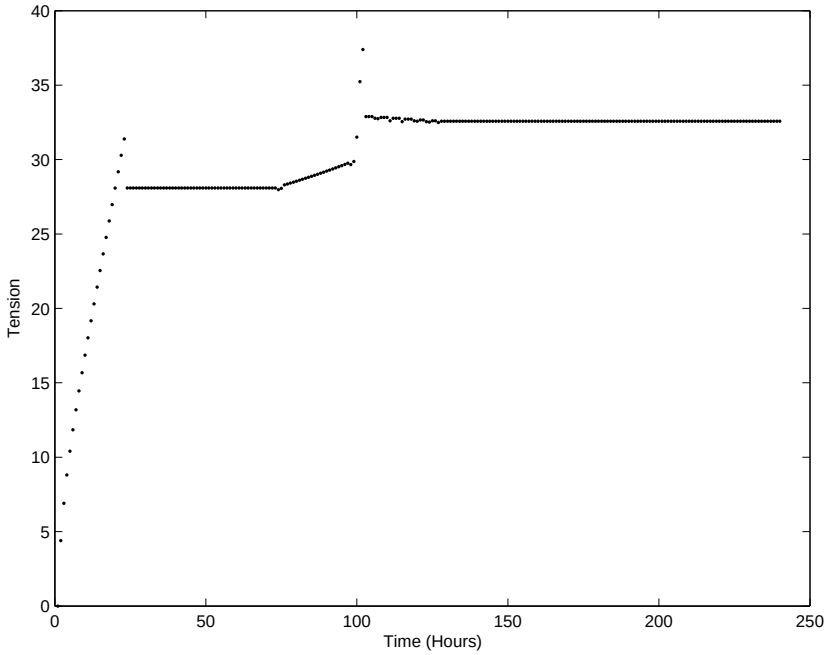

Figure A.26: Tension at Breza (71) Using Method 4.3

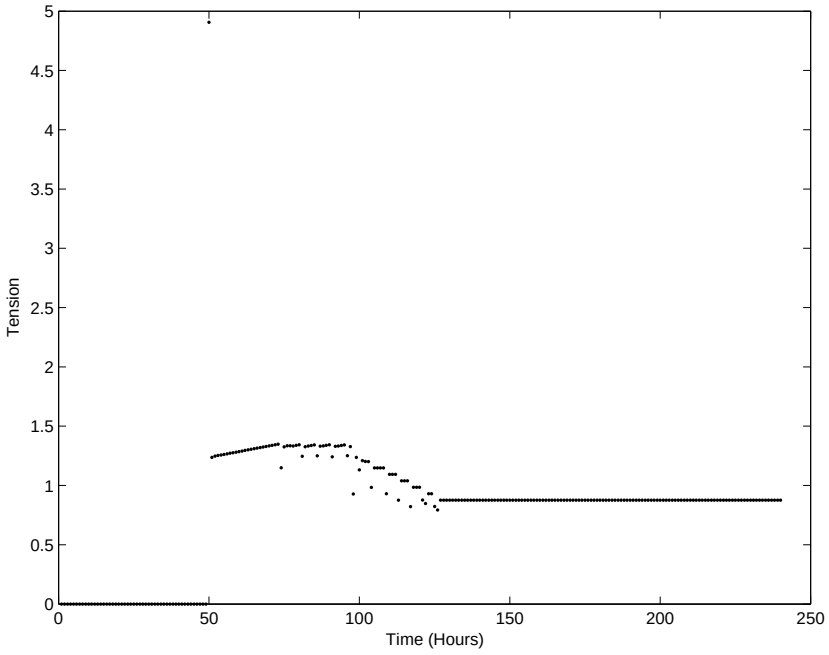

Figure A.27: Tension at Ilijas (72) Using Method 4.3

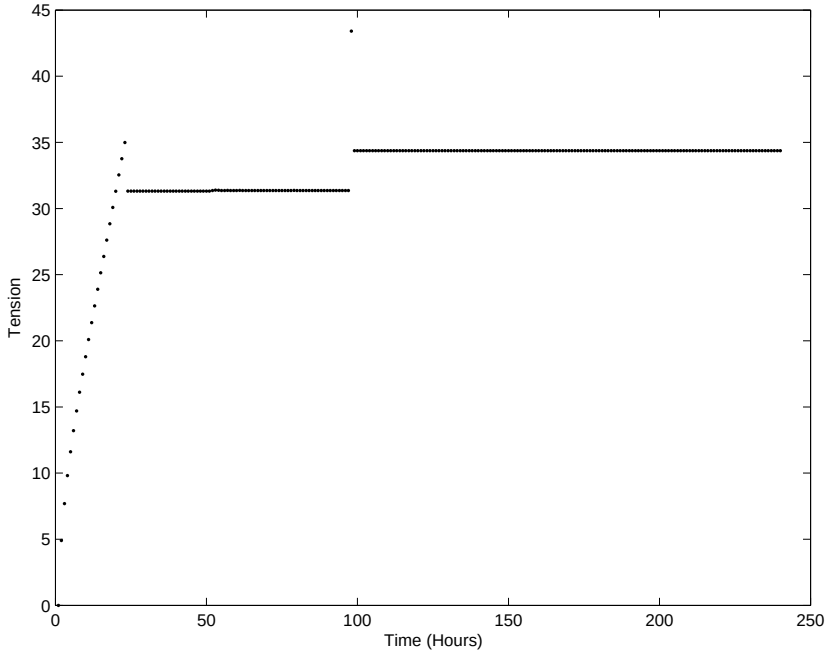

Figure A.28: Tension at Sokolac (73) Using Method 4.3

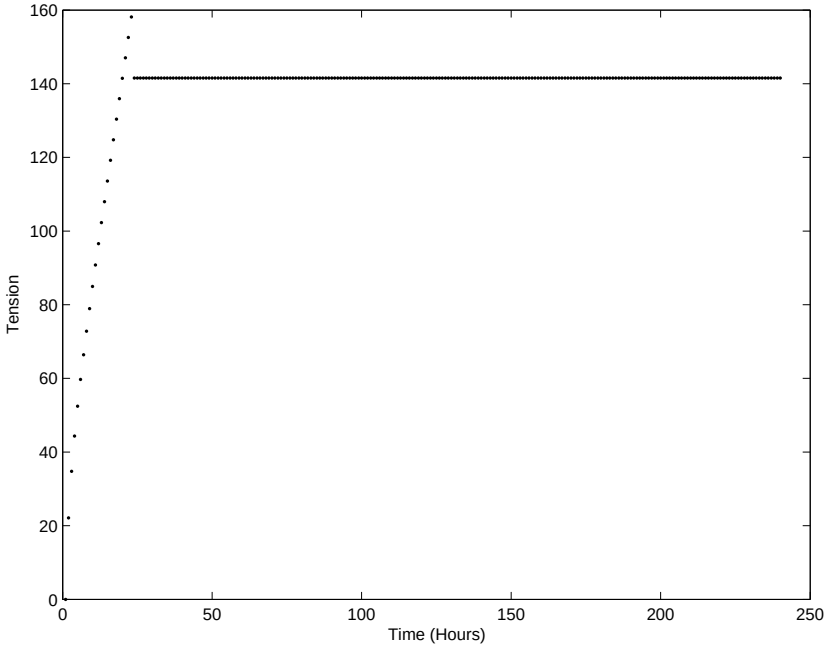

Figure A.29: Tension at Han Pijesak (74) Using Method 4.3

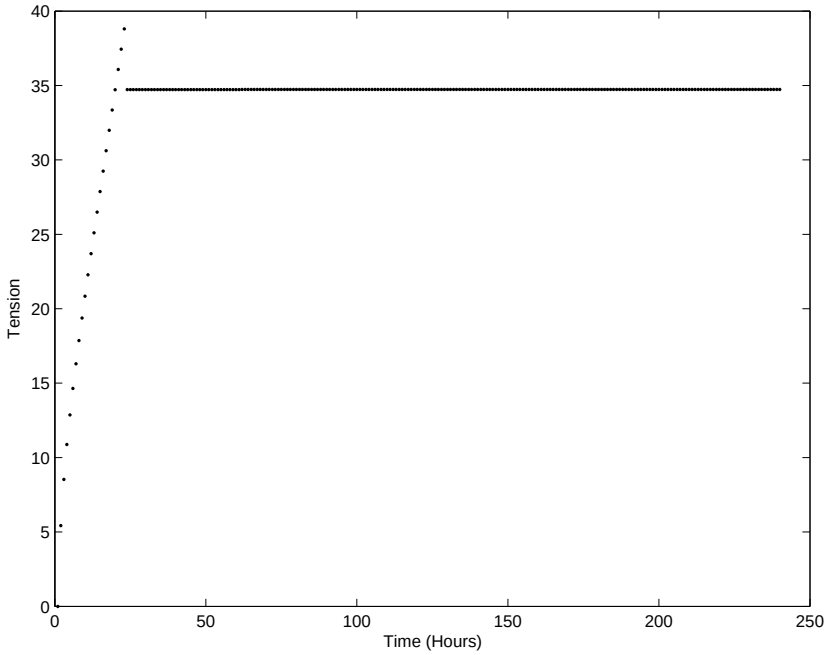

Figure A.30: Tension at Tomislavgrad (75) Using Method 4.3

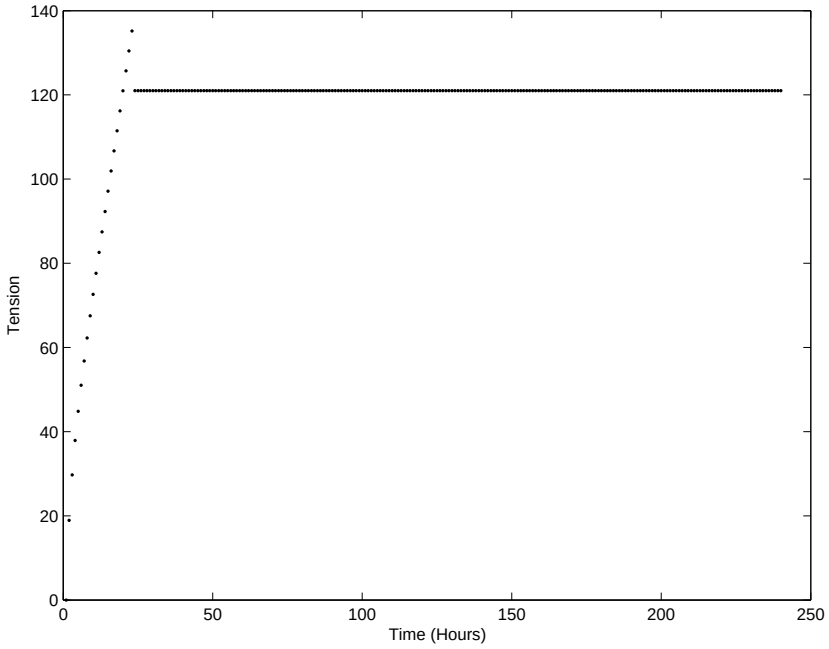

Figure A.31: Tension at Jablanica (77) Using Method 4.3

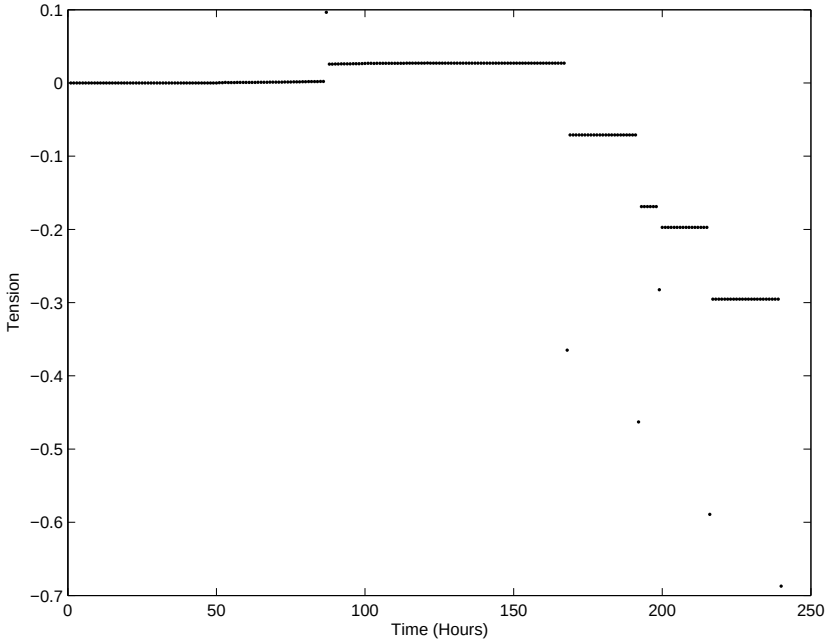

Figure A.32: Tension at Konjic (78) Using Method 4.3

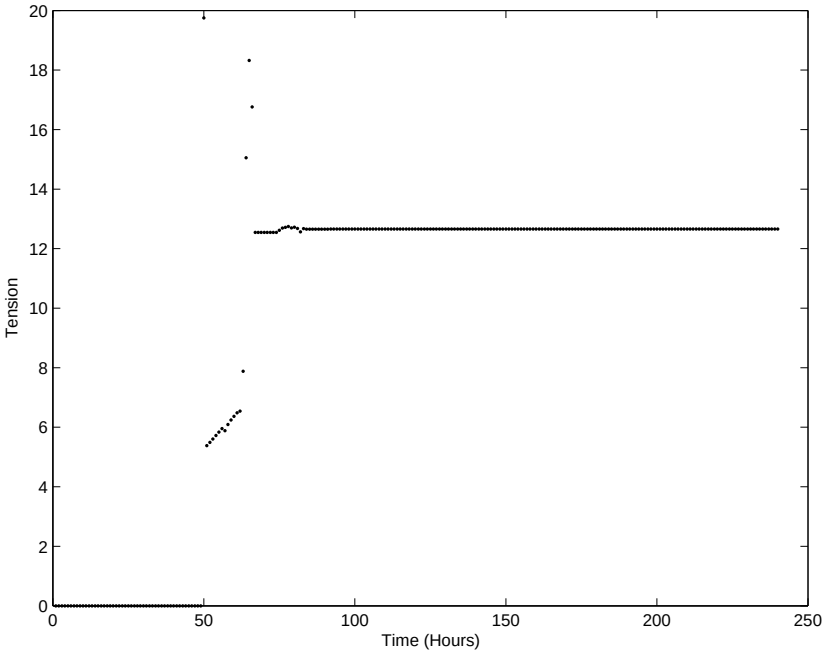

Figure A.33: Tension at Ilidja (81) Using Method 4.3

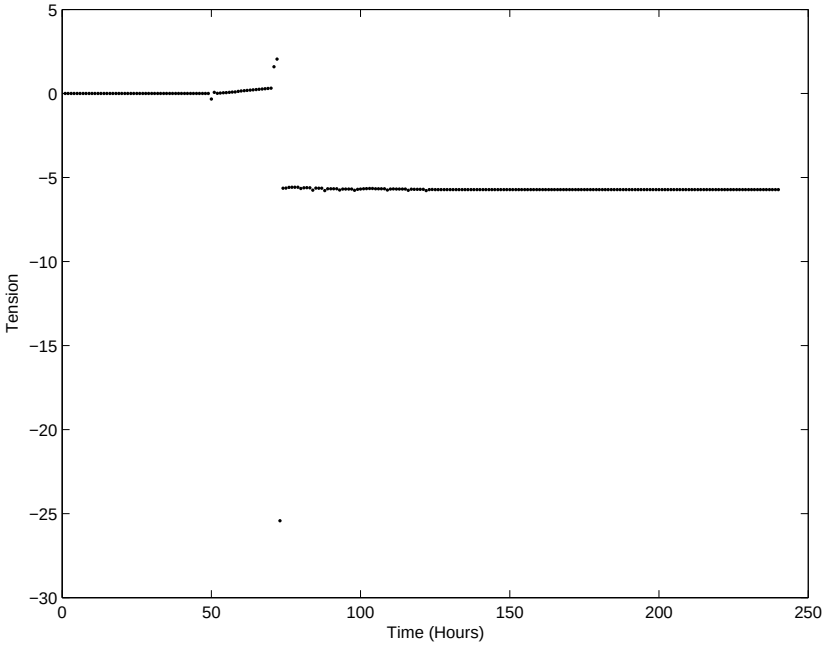

Figure A.34: Tension at Vogosca (83) Using Method 4.3

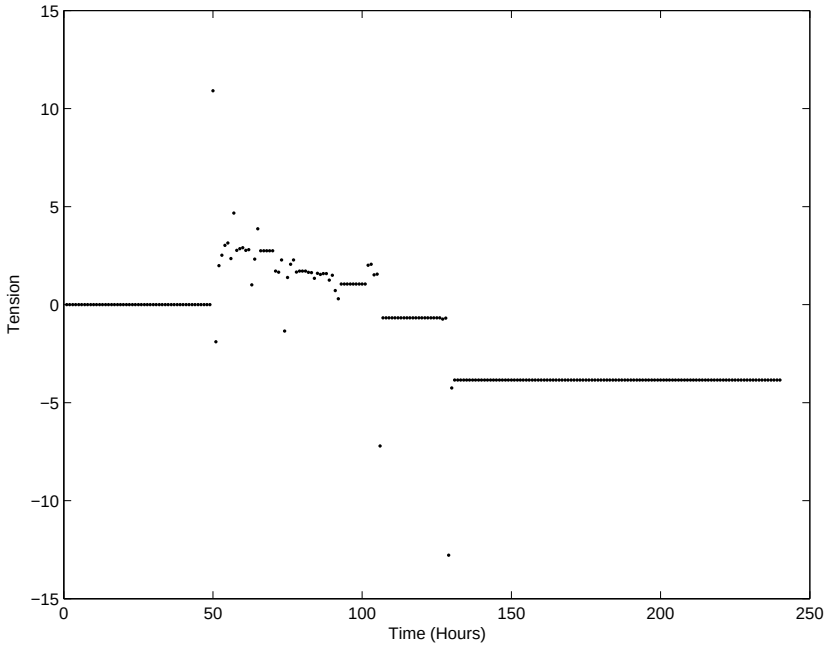

Figure A.35: Tension at Sarajevo Centar (84) Using Method 4.3

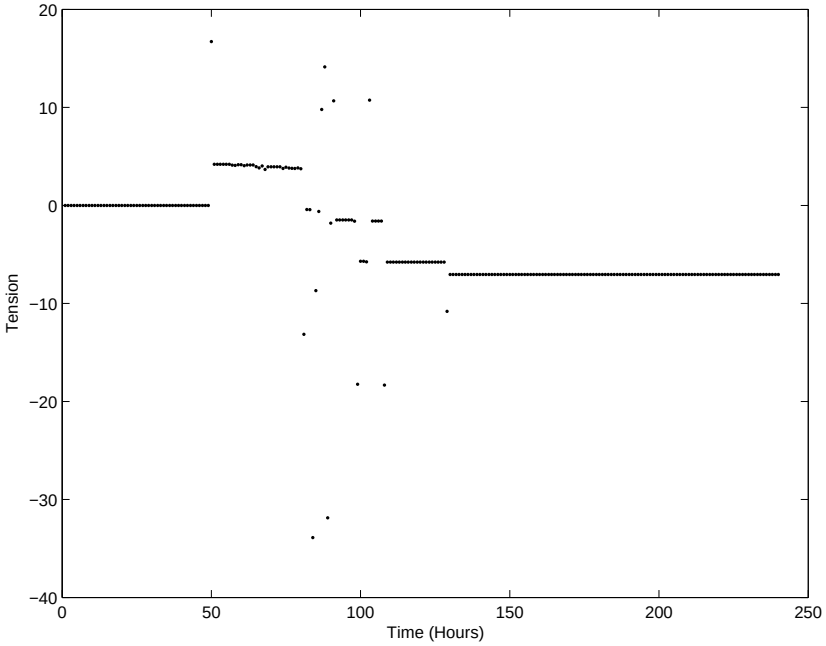

Figure A.36: Tension at Novo Sarajevo (86) Using Method 4.3

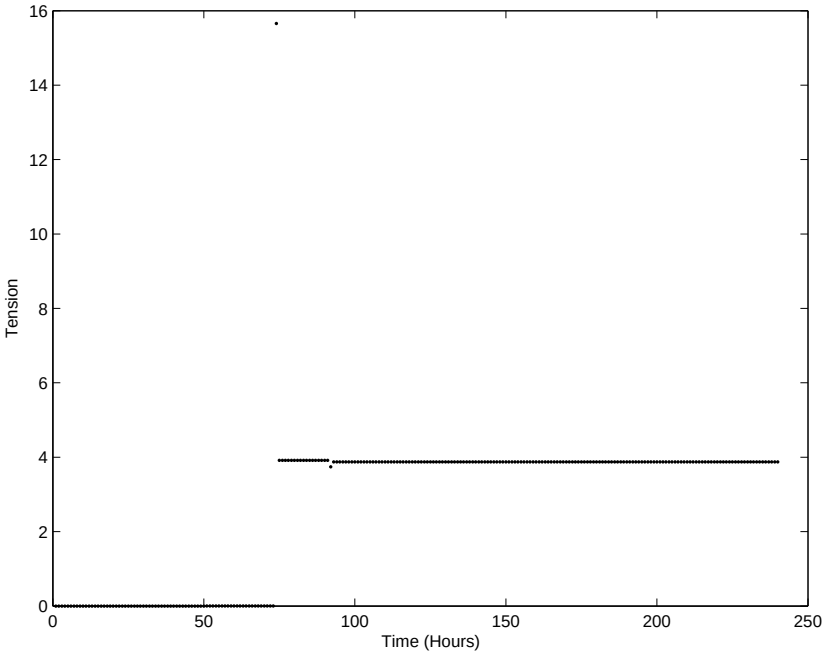

Figure A.37: Tension at Rogatica (89) Using Method 4.3

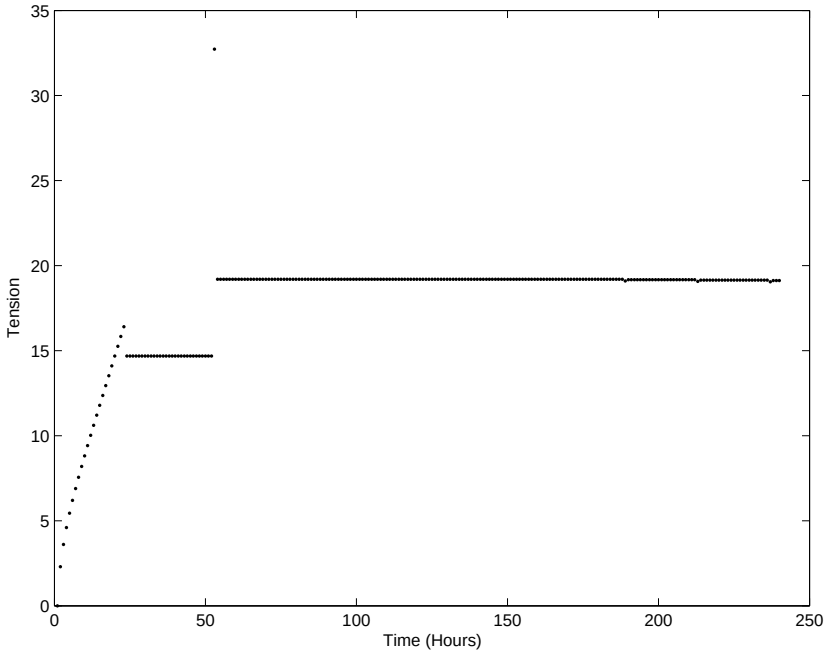

Figure A.38: Tension at Visegrad (90) Using Method 4.3

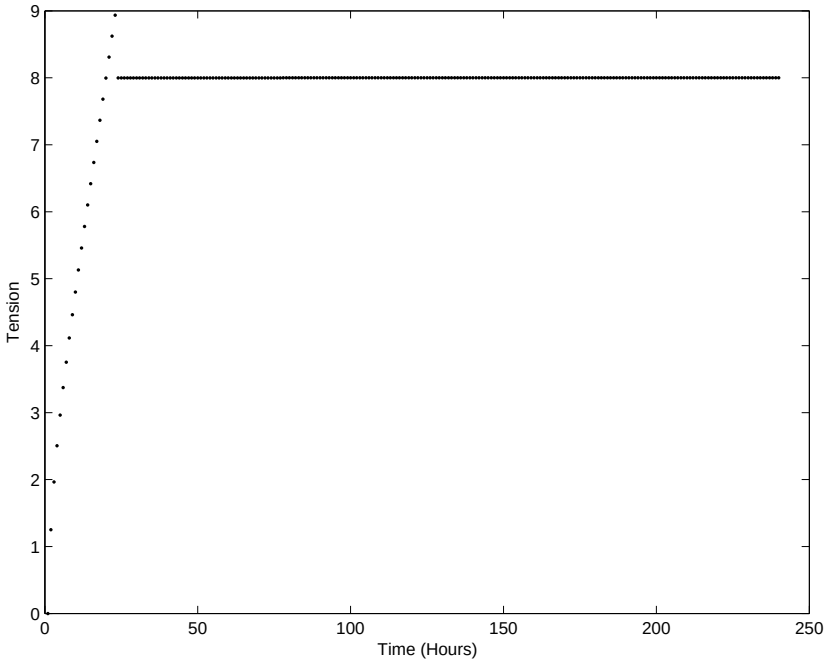

Figure A.39: Tension at Posusje (91) Using Method 4.3

Appendix B

MODEL CODE

This source code comes from Scenario Five, Run One. Changing the parameters in the code as appropriate gives the other scenarios.

B.1 main.cpp

```
// main.cpp
// Main program for model

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"            // Include the header file for the Cell class
#include "agent.h"           // Include agent classes header file
#include "model.h"           // Include general header file

using namespace std;



int main(int nNumberofArgs, char* pszArgs[])
{
    // CONSTANTS AND INITIALISATIONS

    int i, j, k, m, n;                          // Declare counters

    // Initialise the random number generator - Method taken from 'A Complete
    //                                          Guide To Programming In C++'
```

```
//                                              - Kirch-Prinz & Prinz

long sec;
time(&sec);
srand((unsigned)sec);

// Output random number generator seed so can reproduce a run if we need to
cout << "Random number generator seed = " << sec << "\n";

// SET GRID DETAILS

// Initialise grid
Cell grid[GRIDSIZE][GRIDSIZE];

// Declare array of pointers to grid array
Cell *pointGrid[GRIDSIZE][GRIDSIZE];

// Set each pointer to point at relevant cell
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
        pointGrid[j][i] = &(grid[j][i]);
    }
} // End of grid pointer initialisation loop

const int sectorNumber = SECTOR*SECTOR;        // Calculate the total number
                                               // of sectors

// Set sectorNo in the cell properties
int sectorSize;             // Declare variable to hold sector side-length
// Check that the grid can be split into equal sectors, if not give an
// error message and exit the program
if (fmod((float)(GRIDSIZE), (float)(SECTOR)) != 0)
{
    cout << "Error in main.cpp, line 69. \n"
         << "Error in SECTOR! Make sure GRIDSIZE is divisible by SECTOR.\n";
    system("PAUSE");
    exit(1);
}
else
{
    sectorSize = GRIDSIZE/SECTOR;
} // End of if/else loop

for (m = 0; m < SECTOR; m++)
{
    for (n = 0; n < SECTOR; n++)
    {
        // Go through all the cells in the sector and set sectorNo
        for (i = 0; i < sectorSize; i++)
```

```
        {
            for (j = 0; j < sectorSize; j++)
            {
                pointGrid[m*sectorSize + j][n*sectorSize + i]->sectorNo =
                                                                m*SECTOR + n;
            }
        } // End of initialisation of sector (m*SECTOR + n)
    }
} // End of sector initialisations loop

// Change the rest of the Cell properties that haven't worked properly in
// the constructor
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
        pointGrid[j][i]->occSquad = 0;
        pointGrid[j][i]->prevSquad = 0;
        pointGrid[j][i]->agentType = 0;
        pointGrid[j][i]->prevType = 0;
        pointGrid[j][i]->foodAmount = 1000;
        pointGrid[j][i]->actionType = 0;
        pointGrid[j][i]->moveInd = 0;
        pointGrid[j][i]->fixWater = 0;
        pointGrid[j][i]->fixElec = 0;
        pointGrid[j][i]->combat = 0;
        pointGrid[j][i]->shotToNo = 0;
        pointGrid[j][i]->shotInd = 0;
        pointGrid[j][i]->bombBlast = 0;
        pointGrid[j][i]->civInNeed = 0;
    }
} // End of grid parameter initialisations loop

// SET SQUAD DETAILS

int squadType[NOOFSQUADS];                  // Agent type of each squad,
squadType[0] = 1;                           // this gives the squad
squadType[1] = 2;                           // priority order for
squadType[2] = 3;                           // calculating initial
squadType[3] = 4;                           // positions.  Only one squad
                                            // per agent type is allowed
int squadSize[NOOFSQUADS];                  // Declare array for squad
                                            // sizes
int capability[NOOFSQUADS];                 // Declare array for squad
                                            // capabilities
int frequency[NOOFSQUADS];                  // Declare array for squad
                                            // action frequencies

// Set squad size and capability values according to '#define's in file
// agent.h
for (n = 0; n < NOOFSQUADS; n++)
```

```
{
    switch (squadType[n])
    {
        case 1:
            squadSize[n] = PEACENO;
            capability[n] = PEACECAP;
            break;
        case 2:
            squadSize[n] = SUPPORTNO;
            capability[n] = SUPPORTCAP;
            break;
        case 3:
            squadSize[n] = LOCALNO;
            capability[n] = LOCALCAP;
            break;
        case 4:
            squadSize[n] = CIVNO;
            capability[n] = CIVCAP;
            break;
        default:
            cout << "Error in main.cpp, line 160. \n"
                 << "Error in squadType[], check values.\n";
            system("PAUSE");
            exit(1);
    } // End of squadType switch loop
} // End of squad for loop

// Calculate and set maximum capability maxCap
int maxCap = 0;
for (n = 0; n < NOOFSQUADS; n++)
{
    if (capability[n] > maxCap)
    {
        maxCap = capability[n];
    }
} // End of squad for loop

// Set frequency values using capability[] and maxCap
for (n = 0; n < NOOFSQUADS; n++)
{
    switch (squadType[n])
    {
        case 1:
            frequency[n] = (int)(maxCap/PEACECAP);
            break;
        case 2:
            frequency[n] = (int)(maxCap/SUPPORTCAP);
            break;
        case 3:
            frequency[n] = (int)(maxCap/LOCALCAP);
            break;
```

```
        case 4:
            frequency[n] = (int)(maxCap/CIVCAP);
            break;
        default:
            cout << "Error in main.cpp, line 195. \n"
                 << "Error in squadType[], check values.\n";
            system("PAUSE");
            exit(1);
    } // End of squadType switch loop
} // End of for loop

// Construct agents
Peacekeeper peace[PEACENO];
SupportAgent support[SUPPORTNO];
LocalMilitia local[LOCALNO];
Civilian civilian[CIVNO];

// Set pointers to the agents

// Declare array of pointers to Peacekeeper array
Peacekeeper *pointPeace[PEACENO];
// Set each pointer to point at relevant agent
for (i = 0; i < PEACENO; i++)
{
    pointPeace[i] = &(peace[i]);
}
// Declare array of pointers to NGO array
SupportAgent *pointSupport[SUPPORTNO];
// Set each pointer to point at relevant agent
for (i = 0; i < SUPPORTNO; i++)
{
    pointSupport[i] = &(support[i]);
}
// Declare array of pointers to Insurgent array
LocalMilitia *pointLocal[LOCALNO];
// Set each pointer to point at relevant agent
for (i = 0; i < LOCALNO; i++)
{
    pointLocal[i] = &(local[i]);
}
// Declare array of pointers to Civilian array
Civilian *pointCivilian[CIVNO];
// Set each pointer to point at relevant agent
for (i = 0; i < CIVNO; i++)
{
    pointCivilian[i] = &(civilian[i]);
}

short int iniGrid[GRIDSIZE][GRIDSIZE] = {0};  // Declare and initialise
                                               // array for initial agent
                                               // positions
```

```
// CALCULATE INITIAL POSITIONS FOR THE AGENTS

for (n = 0; n < NOOFSQUADS; n++)
{
    for (m = 0; m < squadSize[n]; m++)
    {
        switch (squadType[n])
        {
            case 1:
                // Peacekeeper
                // Initialise position
                peaceInitPos(iniGrid, pointPeace[m]);
                // Amend Cell definition
                pointGrid[(peace[m].yPos)][(peace[m].xPos)]->occSquad = 1;
                pointGrid[(peace[m].yPos)][(peace[m].xPos)]->prevSquad = 1;
                pointGrid[(peace[m].yPos)][(peace[m].xPos)]->agentType = 1;
                pointGrid[(peace[m].yPos)][(peace[m].xPos)]->prevType = 1;
                break;
            case 2:
                // NGO
                suppInitPos(iniGrid, pointSupport[m]);
                // Amend Cell definition
                pointGrid[(support[m].yPos)][(support[m].xPos)]->occSquad
                                                                    = 2;
                pointGrid[(support[m].yPos)][(support[m].xPos)]->prevSquad
                                                                    = 2;
                pointGrid[(support[m].yPos)][(support[m].xPos)]->agentType
                                                                    = 2;
                pointGrid[(support[m].yPos)][(support[m].xPos)]->prevType
                                                                    = 2;
                break;
            case 3:
                // Insurgent
                locInitPos(iniGrid, pointLocal[m]);
                // Amend Cell definition
                pointGrid[(local[m].yPos)][(local[m].xPos)]->occSquad = 3;
                pointGrid[(local[m].yPos)][(local[m].xPos)]->prevSquad = 3;
                pointGrid[(local[m].yPos)][(local[m].xPos)]->agentType = 3;
                pointGrid[(local[m].yPos)][(local[m].xPos)]->prevType = 3;
                break;
            case 4:
                // Civilian
                civInitPos(iniGrid, pointCivilian[m]);
                // Amend Cell definition
                pointGrid[(civilian[m].yPos)][(civilian[m].xPos)]->occSquad
                                                                    = 4;
                pointGrid[(civilian[m].yPos)][(civilian[m].xPos)]->prevSquad
                                                                    = 4;
                pointGrid[(civilian[m].yPos)][(civilian[m].xPos)]->agentType
                                                                    = 4;
```

```
                pointGrid[(civilian[m].yPos)][(civilian[m].xPos)]->prevType
                                                                        = 4;
                break;
            default:
                cout << "Error in main.cpp, line 305. \n"
                     << "Error in squadType[], check values.\n";
                     system("PAUSE");
                     exit(1);
        } // End of squadType switch loop
    } // End of agent for loop
} // End of squad for loop

// Write initial grid to file
ofstream gridoutput;    // Declare file to write output to
gridoutput.open("gridoutput.txt", ios::app);
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        gridoutput << grid[j][i].agentType << " ";
    }
    gridoutput << "\n";
} // End of grid loop
gridoutput << "\n";
gridoutput.close();


// DECLARE VARIABLES

int currentT;                 // Declare counter for timesteps
int subT;                     // Declare counter for sub-timesteps
float x;                      // Declare variables for random numbers
int y, z;
int xFail, yFail;             // Declare variables to hold coordinates for
                              // failed water or electricity supply
int actionInd;                // Declare variable for action type
int *aInd;                    // Declare pointer to actionInd
aInd = &actionInd;

// Initialise shot and bomb counters
int shotCount = 0;
int bombCount = 0;

// Declare pointers to shotCount and bombCount
int *pShotCount;
pShotCount = &shotCount;
int *pBombCount;
pBombCount = &bombCount;

short shotArray[MAXSHOTS][10];      // Declare array to hold details of
                                    // shots fired
```

```
short bombArray[MAXBOMB][6];        // Declare array to hold details of bomb
                                    // attacks

// Declare array of pointers to shotArray
short *pointShot[MAXSHOTS][10];
// Set each pointer to point at relevant array entry
for (j = 0; j < MAXSHOTS; j++)
{
    for (i = 0; i < 10; i++)
    {
        pointShot[j][i] = &(shotArray[j][i]);
    }
} // End of shotArray pointer initialisation loop

// Declare array of pointers to bombArray
short *pointBomb[MAXBOMB][6];
// Set each pointer to point at relevant array entry
for (j = 0; j < MAXBOMB; j++)
{
    for (i = 0; i < 6; i++)
    {
        pointBomb[j][i] = &(bombArray[j][i]);
    }
} // End of bombArray pointer initialisation loop


// Initialise the waterFailure and elecFailure arrays to zero
int waterFailure[sectorNumber] = {0};       // Water supply failure
                                            // indicator for the sectors
int elecFailure[sectorNumber] = {0};        // Electricity supply failure
                                            // indicator for the sectors

// ******  ADDED IN FOR THIS SCENARIO: SET FINAL SECTOR WATER TO FAIL ******

waterFailure[sectorNumber - 1] = 1;

y = (int)(rand()*sectorSize/32768);
z = (int)(rand()*sectorSize/32768);
// Compute the cell coordinates relating to these random integers
xFail = (SECTOR - 1)*sectorSize + y;
yFail = (SECTOR - 1)*sectorSize + z;

// Check (xFail, yFail) is within the array bounds
if ((xFail >= GRIDSIZE) || (yFail >= GRIDSIZE) || (xFail < 0)
                                                        || (yFail < 0))
{
    cout << "Error in repair location in main.cpp line 395"
         << " (" << xFail << "," << yFail << "). \n";
         system("PAUSE");
         exit(1);
} // End of error check if loop
```

```
pointGrid[yFail][xFail]->fixWater = 1;
cout << "Water failure at (" << xFail << "," << yFail << "). \n";

// ************************************************************************


// Record other initial data
ofstream combatind;    // Declare file to write combat indicator output to
combatind.open("combatind.txt", ios::app);
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        combatind << grid[j][i].combat << " ";
    }
    combatind << "\n";
} // End of grid loop
combatind << "\n";
combatind.close();

ofstream civinneedind;    // Declare file to write Civilian in need
                          // indicator output to
civinneedind.open("civinneedind.txt", ios::app);
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        civinneedind << grid[j][i].civInNeed << " ";
    }
    civinneedind << "\n";
} // End of grid loop
civinneedind << "\n";
civinneedind.close();

ofstream waterfailind;    // Declare file to write water failure indicator
                          // to
waterfailind.open("waterfailind.txt", ios::app);
for (k = 0; k < sectorNumber; k++)
{
    waterfailind << waterFailure[k] << " ";
} // End of array loop
waterfailind << "\n";
waterfailind.close();

ofstream elecfailind;    // Declare file to write water failure indicator to
elecfailind.open("elecfailind.txt", ios::app);
for (k = 0; k < sectorNumber; k++)
{
    elecfailind << elecFailure[k] << " ";
} // End of array loop
```

```
elecfailind << "\n";
elecfailind.close();

// TIMESTEPS

for (currentT = 1; currentT <= RUNTIME; currentT++)
{

    // Go through all the sub-timesteps for this timestep
    for (subT = 1; subT <= maxCap; subT++)
    {

        // Go through the entire grid once to determine actions for the
        // agents for this sub-timestep.
        for (i = 0; i < GRIDSIZE; i++)
        {
            for (j = 0; j < GRIDSIZE; j++)
            {
                // Determine the type of agent at the cell, if any, and then
                // determine the course of action for that agent.
                switch (grid[j][i].prevType)
                {
                    case 0:
                        // No agent, no action
                        pointGrid[j][i]->actionType = 0;
                        break;
                    case 1:
                        // Peacekeeper at cell.
                        // First determine whether the agent should take any
                        // action at this sub-timestep.
                        *aInd = actionIndicator(subT,
                                     frequency[(grid[j][i].prevSquad) - 1],
                                   capability[(grid[j][i].prevSquad) - 1]);
                        // If there is no action to be taken set the
                        // actionType cell variable to 0;
                        if (actionInd == 0)
                        {
                            pointGrid[j][i]->actionType = 0;
                        }
                        // Check to see if there have been any shots fired
                        // to the cell in the last timestep, if so set the
                        // actionType cell variable to 1 (combat)
                        else if (grid[j][i].shotInd == 1)
                        {
                            pointGrid[j][i]->actionType = 1;
                        }
                        // Check to see if there have been any bomb blasts
                        // affecting the cell, if so set the actionType
                        // cell variable to 1 (combat)
                        else if (grid[j][i].bombBlast == 1)
                        {
```

```
            pointGrid[j][i]->actionType = 1;
        }

        // Check whether repairs to water or electricity
        // supply are needed at the cell, if so set the
        // actionType cell indicator to 3 (repair)
        else if ((grid[j][i].fixWater == 1) ||
                                  (grid[j][i].fixElec == 1))
        {
            pointGrid[j][i]->actionType = 3;
        }
        // Otherwise set the actionType cell indicator to 4
        // (move)
        else
        {
            pointGrid[j][i]->actionType = 4;
        } // End of if/else if/else loop

        break;
    case 2:
        // NGO at cell
        // First determine whether the agent should take any
        // action at this sub-timestep.
        *aInd = actionIndicator(subT,
                     frequency[(grid[j][i].prevSquad) - 1],
                    capability[(grid[j][i].prevSquad) - 1]);
        // If there is no action to be taken set the
        // actionType cell variable to 0.
        if (actionInd == 0)
        {
            pointGrid[j][i]->actionType = 0;
        }
        // Check to see if there have been any shots fired
        // at the cell in the last timestep, if so set the
        // actionType cell variable to 4 (move)
        else if (grid[j][i].shotInd == 1)
        {
            pointGrid[j][i]->actionType = 4;
        }
        // Check whether repairs to water or electricity
        // supply are needed at the cell, if so set the
        // actionType cell indicator to 3 (repair)
        else if ((grid[j][i].fixWater == 1) ||
                                  (grid[j][i].fixElec == 1))
        {
            pointGrid[j][i]->actionType = 3;
        }
        // Otherwise set the actionType cell indicator to 4
        // (move)
        else
        {
```

```
            pointGrid[j][i]->actionType = 4;
        } // End of if/else if/else loop

        break;
    case 3:
        // Insurgent agent at the cell
        // First determine whether the agent should take any
        // action at this sub-timestep.
        *aInd = actionIndicator(subT,
                     frequency[(grid[j][i].prevSquad) - 1],
                    capability[(grid[j][i].prevSquad) - 1]);
        // If there is no action to be taken set the
        // actionType cell variable to 0 (no action)
        if (actionInd == 0)
        {
            pointGrid[j][i]->actionType = 0;
        }
        // Check to see if there have been any shots fired
        // at the cell since the agent last performed an
        // action, if so set the actionType cell variable to
        // 1 (combat)
        else if (grid[j][i].shotInd == 1)
        {
            pointGrid[j][i]->actionType = 1;
        }
        // Determine whether or not the insurgent agent
        // will attack any hostile agents.  Note that we do
        // not check whether or not there are any valid
        // targets within range at this point, this is done
        // later and if there are no valid targets within
        // range the actionType is changed to move.
        else
        {
            // Generate random number to determine whether
            // or not to set off bomb, if so set actionType
            // cell indicator to 2 (bomb)
            x = (float)rand()/32767;
            if ((LMBOMBPROB != 0.00000) &&
                                        (x <= LMBOMBPROB))
            {
                pointGrid[j][i]->actionType = 2;
            }
            // If no bomb is to be set off determine whether
            // or not to fire, if so set actionType cell
            // indicator to 1 (combat), if not set
            // actionType cell indicator to 4 (move)
            else
            {
                x = (float)rand()/32767;
                if ((LMFIREPROB != 0.00000) &&
                                        (x <= LMFIREPROB))
```

```
                        {
                            pointGrid[j][i]->actionType = 1;
                        }
                        else
                        {
                            pointGrid[j][i]->actionType = 4;
                        } // End of if/else loop
                    } // End of if/else loop
                } // End of if/else if/else loop

                break;
            case 4:
                // Civilian agent at cell
                // First determine whether the agent should take any
                // action at this sub-timestep.
                *aInd = actionIndicator(subT,
                              frequency[(grid[j][i].prevSquad) - 1],
                            capability[(grid[j][i].prevSquad) - 1]);
                // If there is no action to be taken set the
                // actionType cell variable to 0.
                if (actionInd == 0)
                {
                    pointGrid[j][i]->actionType = 0;
                }
                // Otherwise set the actionType cell variable to 4
                // (move)
                else
                {
                    pointGrid[j][i]->actionType = 4;
                }
                // End of if/else loop

                break;
            default:
                // Error message
                cout << "Error in main.cpp, line 515. \n"
                     << "Unknown agent type!\n";
                system("PAUSE");
                exit(1);
        } // End of prevType switch loop
    }
} // End of grid loop to determine action type


// Go through the entire grid to determine all the combat that will
// take place at this sub-timestep.  Combat includes both firing and
// bombs so any agent at a cell with actionType 1 or 2 will be
// involved.  If there are no valid targets for an agent the
// actionType indicator is changed to 4 (move).
for (i = 0; i < GRIDSIZE; i++)
{
```

```
for (j = 0; j < GRIDSIZE; j++)
{
    // Check the actionType indicator for each cell, if it is
    // not 1 or 2 no action is taken at this stage
    switch (grid[j][i].actionType)
    {
        case 0:
            // No action needed
            break;
        case 1:
            // Call relevant combat function depending on what
            // type of agent is at the cell
            if (grid[j][i].prevType == 1)
            {
                // Go through whole squad to find the
                // Peacekeeper at (i, j)
                for (k = 0; k < PEACENO; k++)
                {
                    if ((peace[k].xPrev == i) &&
                                        (peace[k].yPrev == j))
                    {
                        peaceCombat(pointGrid, pointPeace[k],
                                    pointPeace, pointSupport,
                                    pointLocal, pointCivilian,
                                    pointShot, pShotCount,
                                    currentT, subT);

                        arrayCheck(grid, 4);
                    } // End of if loop
                } // End of Peacekeeper agent for loop
            }
            else if (grid[j][i].prevType == 3)
            {
                // Go through the whole squad to find the Local
                // Militia agent at (i, j)
                for (k = 0; k < LOCALNO; k++)
                {
                    if ((local[k].xPrev == i) &&
                                        (local[k].yPrev == j))
                    {
                        locCombat(pointGrid, pointLocal[k],
                                  pointPeace, pointSupport,
                                  pointLocal, pointCivilian,
                                  pointShot, pShotCount,
                                  currentT, subT);

                        arrayCheck(grid, 4);
                    } // End of if loop
                } // End of Insurgent agent for loop
            }
            else
```

```
                {
                    // Error message
                    cout << "Error in main.cpp, line 598. \n"
                         << "Combat called for agentType "
                         << grid[j][i].agentType << " at (" << i
                         << "," << j << ") \n";
                    system("PAUSE");
                    exit(1);
                } // End of if/else if/else loop
                break;
            case 2:
                // Insurgent agent sets off bomb if there are
                // valid targets within range
                // Go through the whole squad to find the Local
                // Militia agent at (i, j)
                for (k = 0; k < LOCALNO; k++)
                {
                    if ((local[k].xPrev == i) &&
                                          (local[k].yPrev == j))
                    {
                        locBomb(pointGrid, pointLocal[k],
                                pointPeace, pointSupport,
                                pointLocal, pointCivilian,
                                pointBomb, pBombCount, currentT,
                                subT);

                    } // End of if loop
                } // End of Insurgent agent for loop
                break;
            case 3:
                // No action needed
                break;
            case 4:
                // No action needed
                break;
            default:
                // Error message
                cout << "Error in main.cpp, line 629. \n"
                     << "Unknown agent type!\n";
                system("PAUSE");
                exit(1);
        } // End of actionType switch loop
    }
} // End of grid loop for determining combat

// Go through the entire grid in order to change the 'dead' agents'
// former location actionType to 0
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
```

```
        if (grid[j][i].agentType == 0)
        {
            pointGrid[j][i]->actionType == 0;
        } // End of if loop
    }
} // End of grid loop

// Update cell and agent parameters so the 'dead' agents are not
// considered when the movement function are called.  This should
// not affect cells without casualties.

// Set cell prevType and prevSquad values
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
        pointGrid[j][i]->prevSquad = grid[j][i].occSquad;
        pointGrid[j][i]->prevType = grid[j][i].agentType;
    }
} // End of grid loop

// Set xPrev and yPrev values for all the agents so that 'dead'
// agents are not on the grid
for (n = 0; n < PEACENO; n++)
{
    pointPeace[n]->xPrev = peace[n].xPos;
    pointPeace[n]->yPrev = peace[n].yPos;
}
for (n = 0; n < SUPPORTNO; n++)
{
    pointSupport[n]->xPrev = support[n].xPos;
    pointSupport[n]->yPrev = support[n].yPos;
}
for (n = 0; n < LOCALNO; n++)
{
    pointLocal[n]->xPrev = local[n].xPos;
    pointLocal[n]->yPrev = local[n].yPos;
}
for (n = 0; n < CIVNO; n++)
{
    pointCivilian[n]->xPrev = civilian[n].xPos;
    pointCivilian[n]->yPrev = civilian[n].yPos;
}

// Call movement functions for all agents at a cell with actionType
// value 4, and repair functions for all agents with actionType
// value 3
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
```

```
if (grid[j][i].actionType == 3)
{
    switch (grid[j][i].prevType)
    {
        case 0:
            // Do nothing
            break;
        case 1:
            // Call Peacekeeper repair function
            // Go through the whole squad to find the
            // Peacekeeper at (i, j)
            for (k = 0; k < PEACENO; k++)
            {
                if ((peace[k].xPrev == i) &&
                                        (peace[k].yPrev == j))
                {
                    peaceRepair(pointPeace[k],
                                            pointGrid[j][i]);
                }
            }
            break;
        case 2:
            // Call NGO repair function
            // Go through the whole squad to find the
            // NGO at (i, j)
            for (k = 0; k < SUPPORTNO; k++)
            {
                if ((support[k].xPrev == i) &&
                                      (support[k].yPrev == j))
                {
                    supportRepair(pointSupport[k],
                                            pointGrid[j][i]);
                }
            }
            break;
        default:
            // Error in agent type
            cout << "Error in main.cpp, line 713. \n"
                 << "Unknown agent type!\n";
            system("PAUSE");
            exit(1);
    }
}
else if (grid[j][i].actionType == 4)
{
    switch (grid[j][i].prevType)
    {
        case 0:
            // Do nothing
            break;
        case 1:
```

```
    // Peacekeeper
    for (k = 0; k < PEACENO; k++)
    {
        if ((peace[k].xPrev == i) &&
                            (peace[k].yPrev == j))
        {
            peaceMovement(pointGrid, pointPeace[k],
                        waterFailure, elecFailure);
        }
    }
    break;
case 2:
    // NGO
    for (k = 0; k < SUPPORTNO; k++)
    {
        if ((support[k].xPrev == i) &&
                           (support[k].yPrev == j))
        {
            suppMovement(pointGrid, pointSupport[k],
                        waterFailure, elecFailure);
        }
    }
    break;
case 3:
    // Insurgent
    for (k = 0; k < LOCALNO; k++)
    {
        if ((local[k].xPrev == i) &&
                            (local[k].yPrev == j))
        {
            locMovement(pointGrid, pointLocal[k],
                       waterFailure, elecFailure);
        }
    }
    break;
case 4:
    // Civilian
    for (k = 0; k < CIVNO; k++)
    {
        if ((civilian[k].xPrev == i) &&
                          (civilian[k].yPrev == j))
        {
            civMovement(pointGrid, pointCivilian[k],
                        waterFailure, elecFailure);
        }
    }
    break;
default:
    // Error
    cout << "Error in main.cpp, line 776. \n"
         << "Unknown agent type!\n";
```

```
                    system("PAUSE");
                    exit(1);
                } // End of prevType
            } // End of if/else if loop for actionType
        }
    } // End of grid loop for repairs and movement


    // Set cell prevType and prevSquad values and reset moveInd
    for (i = 0; i < GRIDSIZE; i++)
    {
        for (j = 0; j < GRIDSIZE; j++)
        {
            pointGrid[j][i]->prevSquad = grid[j][i].occSquad;
            pointGrid[j][i]->prevType = grid[j][i].agentType;
            pointGrid[j][i]->moveInd = 0;
        }
    } // End of grid loop

    // Set xPrev and yPrev values for all the agents in preparation for
    // the next sub-timestep
    for (n = 0; n < PEACENO; n++)
    {
        pointPeace[n]->xPrev = peace[n].xPos;
        pointPeace[n]->yPrev = peace[n].yPos;
    }
    for (n = 0; n < SUPPORTNO; n++)
    {
        pointSupport[n]->xPrev = support[n].xPos;
        pointSupport[n]->yPrev = support[n].yPos;
    }
    for (n = 0; n < LOCALNO; n++)
    {
        pointLocal[n]->xPrev = local[n].xPos;
        pointLocal[n]->yPrev = local[n].yPos;
    }
    for (n = 0; n < CIVNO; n++)
    {
        pointCivilian[n]->xPrev = civilian[n].xPos;
        pointCivilian[n]->yPrev = civilian[n].yPos;
    }

    // Go through entire grid and reset combat, bombBlast and shotInd
    // indicators to zero
    for (i = 0; i < GRIDSIZE; i++)
    {
        for (j = 0; j < GRIDSIZE; j++)
        {
            pointGrid[j][i]->combat = 0;
            pointGrid[j][i]->bombBlast = 0;
            pointGrid[j][i]->shotInd = 0;
```

```
    }
} // End of grid loop for resetting combat parameters

// Now set the combat, bombBlast and shotInd indicators using the
// arrays shotArray[][] and bombArray[][]
// First look at shotArray[][]
if (shotCount > 0)
{

    // Set counter
    m = shotCount - 1;

    // Go through appropriate rows of shotArray to set shotInd and
    // combat indicators
    while (shotArray[m][0] >= (currentT - SHOTMEMORY))
    {
        // Break out of while loop if the time gets to more than
        // SHOTMEMORY timesteps previous
        if ((shotArray[m][0] == (currentT - SHOTMEMORY)) &&
            (shotArray[m][1] < subT))
        {
            break;
        }
        // Otherwise set shotInd indicator at target cell and combat
        // indicator at target and origin cells
        else
        {
            pointGrid[(shotArray[m][7])][(shotArray[m][6])]->shotInd
                                                                = 1;
            pointGrid[(shotArray[m][7])][(shotArray[m][6])]->combat
                                                                = 1;
            pointGrid[(shotArray[m][3])][(shotArray[m][2])]->combat
                                                                = 1;
        } // End of if/else loop

        // Increment counter
        m--;

        // Check m >= 0, if not break out of while loop
        if (m < 0)
        {
            break;
        } // End of if loop

    } // End of while loop

} // End of if loop

// Now look at bombArray[][]
if (bombCount > 0)
{
```

```
    // Set counter
    m = bombCount - 1;

    // Go through appropriate rows of bombArray to set bombBlast and
    // combat indicators
    while (bombArray[m][0] >= (currentT - BOMBMEMORY))
    {
        // Break out of while loop if the time gets to more than
        // BOMBMEMORY timesteps previous
        if ((bombArray[m][0] == (currentT - BOMBMEMORY)) &&
            (bombArray[m][1] < subT))
        {
            break;
        }
        // Otherwise set bombBlast and combat indicators at affected
        // cells
        else
        {
            // Look through all cells affected by the bomb
            for (i = (bombArray[m][2] - bombArray[m][4]);
                 i <= (bombArray[m][2] + bombArray[m][4]); i++)
            {
                for (j = (bombArray[m][3] - bombArray[m][4]);
                     j <= (bombArray[m][3] + bombArray[m][4]); j++)
                {
                    // Check this cell is within the grid
                    if ((i >= 0) && (i < GRIDSIZE) && (j >= 0) &&
                        (j < GRIDSIZE))
                    {
                        pointGrid[j][i]->bombBlast = 1;
                        pointGrid[j][i]->combat = 1;
                    } // End of coordinate check if loop
                }
            } // End of target grid loop

        } // End of if/else loop

        // Increment counter
        m--;

        // Check m >= 0, if not break out of while loop
        if (m < 0)
        {
            break;
        } // End of if loop

    } // End of while loop

} // End of if loop
```

```
// Go through entire grid and adjust food amount at the cells
// that are occupied by Insurgent or Civilians
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
         if ((grid[j][i].agentType == 3) ||
             (grid[j][i].agentType == 4))
         {
             // Decrease amount by one unless there's no food
             if (grid[j][i].foodAmount > 0)
             {
                 (pointGrid[j][i]->foodAmount)--;
             } // End of food amount alteration loop
         } // End of agent check loop
    }
} // End of grid loop

// Go through entire grid and update water and electricity supply
// parameters if they have been fixed
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
        if (grid[j][i].fixedW > 0)
        {
            waterFailure[(grid[j][i].sectorNo)] = 0;
            pointGrid[j][i]->fixWater = 0;
            pointGrid[j][i]->fixedW = 0;
        } // End of if loop
        if (grid[j][i].fixedE > 0)
        {
            elecFailure[(grid[j][i].sectorNo)] = 0;
            pointGrid[j][i]->fixElec = 0;
            pointGrid[j][i]->fixedE = 0;
        } // End of if loop
    }
} // End of grid loop

// Update civInNeed indicator for the entire grid
for (i = 0; i < GRIDSIZE; i++)
{
    for (j = 0; j < GRIDSIZE; j++)
    {
        if (grid[j][i].agentType == 4)
        {
            if (grid[j][i].combat == 1)
            {
                pointGrid[j][i]->civInNeed = 1;
            }
            else if (waterFailure[grid[j][i].sectorNo] == 1)
```

```
                {
                    pointGrid[j][i]->civInNeed = 1;
                }
                else if (elecFailure[grid[j][i].sectorNo] == 1)
                {
                    pointGrid[j][i]->civInNeed = 1;
                }
                else if (grid[j][i].foodAmount == 0)
                {
                    pointGrid[j][i]->civInNeed = 1;
                }
                else
                {
                    pointGrid[j][i]->civInNeed = 0;
                } // End of parameter check if/else if/else loop
            } // End of (agentType = 4)
            else
            {
                pointGrid[j][i]->civInNeed = 0;
            } // End of agentType if/else loop
        }
    } // End of grid loop


} // End of sub-timestep for loop


// Determine whether or not the power or water supply to any of the
// sectors will fail
for (k = 0; k < sectorNumber; k++)
{
    // If the water supply is currently working, generate a random
    // number to determine whether or not it will fail
    if (waterFailure[k] == 0)
    {
        x = (float)rand()/32767;
        if ((WATERFAIL != 0) && (x <= WATERFAIL))
        {
            // Change the indicator in the array and randomly select a
            // cell in the sector where the fault is located
            waterFailure[k] = 1;
            // Generate two random integers between 0 and
            // (sectorSize - 1)
            y = (int)(rand()*sectorSize/32768);
            z = (int)(rand()*sectorSize/32768);
            // Compute the cell coordinates relating to these random
            // integers
            xFail = (int)(fmod((float)k, (float)SECTOR))*sectorSize + y;
            yFail = (int)(floor((float)(k/SECTOR)))*sectorSize + z;
            // Check (xFail, yFail) is within the array bounds
            if ((xFail >= GRIDSIZE) || (yFail >= GRIDSIZE) ||
```

```
                (xFail < 0) || (yFail < 0))
            {
                cout << "Error in repair location in main.cpp line 1007"
                     << " (" << xFail << "," << yFail << "). \n";
                system("PAUSE");
                exit(1);
            } // End of error check if loop
            pointGrid[yFail][xFail]->fixWater = 1;
            cout << "Water failure at (" << xFail << "," << yFail
                 << ") in sector " << k << " \n";

        } // End of waterFailure if loop
    } // End of water supply if loop
    // If the electricity supply is currently working, generate a random
    // number to determine whether or not it will fail
    if (elecFailure[k] == 0)
    {
        x = (float)rand()/32767;
        if ((ELECFAIL != 0) && (x <= ELECFAIL))
        {
            // Change the indicator in the array and randomly select a
            // cell in the sector where the fault is located
            elecFailure[k] = 1;
            // Generate two random integers between 0 and
            // (sectorSize - 1)
            y = (int)(rand()*sectorSize/32768);
            z = (int)(rand()*sectorSize/32768);
            // Compute the cell coordinates relating to these random
            // integers
            xFail = (int)(fmod((float)k, (float)SECTOR))*sectorSize + y;
            yFail = (int)(floor((float)(k/SECTOR)))*sectorSize + z;
            // Check (xFail, yFail) is within the array bounds
            if ((xFail >= GRIDSIZE) || (yFail >= GRIDSIZE) ||
                (xFail < 0) || (yFail < 0))
            {
                cout << "Error in repair location in main.cpp line 1037"
                     << " (" << xFail << "," << yFail << "). \n";
                system("PAUSE");
                exit(1);
            } // End of error check if loop
            pointGrid[yFail][xFail]->fixElec = 1;
            cout << "Electricity failure at (" << xFail << "," << yFail
                 << ") in sector " << k << " \n";

        } // End of elecFailure if loop
    } // End of electricity supply if loop
} // End of sector for loop

// Write data to file

gridoutput.open("gridoutput.txt", ios::app);
```

```
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        gridoutput << grid[j][i].agentType << " ";
    }
    gridoutput << "\n";
} // End of grid loop
gridoutput << "\n";
gridoutput.close();

combatind.open("combatind.txt", ios::app);
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        combatind << grid[j][i].combat << " ";
    }
    combatind << "\n";
} // End of grid loop
combatind << "\n";
combatind.close();

civinneedind.open("civinneedind.txt", ios::app);
for (j = 0; j < GRIDSIZE; j++)
{
    for (i = 0; i < GRIDSIZE; i++)
    {
        civinneedind << grid[j][i].civInNeed << " ";
    }
    civinneedind << "\n";
} // End of grid loop
civinneedind << "\n";
civinneedind.close();

waterfailind.open("waterfailind.txt", ios::app);
for (k = 0; k < sectorNumber; k++)
{
    waterfailind << waterFailure[k] << " ";
} // End of array loop
waterfailind << "\n";
waterfailind.close();

elecfailind.open("elecfailind.txt", ios::app);
for (k = 0; k < sectorNumber; k++)
{
    elecfailind << elecFailure[k] << " ";
} // End of array loop
elecfailind << "\n";
elecfailind.close();
```

```
    } // end of timestep for loop


    // Write shotArray and bombArray to file (if they exist)
    if (shotCount > 0)
    {
        ofstream shotoutput;
        shotoutput.open("shotoutput.txt", ios::app);
        for (k = 0; k < shotCount; k++)
        {
            shotoutput << shotArray[k][0] << " " << shotArray[k][1] << " "
                       << shotArray[k][2] << " " << shotArray[k][3] << " "
                       << shotArray[k][4] << " " << shotArray[k][5] << " "
                       << shotArray[k][6] << " " << shotArray[k][7] << " "
                       << shotArray[k][8] << " " << shotArray[k][9] << "\n";
        }
        shotoutput.close();
    }

    if (bombCount > 0)
    {
        ofstream bomboutput;
        bomboutput.open("bomboutput.txt", ios::app);
        for (k = 0; k < bombCount; k++)
        {
            bomboutput << bombArray[k][0] << " " << bombArray[k][1] << " "
                       << bombArray[k][2] << " " << bombArray[k][3] << " "
                       << bombArray[k][4] << " " << bombArray[k][5] << "\n";
        }
        bomboutput.close();
    }

    // Close output files
    gridoutput.close();
    combatind.close();
    civinneedind.close();
    waterfailind.close();
    elecfailind.close();

    // Wait until user is ready before terminating program to allow the user to
    // see the program results
    system("PAUSE");
    return 0;

} // End of main


// FUNCTIONS

// Action indicator function, determines whether or not an agent performs an
// action at a given sub-timestep.
```

```
int actionIndicator(int subTimestep, int subStep, int squadCapability)
{
    int action;
    double frac;
    frac = (double)subTimestep/(double)subStep;
    if ((fmod((double)subTimestep, (double)subStep) == 0) &&
                                                    (frac <= squadCapability))
    {
        action = 1;
    }
    else
    {
        action = 0;
    }
    return action;
} // End of actionIndicator function

// Array checker function, determines whether there are any agent types or
// previous agent types that are not valid
void arrayCheck(Cell gridArray[GRIDSIZE][GRIDSIZE], int maxValue)
{
    int i, j;           // Declare counters
    for (i = 0; i < GRIDSIZE; i++)
    {
        for (j = 0; j < GRIDSIZE; j++)
        {
            if ((gridArray[j][i].prevType) > maxValue)
            {
                cout << "prevType value " << gridArray[j][i].prevType << " at ("
                     << i << "," << j << ") \n";
            } // End of if loop
            if ((gridArray[j][i].agentType) > maxValue)
            {
                cout << "agentType value " << gridArray[j][i].agentType
                     << " at (" << i << "," << j << ") \n";
            } // End of if loop
        }
    } // End of grid loop
} // End of arrayCheck function
```

B.2 model.h

```
// model.h - contains all function prototypes and #defines

#ifndef _MODEL_    // Check if header file has already been called to
#define _MODEL_    // avoid multiple definitions


#define RUNTIME 500     // Number of timesteps the model will run for
```

```
#define MAXSHOTS 5000   // Maximum number of shots that can be fired
                        // over the model run
#define MAXBOMB 100     // Maximum number of bomb attacks per model
                        // run

// Function prototypes

// Prototype for action indicator function - determines whether or not
// an agent is to perform an action at a given sub-timestep
int actionIndicator(int, int, int);

// Prototype for array checking function - determines whether the grid
// contains any agent types that are not valid
void arrayCheck(Cell[GRIDSIZE][GRIDSIZE], int);

// Prototypes for initial.cpp
void peaceInitPos(short[GRIDSIZE][GRIDSIZE], Peacekeeper*);
void suppInitPos(short[GRIDSIZE][GRIDSIZE], SupportAgent*);
void locInitPos(short[GRIDSIZE][GRIDSIZE], LocalMilitia*);
void civInitPos(short[GRIDSIZE][GRIDSIZE], Civilian*);

// Prototypes for move.cpp
double perChange(double, double);
void civMovement(Cell*[GRIDSIZE][GRIDSIZE], Civilian*,
                 int[SECTOR*SECTOR], int[SECTOR*SECTOR]);
void suppMovement(Cell*[GRIDSIZE][GRIDSIZE], SupportAgent*,
                  int[SECTOR*SECTOR], int[SECTOR*SECTOR]);
void locMovement(Cell*[GRIDSIZE][GRIDSIZE], LocalMilitia*,
                 int[SECTOR*SECTOR], int[SECTOR*SECTOR]);
void peaceMovement(Cell*[GRIDSIZE][GRIDSIZE], Peacekeeper*,
                   int[SECTOR*SECTOR], int [SECTOR*SECTOR]);

// Prototypes for combat.cpp
void peaceCombat(Cell*[GRIDSIZE][GRIDSIZE], Peacekeeper*,
                 Peacekeeper*[PEACENO], SupportAgent*[SUPPORTNO],
                 LocalMilitia*[LOCALNO], Civilian*[CIVNO],
                 short*[MAXSHOTS][10], int*, int, int);
void locCombat(Cell*[GRIDSIZE][GRIDSIZE], LocalMilitia*,
               Peacekeeper*[PEACENO], SupportAgent*[SUPPORTNO],
               LocalMilitia*[LOCALNO], Civilian*[CIVNO],
               short*[MAXSHOTS][10], int*, int, int);
void locBomb(Cell*[GRIDSIZE][GRIDSIZE], LocalMilitia*,
             Peacekeeper*[PEACENO], SupportAgent*[SUPPORTNO],
             LocalMilitia*[LOCALNO], Civilian*[CIVNO],
             short*[MAXBOMB][6], int*, int, int);

// Prototypes for repair.cpp
void peaceRepair(Peacekeeper*, Cell*);
void supportRepair(SupportAgent*, Cell*);
```

```
#endif
```

B.3 agent.cpp

```
// agent.cpp
// Gives the member functions for the agent classes

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"        // Include the header file for the Cell class
#include "agent.h"       // Include agent classes header file
#include "model.h"       // Include general header file

using namespace std;

// Peacekeeper constructor
Peacekeeper::Peacekeeper()
{
    squadNo = 1;
    agentType = 1;
    xHome = 99;
    yHome = 14;
    homeRadius = 10;
    xPos = 0;
    xPrev = 0;
    yPos = 0;
    yPrev = 0;
    alive = 1;
    peaceWeight[0] = 0;
    peaceWeight[1] = 0;
    peaceWeight[2] = 0;
    peaceWeight[3] = 0;
    peaceWeight[4] = 0;
    suppWeight[0] = 0;
    suppWeight[1] = 0;
    suppWeight[2] = 0;
    suppWeight[3] = 0;
    suppWeight[4] = 0;
    locWeight[0] = 0;
    locWeight[1] = 0;
    locWeight[2] = 0;
    locWeight[3] = 0;
    locWeight[4] = -50;
    civWeight[0] = 0;
```

```
    civWeight[1] = 0;
    civWeight[2] = 0;
    civWeight[3] = 0;
    civWeight[4] = 0;
    sensorRange = 200;
    relation[0] = 1;
    relation[1] = 1;
    relation[2] = 5;
    relation[3] = 1;

    civInNeedWeight = 10;
    noWaterWeight = 10;
    noElecWeight = 10;
    combatWeight = 0;
    sSKP = 0.10;
    fireRange = 20;
    shotRadius = 0;
    probFixWater = 1.00;
    probFixElec = 1.00;
} // End of Peacekeeper constructor

// Peacekeeper destructor
Peacekeeper::~Peacekeeper()
{

} // End of Peacekeeper destructor


// NGO constructor
SupportAgent::SupportAgent()
{
    squadNo = 2;
    agentType = 2;
    xHome = 14;
    yHome = 99;
    homeRadius = 10;
    xPos = 0;
    xPrev = 0;
    yPos = 0;
    yPrev = 0;
    alive = 1;
    peaceWeight[0] = 0;
    peaceWeight[1] = 0;
    peaceWeight[2] = 0;
    peaceWeight[3] = 0;
    peaceWeight[4] = 0;
    suppWeight[0] = 0;
    suppWeight[1] = 0;
    suppWeight[2] = 0;
    suppWeight[3] = 0;
    suppWeight[4] = 0;
```

```
    locWeight[0] = 0;
    locWeight[1] = 0;
    locWeight[2] = 0;
    locWeight[3] = -100;
    locWeight[4] = 0;
    civWeight[0] = 0;
    civWeight[1] = 0;
    civWeight[2] = 0;
    civWeight[3] = 0;
    civWeight[4] = 0;
    sensorRange = 200;
    relation[0] = 1;
    relation[1] = 1;
    relation[2] = 4;
    relation[3] = 1;

    civInNeedWeight = 10;
    noWaterWeight = 10;
    noElecWeight = 10;
    probFixWater = 1.00;
    probFixElec = 1.00;
} // End of NGO constructor

// NGO destructor
SupportAgent::~SupportAgent()
{

} // End of NGO destructor


// Insurgent constructor
LocalMilitia::LocalMilitia()
{
    squadNo = 3;
    agentType = 3;
    xHome = 112;
    yHome = 112;
    homeRadius = 87;
    xPos = 0;
    xPrev = 0;
    yPos = 0;
    yPrev = 0;
    alive = 1;
    peaceWeight[0] = 0;
    peaceWeight[1] = 0;
    peaceWeight[2] = 0;
    peaceWeight[3] = 0;
    peaceWeight[4] = 0;
    suppWeight[0] = 0;
    suppWeight[1] = 0;
    suppWeight[2] = 0;
```

```
    suppWeight[3] = 0;
    suppWeight[4] = 0;
    locWeight[0] = 0;
    locWeight[1] = 0;
    locWeight[2] = 0;
    locWeight[3] = 0;
    locWeight[4] = 0;
    civWeight[0] = 0;
    civWeight[1] = 0;
    civWeight[2] = 0;
    civWeight[3] = 0;
    civWeight[4] = 0;
    sensorRange = 50;
    relation[0] = 5;
    relation[1] = 4;
    relation[2] = 1;
    relation[3] = 1;

    combatWeight = 0;
    sSKP = 0.05;
    fireRange = 15;
    shotRadius = 0;
    bombRadius = 5;
    accOccForce = 0.0;
} // End of Insurgent constructor

// Insurgent destructor
LocalMilitia::~LocalMilitia()
{

} // End of Insurgent destructor

// Civilian constructor
Civilian::Civilian()
{
    squadNo = 4;
    agentType = 4;
    xHome = 112;
    yHome = 112;
    homeRadius = 87;
    xPos = 0;
    xPrev = 0;
    yPos = 0;
    yPrev = 0;
    alive = 1;
    peaceWeight[0] = 0;
    peaceWeight[1] = 0;
    peaceWeight[2] = 0;
    peaceWeight[3] = 0;
    peaceWeight[4] = 0;
    suppWeight[0] = 0;
```

```
    suppWeight[1] = 0;
    suppWeight[2] = 0;
    suppWeight[3] = 0;
    suppWeight[4] = 0;
    locWeight[0] = 0;
    locWeight[1] = 0;
    locWeight[2] = 0;
    locWeight[3] = 0;
    locWeight[4] = 0;
    civWeight[0] = 0;
    civWeight[1] = 0;
    civWeight[2] = 0;
    civWeight[3] = 0;
    civWeight[4] = 0;
    sensorRange = 50;
    relation[0] = 2;
    relation[1] = 1;
    relation[2] = 4;
    relation[3] = 1;

    accOccForce = 0.0;
    fear = 0.0;
} // End of Civilian constructor

// Civilian destructor
Civilian::~Civilian()
{

} // End of Civilian destructor
```

B.4 agent.h

```
// agent.h
// Defines the class Agent

#define NOOFSQUADS 4

// Since at present we can only have one squad per agent type we can
// also define the squad sizes and capabilities

// Squad sizes
#define PEACENO 25
#define SUPPORTNO 25
#define LOCALNO 50
#define CIVNO 1000

// Squad capabilities.
#define PEACECAP 1
```

```
#define SUPPORTCAP 1
#define LOCALCAP 1
#define CIVCAP 1

// We define the length of time (in timesteps) a bomb blast and shot
// are "remembered", ie the time the bomb and shot indicators are
// flagged for
#define BOMBMEMORY 20
#define SHOTMEMORY 10

// Also since we only have one Insurgent squad we can define their
// probability of bombing and probability of firing (without
// provocation)
#define LMBOMBPROB 0.002
#define LMFIREPROB 0.01

// First we define the peacekeepers

#ifndef _PEACE_                    // Check to see if already defined
#define _PEACE_
class Peacekeeper
{
    public:

        // CONSTANTS

        // Basic agent properties
        short squadNo;             // Squad number
        short agentType;           // Type of agent: 1 = Peacekeeper,
                                   // 2 = NGO, 3 = Insurgent,
                                   // 4 = Civilian
        short xHome;               // x coordinate for home location
        short yHome;               // y coordinate for home location
        short homeRadius;          // Radius for initial agent
                                   // locations
        short xPos;                // x coordinate of current location
        short xPrev;               // x coordinate of location at
                                   // previous (sub-)timestep
        short yPos;                // y coordinate of current location
        short yPrev;               // y coordinate of location at
                                   // previous (sub-)timestep
        short alive;               // Indicates when an agent gets
                                   // killed, a value of 1 indicates
                                   // the agent is alive, this changes
                                   // to 0 if the agent is killed

        // Define the personality weights
        short peaceWeight[5];      // Weights towards Peacekeepers:
                                   // first entry is for friendly,
                                   // second for cooperative, third
                                   // for neutral, fourth for
```

```
                                // uncooperative, fifth for hostile

        short suppWeight[5];    // Weights towards NGOs

        short locWeight[5];     // Weights towards Insurgents

        short civWeight[5];     // Weights towards Civilians

        // Ranges and constraints
        short sensorRange;      // Sensor range

        // Psychological factors
        short relation[NOOFSQUADS];     // Relationships to all the
                                        // other squads

        // Additional personality weights
        short civInNeedWeight;  // Weight towards civilians in need
        short noWaterWeight;    // Weight towards cells with no
                                // water
        short noElecWeight;     // Weight towards cells with no
                                // electricity
        short combatWeight;     // Weight towards cells with combat

        // Additional ranges and constraints
        float sSKP;             // Single shot kill probability
        short fireRange;        // Firing range
        short shotRadius;       // Radius of fire, > 0 allows for
                                // friendly fire
        float probFixWater;     // Probability the agent can fix
                                // the water supply if at relevant
                                // cell, value between 0 and 1
        float probFixElec;      // Probability the agent can fix
                                // the water supply if at relevant
                                // cell, value between 0 and 1

        // FUNCTION PROTOTYPES

        Peacekeeper();          // Constructor
        ~Peacekeeper();         // Destructor
};

#endif

// Next we have the NGOs

#ifndef _SUPPORT_               // Check to see if already defined
#define _SUPPORT_

class SupportAgent
{
    public:
```

```
// CONSTANTS

// Basic agent properties
short squadNo;              // Squad number
short agentType;            // Type of agent: 1 = Peacekeeper,
                            // 2 = NGO, 3 = Insurgent,
                            // 4 = Civilian
short xHome;                // x coordinate for home location
short yHome;                // y coordinate for home location
short homeRadius;           // Radius for initial agent
                            // locations
short xPos;                 // x coordinate of current location
short xPrev;                // x coordinate of location at
                            // previous (sub-)timestep
short yPos;                 // y coordinate of current location
short yPrev;                // y coordinate of location at
                            // previous (sub-)timestep
short alive;                // Indicates when an agent gets
                            // killed, a value of 1 indicates
                            // the agent is alive, this changes
                            // to 0 if the agent is killed

// Define the personality weights
short peaceWeight[5];       // Weights towards Peacekeepers:
                            // first entry is for friendly,
                            // second for cooperative, third
                            // for neutral, fourth for
                            // uncooperative, fifth for hostile

short suppWeight[5];        // Weights towards NGOs

short locWeight[5];         // Weights towards Insurgents

short civWeight[5];         // Weights towards Civilians

// Ranges and constraints
short sensorRange;          // Sensor range

// Psychological factors
short relation[NOOFSQUADS];     // Relationships to all the
                                // other squads

// Additional personality weights
short civInNeedWeight;      // Weight towards civilians in need
short noWaterWeight;        // Weight towards cells with no
                            // water
short noElecWeight;         // Weight towards cells with no
                            // electricity

// Additional ranges and constraints
```

```
        float probFixWater;        // Probability the agent can fix
                                   // the water supply if at relevant
                                   // cell, value between 0 and 1
        float probFixElec;         // Probability the agent can fix
                                   // the water supply if at relevant
                                   // cell, value between 0 and 1

        // FUNCTION PROTOTYPES

        SupportAgent();            // Constructor
        ~SupportAgent();           // Destructor
};

#endif

// Insurgent agents

#ifndef _LOCAL_                    // Check to see if already defined
#define _LOCAL_

class LocalMilitia
{
    public:

        // CONSTANTS

        // Basic agent properties
        short squadNo;             // Squad number
        short agentType;           // Type of agent: 1 = Peacekeeper,
                                   // 2 = NGO, 3 = Insurgent,
                                   // 4 = Civilian
        short xHome;               // x coordinate for home location
        short yHome;               // y coordinate for home location
        short homeRadius;          // Radius for initial agent
                                   // locations
        short xPos;                // x coordinate of current location
        short xPrev;               // x coordinate of location at
                                   // previous (sub-)timestep
        short yPos;                // y coordinate of current location
        short yPrev;               // y coordinate of location at
                                   // previous (sub-)timestep
        short alive;               // Indicates when an agent gets
                                   // killed, a value of 1 indicates
                                   // the agent is alive, this changes
                                   // to 0 if the agent is killed

        // Define the personality weights
        short peaceWeight[5];      // Weights towards Peacekeepers:
                                   // first entry is for friendly,
                                   // second for cooperative, third
                                   // for neutral, fourth for
```

```
                                // uncooperative, fifth for hostile

        short suppWeight[5];    // Weights towards NGOs

        short locWeight[5];     // Weights towards Insurgents

        short civWeight[5];     // Weights towards Civilians

        // Ranges and constraints
        short sensorRange;      // Sensor range

        // Psychological factors
        short relation[NOOFSQUADS];     // Relationships to all the
                                        // other squads

        // Additional personality weights
        short combatWeight;     // Weight towards cells with combat

        // Additional ranges and constraints
        float sSKP;             // Single shot kill probability
        short fireRange;        // Firing range
        short shotRadius;       // Radius of fire, > 0 allows for
                                // friendly fire
        short bombRadius;       // Radius of effect for bomb blasts

        // VARIABLES

        // Psychological factors
        float accOccForce;      // Acceptance of occupying force

        // FUNCTION PROTOTYPES

        LocalMilitia();         // Constructor
        ~LocalMilitia();        // Destructor

};

#endif

// Finally we have the civilian agents

#ifndef _CIVILIAN_              // Check to see if already defined
#define _CIVILIAN_

class Civilian
{
    public:

        // CONSTANTS

        // Basic agent properties
```

```
short squadNo;              // Squad number
short agentType;            // Type of agent: 1 = Peacekeeper,
                            // 2 = NGO, 3 = Insurgent,
                            // 4 = Civilian
short xHome;                // x coordinate for home location
short yHome;                // y coordinate for home location
short homeRadius;           // Radius for initial agent
                            // locations
short xPos;                 // x coordinate of current location
short xPrev;                // x coordinate of location at
                            // previous (sub-)timestep
short yPos;                 // y coordinate of current location
short yPrev;                // y coordinate of location at
                            // previous (sub-)timestep
short alive;                // Indicates when an agent gets
                            // killed, a value of 1 indicates
                            // the agent is alive, this changes
                            // to 0 if the agent is killed

// Define the personality weights
short peaceWeight[5];       // Weights towards Peacekeepers:
                            // first entry is for friendly,
                            // second for cooperative, third
                            // for neutral, fourth for
                            // uncooperative, fifth for hostile

short suppWeight[5];        // Weights towards NGOs

short locWeight[5];         // Weights towards Insurgents

short civWeight[5];         // Weights towards Civilians

// Ranges and constraints
short sensorRange;          // Sensor range

// Psychological factors
short relation[NOOFSQUADS];     // Relationships to all the
                                // other squads (constant in
                                // first prototype but will be
                                // variable in future models)


// Psychological factors
float accOccForce;          // Acceptance of occupying force
float fear;                 // Fear

// FUNCTION PROTOTYPES

Civilian();                 // Constructor
~Civilian();                // Destructor
```

```
};

#endif
```

B.5 cell.cpp

```
// cell.cpp
// Gives the member functions for the Cell class

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"        // Include the header file for the Cell class
#include "agent.h"       // Include agent classes header file
#include "model.h"       // Include general header file

using namespace std;

// Constructor
Cell::Cell()
{
    sectorNo = 0;
    // Note that the grid is initialised before the agents so the
    // initial occupant settings will be zero
    occSquad = 0;
    prevSquad = 0;
    agentType = 0;
    prevType = 0;
    foodAmount = 1000;
    actionType = 0;
    moveInd = 0;
    fixWater = 0;
    fixedW = 0;
    fixElec = 0;
    fixedE = 0;
    combat = 0;
    shotToNo = 0;
    shotInd = 0;
    bombBlast = 0;
    civInNeed = 0;
} // End of Cell constructor

// Destructor
Cell::~Cell()
{
```

```
} // End of Cell destructor
```

B.6 cell.h

```
// cell.h
// Defines the Cell class

#ifndef _CELL_              // Avoid multiple definitions
#define _CELL_

#define GRIDSIZE 200        // Side-length of grid
#define SECTOR 5            // Number of sectors each side is divided
                            // into (i.e. square root of total number
                            // of sectors) Make sure GRIDSIZE is
                            // divisible by SECTOR so we have equal
                            // sector sizes.
#define WATERFAIL 0.000
#define ELECFAIL 0.000

// Define the cell class which gives all the properties a cell will
// have

class Cell
{
    public:

        // CONSTANTS

        // Basic properties
        short sectorNo;           // Which sector the cell is
                                  // situated in

        // VARIABLES

        // General variables
        short occSquad;           // Squad the occupant belongs to,
                                  // 0 if there is no agent at the
                                  // cell
        short prevSquad;          // Squad the occupant at the last
                                  // (sub-)timestep belonged to
        short agentType;          // Type of agent the occupant is,
                                  // 0 if there is no agent at the
                                  // cell
        short prevType;           // Type of agent that was occupying
                                  // the cell at the last
                                  // (sub-)timestep
        short foodAmount;         // Number of rations at cell (one
                                  // ration feeds one local agent for
```

```
                                // one sub-timestep)
        short actionType;       // Indicates the action to be taken
                                // by the occupying agent at the
                                // current sub-timestep
        short moveInd;          // Indicates whether or not an
                                // agent has moved to the cell in
                                // that (sub-)timestep

        // General indicator functions
        short fixWater;         // Repairs needed to water supply
                                // indicator (0 or 1, where 1
                                // represents repairs needed at the
                                // cell)
        short fixedW;           // Indicates if the water supply
                                // has been fixed at a sub-timestep
        short fixElec;          // Repairs needed to electricity
                                // supply indicator (0 or 1, where
                                // 1 represents repairs needed at
                                // cell)
        short fixedE;           // Indicates if the electricity
                                // supply has been fixed at a
                                // sub-timestep

        // Combat variables
        short combat;           // Combat indicator (0 or 1, where
                                // 1 indicates combat at the cell)
        short shotToNo;         // Number of shots fired to the
                                // cell
        short shotInd;          // Indicates whether or not there
                                // have been any shots fired to the
                                // cell over the course of the last
                                // timestep
        short bombBlast;        // Bomb detonation indicator (0
                                // or 1, where 1 indicates a bomb
                                // has been exploded at the cell)

        // Psychological factors
        short civInNeed;        // Civilians in need indicator (0
                                // or 1, where 1 indicates that
                                // there are civilians in need at
                                // the cell

        // FUNCTION PROTOTYPES

        // Constructor and destructor
        Cell();                             // Constructor
        ~Cell();                            // Destructor

};
```

```
#endif
```

B.7 combat.cpp

```
// combat.cpp gives the combat functions for the peacekeepers and
// insurgent agents.  This is called in the main program whenever an
// agent's action is set to combat.  The functions pick out a target,
// and if there is none within sensor range the action is changed to
// movement.

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"        // Include the header file for the Cell class
#include "agent.h"       // Include agent classes header file
#include "model.h"       // Include general header file

using namespace std;

// Peacekeeping agents' combat function
void peaceCombat(Cell *pGrid[GRIDSIZE][GRIDSIZE], Peacekeeper *pPeace,
                 Peacekeeper *pPeaceArray[PEACENO],
                 SupportAgent *pSupportArray[SUPPORTNO],
                 LocalMilitia *pLocalArray[LOCALNO],
                 Civilian *pCivilianArray[CIVNO],
                 short *pShotArray[MAXSHOTS][10], int *pShotCount,
                 int timestep, int subTime)
{
    short x = pPeace->xPrev;        // Set x and y variables to
    short y = pPeace->yPrev;        // represent the agent's x and y
                                    // positions respectively

    // First check the shots fired to the current cell to see where
    // they're coming from
    // Count the number of shots fired at the cell in the last
    // timestep
    int i, j, k, n;              // Declare counters
    int target[500][2];          // Declare array to hold valid target
                                 // locations
    int tar = 0;                 // Initialise counter for number of
                                 // valid targets
    int tarNo;
    short xTar;                  // Variables to hold target coordinates
    short yTar;
    short rel;                   // Variables to hold details of possible
    short squad;                 // target agents
```

```
short aType;
short shotAtArray[500][4];   // Declare array to hold details of shots
                             // fired to the cell
int count = 0;               // Initialise variable to count number of
                             // shots
float randNo;
// Check to see if any shots have been fired to the cell and if so count
// them
if (pGrid[y][x]->shotToNo > 0)
{
    n = (*pShotCount) - 1;    // Initialise loop counter
    while (*pShotArray[n][0] >= (timestep - 1))
    {
        if ((*pShotArray[n][0] == timestep) &&
            (*pShotArray[n][1] != subTime) && (*pShotArray[n][6] == x) &&
            (*pShotArray[n][7] == y))
        {
            shotAtArray[count][0] = *pShotArray[n][2];
            shotAtArray[count][1] = *pShotArray[n][3];
            shotAtArray[count][2] = *pShotArray[n][4];
            shotAtArray[count][3] = *pShotArray[n][5];
            count++;
        }
        else if ((*pShotArray[n][0] == (timestep - 1)) &&
                 (*pShotArray[n][1] >= subTime) && (*pShotArray[n][6] == x)
                 && (*pShotArray[n][7] == y))
        {
            shotAtArray[count][0] = *pShotArray[n][2];
            shotAtArray[count][1] = *pShotArray[n][3];
            shotAtArray[count][2] = *pShotArray[n][4];
            shotAtArray[count][3] = *pShotArray[n][5];
            count++;
        } // End of if/else if loop

        n--;    // Increment loop counter

        // Check to make sure n >= 0, if not break out of while loop
        if (n < 0)
        {
            break;
        } // End of if loop


    } // End of while loop

} // End of if loop

// If there have been no shots fired then the peacekeeper should look for
// valid insurgent targets within firing range, for the count to be zero
// and combat initiated means a bomb must have been detonated within sensor
// range
```

```
if (count == 0)
{
    // Count the number of valid targets
    for (i = (x - pPeace->fireRange); i <= (x + pPeace->fireRange); i++)
    {
        for (j = (y - pPeace->fireRange); j <= (y + pPeace->fireRange); j++)
        {
            // Check this cell is within the grid
            if ((i >= 0) && (i < GRIDSIZE) && (j >= 0) && (j < GRIDSIZE))
            {
                // Check to see if there is an insurgent agent at the
                // cell
                if (pGrid[j][i]->prevType == 3)
                {
                    // Record the relationship to the insurgent agent
                    // at this cell
                    rel = pPeace->relation[(pGrid[j][i]->prevSquad) - 1];
                    // If the insurgent agent is hostile or
                    // uncooperative then record it as a valid target
                    if ((rel == 5) || (rel == 4))
                    {
                        target[tar][0] = i;
                        target[tar][1] = j;
                        tar++;
                    } // End of relationship if loop
                } // End of prevType if loop

            } // End of coordinate check if loop

        }
    } // End of grid loop

    // If there are no valid targets within range change the action to move
    if (tar == 0)
    {
        pGrid[y][x]->actionType = 4;
    } // End of no target loop
    // Else choose a target to fire at
    else
    {
        // Generate a random integer between 0 and (tar - 1)
        tarNo = (int)(rand()*tar/32768);
        // Set target location
        xTar = target[tarNo][0];
        yTar = target[tarNo][1];
        // Change shotArray
        // Check there haven't already been MAXSHOTS shots fired (array can
        // only hold MAXSHOTS rows)
        if (*pShotCount < MAXSHOTS)
        {
            *pShotArray[*pShotCount][0] = timestep;
```

```
    *pShotArray[*pShotCount][1] = subTime;
    *pShotArray[*pShotCount][2] = x;
    *pShotArray[*pShotCount][3] = y;
    *pShotArray[*pShotCount][4] = 1;
    *pShotArray[*pShotCount][5] = pPeace->squadNo;
    *pShotArray[*pShotCount][6] = xTar;
    *pShotArray[*pShotCount][7] = yTar;
    *pShotArray[*pShotCount][8] = 3;
    *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
    (*pShotCount)++;
    (pGrid[yTar][xTar]->shotToNo)++;
}
else
{
    // Exit program
    cout << "Error: Too many shots fired, array bounds exceeded.";
    system("PAUSE");
    exit(1);
} // End of if/else loop

// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;      // Generate a random number
                                   // between 0 and 1
// If they are killed then change the target agent's status
if ((pPeace->sSKP != 0) && (randNo <= (pPeace->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status and cell values
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
                   pPeaceArray[k]->alive = 0;
                   pPeaceArray[k]->xPos = GRIDSIZE;
                   pPeaceArray[k]->yPos = GRIDSIZE;
                   pGrid[yTar][xTar]->occSquad = 0;
                   pGrid[yTar][xTar]->agentType = 0;
                   // Write details to file
                   ofstream casualties;    // Declare file to write
                                           // output to
                   casualties.open("casualties.txt", ios::app);
                   casualties << "peace[" << k << "] killed at ("
                              << xTar << "," << yTar
```

```
                        << ") at timestep " << timestep
                        << ", subtime " << subTime << "\n";
                casualties.close();
                // Write details to screen
                cout << "peace[" << k << "] killed at (" << xTar
                     << "," << yTar << ") \n";
            } // End of Peacekeeper coordinates if loop
        } // End of Peacekeeper for loop
        break;
    case 2:
        // Go through NGO array to find the relevant
        // agent and then change agent status
        for (k = 0; k < SUPPORTNO; k++)
        {
            if ((pSupportArray[k]->xPrev == xTar) &&
                (pSupportArray[k]->yPrev == yTar))
            {
                pSupportArray[k]->alive = 0;
                pSupportArray[k]->xPos = GRIDSIZE;
                pSupportArray[k]->yPos = GRIDSIZE;
                pGrid[yTar][xTar]->occSquad = 0;
                pGrid[yTar][xTar]->agentType = 0;
                // Write details to file
                ofstream casualties;    // Declare file to write
                                        // output to
                casualties.open("casualties.txt", ios::app);
                casualties << "support[" << k << "] killed at ("
                           << xTar << "," << yTar
                           << ") at timestep " << timestep
                           << ", subtime " << subTime << "\n";
                casualties.close();
                // Write details to screen
                cout << "support[" << k << "] killed at ("
                     << xTar << "," << yTar << ") \n";
            } // End of NGO coordinates if loop
        } // End of NGO for loop
        break;
    case 3:
        // Go through insurgent array to find the relevant
        // agent and then change agent status
        for (k = 0; k < LOCALNO; k++)
        {
            if ((pLocalArray[k]->xPrev == xTar) &&
                (pLocalArray[k]->yPrev == yTar))
            {
                pLocalArray[k]->alive = 0;
                pLocalArray[k]->xPos = GRIDSIZE;
                pLocalArray[k]->yPos = GRIDSIZE;
                pGrid[yTar][xTar]->occSquad = 0;
                pGrid[yTar][xTar]->agentType = 0;
                // Write details to file
```

```
                    ofstream casualties;    // Declare file to write
                                            // output to
                    casualties.open("casualties.txt", ios::app);
                    casualties << "local[" << k << "] killed at ("
                               << xTar << "," << yTar
                               << ") at timestep " << timestep
                               << ", subtime " << subTime << "\n";
                    casualties.close();
                    // Write details to screen
                    cout << "local[" << k << "] killed at (" << xTar
                         << "," << yTar << ") \n";
                } // End of insurgent coordinates if loop
            } // End of insurgent for loop
            break;
        case 4:
            // Go through Civilian array to find the relevant
            // agent and then change agent status
            for (k = 0; k < CIVNO; k++)
            {
                if ((pCivilianArray[k]->xPrev == xTar) &&
                    (pCivilianArray[k]->yPrev == yTar))
                {
                    pCivilianArray[k]->alive = 0;
                    pCivilianArray[k]->xPos = GRIDSIZE;
                    pCivilianArray[k]->yPos = GRIDSIZE;
                    pGrid[yTar][xTar]->occSquad = 0;
                    pGrid[yTar][xTar]->agentType = 0;
                    // Write details to file
                    ofstream casualties;    // Declare file to write
                                            // output to
                    casualties.open("casualties.txt", ios::app);
                    casualties << "civilian[" << k << "] killed at ("
                               << xTar << "," << yTar
                               << ") at timestep " << timestep
                               << ", subtime " << subTime << "\n";
                    casualties.close();
                    // Write details to screen
                    cout << "civilian[" << k << "] killed at ("
                         << xTar << "," << yTar << ") \n";
                } // End if Civilian coordinates if loop
            } // End of Civilian for loop
            break;
        default:
            // Error message
            cout << "Unknown agent type "
                 << pGrid[yTar][xTar]->prevType
                 << " in combat.cpp line 377!\n";
            system("PAUSE");
            exit(1);
    } // End of prevType switch loop
} // End of target killed if loop
```

```
    } // End of target loop.  End of if/else loop

} // End of (count == 0) loop
// If there's only one agent firing at the peacekeeper check that it's a
// valid target, and if so fire at that cell
else if (count == 1)
{
    // Check pGrid[y][x]->shotToNo > 0
    if ((pGrid[y][x]->shotToNo) <= 0)
    {
        cout << "Error in counting targets in combat.cpp line 422. \n";
        system("PAUSE");
        exit(1);
    } // End of error check if loop
    // Check the relationship to the occupying agent of that cell
    squad = shotAtArray[0][3];
    rel = pPeace->relation[squad - 1];
    aType = shotAtArray[0][2];
    // If the relationship is hostile or uncooperative, and the agent is
    // insurgent, shoot at that cell
    if ((aType == 3) && ((rel == 4) || (rel == 5)))
    {
        xTar = shotAtArray[0][0];
        yTar = shotAtArray[0][1];
        // Check xTar and yTar are within array bounds
        if ((xTar < 0) || (xTar >= GRIDSIZE) || (yTar < 0) ||
                                                    (yTar >= GRIDSIZE))
        {
            cout << "Error in target coordinates in combat.cpp line 433, "
                 << "values outside array bounds. \n";
            system("PAUSE");
            exit(1);
        } // End of error check if loop

        // Change shotArray
        // Check there haven't already been MAXSHOTS shots fired (array can
        // only hold MAXSHOTS rows)
        if (*pShotCount < MAXSHOTS)
        {
            *pShotArray[*pShotCount][0] = timestep;
            *pShotArray[*pShotCount][1] = subTime;
            *pShotArray[*pShotCount][2] = x;
            *pShotArray[*pShotCount][3] = y;
            *pShotArray[*pShotCount][4] = 1;
            *pShotArray[*pShotCount][5] = pPeace->squadNo;
            *pShotArray[*pShotCount][6] = xTar;
            *pShotArray[*pShotCount][7] = yTar;
            *pShotArray[*pShotCount][8] = 3;
            *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
            (*pShotCount)++;
```

```
    (pGrid[yTar][xTar]->shotToNo)++;
}
else
{
    // Exit program
    cout << "Error: Too many shots fired, array bounds exceeded.";
    system("PAUSE");
    exit(1);
} // End of if/else loop

// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;          // Generate a random number
                                       // between 0 and 1
// If they are killed then change target agent status
if ((pPeace->sSKP != 0) && (randNo <= (pPeace->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
                   pPeaceArray[k]->alive = 0;
                   pPeaceArray[k]->xPos = GRIDSIZE;
                   pPeaceArray[k]->yPos = GRIDSIZE;
                   pGrid[yTar][xTar]->occSquad = 0;
                   pGrid[yTar][xTar]->agentType = 0;
                   // Write details to file
                   ofstream casualties;    // Declare file to write
                                           // output to
                   casualties.open("casualties.txt", ios::app);
                   casualties << "peace[" << k << "] killed at ("
                              << xTar << "," << yTar
                              << ") at timestep " << timestep
                              << ", subtime " << subTime << "\n";
                   casualties.close();
                   // Write details to screen
                   cout << "peace[" << k << "] killed at (" << xTar
                        << "," << yTar << ") \n";
               } // End of Peacekeeper coordinates if loop
           } // End of Peacekeeper for loop
           break;
       case 2:
```

```
    // Go through NGO array to find the relevant
    // agent and then change agent status
    for (k = 0; k < SUPPORTNO; k++)
    {
        if ((pSupportArray[k]->xPrev == xTar) &&
            (pSupportArray[k]->yPrev == yTar))
        {
            pSupportArray[k]->alive = 0;
            pSupportArray[k]->xPos = GRIDSIZE;
            pSupportArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of NGO coordinates if loop
    } // End of NGO for loop
    break;
case 3:
    // Go through insurgent array to find the relevant
    // agent and then change agent status
    for (k = 0; k < LOCALNO; k++)
    {
        if ((pLocalArray[k]->xPrev == xTar) &&
            (pLocalArray[k]->yPrev == yTar))
        {
            pLocalArray[k]->alive = 0;
            pLocalArray[k]->xPos = GRIDSIZE;
            pLocalArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "local[" << k << "] killed at (" << xTar
```

```
                                << "," << yTar << ") \n";
                        } // End of insurgent coordinates if loop
                    } // End of insurgent for loop
                    break;
                case 4:
                    // Go through Civilian array to find the relevant
                    // agent and then change agent status
                    for (k = 0; k < CIVNO; k++)
                    {
                        if ((pCivilianArray[k]->xPrev == xTar) &&
                            (pCivilianArray[k]->yPrev == yTar))
                        {
                            pCivilianArray[k]->alive = 0;
                            pCivilianArray[k]->xPos = GRIDSIZE;
                            pCivilianArray[k]->yPos = GRIDSIZE;
                            pGrid[yTar][xTar]->occSquad = 0;
                            pGrid[yTar][xTar]->agentType = 0;
                            // Write details to file
                            ofstream casualties;    // Declare file to write
                                                    // output to
                            casualties.open("casualties.txt", ios::app);
                            casualties << "civilian[" << k << "] killed at ("
                                       << xTar << "," << yTar
                                       << ") at timestep " << timestep
                                       << ", subtime " << subTime << "\n";
                            casualties.close();
                            // Write details to screen
                            cout << "civilian[" << k << "] killed at ("
                                 << xTar << "," << yTar << ") \n";
                        } // End of Civilian coordinates if loop
                    } // End of Civilian for loop
                    break;
                default:
                    // Error message
                    cout << "Unknown agent type "
                         << pGrid[yTar][xTar]->prevType
                         << " in combat.cpp line 569!\n";
                    system("PAUSE");
                    exit(1);
                } // End of prevType switch loop
            } // End of target killed if loop

        } // End of target choice loop
        // If the agent is not insurgent and/or the relationship is not
        // uncooperative or hostile then change the peacekeepers action to move
        else
        {
             pGrid[y][x]->actionType = 4;
        } // End of no target loop.  End of if/else loop

    } // End of (count == 1) loop
```

```
// If there was more than one shot fired at the cell, choose a target at
// random from them
else if (count > 1)
{
    // Generate a random integer between 0 and (count - 1) to determine
    // which target to aim for
    tarNo = (int)(rand()*count/32768);
    // Check 0 <= tarNo <= (count - 1)
    if ((tarNo < 0) || (tarNo >= count))
    {
        cout << "Error in calculating target in combat.cpp line 456. \n";
        system ("PAUSE");
        exit(1);
    }
    // Check tarNo < shotToNo
    else if (tarNo >= (pGrid[y][x]->shotToNo))
    {
        cout << "tarNo = " << tarNo << ", count = " << count
             << ", shotToNo = " << pGrid[y][x]->shotToNo
             << ", currentT = " << timestep << ", x = " << x << ", y = "
             << y << "\n";
        cout << "Error in counting targets in combat.cpp line 463. \n";
        // Output shotArray to file
        ofstream shotoutput;
        shotoutput.open("shotoutput.txt", ios::app);
        for (k = 0; k < *pShotCount; k++)
        {
            shotoutput << *pShotArray[k][0] << " " << *pShotArray[k][1] << " "
                       << *pShotArray[k][2] << " " << *pShotArray[k][3] << " "
                       << *pShotArray[k][4] << " " << *pShotArray[k][5] << " "
                       << *pShotArray[k][6] << " " << *pShotArray[k][7] << " "
                       << *pShotArray[k][8] << " " << *pShotArray[k][9] << "\n";
        }
        shotoutput.close();
        system("PAUSE");
        exit(1);
    } // End of error check if/else if loop
    // Check the relationship to the occupying agent of that cell
    squad = shotAtArray[tarNo][3];
    rel = pPeace->relation[squad - 1];
    aType = shotAtArray[tarNo][2];
    // If the relationship is hostile or uncooperative, and the agent is
    // insurgent, shoot at that cell
    if ((aType == 3) && ((rel == 4) || (rel == 5)))
    {
        xTar = shotAtArray[tarNo][0];
        yTar = shotAtArray[tarNo][1];
        // Check xTar and yTar are within array bounds
        if ((xTar < 0) || (xTar >= GRIDSIZE) || (yTar < 0) ||
                                                    (yTar >= GRIDSIZE))
        {
```

```
    cout << "Error in target coordinates in combat.cpp line 483, "
         << "values outside array bounds. \n";
    system("PAUSE");
    exit(1);
} // End of error check if loop
// Change shotArray
// Check there haven't already been MAXSHOTS shots fired (array can
// only hold MAXSHOTS rows)
if (*pShotCount < MAXSHOTS)
{
    *pShotArray[*pShotCount][0] = timestep;
    *pShotArray[*pShotCount][1] = subTime;
    *pShotArray[*pShotCount][2] = x;
    *pShotArray[*pShotCount][3] = y;
    *pShotArray[*pShotCount][4] = 1;
    *pShotArray[*pShotCount][5] = pPeace->squadNo;
    *pShotArray[*pShotCount][6] = xTar;
    *pShotArray[*pShotCount][7] = yTar;
    *pShotArray[*pShotCount][8] = 3;
    *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
    (*pShotCount)++;
    (pGrid[yTar][xTar]->shotToNo)++;
}
else
{
    // Exit program
    cout << "Error: Too many shots fired, array bounds exceeded.";
    system("PAUSE");
    exit(1);
} // End of if/else loop

// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;     // Generate a random number
                                  // between 0 and 1
// If they are killed then change the target agent's status
if ((pPeace->sSKP != 0) && (randNo <= (pPeace->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
```

```
            pPeaceArray[k]->alive = 0;
            pPeaceArray[k]->xPos = GRIDSIZE;
            pPeaceArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "peace[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "peace[" << k << "] killed at (" << xTar
                 << "," << yTar << ") \n";
        } // End of Peacekeeper coordinates if loop
    } // End of Peacekeeper for loop
    break;
case 2:
    // Go through NGO array to find the relevant
    // agent and then change agent status
    for (k = 0; k < SUPPORTNO; k++)
    {
        if ((pSupportArray[k]->xPrev == xTar) &&
            (pSupportArray[k]->yPrev == yTar))
        {
            pSupportArray[k]->alive = 0;
            pSupportArray[k]->xPos = GRIDSIZE;
            pSupportArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of NGO coordinates if loop
    } // End of NGO for loop
    break;
case 3:
    // Go through insurgent array to find the relevant
    // agent and then change agent status
```

```
    for (k = 0; k < LOCALNO; k++)
    {
        if ((pLocalArray[k]->xPrev == xTar) &&
            (pLocalArray[k]->yPrev == yTar))
        {
            pLocalArray[k]->alive = 0;
            pLocalArray[k]->xPos = GRIDSIZE;
            pLocalArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "local[" << k << "] killed at (" << xTar
                 << "," << yTar << ") \n";
        } // End of insurgent coordinates if loop
    } // End of insurgent for loop
    break;
case 4:
    // Go through Civilian array to find the relevant
    // agent and then change agent status
    for (k = 0; k < CIVNO; k++)
    {
        if ((pCivilianArray[k]->xPrev == xTar) &&
            (pCivilianArray[k]->yPrev == yTar))
        {
            pCivilianArray[k]->alive = 0;
            pCivilianArray[k]->xPos = GRIDSIZE;
            pCivilianArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "civilian[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "civilian[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of Civilian coordinates if loop
```

```
                    } // End of Civilian for loop
                    break;
                default:
                    // Error message
                    cout << "Unknown agent type "
                         << pGrid[yTar][xTar]->prevType
                         << "in combat.cpp line 624!\n";
                    system("PAUSE");
                    exit(1);
            }
        }
    } // End of target choice loop
    // If the agent is not insurgent and/or the relationship is not
    // uncooperative or hostile then change the peacekeepers action to move
    else
    {
         pGrid[y][x]->actionType = 4;
    } // End of no target loop.  End of if/else loop

} // End of (count > 1) loop.  End of if/else if loop

} // End of peaceCombat function


// Local milita agents' combat function
void locCombat(Cell *pGrid[GRIDSIZE][GRIDSIZE], LocalMilitia *pLoc,
               Peacekeeper *pPeaceArray[PEACENO],
               SupportAgent *pSupportArray[SUPPORTNO],
               LocalMilitia *pLocalArray[LOCALNO],
               Civilian *pCivilianArray[CIVNO],
               short *pShotArray[MAXSHOTS][10], int *pShotCount,
               int timestep, int subTime)
{
    short x = pLoc->xPrev;           // Set x and y variables to represent the
    short y = pLoc->yPrev;           // agent's x and y positions respectively

    // First check the shots fired to the current cell to see where they're
    // coming from
    // Count the number of shots fired at the cell in the last timestep
    int i, j, k, n;                  // Declare counters
    int target[500][2];              // Declare array to hold valid target
                                     // locations
    int tar = 0;                     // Initialise counter for number of valid
                                     // targets
    int tarNo;
    short xTar;                      // Variables to hold target coordinates
    short yTar;
    short rel;                       // Variables will hold details of possible
    short squad;                     // target agents
    short aType;
    short shotAtArray[500][4];       // Declare array to hold details of shots
```

```
                                // fired to the cell
int count = 0;                  // Initialise variable to count number of
                                // shots
float randNo;
// Check to see if any shots have been fired to the cell and if so count
// them
if (pGrid[y][x]->shotToNo > 0)
{
    n = *pShotCount - 1;    // Initialise loop counter

    while (*pShotArray[n][0] >= (timestep - 1))
    {

        if ((*pShotArray[n][0] == timestep) &&
            (*pShotArray[n][1] != subTime) && (*pShotArray[n][6] == x) &&
            (*pShotArray[n][7] == y))
        {
            shotAtArray[count][0] = *pShotArray[n][2];
            shotAtArray[count][1] = *pShotArray[n][3];
            shotAtArray[count][2] = *pShotArray[n][4];
            shotAtArray[count][3] = *pShotArray[n][5];
            count++;
        }
        else if ((*pShotArray[n][0] == (timestep - 1)) &&
                 (*pShotArray[n][1] >= subTime) && (*pShotArray[n][6] == x)
                 && (*pShotArray[n][7] == y))
        {
            shotAtArray[count][0] = *pShotArray[n][2];
            shotAtArray[count][1] = *pShotArray[n][3];
            shotAtArray[count][2] = *pShotArray[n][4];
            shotAtArray[count][3] = *pShotArray[n][5];
            count++;
        } // End of if/else if loop

        n--;    // Increment loop counter

        // Check to make sure n >= 0, if not break out of while loop
        if (n < 0)
        {
            break;
        } // End of if loop

    } // End of while loop

} // End of if loop

// If there have been no shots fired at the cell then the agent must have
// been chosen to fire at a random target within firing range.
if (count == 0)
{
    // Check to see if there is a valid target within firing range
```

```
for (i = (x - pLoc->fireRange); i <= (x + pLoc->fireRange); i++)
{
    for (j = (y - pLoc->fireRange); j < (y + pLoc->fireRange); j++)
    {
        // Check this cell is within the grid
        if ((i >= 0) && (i < GRIDSIZE) && (j >= 0) && (j < GRIDSIZE))
        {
            // Check to see if there is an agent at the cell
            if (pGrid[j][i]->prevType != 0)
            {
                // Record the relationship to the agent at this cell
                rel = pLoc->relation[(pGrid[j][i]->prevSquad) - 1];
                // If the agent is hostile or uncooperative then record it
                // as a valid target
                if ((rel == 5) || (rel == 4))
                {
                    target[tar][0] = i;
                    target[tar][1] = j;
                    tar++;
                } // End of relationship check if loop
            } // End of occupying agent check if loop
        } // End of coordinate check if loop
    }
} // End of firing range target search loop

// If there are no valid targets within range change the action to move
if (tar == 0)
{
    pGrid[y][x]->actionType = 4;
} // End of no target loop
// Else choose a target to fire at
else
{
    // Generate a random integer between 0 and (tar - 1)
    tarNo = (int)(rand()*tar/32768);
    // Set target location
    xTar = target[tarNo][0];
    yTar = target[tarNo][1];
    // Change shotArray
    // Check there haven't already been MAXSHOTS shots fired (array can
    // only hold MAXSHOTS rows)
    if (*pShotCount < MAXSHOTS)
    {
        *pShotArray[*pShotCount][0] = timestep;
        *pShotArray[*pShotCount][1] = subTime;
        *pShotArray[*pShotCount][2] = x;
        *pShotArray[*pShotCount][3] = y;
        *pShotArray[*pShotCount][4] = 3;
        *pShotArray[*pShotCount][5] = pLoc->squadNo;
        *pShotArray[*pShotCount][6] = xTar;
        *pShotArray[*pShotCount][7] = yTar;
```

```
    *pShotArray[*pShotCount][8] = pGrid[yTar][xTar]->prevType;
    *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
    (*pShotCount)++;
    (pGrid[yTar][xTar]->shotToNo)++;
}
else
{
    // Exit program
    cout << "Error: Too many shots fired, array bounds exceeded.";
    system("PAUSE");
    exit(1);
} // End of if/else loop

// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;      // Generate a random number
                                   // between 0 and 1
// If they are killed then change the target agent's status
if ((pLoc->sSKP != 0) && (randNo <= (pLoc->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
                   pPeaceArray[k]->alive = 0;
                   pPeaceArray[k]->xPos = GRIDSIZE;
                   pPeaceArray[k]->yPos = GRIDSIZE;
                   pGrid[yTar][xTar]->occSquad = 0;
                   pGrid[yTar][xTar]->agentType = 0;
                   // Write details to file
                   ofstream casualties;    // Declare file to write
                                           // output to
                   casualties.open("casualties.txt", ios::app);
                   casualties << "peace[" << k << "] killed at ("
                              << xTar << "," << yTar
                              << ") at timestep " << timestep
                              << ", subtime " << subTime << "\n";
                   casualties.close();
                   // Write details to screen
                   cout << "peace[" << k << "] killed at (" << xTar
                        << "," << yTar << ") \n";
               } // End of Peacekeeper coordinates if loop
```

```
    } // End of Peacekeeper for loop
    break;
case 2:
    // Go through NGO array to find the relevant
    // agent and then change agent status
    for (k = 0; k < SUPPORTNO; k++)
    {
        if ((pSupportArray[k]->xPrev == xTar) &&
            (pSupportArray[k]->yPrev == yTar))
        {
            pSupportArray[k]->alive = 0;
            pSupportArray[k]->xPos = GRIDSIZE;
            pSupportArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of NGO coordinates if loop
    } // End of NGO for loop
    break;
case 3:
    // Go through insurgent array to find the relevant
    // agent and then change agent status
    for (k = 0; k < LOCALNO; k++)
    {
        if ((pLocalArray[k]->xPrev == xTar) &&
            (pLocalArray[k]->yPrev == yTar))
        {
            pLocalArray[k]->alive = 0;
            pLocalArray[k]->xPos = GRIDSIZE;
            pLocalArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
```

```
                        casualties.close();
                        // Write details to screen
                        cout << "local[" << k << "] killed at (" << xTar
                             << "," << yTar << ") \n";
                    } // End of insurgent coordinates if loop
                } // End of insurgent for loop
                break;
            case 4:
                // Go through Civilian array to find the relevant
                // agent and then change agent status
                for (k = 0; k < CIVNO; k++)
                {
                    if ((pCivilianArray[k]->xPrev == xTar) &&
                        (pCivilianArray[k]->yPrev == yTar))
                    {
                        pCivilianArray[k]->alive = 0;
                        pCivilianArray[k]->xPos = GRIDSIZE;
                        pCivilianArray[k]->yPos = GRIDSIZE;
                        pGrid[yTar][xTar]->occSquad = 0;
                        pGrid[yTar][xTar]->agentType = 0;
                        // Write details to file
                        ofstream casualties;    // Declare file to write
                                                // output to
                        casualties.open("casualties.txt", ios::app);
                        casualties << "civilian[" << k << "] killed at ("
                                   << xTar << "," << yTar
                                   << ") at timestep " << timestep
                                   << ", subtime " << subTime << "\n";
                        casualties.close();
                        // Write details to screen
                        cout << "civilian[" << k << "] killed at ("
                             << xTar << "," << yTar << ") \n";
                    } // End of Civilian coordinates if loop
                } // End of Civilian for loop
                break;
            default:
                // Error message
                cout << "Unknown agent type "
                     << pGrid[yTar][xTar]->prevType
                     << "in combat.cpp line 1117!\n";
                system("PAUSE");
                exit(1);
            } // End of prevType switch loop
        } // End of target killed if loop

    } // End of target loop.  End of if/else loop
} // End of (count == 0) loop
// If there was one shot fired at the cell, fire back unless the
// relationship to the occupant agent is not hostile or uncooperative
else if (count == 1)
{
```

```
// Check pGrid[y][x]->shotToNo > 0
if ((pGrid[y][x]->shotToNo) <= 0)
{
    cout << "Error in counting targets in combat.cpp line 1213. \n";
    system("PAUSE");
    exit(1);
} // End of error check if loop
// Check the relationship to the occupying agent of that cell
squad = shotAtArray[0][3];
rel = pLoc->relation[squad - 1];
aType = shotAtArray[0][2];
// If the relationship is hostile or uncooperative shoot at that cell
if ((rel == 4) || (rel == 5))
{
    xTar = shotAtArray[0][0];
    yTar = shotAtArray[0][1];
    // Check xTar and yTar are within array bounds
    if ((xTar < 0) || (xTar >= GRIDSIZE) || (yTar < 0) ||
                                                    (yTar >= GRIDSIZE))
    {
        cout << "Error in target coordinates in combat.cpp line 1216, "
             << "values outside array bounds. \n";
        system("PAUSE");
        exit(1);
    } // End of error check if loop
    // Change shotArray
    // Check there haven't already been MAXSHOTS shots fired (array can
    // only hold MAXSHOTS rows)
    if (*pShotCount < MAXSHOTS)
    {
        *pShotArray[*pShotCount][0] = timestep;
        *pShotArray[*pShotCount][1] = subTime;
        *pShotArray[*pShotCount][2] = x;
        *pShotArray[*pShotCount][3] = y;
        *pShotArray[*pShotCount][4] = 3;
        *pShotArray[*pShotCount][5] = pLoc->squadNo;
        *pShotArray[*pShotCount][6] = xTar;
        *pShotArray[*pShotCount][7] = yTar;
        *pShotArray[*pShotCount][8] = pGrid[yTar][xTar]->prevType;
        *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
        (*pShotCount)++;
        (pGrid[yTar][xTar]->shotToNo)++;
    }
    else
    {
        // Exit program
        cout << "Error: Too many shots fired, array bounds exceeded.";
        system("PAUSE");
        exit(1);
    } // End of if/else loop
```

```
// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;      // Generate a random number
                                   // between 0 and 1
// If they are killed then change the target agent's status
if ((pLoc->sSKP != 0) && (randNo <= (pLoc->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
                   pPeaceArray[k]->alive = 0;
                   pPeaceArray[k]->xPos = GRIDSIZE;
                   pPeaceArray[k]->yPos = GRIDSIZE;
                   pGrid[yTar][xTar]->occSquad = 0;
                   pGrid[yTar][xTar]->agentType = 0;
                   // Write details to file
                   ofstream casualties;    // Declare file to write
                                           // output to
                   casualties.open("casualties.txt", ios::app);
                   casualties << "peace[" << k << "] killed at ("
                              << xTar << "," << yTar
                              << ") at timestep " << timestep
                              << ", subtime " << subTime << "\n";
                   casualties.close();
                   // Write details to screen
                   cout << "peace[" << k << "] killed at (" << xTar
                        << "," << yTar << ") \n";
               } // End of Peacekeeper coordinates if loop
           } // End of Peacekeeper for loop
           break;
       case 2:
           // Go through NGO array to find the relevant
           // agent and then change agent status
           for (k = 0; k < SUPPORTNO; k++)
           {
               if ((pSupportArray[k]->xPrev == xTar) &&
                   (pSupportArray[k]->yPrev == yTar))
               {
                   pSupportArray[k]->alive = 0;
                   pSupportArray[k]->xPos = GRIDSIZE;
                   pSupportArray[k]->yPos = GRIDSIZE;
```

```
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of NGO coordinates if loop
    } // End of NGO for loop
    break;
case 3:
    // Go through insurgent array to find the relevant
    // agent and then change agent status
    for (k = 0; k < LOCALNO; k++)
    {
        if ((pLocalArray[k]->xPrev == xTar) &&
            (pLocalArray[k]->yPrev == yTar))
        {
            pLocalArray[k]->alive = 0;
            pLocalArray[k]->xPos = GRIDSIZE;
            pLocalArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "local[" << k << "] killed at (" << xTar
                 << "," << yTar << ") \n";
        } // End of insurgent coordinates if loop
    } // End of insurgent for loop
    break;
case 4:
    // Go through Civilian array to find the relevant
    // agent and then change agent status
    for (k = 0; k < CIVNO; k++)
    {
        if ((pCivilianArray[k]->xPrev == xTar) &&
```

```
                        (pCivilianArray[k]->yPrev == yTar))
                    {
                        pCivilianArray[k]->alive = 0;
                        pCivilianArray[k]->xPos = GRIDSIZE;
                        pCivilianArray[k]->yPos = GRIDSIZE;
                        pGrid[yTar][xTar]->occSquad = 0;
                        pGrid[yTar][xTar]->agentType = 0;
                        // Write details to file
                        ofstream casualties;    // Declare file to write
                                                // output to
                        casualties.open("casualties.txt", ios::app);
                        casualties << "civilian[" << k << "] killed at ("
                                   << xTar << "," << yTar
                                   << ") at timestep " << timestep
                                   << ", subtime " << subTime << "\n";
                        casualties.close();
                        // Write details to screen
                        cout << "civilian[" << k << "] killed at ("
                             << xTar << "," << yTar << ") \n";
                    } // End of Civilian coordinates if loop
                } // End of Civilian for loop
                break;
            default:
                // Error message
                cout << "Unknown agent type "
                     << pGrid[yTar][xTar]->prevType
                     << " in combat.cpp line 1319!\n";
                system("PAUSE");
                exit(1);
            } // End of prevType switch loop
        } // End of target killed loop
    } // End of relationship check loop
    // If the relationship is not uncooperative or hostile then change the
    // action to move
    else
    {
        pGrid[y][x]->actionType = 4;
    } // End of no shooting loop.  End of if/else loop
} // End of (count == 1) loop
// If there were more than one shots fired at the cell, choose a target at
// random from them
else if (count > 1)
{
    // Generate a random integer between 0 and (count - 1) to determine
    // which target to aim for
    tarNo = (int)(rand()*count/32768);
    // Check 0 <= tarNo <= (count - 1)
    if ((tarNo < 0) || (tarNo >= count))
    {
        cout << "Error in calculating target in combat.cpp line 1400. \n";
        system ("PAUSE");
```

```
    exit(1);
}
// Check tarNo < shotToNo
else if (tarNo >= (pGrid[y][x]->shotToNo))
{
    cout << "Error in counting targets in combat.cpp line 1407. \n";
    system("PAUSE");
    exit(1);
} // End of error check if/else if loop
// Check the relationship to the occupying agent of that cell
squad = shotAtArray[tarNo][3];
rel = pLoc->relation[squad - 1];
aType = shotAtArray[tarNo][2];
// If the relationship is hostile or uncooperative shoot at that cell
if ((rel == 4) || (rel == 5))
{
    xTar = shotAtArray[tarNo][0];
    yTar = shotAtArray[tarNo][1];
    // Check xTar and yTar are within array bounds
    if ((xTar < 0) || (xTar >= GRIDSIZE) || (yTar < 0) ||
                                                    (yTar >= GRIDSIZE))
    {
        cout << "Error in target coordinates in combat.cpp line 1410, "
             << "values outside array bounds. \n";
        system("PAUSE");
        exit(1);
    } // End of error check if loop
    // Change shotArray
    // Check there haven't already been MAXSHOTS shots fired (array can
    // only hold MAXSHOTS rows)
    if (*pShotCount < MAXSHOTS)
    {
        *pShotArray[*pShotCount][0] = timestep;
        *pShotArray[*pShotCount][1] = subTime;
        *pShotArray[*pShotCount][2] = x;
        *pShotArray[*pShotCount][3] = y;
        *pShotArray[*pShotCount][4] = 3;
        *pShotArray[*pShotCount][5] = pLoc->squadNo;
        *pShotArray[*pShotCount][6] = xTar;
        *pShotArray[*pShotCount][7] = yTar;
        *pShotArray[*pShotCount][8] = pGrid[yTar][xTar]->prevType;
        *pShotArray[*pShotCount][9] = pGrid[yTar][xTar]->prevSquad;
        (*pShotCount++);
        (pGrid[yTar][xTar]->shotToNo)++;
    }
    else
    {
        // Exit program
        cout << "Error: Too many shots fired, array bounds exceeded.";
        system("PAUSE");
        exit(1);
```

```
} // End of if/else loop

// Now determine whether or not the target agent is killed
randNo = (float)rand()/32767;    // Generate a random number
                                 // between 0 and 1
// If they are killed then change the target agent's status
if ((pLoc->sSKP != 0) && (randNo <= (pLoc->sSKP)))
{
    switch (pGrid[yTar][xTar]->prevType)
    {
       case 0:
           // No agent at cell - has either moved or been killed -
           // therefore the shot will have no effect
           break;
       case 1:
           // Go through Peacekeeper array to find the relevant
           // agent and then change agent status
           for (k = 0; k < PEACENO; k++)
           {
               if ((pPeaceArray[k]->xPrev == xTar) &&
                   (pPeaceArray[k]->yPrev == yTar))
               {
                   pPeaceArray[k]->alive = 0;
                   pPeaceArray[k]->xPos = GRIDSIZE;
                   pPeaceArray[k]->yPos = GRIDSIZE;
                   pGrid[yTar][xTar]->occSquad = 0;
                   pGrid[yTar][xTar]->agentType = 0;
                   // Write details to file
                   ofstream casualties;    // Declare file to write
                                           // output to
                   casualties.open("casualties.txt", ios::app);
                   casualties << "peace[" << k << "] killed at ("
                              << xTar << "," << yTar
                              << ") at timestep " << timestep
                              << ", subtime " << subTime << "\n";
                   casualties.close();
                   // Write details to screen
                   cout << "peace[" << k << "] killed at (" << xTar
                        << "," << yTar << ") \n";
               } // End of Peacekeeper coordinates if loop
           } // End of Peacekeeper if loop
           break;
       case 2:
           // Go through NGO array to find the relevant
           // agent and then change agent status
           for (k = 0; k < SUPPORTNO; k++)
           {
               if ((pSupportArray[k]->xPrev == xTar) &&
                   (pSupportArray[k]->yPrev == yTar))
               {
                   pSupportArray[k]->alive = 0;
```

```
            pSupportArray[k]->xPos = GRIDSIZE;
            pSupportArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << k << "] killed at ("
                 << xTar << "," << yTar << ") \n";
        } // End of NGO coordinates if loop
    } // End of NGO for loop
    break;
case 3:
    // Go through insurgent array to find the relevant
    // agent and then change agent status
    for (k = 0; k < LOCALNO; k++)
    {
        if ((pLocalArray[k]->xPrev == xTar) &&
            (pLocalArray[k]->yPrev == yTar))
        {
            pLocalArray[k]->alive = 0;
            pLocalArray[k]->xPos = GRIDSIZE;
            pLocalArray[k]->yPos = GRIDSIZE;
            pGrid[yTar][xTar]->occSquad = 0;
            pGrid[yTar][xTar]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to write
                                    // output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << k << "] killed at ("
                       << xTar << "," << yTar
                       << ") at timestep " << timestep
                       << ", subtime " << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "local[" << k << "] killed at (" << xTar
                 << "," << yTar << ") \n";
        } // End of Insurgent coordinates if loop
    } // End of Insurgent for loop
    break;
case 4:
    // Go through Civilian array to find the relevant
    // agent and then change agent status
    for (k = 0; k < CIVNO; k++)
```

```
                        {
                            if ((pCivilianArray[k]->xPrev == xTar) &&
                                (pCivilianArray[k]->yPrev == yTar))
                            {
                                pCivilianArray[k]->alive = 0;
                                pCivilianArray[k]->xPos = GRIDSIZE;
                                pCivilianArray[k]->yPos = GRIDSIZE;
                                pGrid[yTar][xTar]->occSquad = 0;
                                pGrid[yTar][xTar]->agentType = 0;
                                // Write details to file
                                ofstream casualties;    // Declare file to write
                                                        // output to
                                casualties.open("casualties.txt", ios::app);
                                casualties << "civilian[" << k << "] killed at ("
                                           << xTar << "," << yTar
                                           << ") at timestep " << timestep
                                           << ", subtime " << subTime << "\n";
                                casualties.close();
                                // Write details to screen
                                cout << "civilian[" << k << "] killed at ("
                                     << xTar << "," << yTar << ") \n";
                            } // End of Civilian coordinates if loop
                        } // End of Civilian for loop
                        break;
                    default:
                        // Error message
                        cout << "Unknown agent type "
                             << pGrid[yTar][xTar]->prevType
                             << " in combat.cpp line 1481!\n";
                        system("PAUSE");
                        exit(1);
                } // End of prevType switch loop
            } // End of target killed if loop
        } // End of target selection loop
        // If the relationship is not uncooperative or hostile then change the
        // action to move
        else
        {
            pGrid[y][x]->actionType = 4;
        } // End of no target loop.  End of if/else loop
    } // End of (count > 1) loop.  End of if/else if loop
} // End of locCombat function


// Insurgent bomb function
void locBomb(Cell *pGrid[GRIDSIZE][GRIDSIZE], LocalMilitia *pLoc,
             Peacekeeper *pPeaceArray[PEACENO],
             SupportAgent *pSupportArray[SUPPORTNO],
             LocalMilitia *pLocalArray[LOCALNO],
             Civilian *pCivilianArray[CIVNO],  short *pBombArray[MAXBOMB][6],
             int *pBombCount, int timestep, int subTime)
```

```
{
    short x = pLoc->xPrev;        // Set x and y variables to represent the
    short y = pLoc->yPrev;        // agent's x and y positions respectively
    // Check there are hostile or uncooperative agents within bomb range
    int tar = 0;                // Initialise counter
    short rel;
    int i, j;                   // Declare grid counters
    int n;                      // Declare array counter
    // Count the number of valid targets
    for (i = (x - pLoc->bombRadius); i <= (x + pLoc->bombRadius); i++)
    {
        for (j = (y - pLoc->bombRadius); j <= (y + pLoc->bombRadius); j++)
        {
            // Check this cell is within the grid
            if ((i >= 0) && (i < GRIDSIZE) && (j >= 0) && (j < GRIDSIZE))
            {
                // Check there is an agent at the cell
                if (pGrid[j][i]->prevSquad != 0)
                {
                    // Record the relationship to the agent at this cell
                    rel = pLoc->relation[(pGrid[j][i]->prevSquad) - 1];
                    // If the agent is hostile or uncooperative then record it
                    // as a valid target
                    if ((rel == 5) || (rel == 4))
                    {
                        tar++;
                    } // End of relationship check if loop
                } // End of occupying agent check if loop
            } // End of coordinate check if loop
        }
    } // End of bomb radius grid loop
    // If there are no valid targets within bomb range then change the agent's
    // action to move
    if (tar == 0)
    {
        pGrid[y][x]->actionType = 4;
    }
    // Else set off the bomb
    else
    {
        cout << "Bomb set off at (" << x << "," << y << ") \n";
        // Update bombArray
        // Check there haven't already been MAXBOMB bombs (array can
        // only hold MAXBOMB rows)
        if (*pBombCount < MAXBOMB)
        {
            *pBombArray[*pBombCount][0] = timestep;
            *pBombArray[*pBombCount][1] = subTime;
            *pBombArray[*pBombCount][2] = x;
            *pBombArray[*pBombCount][3] = y;
            *pBombArray[*pBombCount][4] = pLoc->bombRadius;
```

```
    *pBombArray[*pBombCount][5] = pLoc->squadNo;
    (*pBombCount)++;
}
else
{
    // Exit program
    cout << "Error: Too many bombs, array bounds exceeded.";
    system("PAUSE");
    exit(1);
} // End of if/else loop

// Go through the cells affected by the bomb
for (i = (x - pLoc->bombRadius); i <= (x + pLoc->bombRadius); i++)
{
    for (j = (y - pLoc->bombRadius); j <= (y + pLoc->bombRadius); j++)
    {
        // Check this cell is within the grid
        if ((i >= 0) && (i < GRIDSIZE) && (j >= 0) && (j < GRIDSIZE))
        {
            // If there's an agent at the cell then change its alive
            // status to 0
            switch (pGrid[j][i]->prevType)
            {
                case 0:
                    // No agent so do nothing
                    break;
                case 1:
                    // Go through Peacekeeper array to find the agent at
                    // the cell
                    for (n = 0; n < PEACENO; n++)
                    {
                        if ((pPeaceArray[n]->xPrev == i) &&
                            (pPeaceArray[n]->yPrev == j))
                        {
                            pPeaceArray[n]->alive = 0;
                            pPeaceArray[n]->xPos = GRIDSIZE;
                            pPeaceArray[n]->yPos = GRIDSIZE;
                            pGrid[j][i]->occSquad = 0;
                            pGrid[j][i]->agentType = 0;
                            // Write details to file
                            ofstream casualties;    // Declare file to
                                                    // write output to
                            casualties.open("casualties.txt", ios::app);
                            casualties << "peace[" << n
                                       << "] killed by a bomb at (" << i
                                       << "," << j << ") at timestep "
                                       << timestep << ", subtime "
                                       << subTime << "\n";
                            casualties.close();
                            // Write details to screen
                            cout << "peace[" << n << "] killed at ("
```

```
                << i << "," << j << ") \n";
        }
    }
    break;
case 2:
    // Go through NGO array to find agent at
    // cell
    for (n = 0; n < SUPPORTNO; n++)
    {
        if ((pSupportArray[n]->xPrev == i) &&
            (pSupportArray[n]->yPrev == j))
        {
            pSupportArray[n]->alive = 0;
            pSupportArray[n]->xPos = GRIDSIZE;
            pSupportArray[n]->yPos = GRIDSIZE;
            pGrid[j][i]->occSquad = 0;
            pGrid[j][i]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to
                                    // write output to
            casualties.open("casualties.txt", ios::app);
            casualties << "support[" << n
                       << "] killed by a bomb at (" << i
                       << "," << j << ") at timestep "
                       << timestep << ", subtime "
                       << subTime << "\n";
            casualties.close();
            // Write details to screen
            cout << "support[" << n << "] killed at ("
                 << i << "," << j << ") \n";
        }
    }
    break;
case 3:
    // Go through insurgent array to find agent at
    // cell
    for (n = 0; n < LOCALNO; n++)
    {
        if ((pLocalArray[n]->xPrev == i) &&
            (pLocalArray[n]->yPrev == j))
        {
            pLocalArray[n]->alive = 0;
            pLocalArray[n]->xPos = GRIDSIZE;
            pLocalArray[n]->yPos = GRIDSIZE;
            pGrid[j][i]->occSquad = 0;
            pGrid[j][i]->agentType = 0;
            // Write details to file
            ofstream casualties;    // Declare file to
                                    // write output to
            casualties.open("casualties.txt", ios::app);
            casualties << "local[" << n
```

```
                                    << "] killed by a bomb at (" << i
                                    << "," << j << ") at timestep "
                                    << timestep << ", subtime "
                                    << subTime << "\n";
                                casualties.close();
                                // Write details to screen
                                cout << "local[" << n << "] killed at ("
                                     << i << "," << j << ") \n";
                            }
                        }
                        break;
                    case 4:
                        // Go through Civilian array to find agent at cell
                        for (n = 0; n < CIVNO; n++)
                        {
                            if ((pCivilianArray[n]->xPrev == i) &&
                                (pCivilianArray[n]->yPrev == j))
                            {
                                pCivilianArray[n]->alive = 0;
                                pCivilianArray[n]->xPos = GRIDSIZE;
                                pCivilianArray[n]->yPos = GRIDSIZE;
                                pGrid[j][i]->occSquad = 0;
                                pGrid[j][i]->agentType = 0;
                                // Write details to file
                                ofstream casualties;    // Declare file to
                                                        // write output to
                                casualties.open("casualties.txt", ios::app);
                                casualties << "civilian[" << n
                                           << "] killed by a bomb at (" << i
                                           << "," << j << ") at timestep "
                                           << timestep << ", subtime "
                                           << subTime << "\n";
                                casualties.close();
                                // Write details to screen
                                cout << "civilian[" << n << "] killed at ("
                                     << i << "," << j << ") \n";
                            }
                        }
                        break;
                    default:
                        // Error message
                        cout << "Unknown agent type in Insurgent bomb "
                             << "function. \n";
                        cout << "grid[" << j << "][" << i << "].prevType = "
                             << pGrid[j][i]->prevType << "\n";
                        system("PAUSE");
                        exit(1);
                } // End of prevType switch loop
            } // End of coordinate check if loop
        }
    } // End of bomb radius grid loop
```

```
    } // End of if/else loop
} // End of locBomb function
```

B.8 initial.cpp

```
// initial.cpp
// Contains the functions for calculating initial positions

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"            // Include the header file for the Cell class
#include "agent.h"           // Include agent classes header file
#include "model.h"           // Include general header file

using namespace std;

// Peacekeeper initial position function

void peaceInitPos(short iniGrid[GRIDSIZE][GRIDSIZE], Peacekeeper *pPeace)
{
    short x, y, length, occupied;    // Initialise variables
    occupied = 0;
    length = (2*(pPeace->homeRadius)) + 1;

    while (occupied == 0)
    {
        x = (short)(rand()*length/32768) + pPeace->xHome - pPeace->homeRadius;
        y = (short)(rand()*length/32768) + pPeace->yHome - pPeace->homeRadius;
        // Check to see if this cell is occupied, if not set the agent's initial
        // position at grid[y][x] and change the value of occupied so we break
        // out of the while loop
        if (iniGrid[y][x] == 0)
        {
            occupied++;
            pPeace->xPos = x;
            pPeace->xPrev = x;
            pPeace->yPos = y;
            pPeace->yPrev = y;
            iniGrid[y][x]++;
        } // End of if loop
    } // End of while loop
```

```
    return;
} // End of peaceInitPos function

// NGO initial position function

void suppInitPos(short iniGrid[GRIDSIZE][GRIDSIZE], SupportAgent *pSupp)
{
    short x, y, length, occupied;    // Initialise variables
    occupied = 0;
    length = (2*(pSupp->homeRadius)) + 1;

    while (occupied == 0)
    {
        x = (short)(rand()*length/32768) + pSupp->xHome - pSupp->homeRadius;
        y = (short)(rand()*length/32768) + pSupp->yHome - pSupp->homeRadius;
        // Check to see if this cell is occupied, if not set the agent's initial
        // position at grid[y][x] and change the value of occupied so we break
        // out of the while loop
        if (iniGrid[y][x] == 0)
        {
            occupied++;
            pSupp->xPos = x;
            pSupp->xPrev = x;
            pSupp->yPos = y;
            pSupp->yPrev = y;
            iniGrid[y][x]++;
        } // End of if loop
    } // End of while loop

    return;
} // End of suppInitPos function

// Insurgent initial position function

void locInitPos(short iniGrid[GRIDSIZE][GRIDSIZE], LocalMilitia *pLoc)
{
    short x, y, length, occupied;    // Initialise variables
    occupied = 0;
    length = (2*(pLoc->homeRadius)) + 1;

    while (occupied == 0)
    {
        x = (short)(rand()*length/32768) + pLoc->xHome - pLoc->homeRadius;
        y = (short)(rand()*length/32768) + pLoc->yHome - pLoc->homeRadius;
        // Check to see if this cell is occupied, if not set the agent's initial
        // position at grid[y][x] and change the value of occupied so we break
        // out of the while loop
        if (iniGrid[y][x] == 0)
        {
            occupied++;
```

```
            pLoc->xPos = x;
            pLoc->xPrev = x;
            pLoc->yPos = y;
            pLoc->yPrev = y;
            iniGrid[y][x]++;
        } // End of if loop
    } // End of while loop

    return;
} // End of locInitPos function

// Civilian initial position function

void civInitPos(short iniGrid[GRIDSIZE][GRIDSIZE], Civilian *pCiv)
{
    short x, y, length, occupied;    // Initialise variables
    occupied = 0;
    length = (2*(pCiv->homeRadius)) + 1;

    while (occupied == 0)
    {
        x = (short)(rand()*length/32768) + pCiv->xHome - pCiv->homeRadius;
        y = (short)(rand()*length/32768) + pCiv->yHome - pCiv->homeRadius;
        // Check to see if this cell is occupied, if not set the agent's initial
        // position at grid[y][x] and change the value of occupied so we break
        // out of the while loop
        if (iniGrid[y][x] == 0)
        {
            occupied++;
            pCiv->xPos = x;
            pCiv->xPrev = x;
            pCiv->yPos = y;
            pCiv->yPrev = y;
            iniGrid[y][x]++;
        } // End of if loop
    } // End of while loop

    return;
} // End of civInitPos function
```

B.9 move.cpp

```
// move.cpp
// Gives the movement functions for the different types of agent

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
```

```
#include<ctime>

#include "cell.h"            // Include the header file for the Cell class
#include "agent.h"           // Include agent classes header file
#include "model.h"           // Include general header file

using namespace std;


// Function for calculating percentage change used for distance changes in the
// incentive function
double perChange(double oldValue, double newValue)
{
    double change;
    change = (oldValue - newValue)/oldValue;
    return change;
} // End of perChange function


// Movement functions

// Civilian movement function
void civMovement(Cell *pGrid[GRIDSIZE][GRIDSIZE], Civilian *pCiv,
                 int waterFailure[SECTOR*SECTOR],
                 int elecFailure[SECTOR*SECTOR])
{
    // Declare variables
    int i, j, a, b;            // Counters for loops
    int m = 0;                 // Counters for incentive function array,
    int n = 0;                 // initialised to be zero
    int weightNo;              // Variable to indicate which array entry should
                               // be used for the weight
    double dx;                 // Variables to hold x and y distances
    double dy;
    double newDistance;        // Variables to hold new and old distances to
    double oldDistance;        // agents, to be used in incentive calculation
    double changeInDistance;   // Variable to store percentage change in
                               // distance between proposed move and current
                               // cell
    double sum[3][3];          // Array to hold incentive calculations
    sum[0][0] = 0.0;
    sum[0][1] = 0.0;
    sum[0][2] = 0.0;
    sum[1][0] = 0.0;
    sum[1][1] = 0.0;
    sum[1][2] = 0.0;
    sum[2][0] = 0.0;
    sum[2][1] = 0.0;
    sum[2][2] = 0.0;

    double *ptr;                // Declare pointer
```

```
// Look at each possible move grid[j][i] and store the incentive
// calculations in the array sum[][]

for (i = ((pCiv->xPrev) - 1); i <= ((pCiv->xPrev) + 1); i++)
{
    for (j = ((pCiv->yPrev) - 1); j <= ((pCiv->yPrev) + 1); j++)
    {
        ptr = &(sum[n][m]);   // ptr points to sum[n][m]

        // Check that i and j are within range, this identifies corner and
        // edge cells and stops referrals to non-existant cells.  If
        // grid[j][i] does not exist then set the incentive to -50000, so
        // the agent cannot choose that square to move to.
        if ((i < 0) || (i >= GRIDSIZE) || (j < 0) || (j >= GRIDSIZE))
        {
            *ptr = -50000.0;
        }

        // Check that grid[j][i] is a valid move.  If not set the incentive
        // to be -50000.
        // First check to see whether the cell was occupied at the previous
        // timestep.  Only do this for cells other than the current position
        else if (((i != pCiv->xPrev) || (j != pCiv->yPrev)) &&
                 (pGrid[j][i]->prevType != 0))
        {
            *ptr = -50000.0;
        }
        // Check the following for all possible moves, including the current
        // cell
        // Check to see if any agent has already moved to the cell in this
        // timestep
        else if (pGrid[j][i]->moveInd == 1)
        {
            *ptr = -50000.0;
        }
        // Check whether there is combat at the cell
        else if ((pGrid[j][i]->combat) == 1)
        {
            *ptr = -49000.0;
        }
        // Check whether there is water at the cell
        else if (waterFailure[(pGrid[j][i]->sectorNo)] == 1)
        {
            *ptr = -48000.0;
        }
        // Check whether there is food at the cell
        else if ((pGrid[j][i]->foodAmount) == 0)
        {
            *ptr = -47000.0;
        }
```

```
// Check whether there is electricity at the cell
else if (elecFailure[(pGrid[j][i]->sectorNo)] == 1)
{
    *ptr = -46000.0;
}
// Now see if the proposed move is the agent's current location, if
// so the incentive value is automatically zero
else if ((i == pCiv->xPrev) && (j == pCiv->yPrev))
{
    *ptr = 0.0;
}
// Now calculate the incentive for all the valid moves grid[j][i]
// that are not the agent's current location
else
{
    // Look at all the agents within sensor range
    for (a = ((pCiv->xPrev) - (pCiv->sensorRange));
         a <= ((pCiv->xPrev) + (pCiv->sensorRange)); a++)
    {
        for (b = ((pCiv->yPrev) - (pCiv->sensorRange));
             b <= ((pCiv->yPrev) + (pCiv->sensorRange)); b++)
        {
            // Check that grid[b][a] is a valid grid point by
            // checking that 0 <= a, b < gridSize and also that the
            // cell is not the agent's current location
            if ((a >= 0) && (a < GRIDSIZE) && (b >= 0) &&
                (b < GRIDSIZE) && ((a != pCiv->xPrev) ||
                (b != pCiv->yPrev)))
            {
                // Calculate (Old dist - New dist)/Old dist,
                // where the distance is that to the Cell (a,b)
                dx = i - a;  // Calculate x difference
                dy = j - b;  // Calculate y difference
                newDistance = sqrt((double)(dx*dx + dy*dy));
                dx = pCiv->xPrev - a;  // Calculate x difference
                dy = pCiv->yPrev - b;  // Calculate y difference
                oldDistance = sqrt((double)(dx*dx + dy*dy));
                changeInDistance = perChange(oldDistance,
                                                    newDistance);

                // Check which agent type is at the cell grid[b][a].
                // Calculate the percentage change in distance then
                // multiply by the appropriate weight and add this
                // value to sum[n][m].  If there is no agent at the
                // cell then sum[n][m] remains unchanged
                switch (pGrid[b][a]->prevType)
                {
                    // No agent at the cell
                    case 0:
                        break;
                    // Peacekeeper at the cell
```

```
case 1:
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
        (pCiv->relation[(pGrid[b][a]->prevSquad)
                        - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pCiv->peaceWeight[weightNo])*changeInDistance;
    break;
// NGO at the cell
case 2:
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
        (pCiv->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pCiv->suppWeight[weightNo])*changeInDistance;
    break;
// Insurgent at the cell
case 3:
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
        (pCiv->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pCiv->locWeight[weightNo])*changeInDistance;
    break;
// Civilian at the cell
case 4:
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
```

```
                                    (pCiv->relation[(pGrid[b][a]->prevSquad)
                                                                - 1]) - 1;
                                // Multiply the percentage change in
                                // distance by the appropriate weight and
                                // add to sum[n][m]
                                *ptr = sum[n][m] +
                                (pCiv->civWeight[weightNo])*changeInDistance;
                                break;
                            default:
                                cout << "Invalid agent type in move.cpp "
                                     << "line 261!\n";
                                system("PAUSE");
                                exit(1);
                        } // End of prevType switch loop
                    } // End of coordinate check if loop
                }
            } // End of sensor range grid loop
        } // End of incentive calculation if/else if/else loop

        // Increment n
        n++;
    }
    // Reset n
    n = 0;
    // Increment m
    m++;
} // End of possible moves grid loop

// Look at the array sum[][] and determine the best move
int x, y;                       // Counters
short xNew, yNew;               // Store best move coordinates
short max[2] = { 0 };           // Stores sum[][] array coordinates for
                                // highest incentive location
short eqPen[9][2];              // Stores sum[][] array coordinates in case
                                // of equal penalties
int equalNo = 0;                // Counts how many cells have the max
                                // incentive
int randNum;                    // Declare integer to hold random number in
                                // case of equal penalties
short *mx, *my;                 // Declare pointers
mx = &(max[0]);                 // Set to point at max[0] and max[1]
my = &(max[1]);

// Look at all the incentive values in the sum[][] array and compare values
// to the highest currently found
for (x = 0; x < 3; x++)
{
    for (y = 0; y < 3; y++)
    {
        // Check to see if sum[y][x] is greater than the current highest
        // value, if so max[] becomes {x, y}
```

```
            if (sum[y][x] > sum[max[1]][max[0]])
            {
                *mx = x;
                *my = y;
                // Reset eqPen[][] and equalNo
                equalNo = 1;
                eqPen[0][0] = x;
                eqPen[0][1] = y;
            }
            // If sum[y][x] is equal to the current highest values then update
            // eqPen[][] and equalNo
            else if (sum[y][x] == sum[max[1]][max[0]])
            {
                eqPen[equalNo][0] = x;
                eqPen[equalNo][1] = y;
                equalNo++;
            } // End of if/else if loop
        }
    } // End of sum[][] array loop

    // If more than one cell has the same incentive then choose one at random
    if (equalNo > 1)
    {
        // Generate a random integer between 0 and (equalNo - 1)
        randNum = (int)(rand()*equalNo/32768);

        *mx = eqPen[randNum][0];
        *my = eqPen[randNum][1];
    } // End of if loop

    // Calculate grid coordinates of max[]
    xNew = (pCiv->xPrev) - 1 + max[0];
    yNew = (pCiv->yPrev) - 1 + max[1];

    // Change the cell occupancy information for the current cell
    pGrid[pCiv->yPos][pCiv->xPos]->occSquad = 0;
    pGrid[pCiv->yPos][pCiv->xPos]->agentType = 0;

    // Change agent position to (xNew, yNew)
    pCiv->xPos = xNew;
    pCiv->yPos = yNew;

    // Change the cell occupancy information for the cell the agent is moving to
    pGrid[yNew][xNew]->occSquad = pCiv->squadNo;
    pGrid[yNew][xNew]->agentType = 4;
    pGrid[yNew][xNew]->moveInd = 1;

    return;
} // End of civMovement function

// NGO movement function
```

```
void suppMovement(Cell *pGrid[GRIDSIZE][GRIDSIZE], SupportAgent *pSupp,
                  int waterFailure[SECTOR*SECTOR],
                  int elecFailure[SECTOR*SECTOR])
{
    // Declare variables
    int i, j, a, b;          // Counters for loops
    int m = 0;               // Counters for incentive function array,
    int n = 0;               // initialised to be zero
    int weightNo;            // Variable to indicate which array entry should be
                             // used for the weight
    double dx;               // Variables to hold x and y distances
    double dy;
    double newDistance;      // Variables to hold new and old distances to
    double oldDistance;      // agents, to be used in incentive calculation
    double changeInDistance; // Variable to store percentage change in distance
                             // between proposed move and current cell
    double sum[3][3];        // Array to hold incentive calculations
    sum[0][0] = 0.0;
    sum[0][1] = 0.0;
    sum[0][2] = 0.0;
    sum[1][0] = 0.0;
    sum[1][1] = 0.0;
    sum[1][2] = 0.0;
    sum[2][0] = 0.0;
    sum[2][1] = 0.0;
    sum[2][2] = 0.0;

    double *ptr;             // Declare pointer

    // Look at each possible move grid[j][i] and store the incentive
    // calculations in the array sum[][]

    for (i = ((pSupp->xPrev) - 1); i <= ((pSupp->xPrev) + 1); i++)
    {
        for (j = ((pSupp->yPrev) - 1); j <= ((pSupp->yPrev) + 1); j++)
        {
            ptr = &(sum[n][m]);   // ptr points to sum[n][m]

            // Check that i and j are within range, this identifies corner and
            // edge cells and stops referrals to non-existant cells.  If
            // grid[j][i] does not exist then set the incentive to -50000, so
            // the agent cannot choose that square to move to.
            if ((i < 0) || (i >= GRIDSIZE) || (j < 0) || (j >= GRIDSIZE))
            {
                *ptr = -50000.0;
            }

            // Check that grid[j][i] is a valid move.  If not set the incentive
            // to be -50000.
            // First check to see whether the cell was occupied at the previous
            // timestep.  Only do this for cells other than the current position
```

```
else if (((i != pSupp->xPrev) || (j != pSupp->yPrev))
         && (pGrid[j][i]->prevType != 0))
{
    *ptr = -50000.0;
}
// Check the following for all possible moves, including the current
// cell
// Check to see if any agent has already moved to the cell in this
// timestep
else if (pGrid[j][i]->moveInd == 1)
{
    *ptr = -50000.0;
}
// Check whether there is combat at the cell
else if ((pGrid[j][i]->combat) == 1)
{
    *ptr = -49000.0;
}
// Now check if the proposed move is the current cell
else if ((i == pSupp->xPrev) && (j == pSupp->yPrev))
{
    // Calculate the incentive using factors relating to civilians
    // in need, no water and no electricity
    *ptr = (pSupp->civInNeedWeight)*(pGrid[j][i]->civInNeed) +
    (pSupp->noWaterWeight)*(waterFailure[(pGrid[j][i]->sectorNo)]) +
       (pSupp->noElecWeight)*(elecFailure[(pGrid[j][i]->sectorNo)]);
}
// Now calculate the incentive for all the valid moves grid[j][i]
else
{
    // Look at all the agents within sensor range
    for (a = ((pSupp->xPrev) - (pSupp->sensorRange));
         a <= ((pSupp->xPrev) + (pSupp->sensorRange)); a++)
    {
        for (b = ((pSupp->yPrev) - (pSupp->sensorRange));
             b <= ((pSupp->yPrev) + (pSupp->sensorRange)); b++)
        {
            // Check that grid[b][a] is a valid grid point by
            // checking that 0 <= a, b < gridSize also check that
            // the cell is not the agent's current location
            if ((a >= 0) && (a < GRIDSIZE) && (b >= 0) &&
                (b < GRIDSIZE) && ((a != pSupp->xPrev) ||
                (b != pSupp->yPrev)))
            {
                // Calculate (Old dist - New dist)/Old dist,
                // where the distance is that to the Cell (a,b)
                dx = i - a;  // Calculate x difference
                dy = j - b;  // Calculate y difference
                newDistance = sqrt((double)(dx*dx + dy*dy));
                dx = pSupp->xPrev - a;  // Calculate x difference
                dy = pSupp->yPrev - b;  // Calculate y difference
```

```
oldDistance = sqrt((double)(dx*dx + dy*dy));
changeInDistance = perChange(oldDistance,
                                        newDistance);

// Check which agent type is at the cell grid[b][a].
// Calculate the percentage change in distance then
// multiply by the appropriate weight and add this
// value to  sum[n][m].  If there is no agent at the
// cell then sum[n][m] remains unchanged
switch (pGrid[b][a]->prevType)
{
    // No agent at the cell
    case 0:
        break;
    // Peacekeeper at the cell
    case 1:
        // Calculate array entry number for
        // relationship, remembering that arrays
        // start from zero, but relationships and
        // squads are numbered from one
        weightNo =
           (pSupp->relation[(pGrid[b][a]->prevSquad)
                                - 1]) - 1;
        // Multiply the percentage change in
        // distance by the appropriate weight and
        // add to sum[n][m]
        *ptr = sum[n][m] +
        (pSupp->peaceWeight[weightNo])*changeInDistance;
        break;
    // NGO at the cell
    case 2:
        // Calculate array entry number for
        // relationship, remembering that arrays
        // start from zero, but relationships and
        // squads are numbered from one
        weightNo =
           (pSupp->relation[(pGrid[b][a]->prevSquad)
                                - 1]) - 1;
        // Multiply the percentage change in
        // distance by the appropriate weight and
        // add to sum[n][m]
        *ptr = sum[n][m] +
        (pSupp->suppWeight[weightNo])*changeInDistance;
        break;
    // Insurgent at the cell
    case 3:
        // Calculate array entry number for
        // relationship, remembering that arrays
        // start from zero, but relationships and
        // squads are numbered from one
        weightNo =
```

```
                                    (pSupp->relation[(pGrid[b][a]->prevSquad)
                                                                - 1]) - 1;
                                // Multiply the percentage change in
                                // distance by the appropriate weight and
                                // add to sum[n][m]
                                *ptr = sum[n][m] +
                                (pSupp->locWeight[weightNo])*changeInDistance;
                                break;
                            // Civilian at the cell
                            case 4:
                                // Calculate array entry number for
                                // relationship, remembering that arrays
                                // start from zero, but relationships and
                                // squads are numbered from one
                                weightNo =
                                    (pSupp->relation[(pGrid[b][a]->prevSquad)
                                                                - 1]) - 1;
                                // Multiply the percentage change in
                                // distance by the appropriate weight and
                                // add to sum[n][m], also add in the
                                // Civilians in need factor
                                *ptr = sum[n][m] +
                             (pSupp->civWeight[weightNo])*changeInDistance +
          (pSupp->civInNeedWeight)*(pGrid[b][a]->civInNeed)*changeInDistance;
                                break;
                            default:
                                cout << "Invalid agent type in move.cpp "
                                     << "line 576!\n";
                                system("PAUSE");
                                exit(1);
                        } // End of prevType switch loop

                        // Add in no water and no electricity factors
                        *ptr = sum[n][m] +
 (pSupp->noWaterWeight)*(waterFailure[(pGrid[b][a]->sectorNo)])*changeInDistance
+ (pSupp->noElecWeight)*(elecFailure[(pGrid[b][a]->sectorNo)])*changeInDistance;

                    } // End of coordinate check loop
                }
            } // End of sensor range grid loop

            // Now add in the final factors relating to the need to fix the
            // water or electricity supply in the proposed move Cell
            *ptr = sum[n][m] +
                   (pSupp->noWaterWeight)*(pGrid[j][i]->fixWater) +
                   (pSupp->noElecWeight)*(pGrid[j][i]->fixElec);

        } // End of if/else if/else loop

        // Increment n
        n++;
```

```
    }
    // Reset n
    n = 0;
    // Increment m
    m++;
} // End of possible moves grid loop

// Look at the array sum[][] and determine the best move
int x, y;                       // Counters
short xNew, yNew;               // Store best move coordinates
short max[2] = { 0 };           // Stores sum[][] array coordinates for
                                // highest incentive location
short eqPen[9][2];              // Stores sum[][] array coordinates in case
                                // of equal penalties
int equalNo = 0;                // Counts how many cells have the max
                                // incentive
int randNum;                    // Declare integer to hold random number in
                                // case of equal penalties
short *mx, *my;                 // Declare pointers
mx = &(max[0]);                 // Set to point at max[0] and max[1]
my = &(max[1]);
// Look at all the incentive values in the sum[][] array and compare values
// to the highest currently found
for (x = 0; x < 3; x++)
{
    for (y = 0; y < 3; y++)
    {
        // Check to see if sum[y][x] is greater than the current highest
        // value, if so max[] becomes {x, y}
        if (sum[y][x] > sum[max[1]][max[0]])
        {
            *mx = x;
            *my = y;
            // Reset eqPen[][] and equalNo
            equalNo = 1;
            eqPen[0][0] = x;
            eqPen[0][1] = y;
        }
        // If sum[y][x] is equal to the current highest values then update
        // eqPen[][] and equalNo
        else if (sum[y][x] == sum[max[1]][max[0]])
        {
            eqPen[equalNo][0] = x;
            eqPen[equalNo][1] = y;
            equalNo++;
        } // End of incentive calculation if/else if loop
    }
} // End of sum[][] array loop
// If more than one cell has the same incentive then choose one at random
if (equalNo > 1)
{
```

```
        // Generate a random integer between 0 and (equalNo - 1)
        randNum = (int)(rand()*equalNo/32768);
        *mx = eqPen[randNum][0];
        *my = eqPen[randNum][1];
    } // End of if loop


    // Calculate grid coordinates of max[]
    xNew = (pSupp->xPrev) - 1 + max[0];
    yNew = (pSupp->yPrev) - 1 + max[1];

    // Change the cell occupancy information for the current cell
    pGrid[pSupp->yPos][pSupp->xPos]->occSquad = 0;
    pGrid[pSupp->yPos][pSupp->xPos]->agentType = 0;

    // Change agent position to (xNew, yNew)
    pSupp->xPos = xNew;
    pSupp->yPos = yNew;

    // Change the cell occupancy information for the cell the agent is moving to
    pGrid[yNew][xNew]->occSquad = pSupp->squadNo;
    pGrid[yNew][xNew]->agentType = 2;
    pGrid[yNew][xNew]->moveInd = 1;

    return;
} // End of suppMovement function


// Insurgent movement function
void locMovement(Cell *pGrid[GRIDSIZE][GRIDSIZE], LocalMilitia *pLoc,
                 int waterFailure[SECTOR*SECTOR],
                 int elecFailure[SECTOR*SECTOR])
{
    // Declare variables
    int i, j, a, b;               // Counters for loops
    int m = 0;                    // Counters for incentive function array,
    int n = 0;                    // initialised to be zero
    int weightNo;                 // Variable to indicate which array entry
                                  // should be used for the weight
    //short changeInTension;      // Variable to hold change in tension
    double dx;                    // Variables to hold x and y distances
    double dy;
    double newDistance;           // Variables to hold new and old distances to
    double oldDistance;           // agents, to be used in incentive
                                  // calculation
    double changeInDistance;      // Variable to store percentage change in
                                  // distance between proposed move and current
                                  // cell
    double sum[3][3];             // Array to hold incentive calculations
    sum[0][0] = 0.0;
    sum[0][1] = 0.0;
```

```
sum[0][2] = 0.0;
sum[1][0] = 0.0;
sum[1][1] = 0.0;
sum[1][2] = 0.0;
sum[2][0] = 0.0;
sum[2][1] = 0.0;
sum[2][2] = 0.0;

double *ptr;                    // Declare pointer

// Look at each possible move grid[j][i] and store the incentive
// calculations in the array sum[][]

for (i = ((pLoc->xPrev) - 1); i <= ((pLoc->xPrev) + 1); i++)
{
    for (j = ((pLoc->yPrev) - 1); j <= ((pLoc->yPrev) + 1); j++)
    {
        ptr = &(sum[n][m]);   // ptr points to sum[n][m]

        // Check that i and j are within range, this identifies corner and
        // edge cells and stops referrals to non-existant cells.  If
        // grid[j][i] does not exist then set the incentive to -50000, so
        // the agent cannot choose that square to move to.
        if ((i < 0) || (i >= GRIDSIZE) || (j < 0) || (j >= GRIDSIZE))
        {
            *ptr = -50000.0;
        }

        // Check that grid[j][i] is a valid move.  If not set the incentive
        // to be -50000 and break out of the loop to use the incentive
        // function.
        // First check to see whether the cell was occupied at the previous
        // timestep.  Only do this for cells other than the current position
        else if (((i != pLoc->xPrev) || (j != pLoc->yPrev)) &&
                 (pGrid[j][i]->prevType != 0))
        {
            *ptr = -50000.0;
        }
        // Check the following for all possible moves, including the current
        // cell
        // Check to see if any agent has already moved to the cell in this
        // timestep
        else if (pGrid[j][i]->moveInd == 1)
        {
            *ptr = -50000.0;
        }
        // Check whether there is water at the cell
        else if (waterFailure[(pGrid[j][i]->sectorNo)] == 1)
        {
            *ptr = -48000.0;
        }
```

```
// Now check if the proposed move is the current cell, if so the
// only possible nonzero factor will be combat
else if ((i == pLoc->xPrev) && (j == pLoc->yPrev))
{
    *ptr = (pLoc->combatWeight)*(pGrid[j][i]->combat);
}
// Now calculate the incentive for all the valid moves grid[j][i]
else
{
    // Look at all the agents within sensor range
    for (a = ((pLoc->xPrev) - (pLoc->sensorRange));
         a <= ((pLoc->xPrev) + (pLoc->sensorRange)); a++)
    {
        for (b = ((pLoc->yPrev) - (pLoc->sensorRange));
             b <= ((pLoc->yPrev) + (pLoc->sensorRange)); b++)
        {
            // Check that grid[b][a] is a valid grid point by
            // checking that 0 <= a, b < gridSize
            if ((a >= 0) && (a < GRIDSIZE) && (b >= 0) &&
                (b < GRIDSIZE) && ((a != pLoc->xPrev) ||
                (b != pLoc->yPrev)))
            {
                // Calculate (Old dist - New dist)/Old dist,
                // where the distance is that to the Cell (a,b)
                dx = i - a;  // Calculate x difference
                dy = j - b;  // Calculate y difference
                newDistance = sqrt((double)(dx*dx + dy*dy));
                dx = pLoc->xPrev - a;  // Calculate x difference
                dy = pLoc->yPrev - b;  // Calculate y difference
                oldDistance = sqrt((double)(dx*dx + dy*dy));
                changeInDistance = perChange(oldDistance,
                                                    newDistance);

                // Check which agent type is at the cell grid[b][a].
                // Calculate the percentage change in distance then
                // multiply by the appropriate weight and add this
                // value to sum[n][m].  If there is no agent at the
                // cell then sum[n][m] remains unchanged
                switch (pGrid[b][a]->prevType)
                {
                    // No agent at the cell
                    case 0:
                        break;
                    // Peacekeeper at the cell
                    case 1:
                        // Calculate (Old dist - New dist)/Old dist,
                        // where the distance is that to the
                        // appropriate agent
                        dx = i - a;  // Calculate x difference
                        dy = j - b;  // Calculate y difference
                        newDistance = sqrt((double)(dx*dx + dy*dy));
```

```
    dx = pLoc->xPrev - a;  // Calculate x
                           // difference
    dy = pLoc->yPrev - b;  // Calculate y
                           // difference
    oldDistance = sqrt((double)(dx*dx + dy*dy));
    changeInDistance = perChange(oldDistance,
                                 newDistance);
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
        (pLoc->relation[(pGrid[b][a]->prevSquad)
                         - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pLoc->peaceWeight[weightNo])*changeInDistance;
    break;
// NGO at the cell
case 2:
    // Calculate (Old dist - New dist)/Old dist,
    // where the distance is that to the
    // appropriate agent
    dx = i - a;  // Calculate x difference
    dy = j - b;  // Calculate y difference
    newDistance = sqrt((double)(dx*dx + dy*dy));
    dx = pLoc->xPrev - a;  // Calculate x
                           // difference
    dy = pLoc->yPrev - b;  // Calculate y
                           // difference
    oldDistance = sqrt((double)(dx*dx + dy*dy));
    changeInDistance = perChange(oldDistance,
                                 newDistance);
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
        (pLoc->relation[(pGrid[b][a]->prevSquad)
                                    - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pLoc->suppWeight[weightNo])*changeInDistance;
    break;
// Insurgent at the cell
case 3:
    // Calculate (Old dist - New dist)/Old dist,
```

```
    // where the distance is that to the
    // appropriate agent
    dx = i - a;  // Calculate x difference
    dy = j - b;  // Calculate y difference
    newDistance = sqrt((double)(dx*dx + dy*dy));
    dx = pLoc->xPrev - a;  // Calculate x
                           // difference
    dy = pLoc->yPrev - b;  // Calculate y
                           // difference
    oldDistance = sqrt((double)(dx*dx + dy*dy));
    changeInDistance = perChange(oldDistance,
                                 newDistance);
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
    (pLoc->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
    (pLoc->locWeight[weightNo])*changeInDistance;
    break;
// Civilian at the cell
case 4:
    // Calculate (Old dist - New dist)/Old dist,
    // where the distance is that to the
    // appropriate agent
    dx = i - a;  // Calculate x difference
    dy = j - b;  // Calculate y difference
    newDistance = sqrt((double)(dx*dx + dy*dy));
    dx = pLoc->xPrev - a;  // Calculate x
                           // difference
    dy = pLoc->yPrev - b;  // Calculate y
                           // difference
    oldDistance = sqrt((double)(dx*dx + dy*dy));
    changeInDistance = perChange(oldDistance,
                                 newDistance);
    // Calculate array entry number for
    // relationship, remembering that arrays
    // start from zero, but relationships and
    // squads are numbered from one
    weightNo =
    (pLoc->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
    // Multiply the percentage change in
    // distance by the appropriate weight and
    // add to sum[n][m]
    *ptr = sum[n][m] +
```

```
                                (pLoc->civWeight[weightNo])*changeInDistance;
                                break;
                            default:
                                cout << "Invalid agent type in move.cpp "
                                     << "line 891!\n";
                                system("PAUSE");
                                exit(1);
                        } // End of prevType switch loop

                        // Add in combat factor
                        *ptr = sum[n][m] +
                    (pLoc->combatWeight)*(pGrid[b][a]->combat)*changeInDistance;

                    } // End of coordinate check loop
                }
            } // End of sensor range grid loop

        } // End of incentive calculation if/else if/else loop
        // Increment n
        n++;
    }
    // Reset n
    n = 0;
    // Increment m
    m++;
} // End of possible moves loop

// Look at the array sum[][] and determine the best move
int x, y;                    // Counters
short xNew, yNew;              // Store best move coordinates
short max[2] = { 0 };          // Stores sum[][] array coordinates for
                               // highest incentive location
short eqPen[9][2];             // Stores sum[][] array coordinates in case
                               // of equal penalties
int equalNo = 0;             // Counts how many cells have the max
                               // incentive
int randNum;                 // Declare integer to hold random number in
                               // case of equal penalties
short *mx, *my;                // Declare pointers
mx = &(max[0]);                // Set to point at max[0] and max[1]
my = &(max[1]);
// Look at all the incentive values in the sum[][] array and compare values
// to the highest currently found
for (x = 0; x < 3; x++)
{
    for (y = 0; y < 3; y++)
    {
        // Check to see if sum[y][x] is greater than the current highest
        // value, if so max[] becomes {x, y}
        if (sum[y][x] > sum[max[1]][max[0]])
        {
```

```
                *mx = x;
                *my = y;
                // Reset eqPen[][] and equalNo
                equalNo = 1;
                eqPen[0][0] = x;
                eqPen[0][1] = y;
            }
            // If sum[y][x] is equal to the current highest values then update
            // eqPen[][] and equalNo
            else if (sum[y][x] == sum[max[1]][max[0]])
            {
                eqPen[equalNo][0] = x;
                eqPen[equalNo][1] = y;
                equalNo++;
            } // End if if/else if loop
        }
    } // End of sum[][] array loop
    // If more than one cell has the same incentive then choose one at random
    if (equalNo > 1)
    {
        // Generate a random integer between 0 and (equalNo - 1)
        randNum = (int)(rand()*equalNo/32768);
        *mx = eqPen[randNum][0];
        *my = eqPen[randNum][1];
    } // End of if loop

    // Calculate grid coordinates of max[]
    xNew = (pLoc->xPrev) - 1 + max[0];
    yNew = (pLoc->yPrev) - 1 + max[1];

    // Change the cell occupancy information for the current cell
    pGrid[pLoc->yPos][pLoc->xPos]->occSquad = 0;
    pGrid[pLoc->yPos][pLoc->xPos]->agentType = 0;

    // Change agent position to (xNew, yNew)
    pLoc->xPos = xNew;
    pLoc->yPos = yNew;

    // Change the cell occupancy information for the cell the agent is moving to
    pGrid[yNew][xNew]->occSquad = pLoc->squadNo;
    pGrid[yNew][xNew]->agentType = 3;
    pGrid[yNew][xNew]->moveInd = 1;

    return;
} // End of locMovement function

// Peacekeeper movement function
void peaceMovement(Cell *pGrid[GRIDSIZE][GRIDSIZE], Peacekeeper *pPeace,
                   int waterFailure[SECTOR*SECTOR],
                   int elecFailure[SECTOR*SECTOR])
{
```

```
// Declare variables
int i, j, a, b;          // Counters for loops
int m = 0;               // Counters for incentive function array,
int n = 0;               // initialised to be zero
int weightNo;            // Variable to indicate which array entry should be
                         // used for the weight
double dx;               // Variables to hold x and y distances
double dy;
double newDistance;      // Variables to hold new and old distances to
double oldDistance;      // agents, to be used in incentive calculation
double changeInDistance; // Variable to store percentage change in distance
                         // between proposed move and current cell
double sum[3][3];        // Array to hold incentive calculations
sum[0][0] = 0.0;
sum[0][1] = 0.0;
sum[0][2] = 0.0;
sum[1][0] = 0.0;
sum[1][1] = 0.0;
sum[1][2] = 0.0;
sum[2][0] = 0.0;
sum[2][1] = 0.0;
sum[2][2] = 0.0;

double *ptr;             // Declare pointer

// Look at each possible move pGrid[j][i] and store the incentive
// calculations in the array sum[][]

for (i = ((pPeace->xPrev) - 1); i <= ((pPeace->xPrev) + 1); i++)
{
    for (j = ((pPeace->yPrev) - 1); j <= ((pPeace->yPrev) + 1); j++)
    {
        ptr = &(sum[n][m]);   // ptr points to sum[n][m]

        // Check that i and j are within range, this identifies corner and
        // edge cells and stops referrals to non-existant cells.  If
        // grid[j][i] does not exist then set the incentive to -50000, so
        // the agent cannot choose that square to move to.
        if ((i < 0) || (i >= GRIDSIZE) || (j < 0) || (j >= GRIDSIZE))
        {
            *ptr = -50000.0;
        }

        // Check that grid[j][i] is a valid move.  If not set the incentive
        // to be -50000 and break out of the loop to use the incentive
        // function.
        // First check to see whether the cell was occupied at the previous
        // timestep.  Only do this for cells other than the current position
        else if (((i != pPeace->xPrev) || (j != pPeace->yPrev)) &&
                  (pGrid[j][i]->prevType != 0))
        {
```

```
    *ptr = -50000.0;
}
// Check the following for all possible moves, including the current
// cell
// Check to see if any agent has already moved to the cell in this
// timestep
else if (pGrid[j][i]->moveInd == 1)
{
    *ptr = -50000.0;
}
// Now check if the proposed move is the current cell and if so
// calculate the incentive
else if ((i == pPeace->xPrev) && (j == pPeace->yPrev))
{
    // Now add the civilians in need, no water, no electricity and
    // combat factors
    *ptr = (pPeace->civInNeedWeight)*(pGrid[j][i]->civInNeed) +
    (pPeace->noWaterWeight)*(waterFailure[(pGrid[j][i]->sectorNo)]) +
     (pPeace->noElecWeight)*(elecFailure[(pGrid[j][i]->sectorNo)]) +
           (pPeace->combatWeight)*(pGrid[j][i]->combat);
}
// Now calculate the incentive for all the valid moves grid[j][i]
else
{
    // Look at all the agents within sensor range
    for (a = ((pPeace->xPrev) - (pPeace->sensorRange));
         a <= ((pPeace->xPrev) + (pPeace->sensorRange)); a++)
    {
        for (b = ((pPeace->yPrev) - (pPeace->sensorRange));
             b <= ((pPeace->yPrev) + (pPeace->sensorRange)); b++)
        {
            // Check that grid[b][a] is a valid grid point by
            // checking that 0 <= a, b < gridSize
            if ((a >= 0) && (a < GRIDSIZE) && (b >= 0) &&
                (b < GRIDSIZE) && ((a != pPeace->xPrev) ||
                (b != pPeace->yPrev)))
            {
                // Calculate (Old dist - New dist)/Old dist,
                // where the distance is that to the Cell (a,b)
                dx = i - a;  // Calculate x difference
                dy = j - b;  // Calculate y difference
                newDistance = sqrt((double)(dx*dx + dy*dy));
                dx = pPeace->xPrev - a;  // Calculate x difference
                dy = pPeace->yPrev - b;  // Calculate y difference
                oldDistance = sqrt((double)(dx*dx + dy*dy));
                changeInDistance = perChange(oldDistance,
                                                    newDistance);

                // Check which agent type is at the cell grid[b][a].
                // Calculate the percentage change in distance then
                // multiply by the appropriate weight and add this
```

```
// value to sum[n][m].  If there is no agent at the
// cell then sum[n][m] remains unchanged
switch (pGrid[b][a]->prevType)
{
    // No agent at the cell
    case 0:
        break;
    // Peacekeeper at the cell
    case 1:
        // Calculate (Old dist - New dist)/Old dist,
        // where the distance is that to the
        // appropriate agent
        dx = i - a;  // Calculate x difference
        dy = j - b;  // Calculate y difference
        newDistance = sqrt((double)(dx*dx + dy*dy));
        dx = pPeace->xPrev - a;  // Calculate x
                                 // difference
        dy = pPeace->yPrev - b;  // Calculate y
                                 // difference
        oldDistance = sqrt((double)(dx*dx + dy*dy));
        changeInDistance = perChange(oldDistance,
                                     newDistance);
        // Calculate array entry number for
        // relationship, remembering that arrays
        // start from zero, but relationships and
        // squads are numbered from one
        weightNo =
          (pPeace->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
        // Multiply the percentage change in
        // distance by the appropriate weight and
        // add to sum[n][m]
        *ptr = sum[n][m] +
        (pPeace->peaceWeight[weightNo])*changeInDistance;
        break;
    // NGO at the cell
    case 2:
        // Calculate array entry number for
        // relationship, remembering that arrays
        // start from zero, but relationships and
        // squads are numbered from one
        weightNo =
          (pPeace->relation[(pGrid[b][a]->prevSquad)
                            - 1]) - 1;
        // Multiply the percentage change in
        // distance by the appropriate weight and
        // add to sum[n][m]
        *ptr = sum[n][m] +
        (pPeace->suppWeight[weightNo])*changeInDistance;
        break;
    // Insurgent at the cell
```

```
                                case 3:
                                    // Calculate array entry number for
                                    // relationship, remembering that arrays
                                    // start from zero, but relationships and
                                    // squads are numbered from one
                                    weightNo =
                                      (pPeace->relation[(pGrid[b][a]->prevSquad)
                                                          - 1]) - 1;
                                    // Multiply the percentage change in
                                    // distance by the appropriate weight and
                                    // add to sum[n][m]
                                    *ptr = sum[n][m] +
                                    (pPeace->locWeight[weightNo])*changeInDistance;
                                    break;
                                // Civilian at the cell
                                case 4:
                                    // Calculate array entry number for
                                    // relationship, remembering that arrays
                                    // start from zero, but relationships and
                                    // squads are numbered from one
                                    weightNo =
                                      (pPeace->relation[(pGrid[b][a]->prevSquad)
                                                        - 1]) - 1;
                                    // Multiply the percentage change in
                                    // distance by the appropriate weight and
                                    // add to sum[n][m], also add in the
                                    // Civilian in need factor
                                    *ptr = sum[n][m] +
                                (pPeace->civWeight[weightNo])*changeInDistance +
            (pPeace->civInNeedWeight)*(pGrid[b][a]->civInNeed)*changeInDistance;
                                    break;
                                default:
                                    cout << "Invalid agent type in move.cpp "
                                         << "line 1202!\n";
                                    system("PAUSE");
                                    exit(1);
                            } // End of prevType switch loop

                            *ptr = sum[n][m] +
(pPeace->noWaterWeight)*(waterFailure[(pGrid[b][a]->sectorNo)])*changeInDistance
+ (pPeace->noElecWeight)*(elecFailure[(pGrid[b][a]->sectorNo)])*changeInDistance
                + (pPeace->combatWeight)*(pGrid[b][a]->combat)*changeInDistance;

                        } // End of coordinate check if loop
                    }
                } // End of sensor range grid loop

                // Now add in the final factors relating to the need to fix the
                // water or electricity supply in the proposed move Cell
                *ptr = sum[n][m] +
                       (pPeace->noWaterWeight)*(pGrid[j][i]->fixWater) +
```

```
                    (pPeace->noElecWeight)*(pGrid[j][i]->fixElec);

        } // End of incentive calculation if/else if/else loop
        // Increment n
        n++;
    }
    // Reset n
    n = 0;
    // Increment m
    m++;
} // End of possible moves grid loop

// Look at the array sum[][] and determine the best move
int x, y;                    // Counters
short xNew, yNew;              // Store best move coordinates
short max[2] = { 0 };          // Stores sum[][] array coordinates for
                               // highest incentive location
short eqPen[9][2];             // Stores sum[][] array coordinates in case
                               // of equal penalties
int equalNo = 0;             // Counts how many cells have the max
                               // incentive
int randNum;                 // Declare integer to hold random number in
                               // case of equal penalties
short *mx, *my;                // Declare pointers
mx = &(max[0]);                // Set to point at max[0] and max[1]
my = &(max[1]);

// Look at all the incentive values in the sum[][] array and compare values
// to the highest currently found
for (x = 0; x < 3; x++)
{
    for (y = 0; y < 3; y++)
    {
        // Check to see if sum[y][x] is greater than the current highest
        // value, if so max[] becomes {x, y}
        if (sum[y][x] > sum[max[1]][max[0]])
        {
            *mx = x;
            *my = y;
            // Reset eqPen[][] and equalNo
            equalNo = 1;
            eqPen[0][0] = x;
            eqPen[0][1] = y;
        }
        // If sum[y][x] is equal to the current highest values then update
        // eqPen[][] and equalNo
        else if (sum[y][x] == sum[max[1]][max[0]])
        {
            eqPen[equalNo][0] = x;
            eqPen[equalNo][1] = y;
            equalNo++;
```

```
            } // End of if/else if loop
        }
    } // End of sum[][] array loop

    // If more than one cell has the same incentive then choose one at random
    if (equalNo > 1)
    {
        // Generate a random integer between 0 and (equalNo - 1)
        randNum = (int)(rand()*equalNo/32768);
        *mx = eqPen[randNum][0];
        *my = eqPen[randNum][1];
    } // End of if loop

    // Calculate grid coordinates of max[]
    xNew = (pPeace->xPrev) - 1 + max[0];
    yNew = (pPeace->yPrev) - 1 + max[1];

    // Change the cell occupancy information for the current cell
    pGrid[pPeace->yPos][pPeace->xPos]->occSquad = 0;
    pGrid[pPeace->yPos][pPeace->xPos]->agentType = 0;

    // Change agent position to (xNew, yNew)
    pPeace->xPos = xNew;
    pPeace->yPos = yNew;

    // Change the cell occupancy information for the cell the agent is moving to
    pGrid[yNew][xNew]->occSquad = pPeace->squadNo;
    pGrid[yNew][xNew]->agentType = 1;
    pGrid[yNew][xNew]->moveInd = 1;

    return;
} // End of peaceMovement function
```

B.10 repair.cpp

```
// repair.cpp
// Contains the repair functions for the Peacekeepers and NGOs

#include<cstdio>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include<cmath>
#include<ctime>

#include "cell.h"            // Include the header file for the Cell class
#include "agent.h"           // Include agent classes header file
#include "model.h"           // Include general header file
```

```
using namespace std;

// Peacekeeper repair function

void peaceRepair(Peacekeeper *pPeace, Cell *pCell)
{
    float randNumber;        // Declare variable to hold random numbers

    // First check to see if the water supply needs fixing
    if (pCell->fixWater == 1)
    {
        // Generate a random number between 0 and 1 to see if the agent is able
        // to fix the fault
        randNumber = (float)rand()/32767;
        if ((pPeace->probFixWater != 0) &&
                                        (randNumber <= (pPeace->probFixWater)))
        {
            (pCell->fixedW)++;
            cout << "Water supply fixed in sector " << pCell->sectorNo << "\n";
        } // End of if loop
    }
    // If not, then check the electricity supply needs fixing
    else if (pCell->fixElec == 1)
    {
        // Generate a random number between 0 and 1 to see if the agent is able
        // to fix the fault
        randNumber = (float)rand()/32767;
        if ((pPeace->probFixElec != 0) && (randNumber <= (pPeace->probFixElec)))
        {
            (pCell->fixedE)++;
            cout << "Electricity supply fixed in sector " << pCell->sectorNo
                 << "\n";
        } // End of if loop
    }
    // In the case of neither the function must have been called in error
    else
    {
        cout << "Repair function called in error. \n";
        system("PAUSE");
        exit(1);
    } // End of if/else if/else loop


    return;
} // End of peaceRepair function

// NGO repair function

void supportRepair(SupportAgent *pSupport, Cell *pCell)
{
```

```
    float randNumber;        // Declare variable to hold random numbers

    // First check to see if the water supply needs fixing
    if (pCell->fixWater == 1)
    {
        // Generate a random number between 0 and 1 to see if the agent is able
        // to fix the fault
        randNumber = (float)rand()/32767;
        if ((pSupport->probFixWater != 0) &&
                                      (randNumber <= (pSupport->probFixWater)))
        {
            (pCell->fixedW)++;
            cout << "Water supply fixed in sector " << pCell->sectorNo << "\n";
        } // End of if loop
    }
    // If not, then check the electricity supply needs fixing
    else if (pCell->fixElec == 1)
    {
        // Generate a random number between 0 and 1 to see if the agent is able
        // to fix the fault
        randNumber = (float)rand()/32767;
        if ((pSupport->probFixElec != 0) &&
                                       (randNumber <= (pSupport->probFixElec)))
        {
            (pCell->fixedE)++;
            cout << "Electricity supply fixed in sector " << pCell->sectorNo
                 << "\n";
        } // End of if loop
    }
    // In the case of neither the function must have been called in error
    else
    {
        cout << "Repair function called in error. \n";
        system("PAUSE");
        exit(1);
    } // End of if/else if/else loop


    return;
} // End of supportRepair function
```

Appendix C

FULL LIST OF MODEL PARAMETERS

$RUNTIME$	Defines the number of timesteps the model will run for.
$MAXSHOTS$	Defines the maximum total number of shots that can be fired during the model run.
$MAXBOMB$	Defines the maximum total number of bombs that can be set off during the model run.
$GRIDSIZE$	Defines the side length of the grid.
$SECTOR$	Defines the number of sectors each side of the grid is divided into, and thus gives the square root of the total number of sectors. Note that $SECTOR$ must be a divisor of $GRIDSIZE$.
$WATERFAIL$	Defines the probability that the water supply in a sector will fail at a timestep, $0 \leq WATERFAIL \leq 1$.
$ELECFAIL$	Defines the probability that the electricity supply in a sector will fail at a timestep, $0 \leq ELECFAIL \leq 1$.
$NOOFSQUADS$	Defines the total number of squads in the model. As it stands we can only have one squad per agent type so we must have $1 \leq NOOFSQUADS \leq 4$.

PEACENO	Total number of agents in the Peacekeeper squad.
SUPPORTNO	Total number of agents in the NGO squad.
LOCALNO	Total number of agents in the Insurgent squad.
CIVNO	Total number of agents in the Civilian squad.
PEACECAP	Defines the Peacekeeper agent capability.
SUPPORTCAP	Defines the NGO agent capability.
LOCALCAP	Defines the Insurgent agent capability.
CIVCAP	Defines the Civilian agent capability.
BOMBMEMORY	Number of timesteps over which the agents can 'remember' that a bomb has affected a cell.
SHOTMEMORY	Number of timesteps over which the agents can 'remember' that a shot has been fired to the cell.
LMBOMBPROB	Defines the probability that an Insurgent will set off a bomb at a sub-timestep.
LMFIREPROB	Defines the probability that an Insurgent will fire at an enemy agent without provocation at a sub-timestep.

C.1 Cells

$sectorNo$	Identifies the sector in which the cell is located, $0 \leq sectorNo \leq SECTOR^2$.
$occSquad$	Identifies the squad the occupying agent belongs to, this value is zero if there is no agent at the cell, $1 \leq occSquad \leq NOOFSQUADS$.
$prevSquad$	Identifies the squad the occupying agent at the last sub-timestep belonged to, this value is zero if there was no agent at the cell, $1 \leq prevSquad \leq NOOFSQUADS$.
$agentType$	Identifies the type of agent at the cell: one signifies a Peacekeeper, two is an NGO, three is an insurgent, four a Civilian and zero shows the cell is empty.
$prevType$	Identifies the type of agent at the cell at the last sub-timestep: one signifies a Peacekeeper, two is an NGO, three is an Insurgent, four a Civilian and zero shows the cell is empty.
$foodAmount$	Gives the number of food rations available at the cell, one ration gives enough food for one Insurgent or Civilian for one sub-timestep.
$actionType$	Indicates the action to be taken by the occupying agent at the current sub-timestep: one indicates combat, two is set off a suicide bomb, three is repair water or electricity, four is move and zero means no action is to be taken.
$moveInd$	Indicates whether or not an agent has moved to the cell in the current sub-timestep, where one is yes and zero is no.
$fixWater$	Indicates whether or not repairs to the water supply are needed at the cell where one means yes and zero means no.

$fixedW$	Indicates whether or not the water supply has been fixed during the current sub-timestep where one means yes and zero means no.
$fixElec$	Indicates whether or not repairs to the electricity supply are needed at the cell where one is yes and zero is no.
$fixedElec$	Indicates whether or not the electricity supply has been fixed during the current sub-timestep where one means yes and zero means no.
$combat$	Indicates whether or not there have been any shots fired to or from the cell in the previous $SHOTMEMORY$ timesteps or any bombs affecting the cell in the previous $BOMBMEMORY$ timesteps, where one means yes and zero means no.
$shotToNo$	Counter for the number of shots fired to the cell, $0 \leq shotToNo \leq MAXSHOTS$.
$shotInd$	Indicates whether or not there have been any shots fired to the cell in the previous $SHOTMEMORY$ timesteps where one means yes and zero means no.
$bombBlast$	Indicates whether or not the cell has been affected by a bomb during the previous $BOMBMEMORY$ timesteps where one means yes and zero means no.
$civInNeed$	Indicates whether or not there is a Civilian at the cell requiring assistance, this could be because there is either no water, no electricity or no food or if there is combat. A value of one indicates there is such a situation, zero that there is not.

C.2 Agents

$squadNo$	Identifies the squad the agent belongs to, $1 \leq squadNo \leq NOOFSQUADS$.
$agentType$	Identifies the type of agent: one signifies a Peacekeeper, two is an NGO, three is an Insurgent and four a Civilian.
$xHome$	Gives the $x-$coordinate for the centre of the initial distribution of the squad's agents, $0 \leq xHome \leq GRIDSIZE$.
$yHome$	Gives the $y-$coordinate for the centre of the initial distribution of the squad's agents, $0 \leq yHome \leq GRIDSIZE$.
$homeRadius$	Radius of the squad's initial distribution. Along with $xHome$ and $yHome$ this defines the square in which the squad is initially distributed.
$xPos$	Gives the $x-$coordinate of the agent's current location, $0 \leq xPos \leq GRIDSIZE$.
$xPrev$	Gives the $x-$coordinate of the agent's position at the previous timestep, $0 \leq xPrev \leq GRIDSIZE$.
$yPos$	Gives the $y-$coordinate of the agent's current location, $0 \leq yPos \leq GRIDSIZE$.
$yPrev$	Gives the $y-$coordinate of the agent's position at the previous timestep, $0 \leq yPrev \leq GRIDSIZE$.
$alive$	Indicates whether an agent has been killed, a value of one signals that the agent is still alive and zero that he has been killed.

$peaceWeight[5]$	Array to hold the weights towards the Peacekeepers, that is W_1 to W_5.
$suppWeight[5]$	Array to hold the weights towards the NGOs, that is W_6 to W_{10}.
$locWeight[5]$	Array to hold the weights towards the Insurgents, that is W_{11} to W_{15}.
$civWeight[5]$	Array to hold the weights towards the Civilians, that is W_{16} to W_{20}.
$sensorRange$	Defines the area over which the agent can detect other agents and situations. Taking the agent's current location as the centre, the agent can 'see' over a square with side length $2 * sensorRange + 1$.
$relation[NOOFSQUADS]$	Defines an array of the relationships to the other squads, given in numerical order, where a value of one is friendly, two is cooperative, three is neutral, four is uncooperative and five is hostile.

C.2.1 Additional Peacekeeper Parameters

$civInNeedWeight$	Gives the weight towards cells with Civilians in need.
$noWaterWeight$	Gives the weight towards cells with no water supply.
$noElecWeight$	Gives the weight towards cells with no electricity supply.
$combatWeight$	Gives the weight towards cells with conflict.
$sSKP$	Gives the single shot kill probability for the agent, $0 \leq sSKP \leq 1$.
$fireRange$	Gives the maximum number of cells over which the agent can fire.
$probFixWater$	Gives the probability that the agent will fix the water supply at a sub-timestep if they are situated at the cell with the fault, $0 \leq probFixWater \leq 1$.
$probFixElec$	Gives the probability that the agent will fix the electricity supply at a sub-timestep if they are situated at the cell with the fault, $0 \leq probFixElec \leq 1$.

C.2.2 Additional NGO Parameters

$civInNeedWeight$	Gives the weight towards cells with Civilians in need.
$noWaterWeight$	Gives the weight towards cells with no water supply.
$noElecWeight$	Gives the weight towards cells with no electricity supply.
$probFixWater$	Gives the probability that the agent will fix the water supply at a sub-timestep if they are situated at the cell with the fault, $0 \leq probFixWater \leq 1$.
$probFixElec$	Gives the probability that the agent will fix the electricity supply at a sub-timestep if they are situated at the cell with the fault, $0 \leq probFixElec \leq 1$.

C.2.3 Additional Insurgent Parameters

$combatWeight$	Gives the weight towards cells with conflict.
$sSKP$	Gives the single shot kill probability for the agent, $0 \leq sSKP \leq 1$.
$fireRange$	Gives the maximum number of cells over which the agent can fire.
$bombRadius$	Defines the area that is affected by a bomb blast. Taking the agent's current location as the centre, the bomb affects the square of cells with side length $2 * bombRadius + 1$.

Appendix D

MODEL VERIFICATION

D.1 Attract

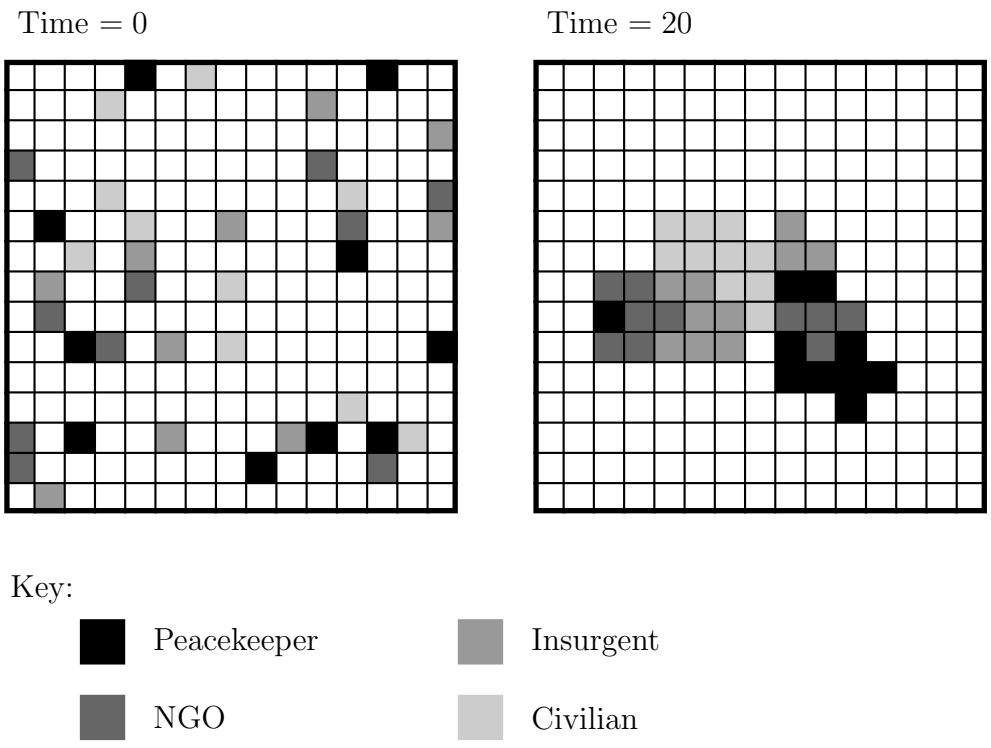

Figure D.1: Model Verification A2

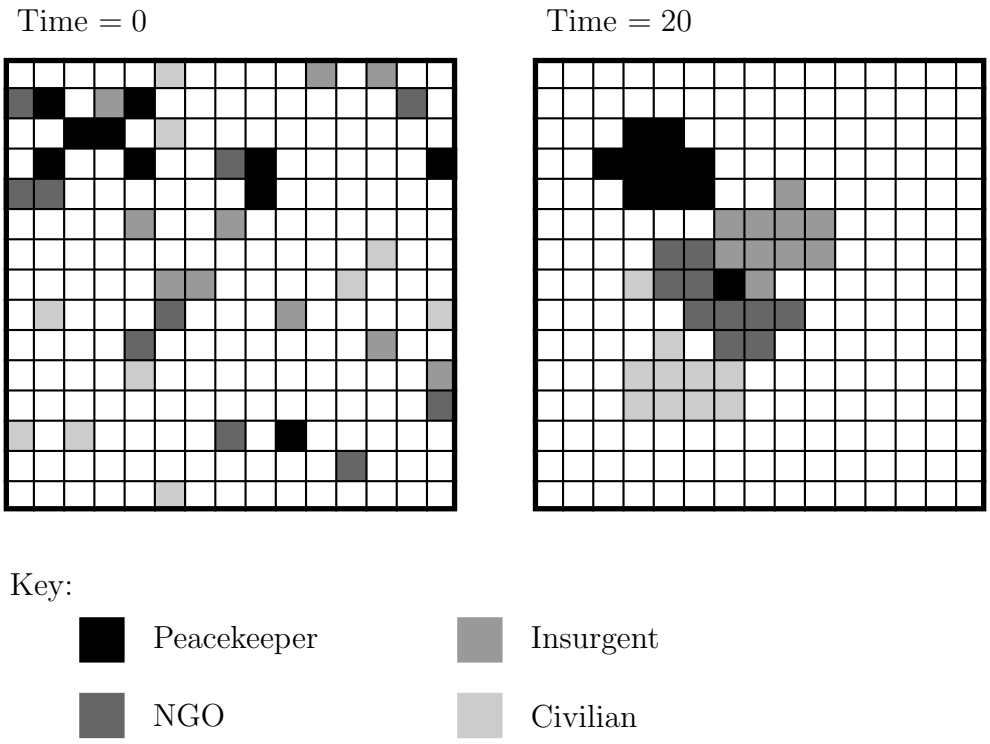

Figure D.2: Model Verification A3

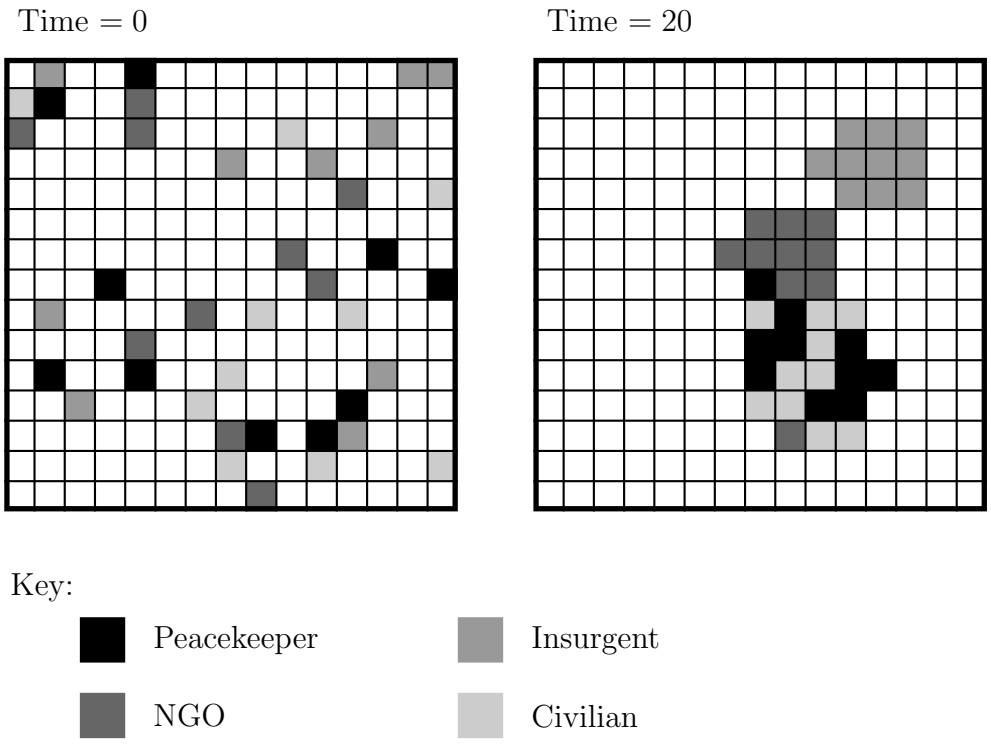

Figure D.3: Model Verification A4

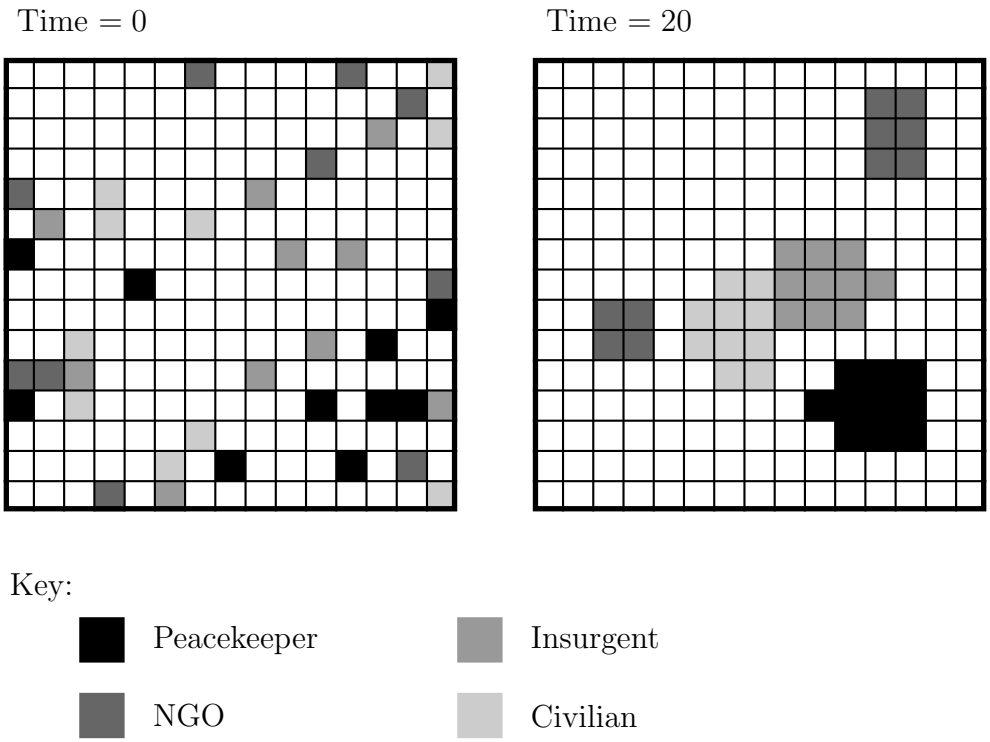

Figure D.4: Model Verification A5

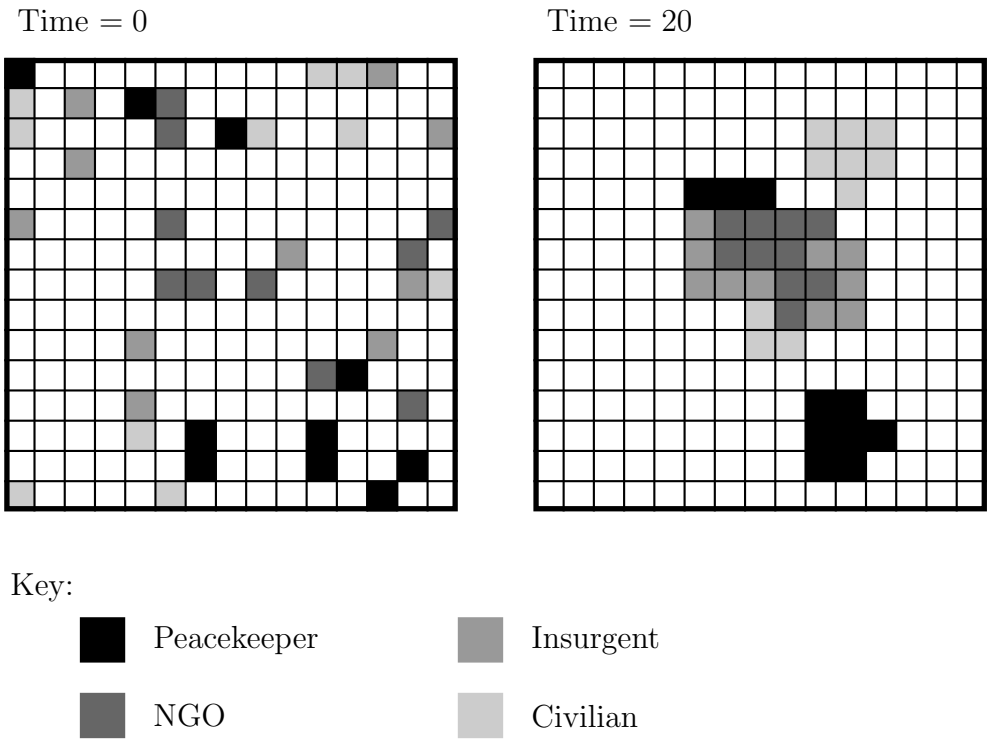

Figure D.5: Model Verification A6

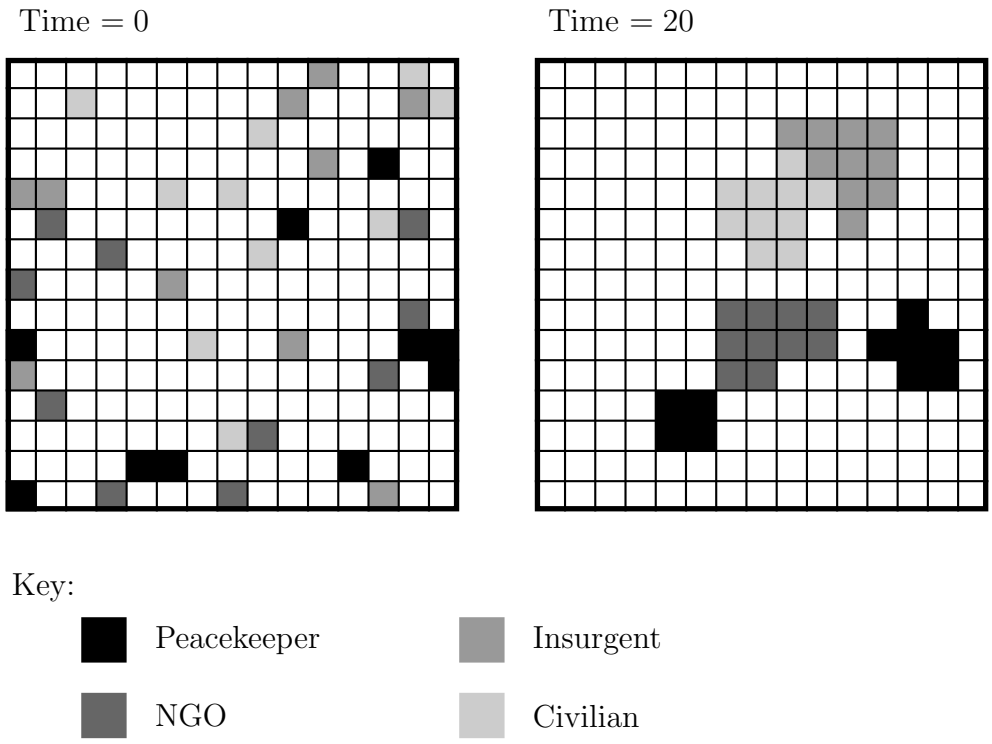

Figure D.6: Model Verification A7

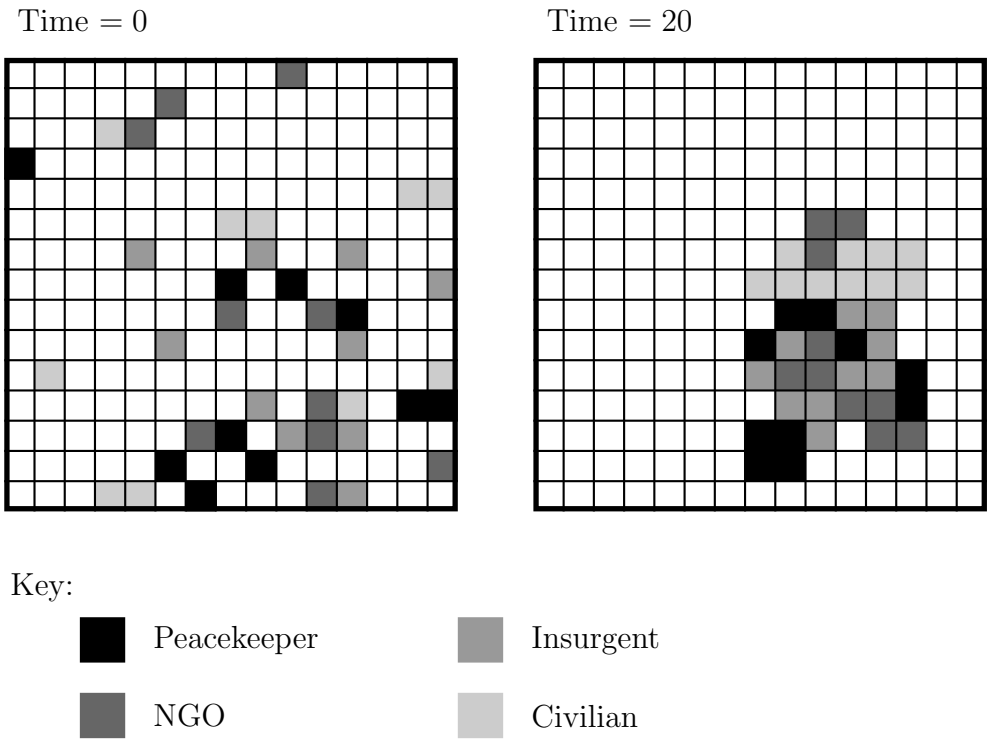

Figure D.7: Model Verification A8

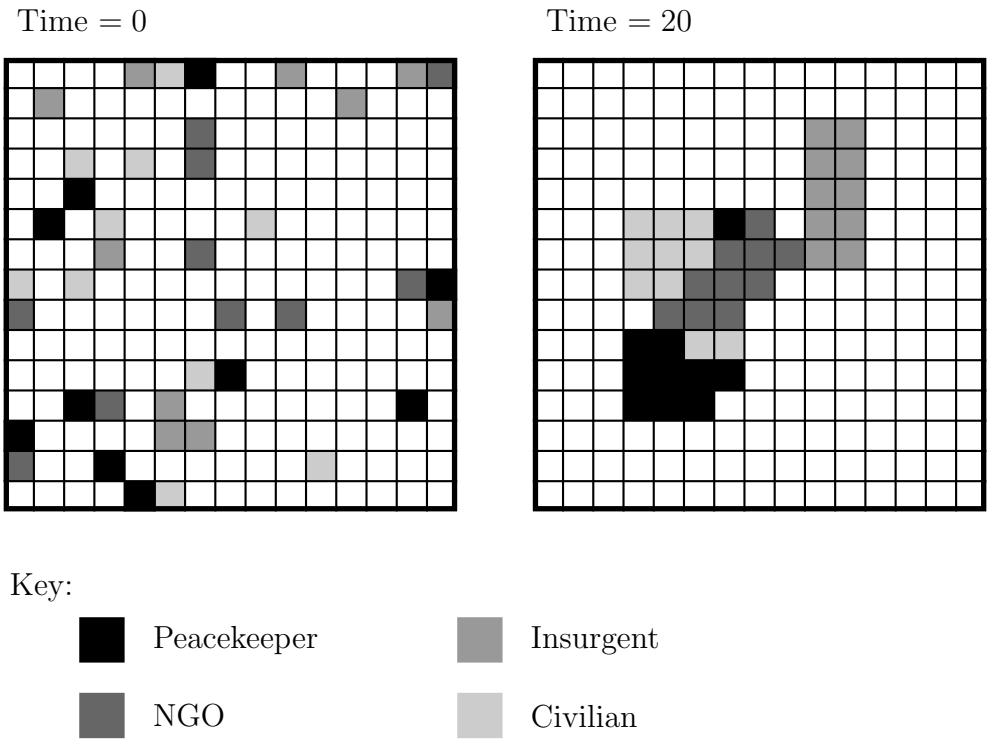

Figure D.8: Model Verification A9

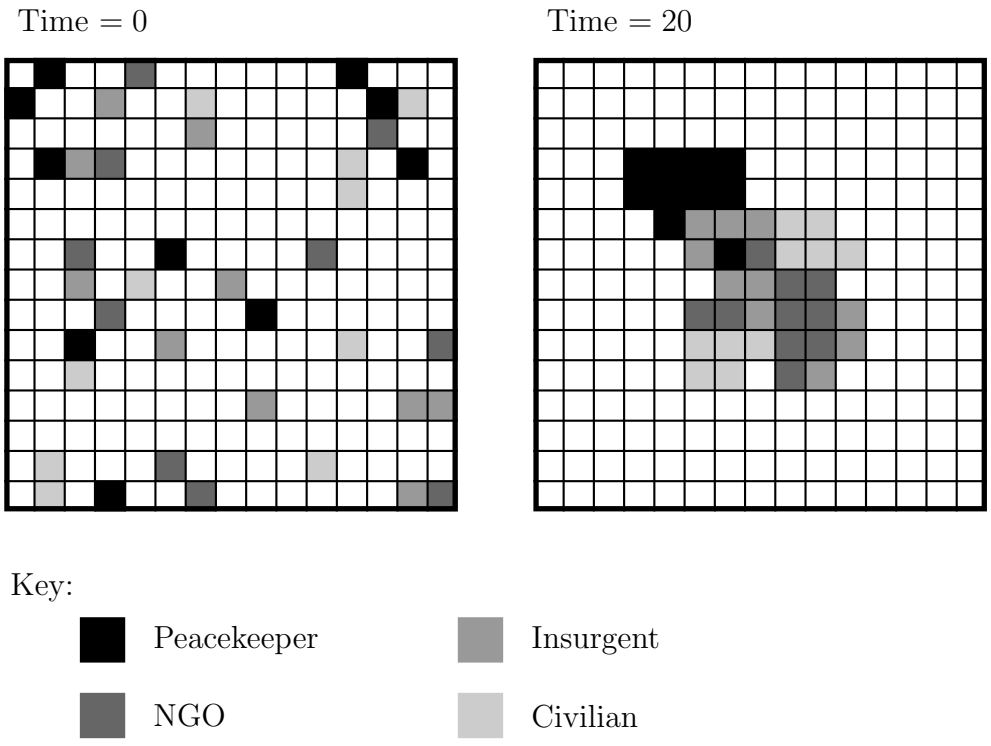

Figure D.9: Model Verification A10

D.2 Attract and Repel

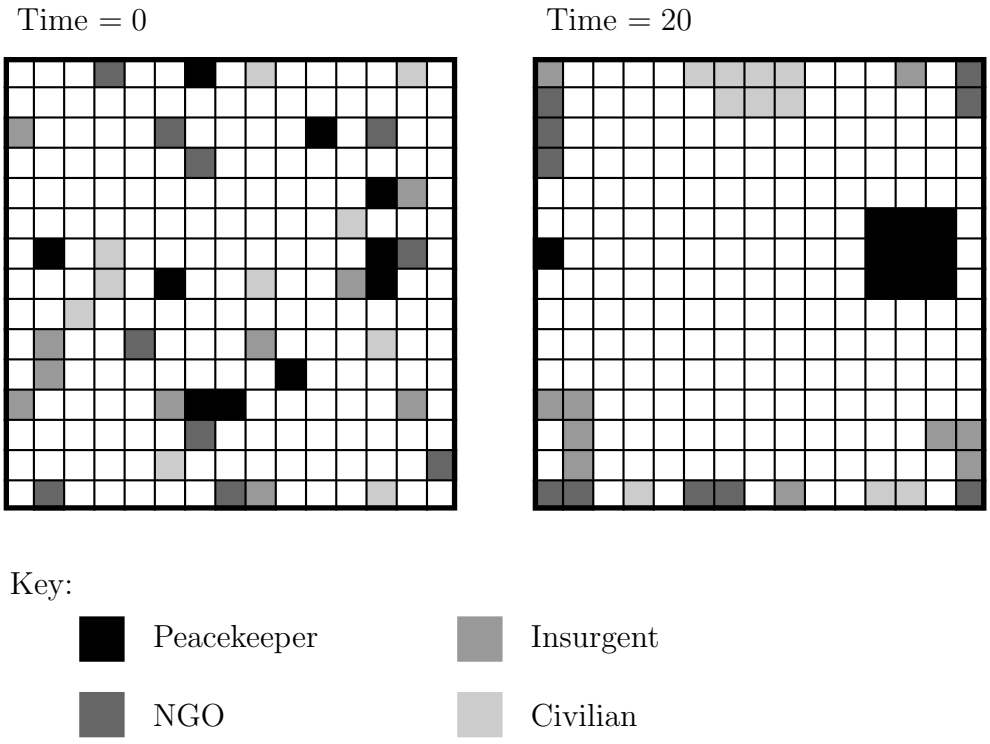

Figure D.10: Model Verification AR2

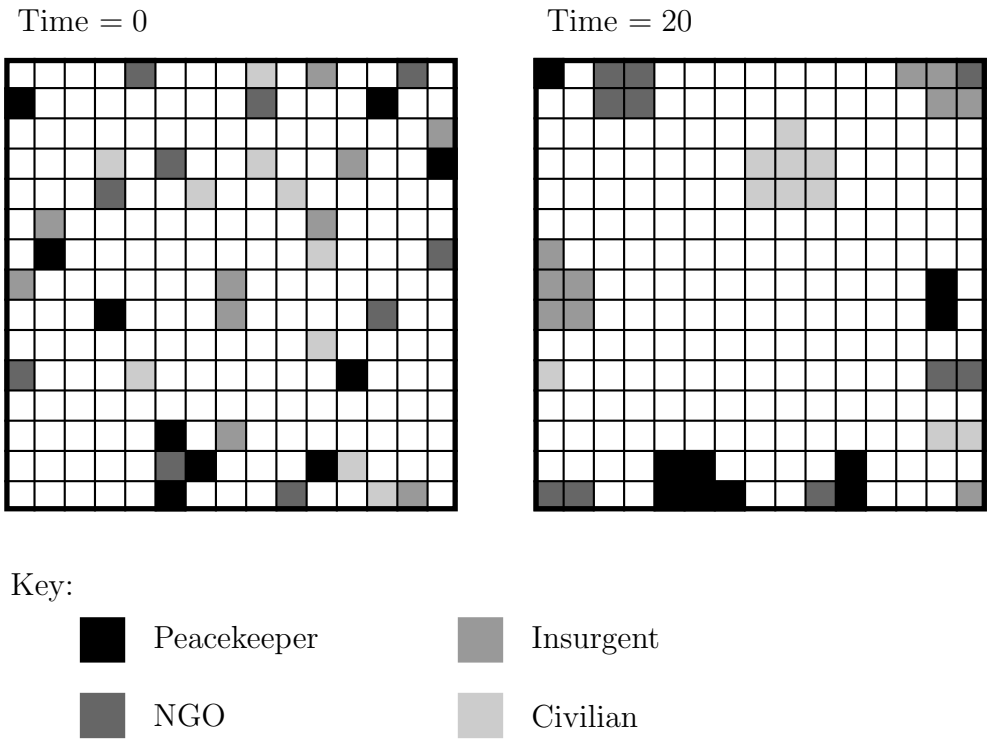

Figure D.11: Model Verification AR3

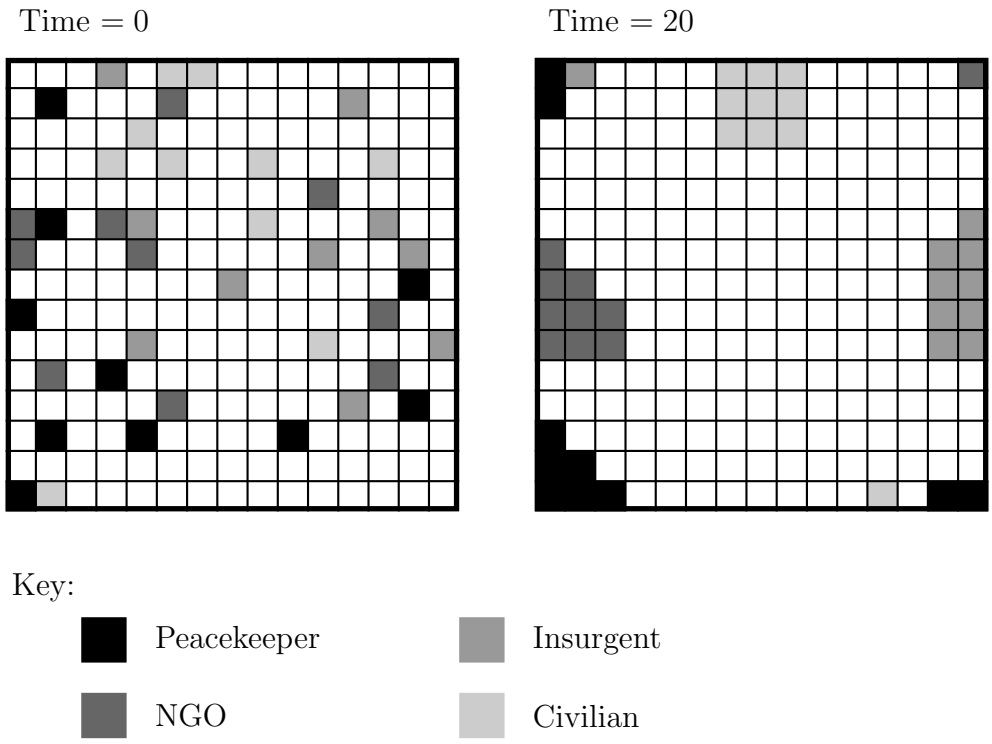

Figure D.12: Model Verification AR4

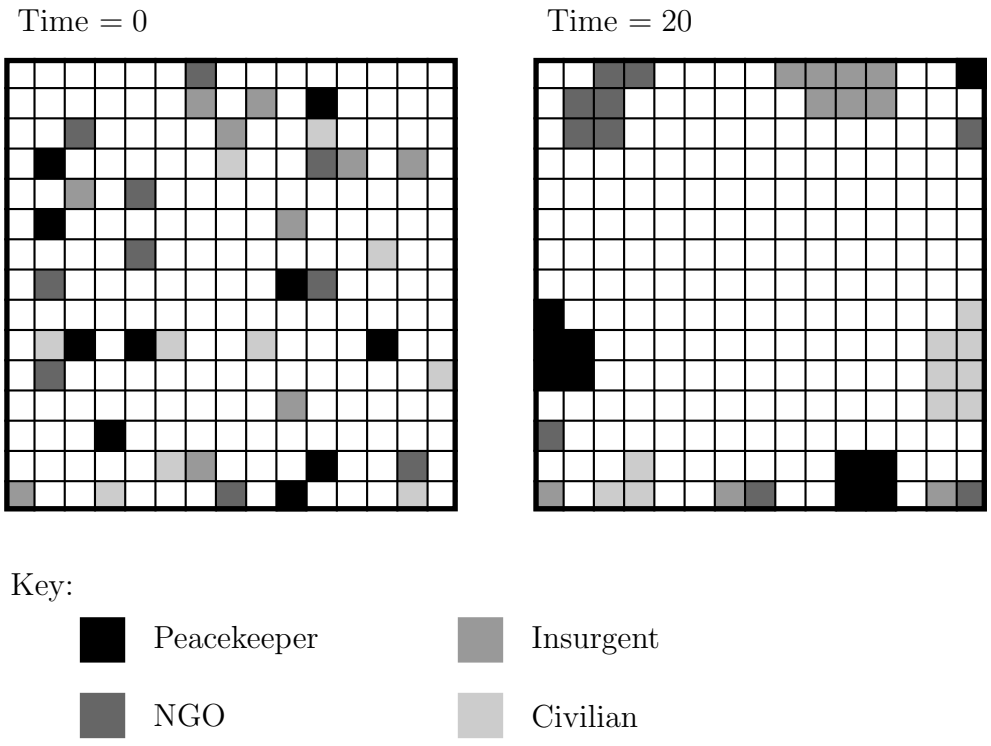

Figure D.13: Model Verification AR5

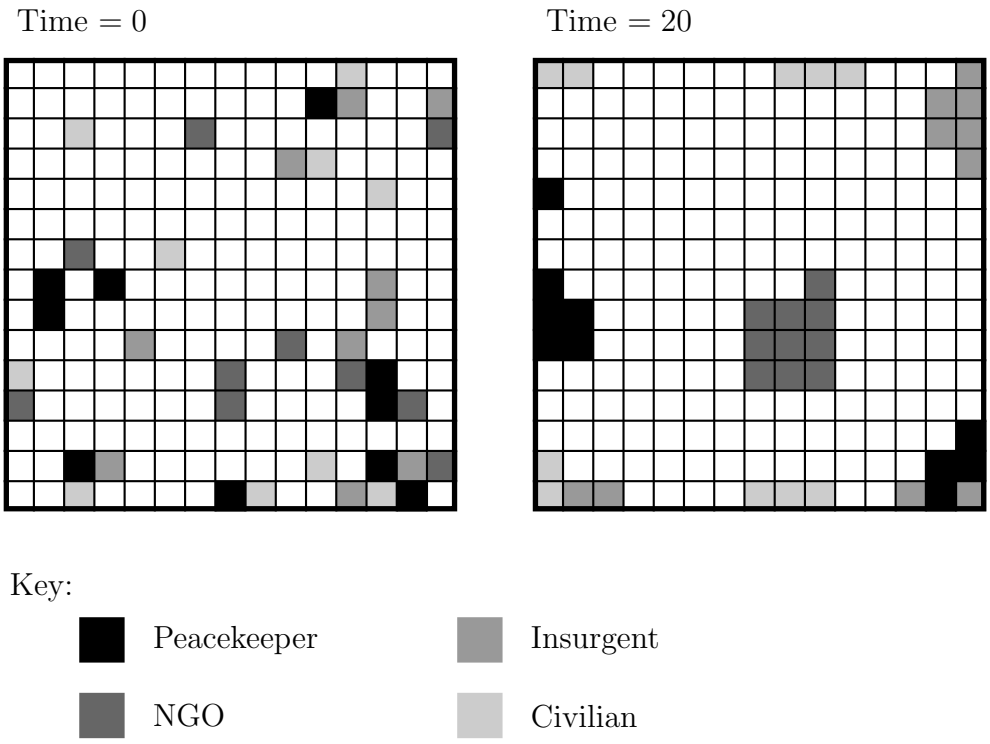

Figure D.14: Model Verification AR6

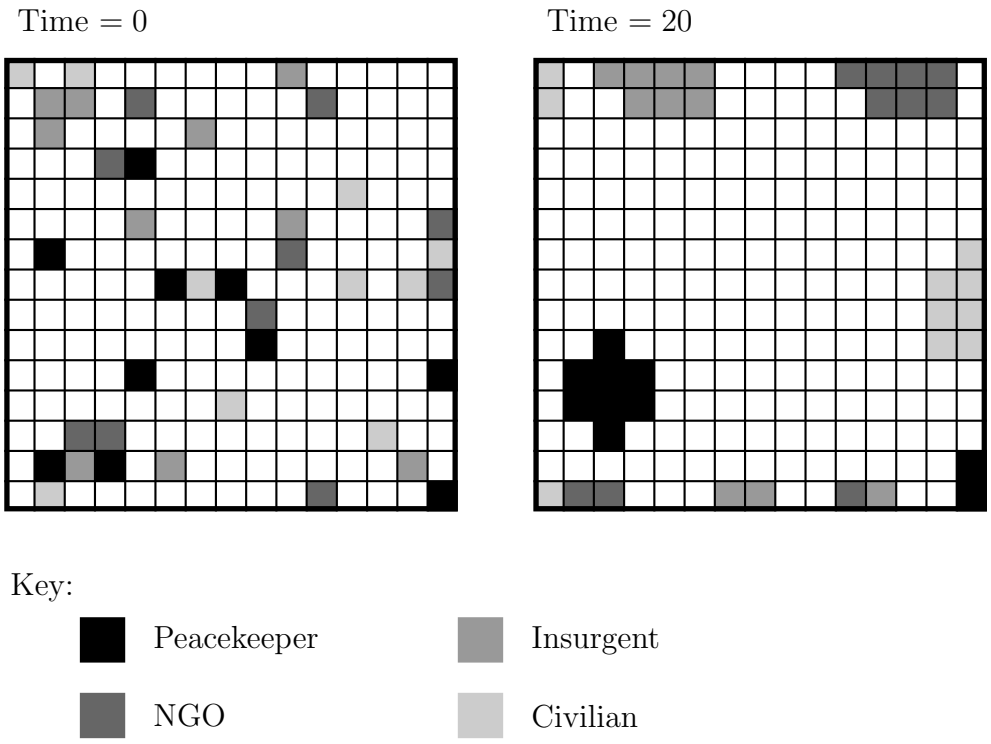

Figure D.15: Model Verification AR7

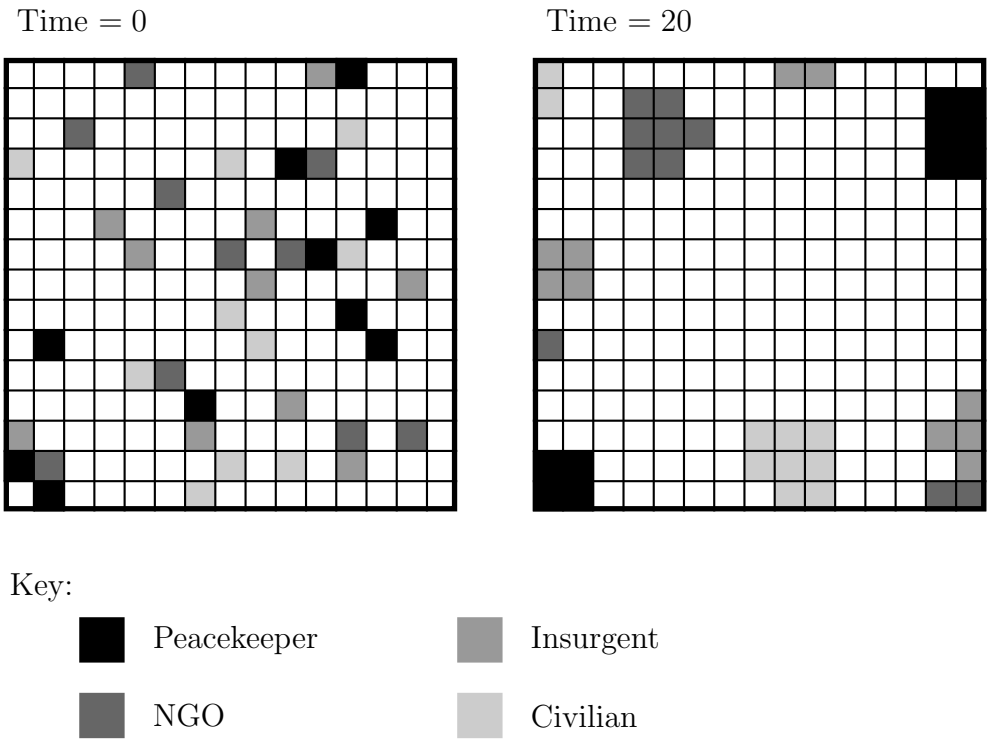

Figure D.16: Model Verification AR8

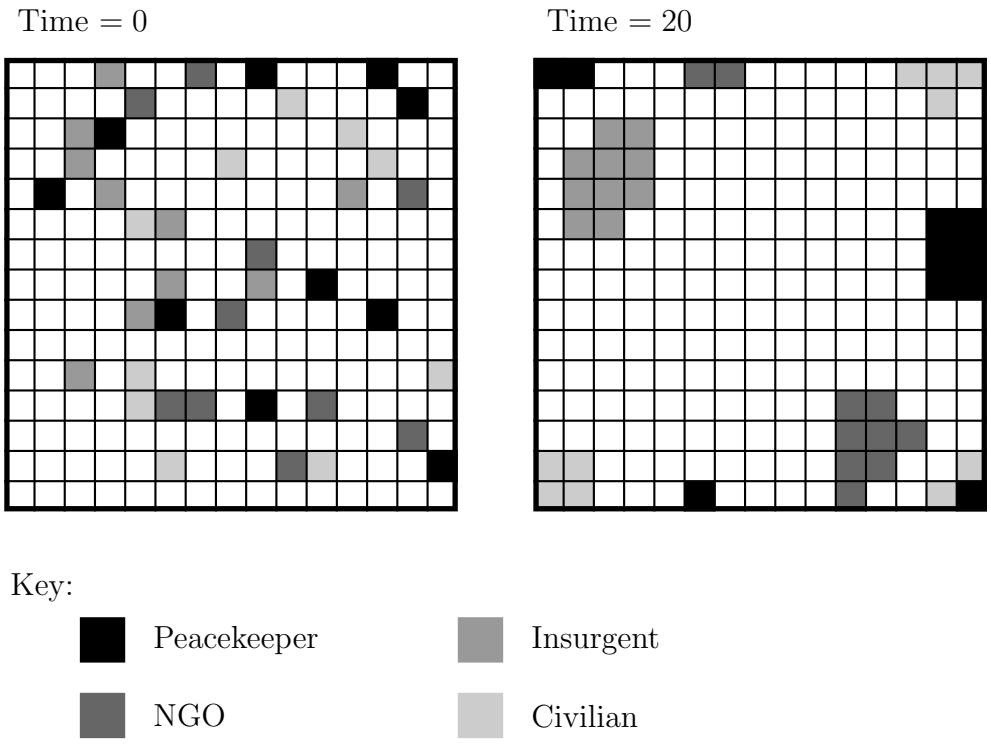

Figure D.17: Model Verification AR9

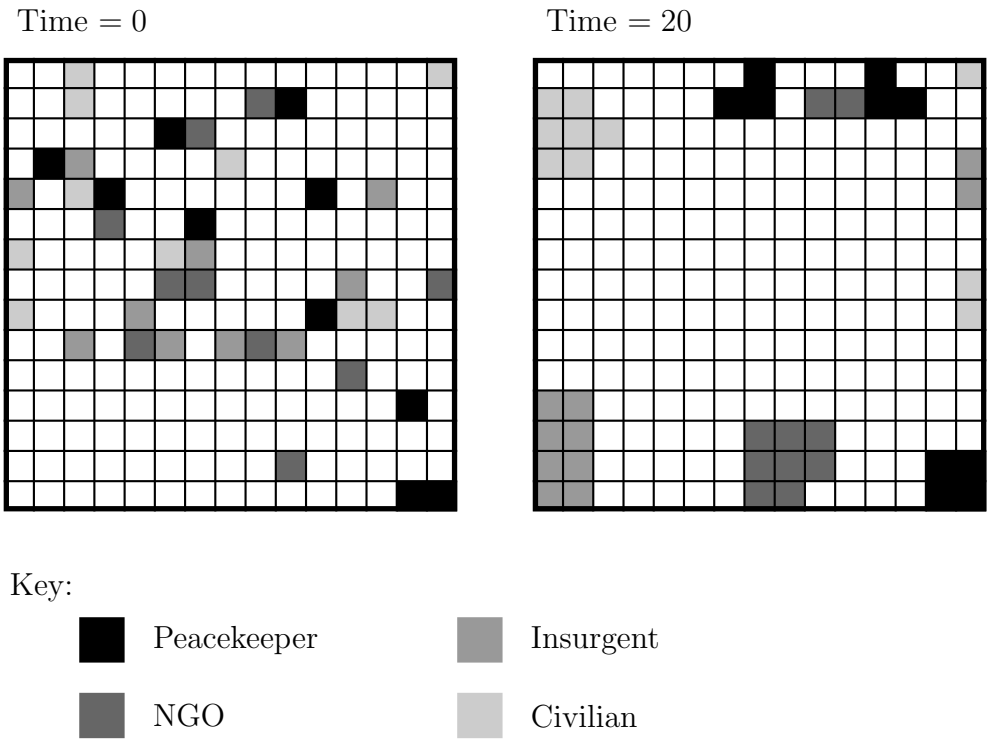

Figure D.18: Model Verification AR10

Appendix E

MATLAB PROGRAMS

E.1 Model Visualisation

```
function gridPositions(GridArray, sidelength, time)
% Function gridPositions(GridArray, sidelength, time).
% Function gridPositions takes the 'GridArray' output matrix from the PSO
% ABM and creates a series of plots that show how the agents move around
% the grid. The agents are given as coloured squares on the plots, the
% peacekeepers are blue, the insurgents red, civilians green and NGOs
% yellow.  The input values 'sidelength' and 'time' are the grid side
% length and the model run time respectively.

pause on

% First plot the initial grid, this corresponds to the first 'sidelength'
% rows of the GridArray matrix.

Initial = GridArray(1:sidelength,:);

% Find the peacekeepers and plot them as blue squares
[a,b] = find(Initial==1);
figure(gcf)
plot(b,-a,'bs','MarkerFaceColor','b','MarkerSize',2);
axis([0 (sidelength+1) -(sidelength+1) 0])
axis square
set(gca,'XTickLabel',[],'YTickLabel',[]);

hold on

% Find the insurgents and plot them as red squares
[c,d] = find(Initial==3);
figure(gcf)
plot(d,-c,'rs','MarkerFaceColor','r','MarkerSize',2);
axis([0 (sidelength+1) -(sidelength+1) 0])
```

```
axis square
set(gca,'XTickLabel',[],'YTickLabel',[]);

hold on

% Find the NGOs and plot them as green squares
[e,f] = find(Initial==2);
figure(gcf)
plot(f,-e,'gs','MarkerFaceColor','g','MarkerSize',2);
axis([0 (sidelength+1) -(sidelength+1) 0])
axis square
set(gca,'XTickLabel',[],'YTickLabel',[]);

hold on

% Find the civilians and plot them as black squares
[g,h] = find(Initial==4);
figure(gcf)
plot(h,-g,'ks','MarkerFaceColor','k','MarkerSize',2);
axis([0 (sidelength+1) -(sidelength+1) 0])
axis square
set(gca,'XTickLabel',[],'YTickLabel',[]);

title(['Time = 0']);

hold off

% Now move on to plot the positions as the model is run over 'time'
% timesteps

for t = 1:time

    % Isolate the relevant part of the 'GridArray' matrix
    Grid = GridArray((t*sidelength+1):(t+1)*sidelength,:);

    % Find the peacekeepers and plot them as blue squares
    [i,j] = find(Grid==1);
    figure(gcf)
    plot(j,-i,'bs','MarkerFaceColor','b','MarkerSize',2);
    axis([0 (sidelength+1) -(sidelength+1) 0])
    axis square
    set(gca,'XTickLabel',[],'YTickLabel',[]);

    hold on

    % Find the insurgents and plot them as red squares
    [i,j] = find(Grid==3);
    figure(gcf)
    plot(j,-i,'rs','MarkerFaceColor','r','MarkerSize',2);
    axis([0 (sidelength+1) -(sidelength+1) 0])
    axis square
```

```
    set(gca,'XTickLabel',[],'YTickLabel',[]);

    hold on

    % Find the NGOs and plot them as green squares
    [i,j] = find(Grid==2);
    figure(gcf)
    plot(j,-i,'gs','MarkerFaceColor','g','MarkerSize',2);
    axis([0 (sidelength+1) -(sidelength+1) 0])
    axis square
    set(gca,'XTickLabel',[],'YTickLabel',[]);

    hold on

    % Find the civilians and plot them as black squares
    [i,j] = find(Grid==4);
    figure(gcf)
    plot(j,-i,'ks','MarkerFaceColor','k','MarkerSize',2);
    axis([0 (sidelength+1) -(sidelength+1) 0])
    axis square
    set(gca,'XTickLabel',[],'YTickLabel',[]);

    title(['Time = ',int2str(t)]);

    hold off

    pause(0.1)

end
```

E.2 Avalanche Analysis

E.2.1 Avalanches (Firing Only)

```
function ConflictSize = avalanche(ShotArray, shotmemory)
% Function ConflictSize = avalanche(ShotArray, shotmemory).
% Function avalanche uses the 'ShotArray' output and the shotmemory
% constant from the PSO ABM and uses it to calculate the number of shots
% fired as a result of an Insurgent firing an unprovoked shot.  The results
% are given in the array ConflictSize where the first entry is the size of
% the first attack, the second is the size of the second attack and so on.

% Find the number of shots by using the 'size' function to give the number
% of rows in ShotArray
Dim = size(ShotArray);
numberofshots = Dim(1);

% If there's only one row in ShotArray then there is only one conflict of
% size one.  Record this and exit the program.
if numberofshots==1
```

```
    ConflictSize = 1;
    return
end

% If there's more than one shot then there may be more than one conflict,
% we go through the ShotArray matrix to determine the conflict each entry
% belongs to and store this in the ConflictNumber array.
% Initialise the ConflictNumber array to zero.
ConflictNumber = zeros(numberofshots,1);

% The first entry in the matrix must be part of the first conflict, store
% this as the first entry of the ConflictNumber array.
ConflictNumber(1) = 1;

% Initialise the counter for the number of conflicts.
count = 1;

% Go through the rest of the shots and determine which conflict they belong
% to and record this number in the array ConflictNumber
for i = 2:numberofshots

    for j = 1:(i-1)

        % Check to see if the location matches a previous target or shot
        % source
        if (((ShotArray(j,7)==ShotArray(i,3)) & ...
                (ShotArray(j,8)==ShotArray(i,4))) | ...
            ((ShotArray(j,3)==ShotArray(i,3)) & ...
                           (ShotArray(j,4)==ShotArray(i,4))))

            % Then check to see if the shot was fired within the previous
            % 'shotmemory' timesteps, if so then the shot was part of this
            % conflict.
            if ((ShotArray(j,1)>(ShotArray(i,1)-shotmemory)) | ...
                ((ShotArray(j,1)==(ShotArray(i,1)-shotmemory)) ...
                             & (ShotArray(j,2)>=ShotArray(i,2))))

                ConflictNumber(i) = ConflictNumber(j);

            end

        end

    end

    % If ConflictNumber(i) is still zero then the shot must be part of a
    % new conflict.  If so, increment 'count' and set ConflictNumber(i) to
    % this new value.
    if ConflictNumber(i)==0

        count = count + 1;
```

```
        ConflictNumber(i) = count;

    end

end

% Should now have 'count' conflicts, initialise an array ConflictSize to
% hold the size of each of these.
ConflictSize = zeros(count,1);

% Go through the ConflictNumber array to calculate the values for the
% ConflictArray matrix.
for k = 1:numberofshots

    ConflictSize(ConflictNumber(k)) = ConflictSize(ConflictNumber(k)) + 1;

end
```

E.2.2 Avalanches (Bombs and Firing)

```
function ConflictSize = mixedAvalanche(ShotArray, shotmemory, BombArray, ...
                                       bombmemory)
% Function ConflictSize = mixedAvalanche(ShotArray, shotmemory,
%                                        BombArray, bombmemory).
% Function mixedAvalanche uses the 'ShotArray' and 'BombArray' output and
% the shotmemory and bombmemory constants from the PSO ABM and uses it to
% calculate the sizes of the conflicts resulting from either a suicide bomb
% attack or an unprovoked shot being fired by an Insurgent.  The results
% are given in the array ConflictSize where the first entry is the size of
% the first attack, the second is the size of the second attack and so on.

% Find the number of shots and number of bombs by using the 'size' function
% to give the number of rows in ShotArray and BombArray respectively.
Dim = size(ShotArray);
numberofshots = Dim(1);
Dim = size(BombArray);
numberofbombs = Dim(1);

% We go through the BombArray and ShotArray matrices to determine the
% conflict each entry belongs to and store this in the ConflictNumber
% array.  The first 'numberofbombs' entries refer to the bomb attacks, the
% next 'numberofshots' entries refer to the shots.
% Initialise the ConflictNumber array to zero.
ConflictNumber = zeros((numberofbombs+numberofshots),1);

% The bomb attacks must be at the start of a conflict so label them as
% conflict numbers one to 'numberofbombs'
for i = 1:numberofbombs

    ConflictNumber(i) = i;
```

```
end

% Go through the ShotArray matrix for each bomb attack to determine which,
% if any, shots are a direct result of this bomb.
for i = 1:numberofbombs

    for j = 1:numberofshots

        % Check if the location of the shooter is within bomb range.
        if ((abs(ShotArray(j,3)-BombArray(i,3))<=BombArray(i,5)) & ...
                (abs(ShotArray(j,4)-BombArray(i,4))<=BombArray(i,5)))

            % Then check to see if the time is within 'bombmemory'
            % timesteps of the bomb.  If so then the shot was part of
            % this conflict.
            if ((BombArray(i,1)>=(ShotArray(j,1)-bombmemory)) | ...
                ((BombArray(i,1)==(ShotArray(j,1)-bombmemory)) ...
                                & (BombArray(i,2)>=ShotArray(j,2))))

                ConflictNumber(j+numberofbombs) = i;

            end

        end

    end

end

% Initialise the counter for the number of conflicts, we already know we
% have at least 'numberofbombs' conflicts.
count = numberofbombs;

% Go through ShotArray again and determine which conflict the shots that
% haven't been accounted for yet belong to, record this number in the array
% ConflictNumber
for j = 1:numberofshots

    % If ConflictNumber(j+numberofbombs) is zero then search through the
    % previous shots to find which conflict it belongs to.  If it does not
    % belong to a previous conflict then it must be the start of a new one.

    if ConflictNumber(j+numberofbombs)==0

        % Search through the previous shots
        for i = 1:(j-1)

            % Check to see if the location matches a previous target or
            % shot source
            if (((ShotArray(i,7)==ShotArray(j,3)) & ...
```

```
                    (ShotArray(i,8)==ShotArray(j,4))) | ...
                ((ShotArray(i,3)==ShotArray(j,3)) & ...
                     (ShotArray(i,4)==ShotArray(j,4)))

                % Then check to see if the shot was fired within the
                % previous 'shotmemory' timesteps, if so then the shot was
                % part of this conflict.
                if ((ShotArray(i,1)>(ShotArray(j,1)-shotmemory)) | ...
                    ((ShotArray(i,1)==(ShotArray(j,1)-shotmemory)) ...
                                & (ShotArray(i,2)>=ShotArray(j,2))))

                    ConflictNumber(j+numberofbombs) = ...
                                        ConflictNumber(i+numberofbombs);

                end

            end

        end

        % If ConflictNumber(j+numberofbombs) is still zero then the shot
        % must be part of a new conflict.  If so, increment 'count' and set
        % ConflictNumber(j+numberofbombs) to this new value.
        if ConflictNumber(j+numberofbombs)==0

            count = count + 1;
            ConflictNumber(j+numberofbombs) = count;

        end

    end

end


% Should now have 'count' conflicts, initialise an array ConflictSize to
% hold the size of each of these.
ConflictSize = zeros(count,1);

% *******NB: This version gives a value for conflict size where a bomb has
% the same value as a shot ***********************************************

% Go through the ConflictNumber array to calculate the values for the
% ConflictArray matrix.
for k = 1:(numberofbombs+numberofshots)

    ConflictSize(ConflictNumber(k)) = ConflictSize(ConflictNumber(k)) + 1;

end
```

E.2.3 Time Avalanches (Firing Only)

```
function ConflictSize = time_avalanche(ShotArray, shotmemory)
% Function ConflictSize = time_avalanche(ShotArray, shotmemory).
% Function avalanche uses the 'ShotArray' output and the shotmemory
% constant from the PSO ABM and uses it to calculate the number of
% timesteps each period of conflict covers.  The results are given in the
% array ConflictSize where the first entry is the size of the first attack,
% the second is the size of the second attack and so on.

% The first part of the program is the same as avalanche.m where we
% determine which conflict each shot belongs to.

% Find the number of shots by using the 'size' function to give the number
% of rows in ShotArray
Dim = size(ShotArray);
numberofshots = Dim(1);

% If there's only one row in ShotArray then there is only one conflict of
% size one.  Record this and exit the program.
if numberofshots==1
    ConflictSize = 1;
    return
end

% If there's more than one shot then there may be more than one conflict,
% we go through the ShotArray matrix to determine the conflict each entry
% belongs to and store this in the ConflictNumber array.
% Initialise the ConflictNumber array to zero.
ConflictNumber = zeros(numberofshots,1);

% The first entry in the matrix must be part of the first conflict, store
% this as the first entry of the ConflictNumber array.
ConflictNumber(1) = 1;

% Initialise the counter for the number of conflicts.
count = 1;

% Go through the rest of the shots and determine which conflict they belong
% to and record this number in the array ConflictNumber
for i = 2:numberofshots

    for j = 1:(i-1)

        % Check to see if the location matches a previous target or shot
        % source
        if (((ShotArray(j,7)==ShotArray(i,3)) & ...
                (ShotArray(j,8)==ShotArray(i,4))) | ...
            ((ShotArray(j,3)==ShotArray(i,3)) & ...
                        (ShotArray(j,4)==ShotArray(i,4))))

            % Then check to see if the shot was fired within the previous
```

```
            % 'shotmemory' timesteps, if so then the shot was part of this
            % conflict.
            if ((ShotArray(j,1)>(ShotArray(i,1)-shotmemory)) | ...
                ((ShotArray(j,1)==(ShotArray(i,1)-shotmemory)) ...
                            & (ShotArray(j,2)>=ShotArray(i,2))))

                ConflictNumber(i) = ConflictNumber(j);

            end

        end

    end

    % If ConflictNumber(i) is still zero then the shot must be part of a
    % new conflict.  If so, increment 'count' and set ConflictNumber(i) to
    % this new value.
    if ConflictNumber(i)==0

        count = count + 1;
        ConflictNumber(i) = count;

    end

end

% Now instead of counting the number of shots, we calculate the number of
% timesteps that include some part of the conflict.

% Go through each conflict in turn.
for k = 1:count

    % Find the first shot from this conflict and record its position in the
    % ConflictNumber array.
    firstattack = find(ConflictNumber == k, 1, 'first');

    % Find the last shot from this conflict and record its position in the
    % ConflictNumber array.
    lastattack = find(ConflictNumber == k, 1, 'last');

    % Now calculate the number of timesteps the conflict affects and record
    % this in the ConflictSize array.
    ConflictSize(k) = ShotArray(lastattack, 1) - ...
                          ShotArray(firstattack, 1) + 1;

end
```

E.2.4 Time Avalanches (Bombs and Firing)

```
function ConflictSize = time_mixedAvalanche(ShotArray, shotmemory, ...
```

```
                                                BombArray, bombmemory)
% Function ConflictSize = time_mixedAvalanche(ShotArray, shotmemory,
%                                               BombArray, bombmemory).
% Function mixedAvalanche uses the 'ShotArray' and 'BombArray' output and
% the shotmemory and bombmemory constants from the PSO ABM and uses it to
% calculate the sizes of the conflicts resulting from either a suicide bomb
% attack or an unprovoked shot being fired by an Insurgent.  Here the size
% refers to the number of timesteps affected by a conflict.  The results
% are given in the array ConflictSize where the first entry is the size of
% the first attack, the second is the size of the second attack and so on.

% The first part of the program is the same as mixedAvalanche.m where we
% determine which conflict each shot belongs to.

% Find the number of shots and number of bombs by using the 'size' function
% to give the number of rows in ShotArray and BombArray respectively.
Dim = size(ShotArray);
numberofshots = Dim(1);
Dim = size(BombArray);
numberofbombs = Dim(1);

% We go through the BombArray and ShotArray matrices to determine the
% conflict each entry belongs to and store this in the ConflictNumber
% array.  The first 'numberofbombs' entries refer to the bomb attacks, the
% next 'numberofshots' entries refer to the shots.
% Initialise the ConflictNumber array to zero.
ConflictNumber = zeros((numberofbombs+numberofshots),1);

% The bomb attacks must be at the start of a conflict so label them as
% conflict numbers one to 'numberofbombs'
for i = 1:numberofbombs

    ConflictNumber(i) = i;

end

% Go through the ShotArray matrix for each bomb attack to determine which,
% if any, shots are a direct result of this bomb.
for i = 1:numberofbombs

    for j = 1:numberofshots

        % Check if the location of the shooter is within bomb range.
        if ((abs(ShotArray(j,3)-BombArray(i,3))<=BombArray(i,5)) & ...
                (abs(ShotArray(j,4)-BombArray(i,4))<=BombArray(i,5)))

            % Then check to see if the time is within 'bombmemory'
            % timesteps of the bomb.  If so then the shot was part of
            % this conflict.
            if ((BombArray(i,1)>=(ShotArray(j,1)-bombmemory)) | ...
                ((BombArray(i,1)==(ShotArray(j,1)-bombmemory)) ...
```

```
                                & (BombArray(i,2)>=ShotArray(j,2))))

                ConflictNumber(j+numberofbombs) = i;

            end

        end

    end

end

% Initialise the counter for the number of conflicts, we already know we
% have at least 'numberofbombs' conflicts.
count = numberofbombs;

% Go through ShotArray again and determine which conflict the shots that
% haven't been accounted for yet belong to, record this number in the array
% ConflictNumber
for j = 1:numberofshots

    % If ConflictNumber(j+numberofbombs) is zero then search through the
    % previous shots to find which conflict it belongs to.  If it does not
    % belong to a previous conflict then it must be the start of a new one.

    if ConflictNumber(j+numberofbombs)==0

        % Search through the previous shots
        for i = 1:(j-1)

            % Check to see if the location matches a previous target or
            % shot source
            if (((ShotArray(i,7)==ShotArray(j,3)) & ...
                        (ShotArray(i,8)==ShotArray(j,4))) | ...
                ((ShotArray(i,3)==ShotArray(j,3)) & ...
                         (ShotArray(i,4)==ShotArray(j,4))))

                % Then check to see if the shot was fired within the
                % previous 'shotmemory' timesteps, if so then the shot was
                % part of this conflict.
                if ((ShotArray(i,1)>(ShotArray(j,1)-shotmemory)) | ...
                    ((ShotArray(i,1)==(ShotArray(j,1)-shotmemory)) ...
                                & (ShotArray(i,2)>=ShotArray(j,2))))

                    ConflictNumber(j+numberofbombs) = ...
                                        ConflictNumber(i+numberofbombs);

                end

            end
```

```
        end

        % If ConflictNumber(j+numberofbombs) is still zero then the shot
        % must be part of a new conflict.  If so, increment 'count' and set
        % ConflictNumber(j+numberofbombs) to this new value.
        if ConflictNumber(j+numberofbombs)==0

            count = count + 1;
            ConflictNumber(j+numberofbombs) = count;

        end

    end

end


% Now instead of counting the number of bombs and shots, we calculate the
% number of timesteps that include some part of the conflict.

% Go through each conflict in turn.
for k = 1:count

    % Find the first bomb/shot from this conflict and record its position
    % in the ConflictNumber array.
    firstattack = find(ConflictNumber == k, 1, 'first');
    % Find the timestep at which this bomb/shot occured.  We can tell from
    % the position in the ConflictNumber array whether this was a bomb or
    % shot: if first <= numberofbombs then it is a bomb, else it is a shot.
    if (firstattack <= numberofbombs)
        time_1 = BombArray(firstattack, 1);
    else
        time_1 = ShotArray((firstattack-numberofbombs), 1);
    end

    % Find the last bomb/shot from this conflict and record its position in
    % the ConflictNumber array.
    lastattack = find(ConflictNumber == k, 1, 'last');
    % Find the timestep at which this bomb/shot occured.  We can tell from
    % the position in the ConflictNumber array whether this was a bomb or
    % shot: if last <= numberofbombs then it is a bomb, else it is a shot.
    if (lastattack <= numberofbombs)
        time_2 = BombArray(lastattack, 1);
    else
        time_2 = ShotArray((lastattack-numberofbombs), 1);
    end

    % Now calculate the number of timesteps the conflict affects and record
    % this in the ConflictSize array.
    ConflictSize(k) = time_2 - time_1 + 1;
```

```
end
```

E.3 Peacekeeper Box-Counting Dimension

```
function Count = boxCount_200(GridArray, time)
% function Count = boxCount_200(GridArray, time)
% Function boxCount_200 takes the array of grid positions on a 200x200 grid
% for 'time' timesteps and calculates the box-counting dimension of the
% cluster of Peacekeepers at each timestep, these results are recorded in
% the 'Count' vector.  The box-counting dimension is an estimation of the
% fractal dimension of a cluster.

% First define a vector of sidelengths for the 'boxes'.  For this we use
% the divisors of 200 for simplicity.
Sidelength = [1 2 4 5 8 10 20 25 40 50 100 200];
% Also calculate the logarithm of the vector for use in calculating the
% box-counting dimension.
L_Sidelength = log(Sidelength);

% Go through each timestep in turn and calculate the box-counting
% dimension.
for t = 0:time

    % Find the coordinates of the Peacekeepers for this timestep.
    [a, b] = find(GridArray((t*200 + 1):((t+1)*200), :));

    % Reset the vector X to hold the count for each sidelength.
    X = zeros(1, 12);

    % The count for box size 200 will clearly be one and the count for size
    % one will be the number of agents.
    X(12) = 1;
    X(1) = size(a, 1);

    % Go through the remaining box sizes in turn.
    for s = 2:11

        % Go through each box in turn and search through the Peacekeeper
        % coordinates until we either find a cell in the box or get the
        % the end of the array.
        for i = 0:(200/Sidelength(s) - 1)

            for j = 0:(200/Sidelength(s) - 1)

                % Go through each coordinate pair and see if they are in
                % this 'box', if so increment X(s) and break out of the for
                % loop.
                for k = 1:(size(a, 1))
```

```
                    if ((a(k) > i*Sidelength(s)) & ...
                            (a(k) <= (i+1)*Sidelength(s)) &...
                                  (b(k) > j*Sidelength(s)) ...
                                        & (b(k) <= (j+1)*Sidelength(s)))

                        X(s) = X(s) + 1;
                        break;

                    end

                end

            end

        end

    end

    % Calculate the logarithm of the vector X.
    L_X = log(X);

    % Estimate the box-counting dimension using a least-squares line of best
    % fit for the log-log plot of (Sidelength, X).
    linedef = polyfit(L_Sidelength, L_X, 1);
    Count(t+1) = -linedef(1);

end
```