CRANFIELD UNIVERSITY

Firas Al-Sawaf

Efficient Phase Unwrapping

SCHOOL OF ENGINEERING

PhD THESIS

CRANFIELD UNIVERSITY

SCHOOL OF ENGINEERING

PhD THESIS

Firas Al-Sawaf

Efficient Phase Unwrapping

Supervisor: R P Tatam

June 2005

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Abstract

In the field of optical interferometry, two-dimensional projections of light interference patterns are often analysed in order to obtain measurements of interest. Such interference patterns, or interferograms, contain phase information which is inherently wrapped onto the range $-\pi$ to π.

Phase unwrapping is the processes of the restoration of the unknown multiple of 2π, and therefore plays a major role in the overall process of interferogram analysis.

Unwrapping phase information correctly becomes a challenging process in the presence of noise. This is particularly the case for speckle interferograms, which are noisy by nature.

Many phase unwrapping algorithms have been devised by workers in the field, in order to achieve better noise rejection and improve the computational performance.

This thesis focuses on the computational efficiency aspect, and picks as a starting point an existing phase unwrapping algorithm which has been shown to have inherent noise immunity. This is, namely, the tile-based phase unwrapping method, which attains its enhanced noise immunity through the application of the minimum spanning tree concept from graph theory.

The thesis examines the problem of finding a minimum spanning tree, for this particular application, from a graph theory perspective, and shows that a more efficient class of minimum spanning tree algorithms can be applied to the problem.

The thesis then goes on to show how a novel algorithm can be used to significantly reduce the size of the minimum spanning tree problem in an efficient manner.

Efficient Phase Unwrapping

Contents

Figures

Equations

Chapter 1 Introduction and thesis outline

Automatic fringe analysis techniques, such as the Fourier transform [1, 2] and phase stepping[1] [3, 4, 5, 6], produce phase information which is inherently wrapped onto the range -π to π. The restoration of the unknown multiple of 2π, Figure 1, is called phase unwrapping, and is central to most such algorithms [7].

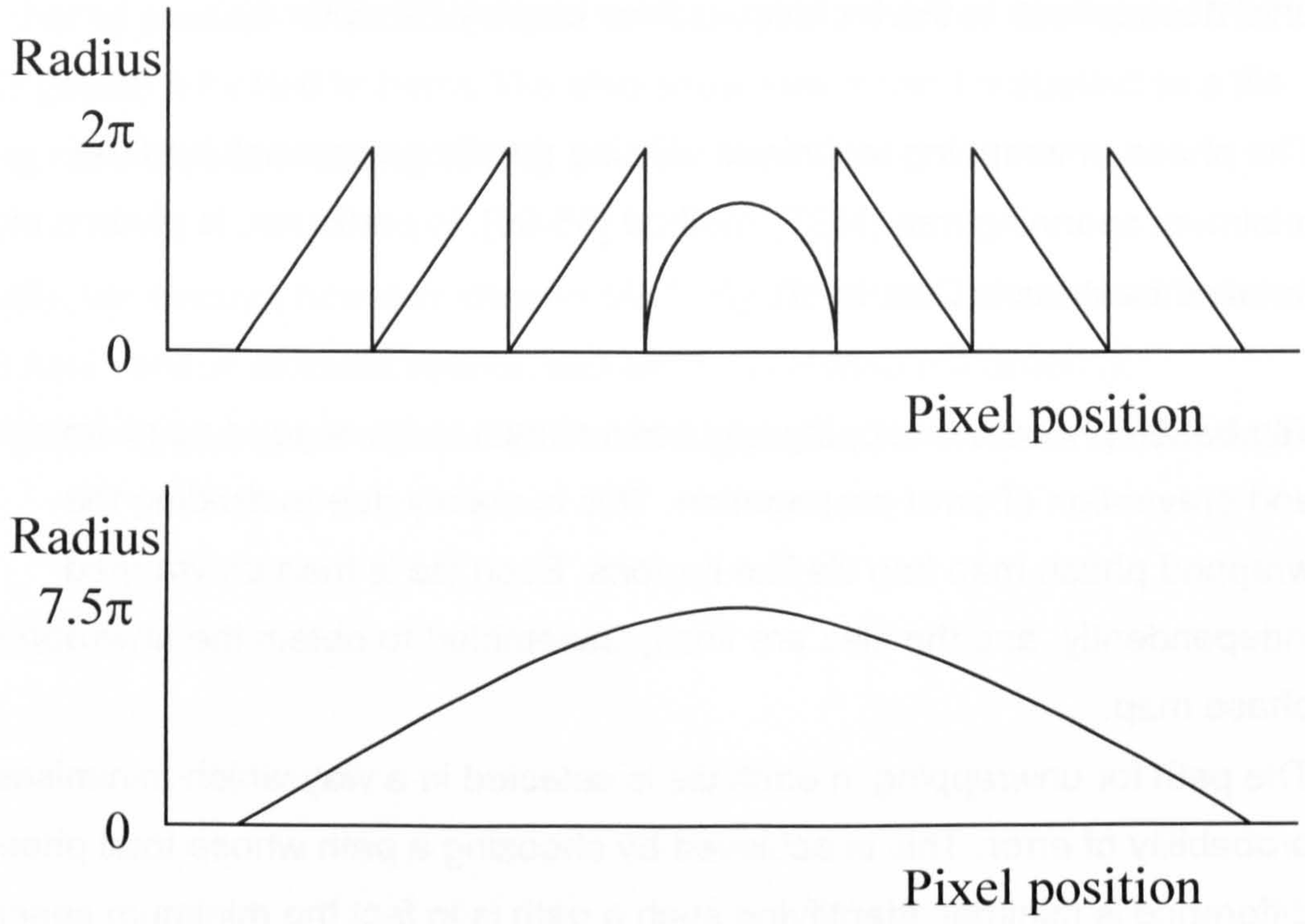


Figure 1 Wrapped phase data (top), and after unwrapping (bottom).

Phase unwrapping algorithms in general seek to achieve correctness of operation, which is particularly an issue in the presence on noise. In addition to operational correctness, phase unwrapping algorithms may also need to be computationally efficient. This can be the case, for example, when it is required for large amounts of fringe information to be processed in

---

[1] Phase stepping is also known as quasi-heterodyning.

real time. Such applications include processing vibration fringes [20], and real time control of dynamic forming [21].

In this thesis we focus on the computational efficiency aspect. We start with a brief introduction to automatic fringe analysis, highlighting the role of phase unwrapping in the process (Chapter 1).

We then review some of the most popular phase unwrapping methods, giving brief descriptions to the techniques they employ (Chapter 1).

The phase unwrapping technique utilising the tiling approach and the minimum spanning tree (MST) method [56-60], in particular, is given a more detailed treatment (Chapter 4).

Tile-based phase unwrapping has been shown to have good noise immunity and prevention of error propagation. This is mainly due to dividing the wrapped phase map into tile like regions. Each tile is then unwrapped independently, and the tiles are finally assembled to obtain the unwrapped phase map.
The path for unwrapping in each tile is selected in a way which minimises the probability of error. This is achieved by choosing a path whose total phase difference is minimal. Identifying such a path is in fact the minimum spanning tree graph theory problem.

Finding a MST for each tile is computationally demanding, and impacts the efficiency of the overall unwrapping algorithm. We examine how phase information in a tile is converted into a weighted graph, and how the topology of such a graph enables the use of a more efficient class of MST algorithms (Chapter 5).

We review some of the most popular MST algorithms in the graph theory literature (Chapter 1), including Prim which is traditionally employed by tile-based phase unwrapping.

We then proceed to describe a novel algorithm tailored specifically to a tile's topology, which helps to reduce the size of the minimum spanning tree problem to be solved for each tile in the wrapped phase map (Chapter 7). We show that our algorithm is linear-time efficient, O(n), where n is the number of pixels in a tile, and that it can be used to reduce the size of the MST problem by half or more. We also show how it can be applied to a tile using an image processing sliding window technique.

Finally, we discuss how any chosen MST algorithm can be attached to ours, and how various efficient hybrids can be thus created (Chapter 1).
We close by giving our conclusions and suggestion for future work (Chapter 1).

Blank

In

Original

Chapter 2 Automatic Fringe Analysis

The analysis of interference patterns[2] has become an established subject of research in its own right [8]. The advent in personal computers and image capturing devices, such as CCD [3] cameras, has helped simplify and advance the process. Automatic fringe analysis tools are commercially available and used widely in the industry.

An excellent introduction to the subject of fringe analysis is given in [9], and a more recent one in [10]. We give here a brief description of the general landscape of the subject, which serves as a background to the chapters which follow.

2.1. Interferometry

Interferometry is a term used to refer to the various phenomena of light interference. Interferograms are the patterns produced by the interference of light.

Light interference can be accomplished through various techniques, such as classical wavefront division and amplitude division, which are briefly described in the following sections.

Holographic techniques [11] pioneered over 40 years ago have given rise to the development of many advanced fringe measurement techniques such as holographic double-exposure interferometry, real-time interferometry, and TV-holography.

TV-holography is perhaps better known as electronic speckle pattern interferometry (ESPI). It has found wide application and is also briefly described in this chapter.

---

[2] Also known as fringe patterns, or interferograms

[3] Charge Coupled Device.

ESPI is particularly of interest as it provides the basis for the interferometric methods often used to obtain the class of speckled and noisy interferograms which are the most relevant to our research.
On the other hand, other types of interferometry utilising fringe projection techniques [12, 13, 14], based on Moiré fringes [16], typically produce interferograms with negligible noise. Thus such interferograms can be usually analysed in a straightforward manner. There are naturally exceptions to this generalisation, such as the cases whereby the object contains features which introduce complexity, such as a step change in surface profile or a high fringe density where the fringes are locally under-sampled by the detector.

Broadly speaking, however, elaborate phase unwrapping algorithms are targeted at complex and noisy interferograms, which our research focuses on.

2.2. Wavefront division interferometry

Figure 2 shows interference patterns produced by Thomas Young in an experiment conducted in 1801 [15].
The two slits are a type of interferometer which operates by dividing the light source's wavefronts.
A wavefront incident on the two slits becomes divided and emerges from the slits as two spherical wavefronts. These two wavefronts interfere, and the resulting interference pattern is observed on the screen.

Other types of wavefront division interferometers are Fresnel's biprism, Lloyd's mirror and Michelson's stellar interferometer. A good introduction to these is given in [17].

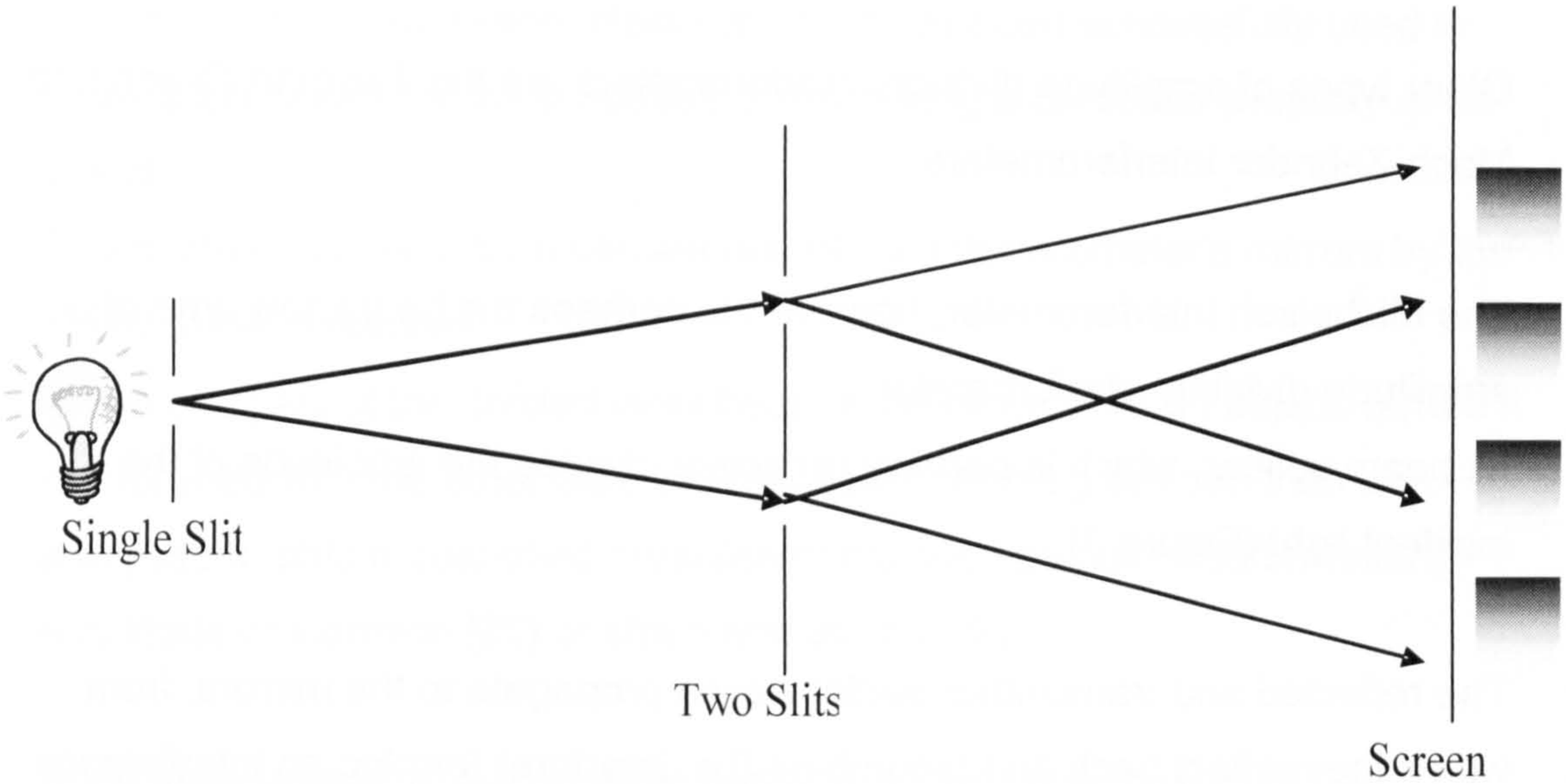


Figure 2 Light interference (Young's slits experiment)

2.3. Amplitude division

One laser source can be used to produce two coherent secondary sources [17], as is the case with the Michelson interferometer (Figure 3).

Lasers are used mainly because of their long coherence time (of the order of milliseconds) relative to conventional light sources (of the order of nano seconds).

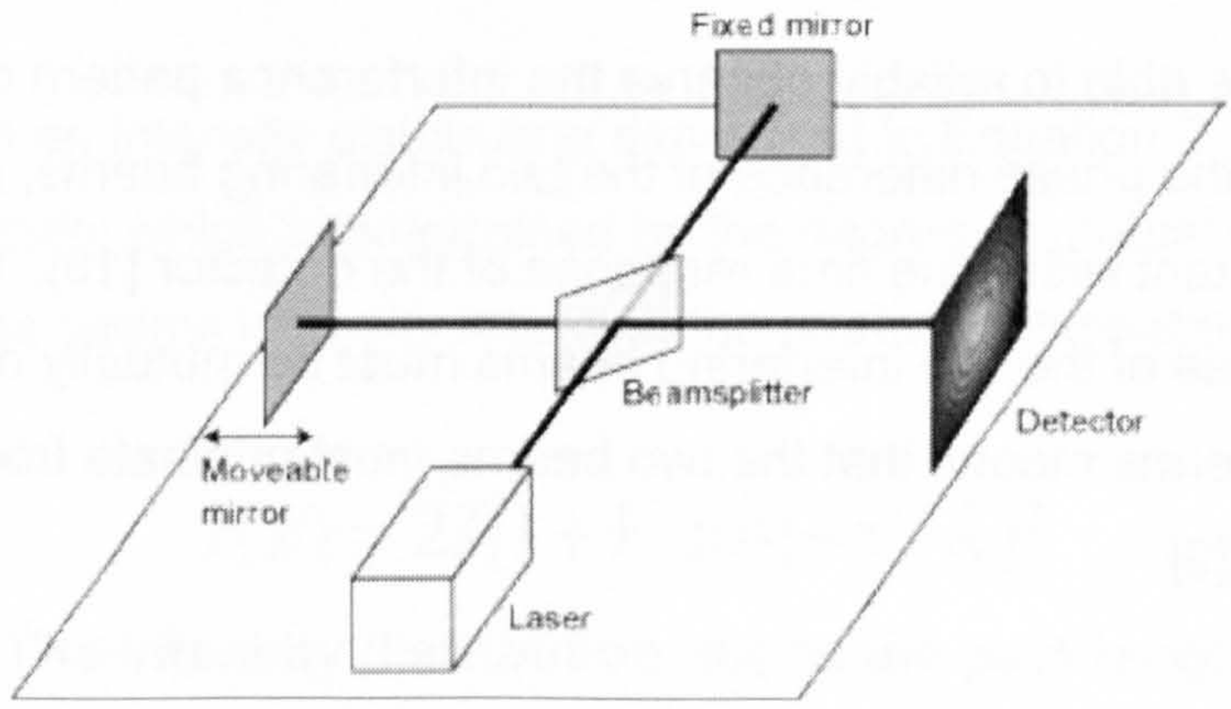


Figure 3 The Michelson Interferometer[5]. A fringe pattern is observed at its output (detector).

Other types of amplitude division interferometers are the Twyman-Green and Mach-Zehnder interferometers.

The Michelson interferometer, however, is perhaps the best know type of an amplitude dividing interferometer.

Its beam splitter, which is partially reflective, divides the amplitude of the incident light (Figure 3).

The reflected and transmitted partial waves propagate to the mirrors, from which they reflect back and recombine (i.e. interfere) forming an interference pattern which can be detected at the interferometer's output as an intensity interference distribution. The intensity of the interference field in the interferometer can be stated mathematically [18] in Equation 1:

$$I = I_1(x,y) + I_2(x,y) + 2\left(\sqrt{I_1(x,y)I_2(x,y)}\right)\cos[\phi(x,y)]$$

Equation 1 The two-beam interference equation [18]

The first two terms (Equation 1) are the intensities of the two beams at the spatial coordinates x, y. The third term is the cross-interference between the two beams, and Ø(x, y) is the phase of the interference pattern.

In order to be able to reliably observe the interference pattern described by Equation 1, the phase difference of the two interfering beams, Ø(x, y), must remain constant within the time response of the detector [19]. This means that the waves of the two interfering beams must be mutually coherent, which in practical terms means that the two beams must originate from the same light source [9].

---

[5] Image obtained from http://en.wikipedia.org/wiki/Michelson_interferometer

Variations of the Michelson interferometer have been successfully used in many applications for the purposes of measuring a particular property of an object.

This is often achieved by replacing one of the interferometer's mirrors by the object to be measured.

When one part of the divided laser beam is reflected from an object, before it is interfered with the other part, the resultant interferogram can then be analysed to obtain quantified measurements such as object deformation, amplitude of vibration [20] or strain and stress [22].

Such measurements are possible due to the direct relationship of the displacement of the object's surface to the intensity distribution of the interference pattern detected at the interferometer's output.

This can be demonstrated by moving one of the mirrors in the Michelson interferometer. A displacement of x of the movable mirror, Figure 3, gives a path length difference:

$$PLD = 2x$$

and phase difference:

$$\varnothing = (2\pi/\lambda)\ 2x$$

This results in an intensity distribution expressed in Equation 2, where V is the fringe visibility which is determined by the degree of mutual coherence of the beams, the beams intensity ratio and the relative polarisations of the beams:

$$I(x) = 2I[1 + V\cos(4\pi / \lambda)]$$

Equation 2 The intensity distribution due to the path length difference of the two interfering beams in the interferometer

As the mirror moves, its displacement is measured by counting the number of light maxima registered by the detector. By counting the number of maxima per unit of time, one can also find the speed of the object [9].

2.4. Electronic Speckle Pattern Interferometry (ESPI)

A Speckle pattern is light of speckled appearance reflected from an optically rough surface when it is illuminated by a laser source (Figure 4).

This is due to interference of waves with random phases from neighbouring regions of the surface.

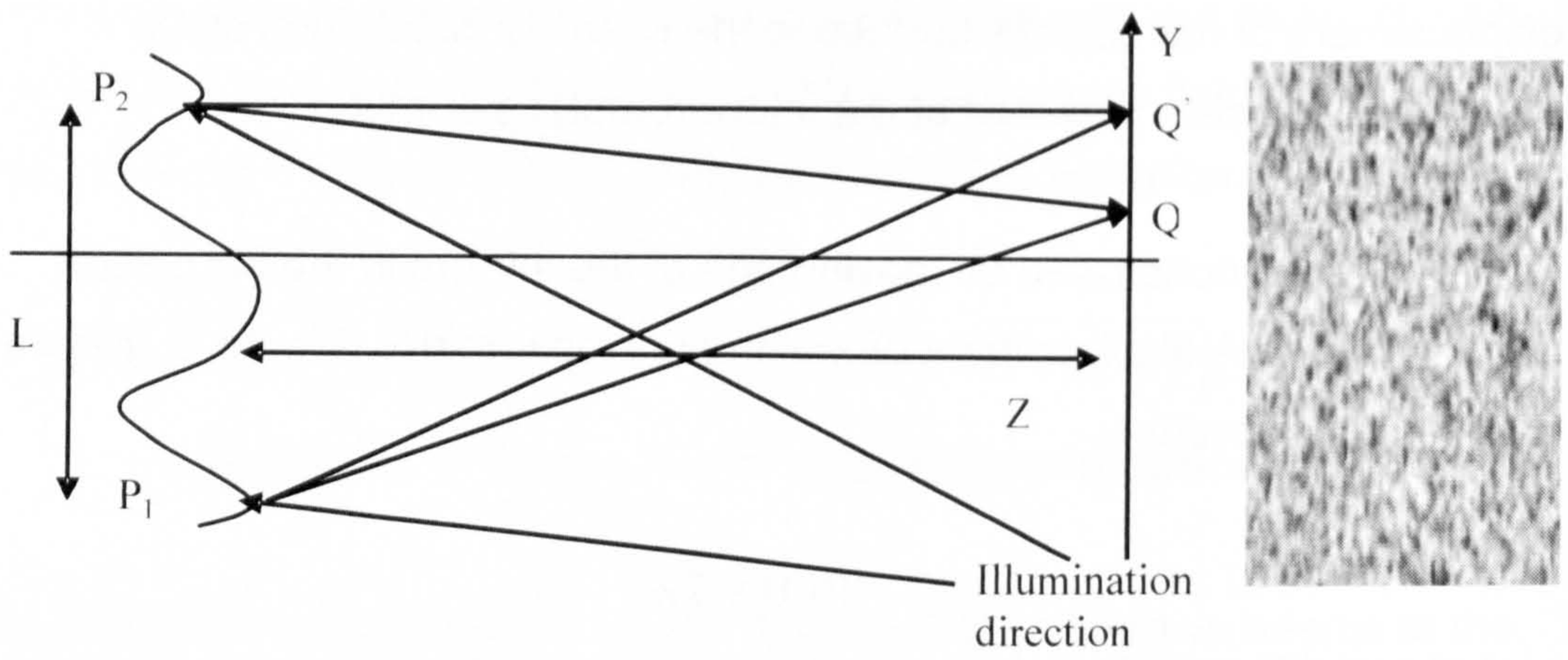


Figure 4 The speckle phenomenon

In electronic speckle pattern interferometry (ESPI) [23], the surface of an object is imaged using a camera (CCD usually) through an interferometer, such as the Michelson interferometer described above.

If the object surface is then loaded by stress or vibration, for example, a different speckle pattern is produced[6]. The two speckle patterns can then be correlated, by subtracting one's intensity from the other. This produces a new image (interferogram) containing fringes.

---

[6] For small deformations, the speckle pattern remains the same but undergoes a phase shift and hence each speckle cycles in intensity.

These are called correlation fringes and can be used to measure the deformation occurring due to the load applied.
It is not possible, however, to distinguish between the hills and the valleys of this phase map (Figure 5).

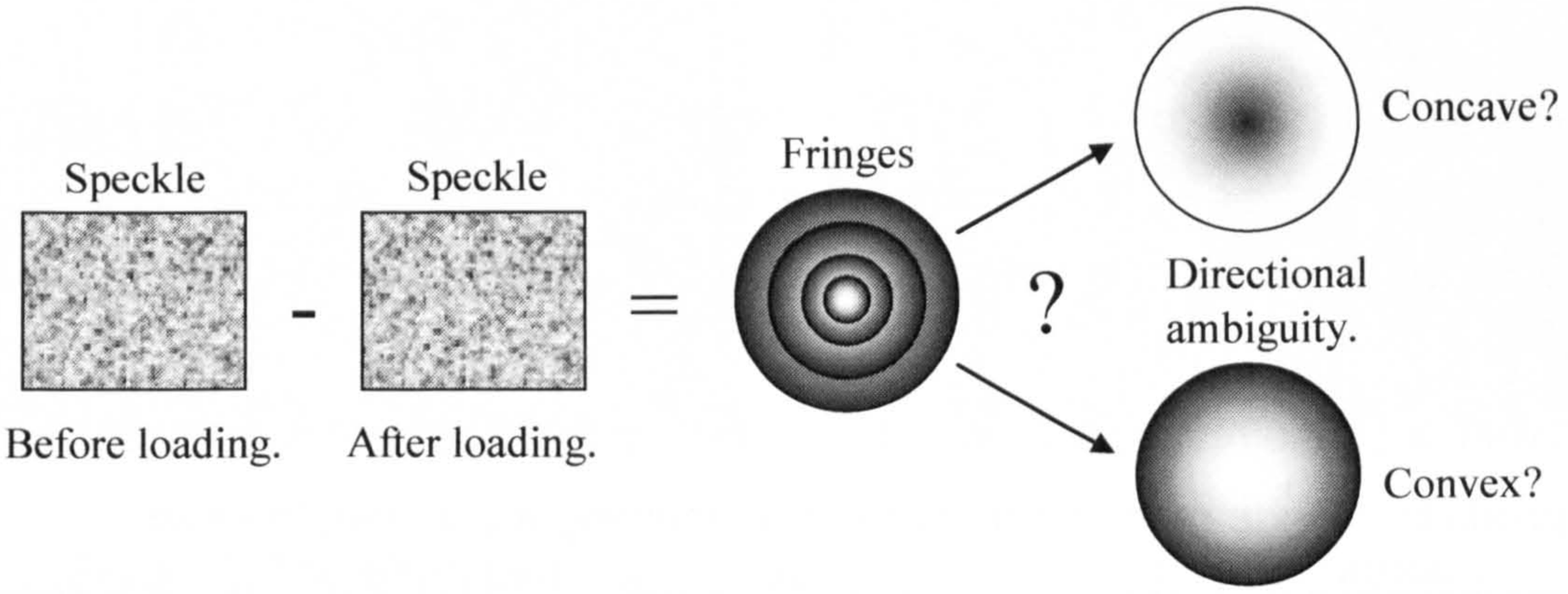


Figure 5 Correlation fringes after further processing, including phase unwrapping, can still have directional ambiguity. Images are for illustration purposes only and are not actual.

This is known as the directional ambiguity of an interferogram. This ambiguity can be resolved by employing one of several techniques, such as fringe tracking [30], the Fourier-transform technique [1, 2], and phase stepping[7] [3, 4, 5, 6].

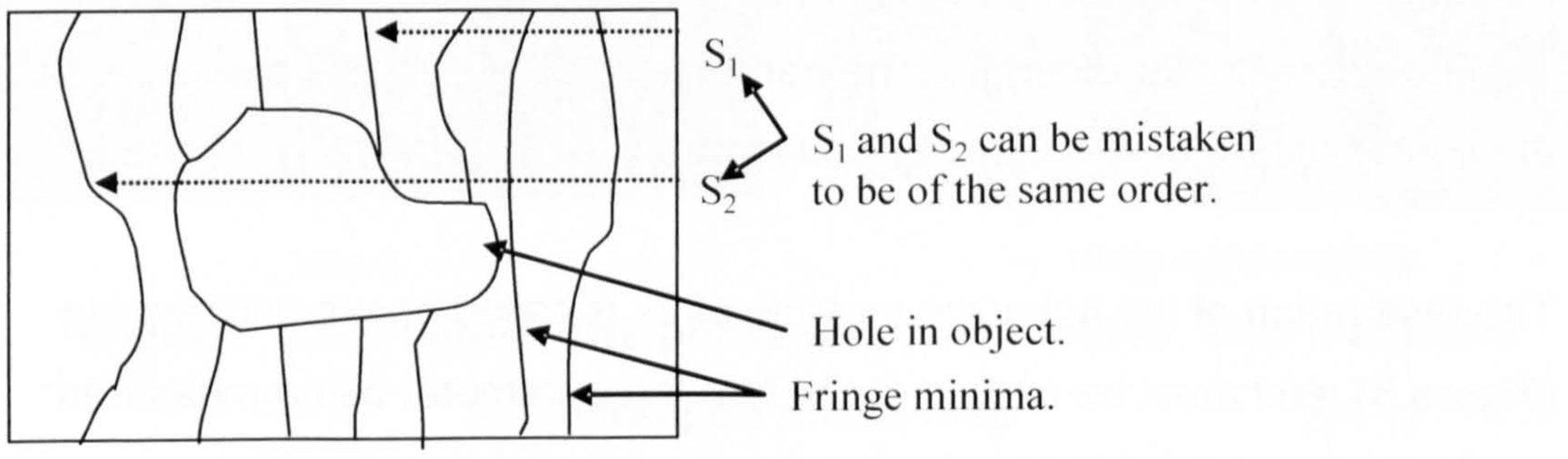


Figure 6 The fringe tracking method and common causes for its failure

[7] Phase stepping is also known as quasi-heterodyning.

Fringe tracking is manual, at least in part, and it is up to the operator to determine the actual direction, be it convex or concave.

Some of the problems with the fringe tracking method are illustrated in Figure 6, the main one being its unsuitability to automatic fringe analysis in the presence of inconsistencies, such as holes, in the object being measured.

Using the Fourier transform and the phase stepping techniques (Figure 7), the obtained fringe pattern contains unambiguous directional information. Such a fringe pattern is called a wrapped phase map.

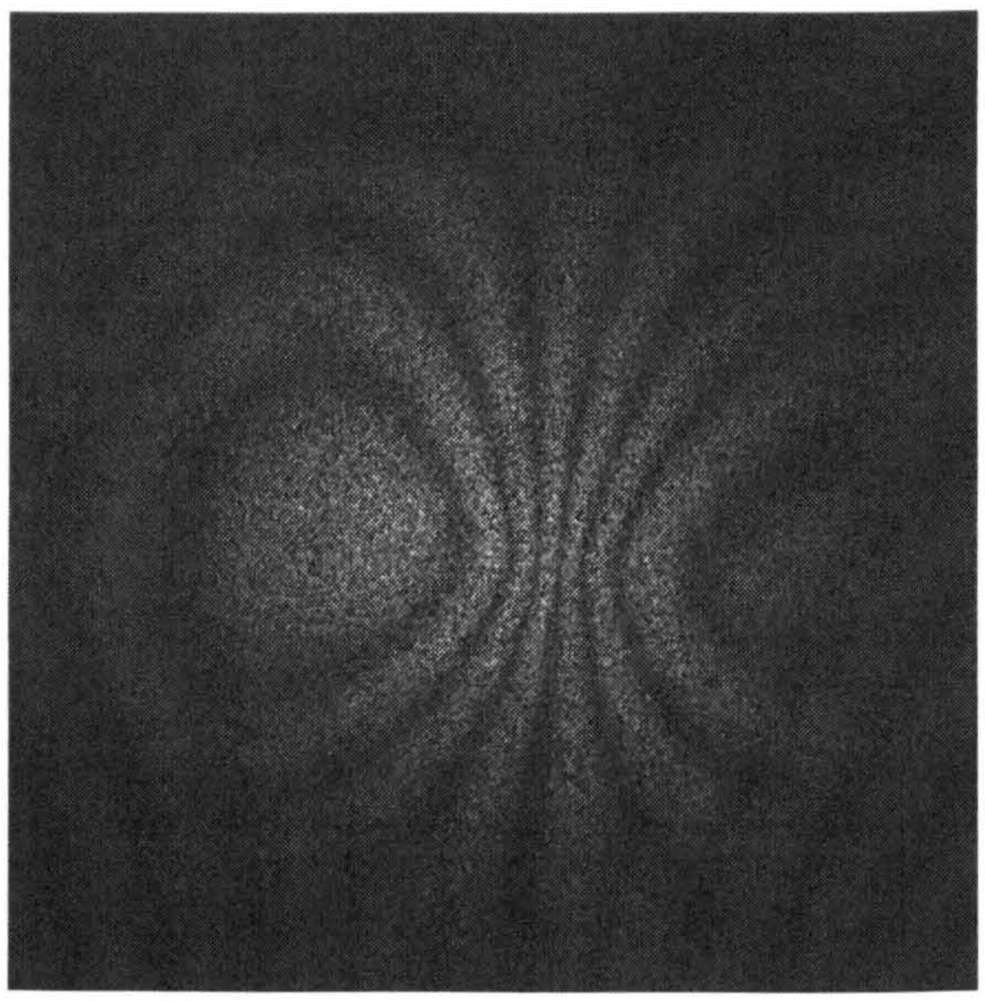
When a phase map is displayed as a grey scale image, the intensities of pixels in a fringe ranges from black, representing a 0 phase, to white, representing a 2π phase (or $-\pi$ and π respectively). This is true for all fringes in the phase map. This is because of the inherent phase wrapping onto the range $-\pi$ to π.

This artefact can be understood by examination of the phase factor in Equation 1; the maxima occur when $\varnothing = \pm 2n\pi$ and minima occur when $\varnothing = \pm(2n - 1)\pi$ where n = 0, 1, 2...

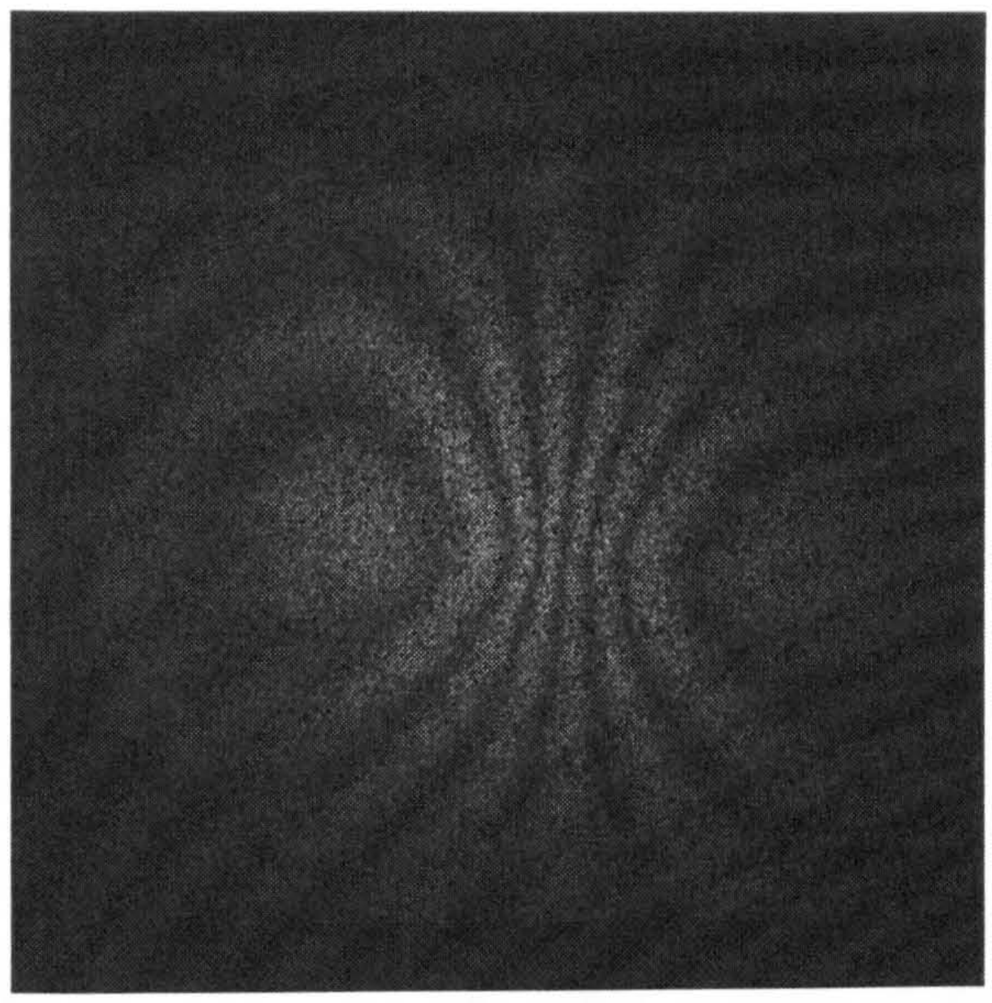
The optical setup is usually constructed so that it is possible to obtain a measurement of physical parameters such as displacement, strain, vibration or surface profile, for example. The parameter being measured is encoded in the interferogram of the wrapped phase map.

The restoration of the unknown multiple of 2π is called phase unwrapping (Figure 8), and must be carried out before the parameter being measured can be deduced from the phase map.

Phase unwrapping is therefore central to most algorithms for automatic fringe analysis [7].

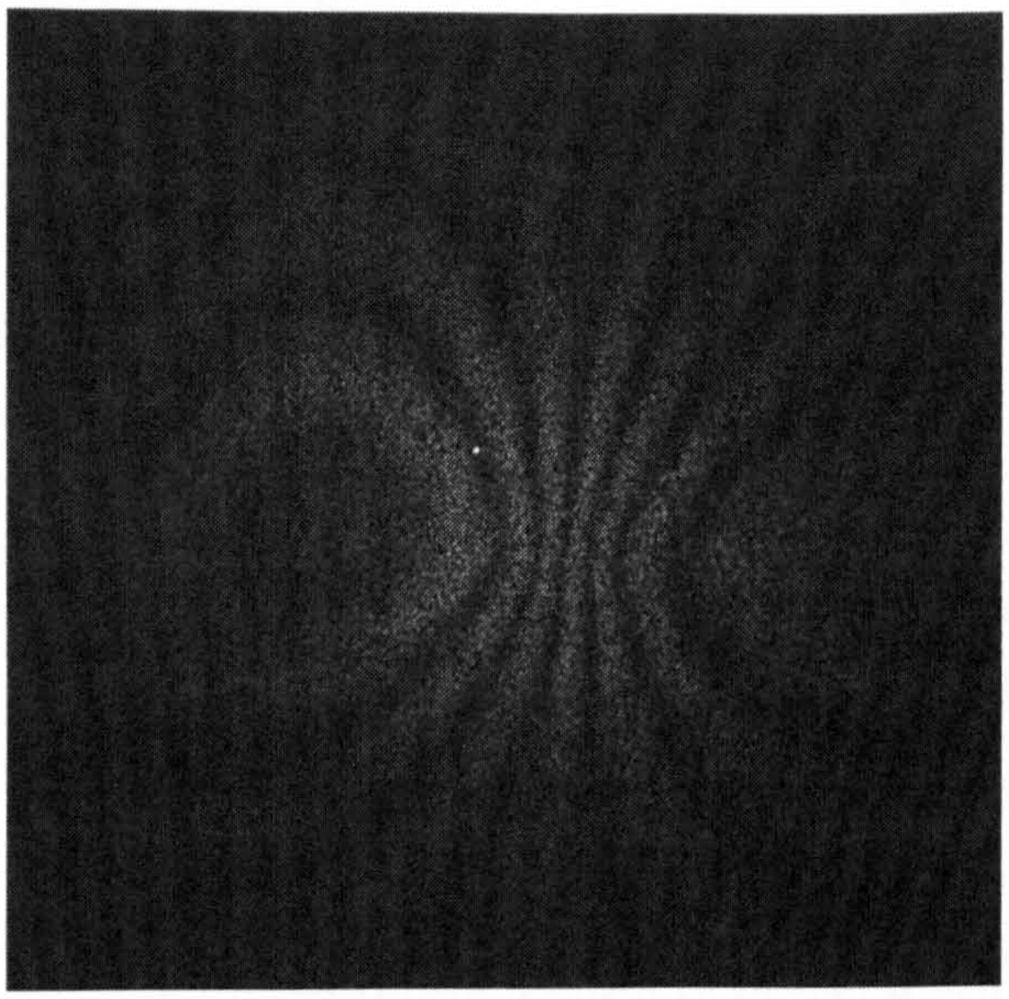


step 1

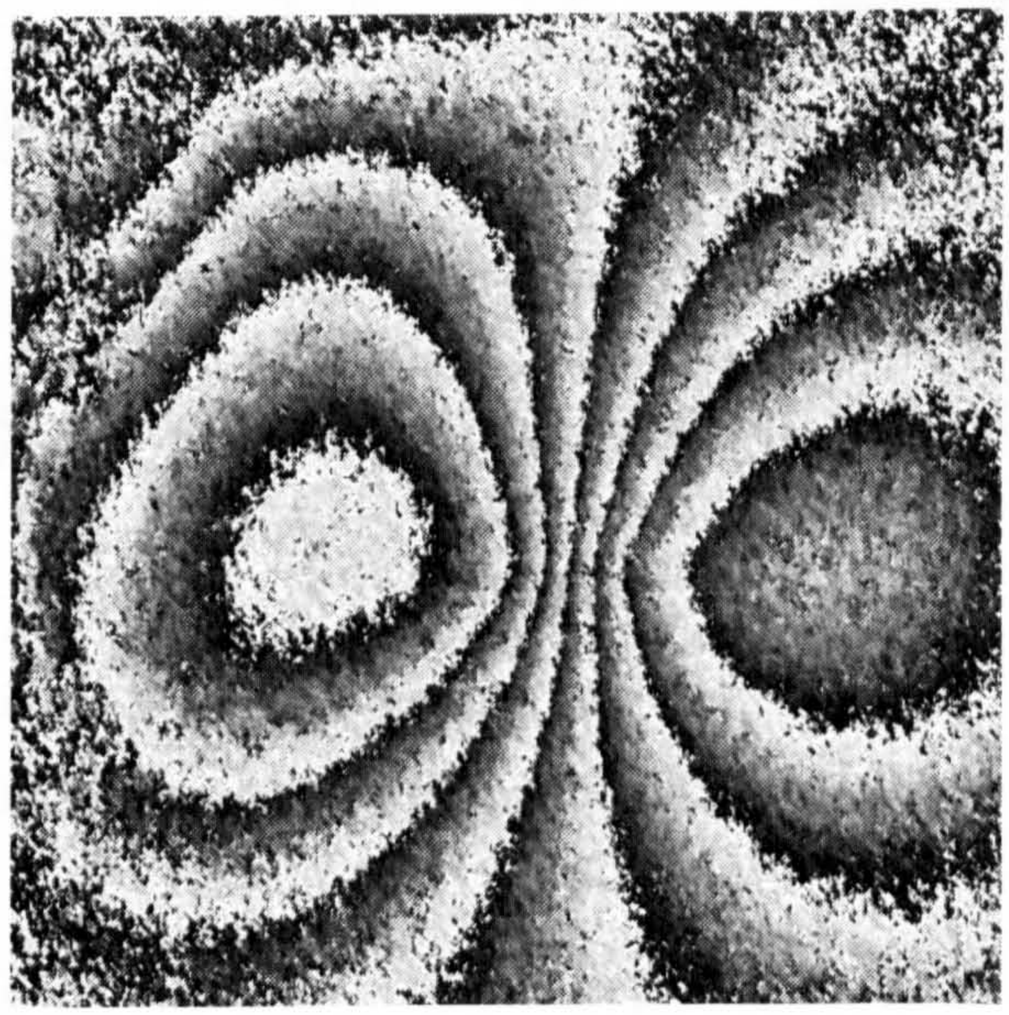


step 2

Figure 7 The phase stepping method, steps 1 and 2, continued…



step 3

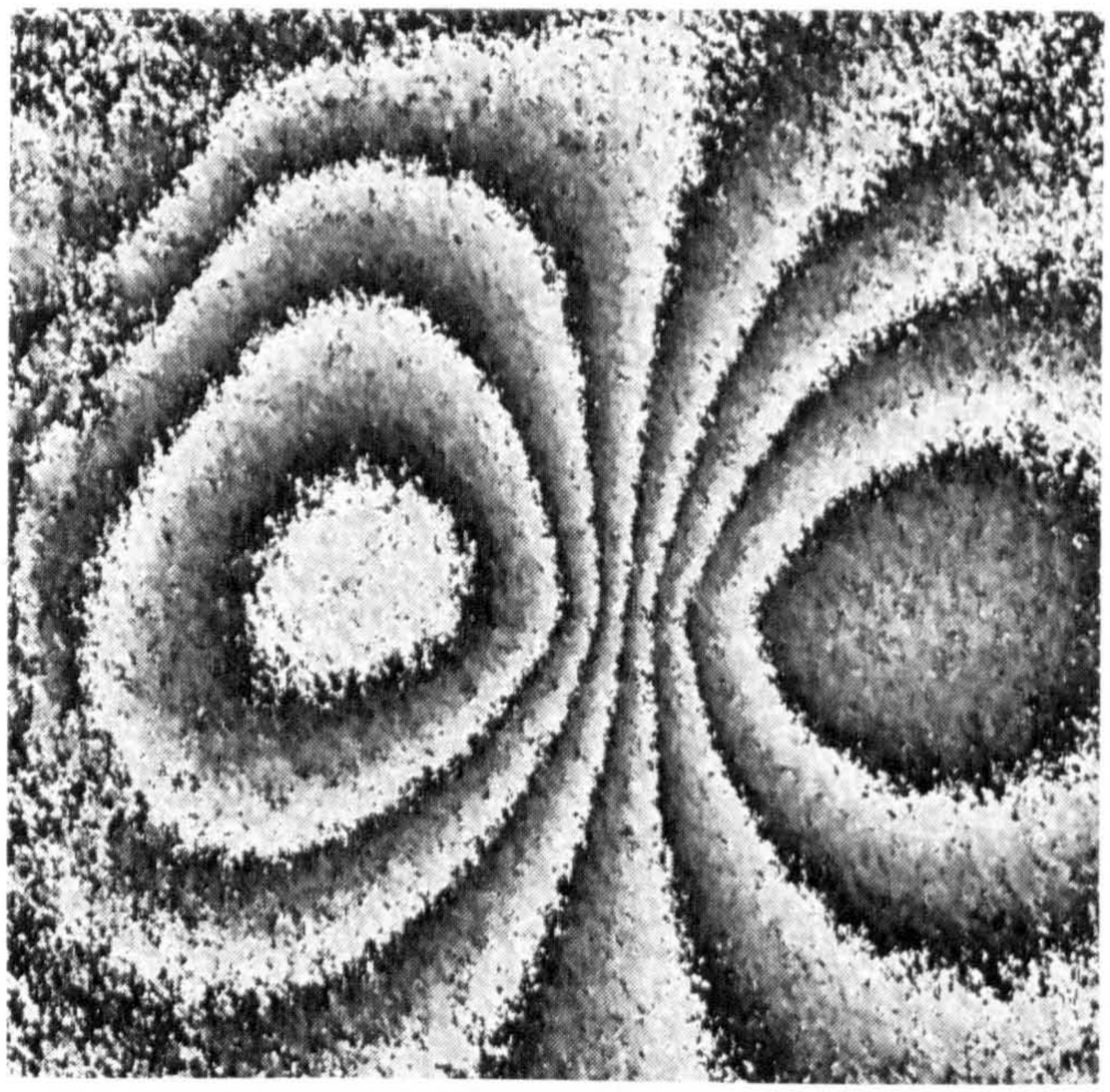


wrapped phase map

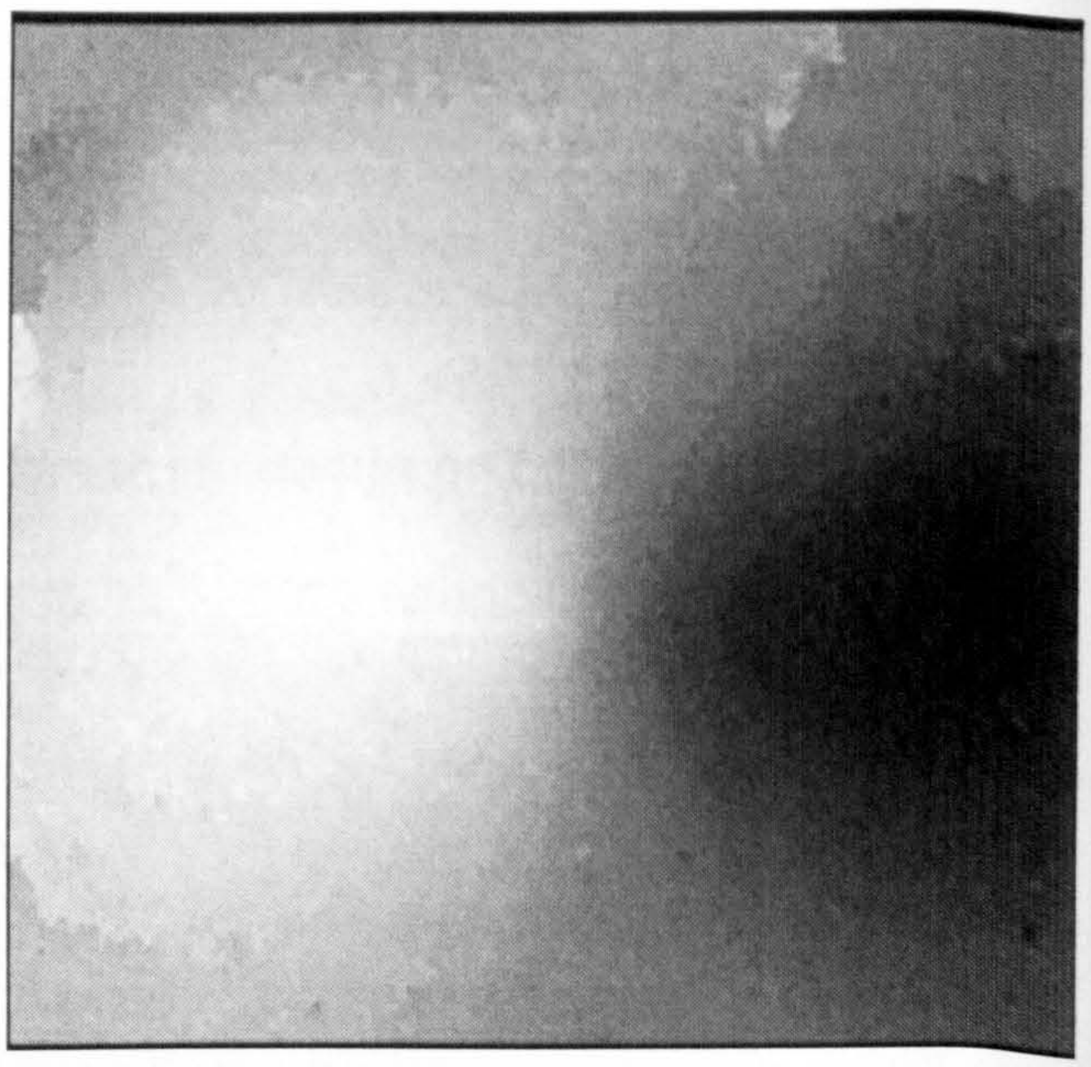
Figure 7 …continued. The phase stepping method, step 3 and the wrapped phase map

In practice, high quality fringes are rarely obtainable due to many sources of error including [24]:

a) Noise. Speckle patterns are noisy by nature. Electronic noise also occurs during the image acquisition.
b) Low-modulation pixels. These are due to areas of low visibility, and appear as fluctuation in the phase modulo 2π. The modulation, which is also referred to as the contrast or the visibility, of the speckle pattern is given by the modulus of the cross interference term in Equation 1 (the third term).



filtered wrapped phase map



unwrapped phase map

Figure 8 Phase map filtering and unwrapping. Images obtained using a commercial phase unwrapping package (ISTRA software by Ettemeyer AG) which uses the minimum spanning tree technique [56]

c) Object surface discontinuities. These result in logical inconsistencies in the phase difference between two pixels. This particularly impacts path-dependent phase unwrapping, such as the fringe tracking method.

d) Aliasing. Fringes usually have to be at least three sampling points (pixels) wide to satisfy the sampling theorem [25]. If this is violated, aliasing occurs resulting in erroneous phase information.

All these problems contribute to the complexity of the phase unwrapping process.

Numerous phase unwrapping algorithms, present in the literature, were devised to overcome these problems. Some of the most common of these algorithms are reviewed and their advantages and shortcomings summarised in the next chapter.

Blank

In

Original

Chapter 3 Phase unwrapping methods

3.1. Introduction

In the previous chapter we briefly introduced interferometry and the role which phase unwrapping algorithms play within it.

Robust and effective phase unwrapping methods are an essential part of the overall process of an automatic fringe analysis system, particularly when the wrapped phase maps obtained by such a system are noisy or contain complex features such as holes [9].
Wrapped phase maps obtained through the various applications of speckle-based interferometry are a prime example, due to the inherently noisy nature of speckle images.

In this chapter, we start by describing the landscape of the phase unwrapping field of research.
We then go on to discuss in more detail the various merits of some of the unwrapping approaches which were considered at the early stages of this research.

Finally, we close this chapter by identifying one algorithm in particular, namely the tile-based method, as the chosen subject for our further research. In the next chapter (Chapter 4), we give a more detailed description of the tile-based method, and in subsequent chapters we discuss one of its fundamental aspects and describe a novel algorithm to improve on its computational performance.

3.2. Phase unwrapping methods classification

Various reviews of phase unwrapping techniques exist in the literature [8, 9, 26, 27, 28, 29, 36], and a text book [10] is entirely dedicated to the subject. Such reviews generally strive to group the various algorithms into appropriately named classes.

A comprehensive classification system, however, appears to be unattainable. This is mainly due to the ever increasing diversity of the algorithms and their approaches, such as:

- The use of fractal geometry and fractal properties [37, 38].
- Constructing a local histogram of wrapped phase information to segment inter-fringe and fringe boundary areas [39].
- Combining the principles of agglomerative clustering and use of heuristics to construct a quality-guided path for unwrapping [40]. Unlike other quality-guided algorithms, which establish the path at the start of the unwrapping process, this technique constructs the path as the unwrapping process evolves. This makes the technique less prone to error propagation, but more computationally demanding however.
- Employing a multi-channel least-mean-square algorithm [41].
- Adopting statistical and probability estimation based approaches [42, 43].
- Evolving genetic algorithms for a suitable unwrapping solution [44, 45].
- Performing stochastic and simulated annealing [46, 47].
- Utilising neural networks for filtering the wrapped phase map in order to reduce the computational cost of unwrapping [48 – 50].
- Converting phase unwrapping to a problem of minimum cost network flow, in order to make use of the latter's efficient algorithms [51 – 55].
- Incorporating a minimum spanning tree approach [56 – 65] to reduce the extent of error propagation from the noisy regions of the phase map.
- Unwrapping along the maximum spanning tree path [66, 67] to reduce the extent of error propagation from the low modulation regions of the phase map. This algorithm is particularly suited to object surface profiling measurements, which are sensitive to degradation caused by low modulation regions. The weights of the graph constructed

represent the cross amplitude multiple of adjacent pixels. The maximum spanning tree algorithm identifies a path whose total weights is maximum, thus reducing error propagation from the less well modulated areas of the image to the rest of the field.

This diversity is perhaps a reflection of the complexity of the underlying problem and the wide range of the targeted applications.

There are, nonetheless, broad classes which are typically used, such as the clear distinction between spatial and temporal methods, and between path-dependent and path-independent methods.

3.2.1. Spatial and temporal phase unwrapping

The main steps followed in the analysis of speckle Interferograms, as currently applied in research laboratories and the industry, can be typically described by one of the paths in flow chart shown in Figure 9 [36].

The various blocks of the follow chart are described in the following sections.

Speckle interferogram

The interferogram is usually recorded using a CCD camera as a two dimensional intensity distribution at a particular instant of time I(x, y, t), where *x* and *y* are the spatial coordinates, and *t* is the time index, which is often referred to as the temporal coordinate.

Phase shifting

The speckle phase can be calculated from the change in the interferogram intensity detected when known phase shifts are introduced between the two interfering light waves.

The speckle phase thus calculated, $\emptyset_W(x, y, t)$, is naturally wrapped onto the range $-\pi$ to π.

The phase shifting required is often obtained through a phase stepping method. Phase stepping can be applied either along the time axis, and is

thus termed temporal phase stepping, or across the two-dimensional spatial coordinates, and thus termed spatial phase stepping.

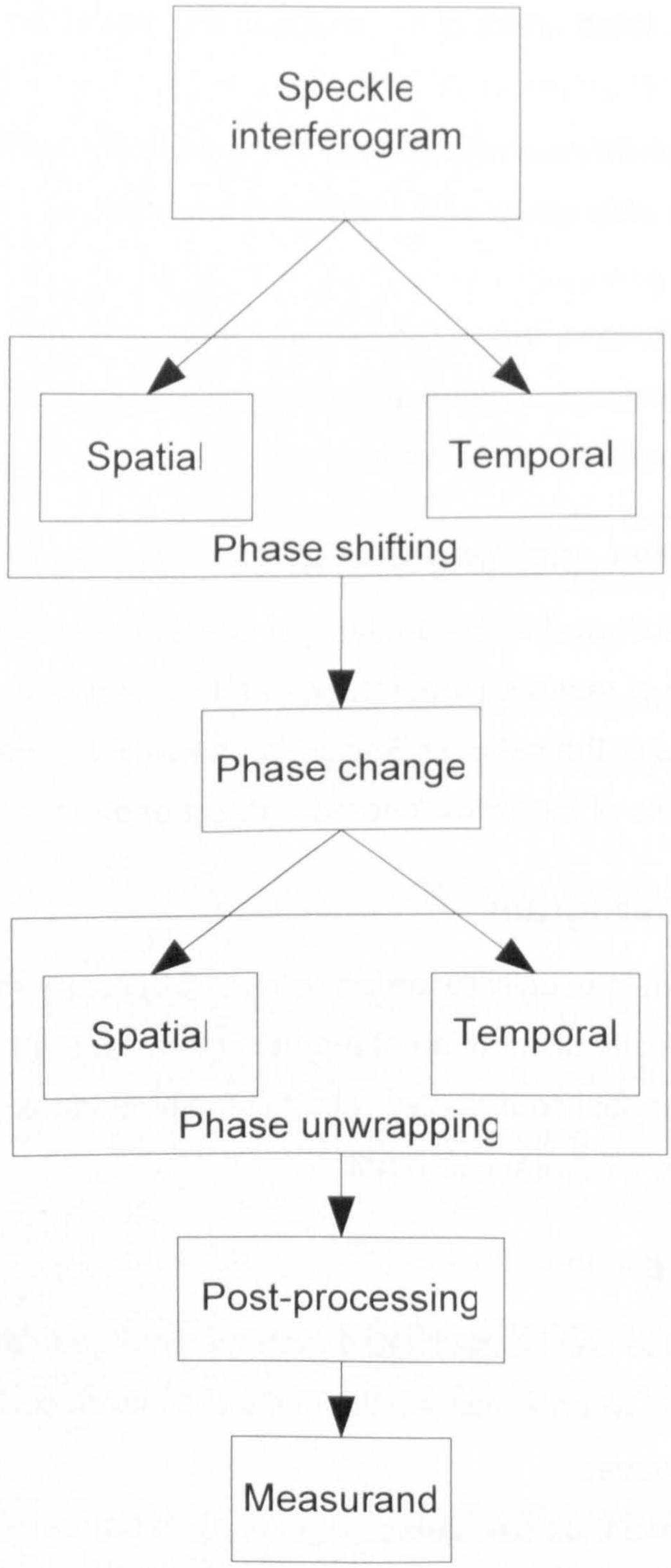


Figure 9 Flow chart of typical speckle interferogram analysis [36]

It is necessary to perform at least three measurements to determine the phase, in order to solve the three unknown interference variables; these are namely the dc intensity, the modulation of the interference fringes and the wavefront phase [25].

Spatial phase stepping

In essence, spatial phase stepping aims to inject a known phase change across the pixels of the recorded two-dimensional interferogram image. In addition, the known phase shift is applied simultaneously, and thus can be recorded in a single instance of time.

This can be achieved in various ways. Perhaps the most straightforward is using the tilted mirror technique.
The tilt is applied to a mirror in one arm of the interferometer (e.g. the Michelson interferometer, Figure 3). The maximum tilt extent which can be applied is equal to half of the wavelength of the laser source employed. This is to ensure that the amplitude of the wave to be measured does not change detrimentally within the period of the spatial carrier frequency, resulting in aliasing due to insufficient sampling. Theoretically, it is necessary that the phase change between adjacent detector elements, or pixels, must be less than 180 degrees. That is, there must be at least two pixels (samples) per fringe to meet the Nyquist frequency and avoid aliasing.

The simultaneous phase change across the spatial coordinates can be also achieved by employing a discrete approach of the carrier technique [68, 69, 70]. This works by introducing into the interferometer three (or more) parallel channels to produce separate interferograms simultaneously.

There are various ways in which phase-stepped interferograms can be generated simultaneously [9], such as by employing polarisation optics [69], diffraction gratings [70], and colour separation [71].

Temporal phase stepping

In the case of temporal phase stepping, the known phase change is applied along the time axis, i.e. as a function of time.
This generally allows for high accuracy measurements, provided of course the object load, and hence the measurand, does not significantly change over the period of time required for obtaining the phase stepped sequence of interferograms.

The same techniques which are used to generate the known phase shift for spatial phase stepping, such as a tilting mirror, polarisation optics, and gratings, can also be used for temporal phase stepping. The difference being that the detected interferograms are recorded sequentially in the temporal case, as opposed to simultaneously in the spatial case.

Temporal phase stepping also lends itself to a more simplified optical setup, and the advent in piezo transducer technology provides for a more accurate way of controlling the phase of the light source.

Phase change

The phase change is applied mainly in order to remove the directional ambiguity from the detected interferograms. This ambiguity stems from the fact that the relationship between the pixels intensities recorded in the interferograms is smooth and periodic.

Obtaining another interferogram after a phase shift has been applied, and then correlating the two interferograms, by subtracting the detected intensities for example, removes (or cancels out) the unknown initial phase term.

The phase change can be introduced as a result of the object loading, in which case the sequence is as follows:

1. A reference interferogram of the object's initial state is first recorded.
2. The object is then loaded (say by applying a mechanical load).
3. A second interferogram of the object, now in the loaded (and therefore deformed) state, is recorded.
4. The intensities of the second image are subtracted from the reference image. The resulting image is called a correlation interferogram, whose phase difference $\Delta\emptyset_W(x, y, t)$ is directly proportional to the load, and hence the measurand (e.g. object displacement).

Phase unwrapping

The phase information in the correlograms (correlation interferogram), obtained by the phase change described above, is also naturally wrapped onto the range $-\pi$ to π. Phase unwrapping restores the unknown multiple of 2π in the wrapped phase map $\Delta\emptyset_w(x, y, t)$, and thus the unwrapped phase map $\Delta\emptyset_u(x, y, t)$ is obtained.

The 2π phase discontinuities can be removed either along one or more of the spatial coordinates, or along the time axis. The two approaches are termed spatial and temporal phase unwrapping respectively.

Spatial phase unwrapping

Spatial phase unwrapping is perhaps the most common type, and is often referred to as two-dimensional phase unwrapping.

The phase is unwrapped along the two-dimensional spatial axis to resolve the 2π phase discontinuities.

Either local neighbourhood phase information in the wrapped phase map, or indeed the full-field information is consulted to ascertain the multiple of 2π which needs to be added or removed. A phase correction value of 2π is added if is added if the values of the pixels are increasing in value along the unwrapping path leading up to the encountered phase jump. Similarly, a phase correction value of 2π is subtracted if the values of the pixels are

decreasing in value along the unwrapping path leading up to the encountered phase jump.

This works very well for interferograms containing little or no noise. However, speckle interferograms are noisy by nature, and therefore there are many sources of error which hinders the performance of the unwrapping algorithm.

The main source of error is areas of low modulation. These areas of the speckle image tend to be very low in intensity due to various factors, such as the reflective properties of the object being examined.

If the unwrapping of less noisy areas of the image happens to rely on the phase information of a neighbouring more noisy area, then the unwrapping errors are said to have propagated through the unwrapping field.
Such unwrapping errors are inevitable by nature, and would be detrimental to the ultimate measurement, unless specific care is taken by the unwrapping algorithm in order to limit its propagation.

The challenge of unwrapping noisy interferograms has called for the development of more sophisticated unwrapping algorithms, which is the main topic of our research.
Whilst spatial phase unwrapping is more challenging in terms of algorithm development, it is generally considered an advantageous approach.
This is mainly because it requires a relatively simple optical setup and off the shelf image capture and storage equipment, and the ubiquitous personal computer.

Temporal phase unwrapping

Temporal phase unwrapping was pioneered over a decade ago [7], as an alternative to the spatial phase unwrapping methods.
This can be achieved by employing phase stepping techniques to introduce a known phase shift along the time axis, while the object undergoes continual

and gradual loading. Thus, a sequence of correlation interferograms is obtained along the time axis.

The sampling rate requirement for temporal phase unwrapping is identical to that of spatial unwrapping; two or more samples (pixels) per cycle. This imposes a constraint on the tolerable rate of the deformation (due to loading) of the object: the phase change due to deformation should be negligibly small over the period of time a set of sequential correlation interferograms is obtained.
This temporal resolution requirement generally means that, for a given application, faster image capturing is required as compared to what would be required for spatial phase unwrapping.

Additionally, for the phase discontinuities to be unwrapped correctly, the positions of the discontinuities (i.e. fringes) should remain constant throughout the time needed to capture the required sequential set of interferograms.

The unwrapped phase information of each pixel $p(x_u, y_u)$ in the unwrapped phase map is calculated by unwrapping the corresponding sequence of pixels $p(x_w, y_w, t_1)$, $p(x_w, y_{w,}, t_2)$ .. $p(x_w, y_{w,}, t_n)$ in the n wrapped phase maps of the correlation interferograms.

The main advantage of temporal phase unwrapping is the fact that each pixel is unwrapped independently of all others in the spatial field, and is thus guaranteed not to suffer from the existence of low modulation points or other phase information defects in its neighbourhood. Phase unwrapping error propagation is therefore completely avoided.

The main disadvantage of this method is its requirement for more advanced image capturing, recording and processing equipment. However, this is

becoming less of an issue with the advent of high performance hardware at increasingly lower prices.

Temporal phase unwrapping and its application [31] remain a subject of active research [32, 33, 34, 35, 36]. Our research is nevertheless primarily focused on spatial phase unwrapping techniques.

Post-processing and the measurand

An unwrapped phase map obtained through the above steps holds unambiguous and continuous phase information proportional to the measurand.

A further post-processing step is required to obtain the absolute quantitative measurement values in the meausrand's units, such as object deformation, amplitude of vibration [20] or strain and stress [22].

3.3. Path independent methods

Rather than relying only on a local neighbourhood of phase information, path independent methods take into account all of the information of the interferogram at each stage of the process. The aim of this approach is to minimise unwrapping errors.

This generally results in a significant computational performance demand, although some developments [76], and more recently [77], have shown promising improvements in this respect.

3.3.1. The cellular automata method [72]

Cellular automata consist mainly of two parts; a collection (neighbourhood) of sites (pixels in this case), which represent the lattice, and the automation rules that govern the evolution of the values of the elements in each site at each time step.
This value can be generally any number of a finite set, and the rules are deterministic and are applied to each cell at each discrete time step.

Thus, the values of the individual sites depend only on the neighbouring sites and the evolution rules.
This makes the cellular automata attractive to the phase unwrapping algorithm developer, because it has the potential for applying the phase unwrapping rules, which are well defined, to predefined neighbourhoods. This, in turn, aids noise immunity and prevents error propagation.

The steps of the original algorithm [72] to adopt the cellular automata approach were as follows:

- The neighbourhood is defined as the four adjacent orthogonal sites (Figure 10).
- A voting system is used, by which each neighbouring site is consulted for the multiple of a 2π-phase shift required for unwrapping the phase difference between the current site and the consulted neighbour. The votes are then accumulated, and if none of the neighbours differ from the current site by more than π then no change is made. Otherwise, a 2π is added or subtracted to the current site according to the majority vote. If all four votes cancel each other out then the positive direction is favoured.

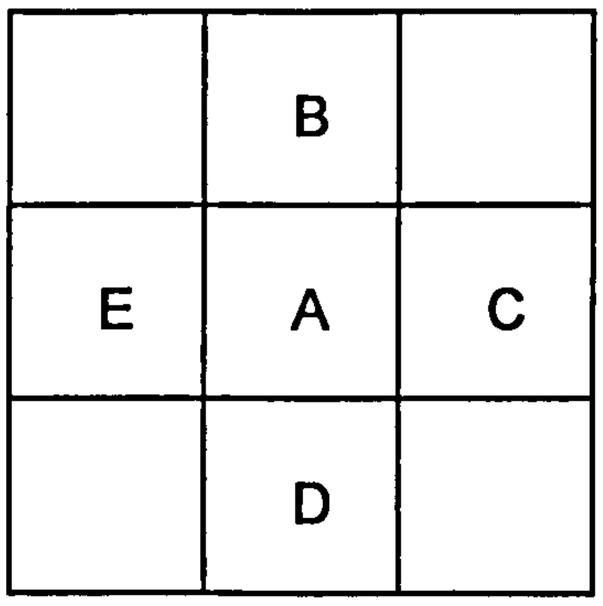


Figure 10 The cellular automata neighbourhood: Cells B, C, D and E are the neighbours of cell A.

- This vote based unwrapping is repeated until the cellular automaton reaches its period-two oscillation state. This is a common feature of the cellular automaton algorithms. It occurs when applying the neighbourhood rules make the values of all the sites to change from one state (S_a) to another (S_b), and then back to the former state (S_a) at consecutive time steps.
  - At this point the two oscillatory states (S_a and S_b) are averaged, by averaging the intensity values of the two respective images.
  - If at this point the image is not completely unwrapped, the entire process is repeated.

Each complete repetition unwraps a single fringe [73] (the outer most fringe). And, as can be seen from the above description, each iteration involves a large number of calculations.

The main disadvantages of this method are its computational demands and the possibility of running into a rogue type of period-two oscillation.
This occurs when inconsistencies and noise in the phase map lead to the situation whereby one iteration removes a fringe and the next iteration erroneously introduce a fringe, thus never converging onto a solution. This further complicates the algorithm.

The main advantages of this method are its path independence and natural parallelism.
The latter is especially important when parallel programming facilities [74] are viable.
This is rarely the case, however, due to the dominance of personal computers which usually comprise only one or two processors, and thus do not lend themselves to the parallel processing requirements of the cellular automata technique, which (optimally) requires one processor for each site.

Despite these well known problems, the cellular automata method remains the subject of active research [75].
The method has also been recently built on to create a new region-growing algorithm, which is claimed to be more robust and to have a better computational performance [77].

3.3.2. The branch cut methods [78, 79]

The main principal of the branch cut methods is placing cuts (separators) on a scan line by scan line bases between the points of phase discontinuity [78, 79].
The aim is to mask the discontinuities, thus minimising the propagation of errors to the rest of the phase field.

The algorithm proceeds by seeking fringe order changes in the image. This is done usually by scanning the image with a 2 by 2 pixel kernel.
The fringe order is, or a fraction thereof, is accumulated following the perimeter of the square kernel, in a closed-path clockwise fashion. The number arrived at is termed the residue, and its sign, positive or negative, is determined by the difference between the start and the end phase pixels.
All four pixels are then marked consistent if the residue along the closed-path equals zero. Otherwise, all four pixels are marked inconsistent.
In general, all consistent pixels have a zero residue, and all inconsistent ones have a positive or negative non-zero residue.

Once the first residue is found, a box of size three is placed around the residue, and a search for another residue is carried out. Once another residue is detected, a cut is placed between the residue pair if they are of opposite signs. If however, the second residue fond is of the same sign as the original's, the box is moved to the second residue and the search is continued until either an opposite sign residue is found, in which case the cut is placed, or a new residue can be found within the boxes, in which case the box size is increased by two and the search then repeats from that point.

Although this procedure may appear complex at first, it achieves a goal that can be easily understood; joining the residues by cuts of minimum length. The minimum length of the cuts is achieved by simply joining residues to their nearest neighbour residue until the sum of all the residues in the particular branch cut is zero, at which point a new "seed" residue is found and another branch cut is started.

This algorithm effectively partitions the field, and the pixels marked as consistent can then be unwrapped using a conventional fringe counting method.
This works very well when all the inconsistencies can be caught by the residue-searching step, as the case would be for localised random noise.

If the inconsistencies are larger than the two by two kernel, as the in the case of aliasing or holes in the object for example, then the algorithm is vulnerable to error propagation into large areas of the data.

A variation on this algorithm is the flood unwrap algorithm [79]. Here, the inconsistencies are simply masked out (left unprocessed) rather than joined by cuts.
The algorithm then proceeds to use a standard recursive flood-fill algorithm (for filling a bit-map region) to unwrap the remaining pixels in the image. This algorithm, perhaps, does not offer much of an advantage over the original cut methods in terms of noise immunity.
It does, however, greatly simplify the original algorithm by doing away with the computationally expensive searching process needed to construct the branches of the cuts.

The branch cut method remains the subject of active research, and has been recently extended to three dimensions [80].

In addition, [81] has shown a connection between the branch cut method and what is termed the "global L^0-norm solution". This has led to the development of efficient algorithms based on network flow techniques which find an approximation to this solution [82 – 84].

3.4. Path dependent algorithms

3.4.1. The Fringe counting approach

The fringe edges are first found using an edge detection technique.
The edge detection seeks a phase change (jump) greater than π between adjacent pixels in the wrapped phase map.
The image is then scanned (usually horizontally) and when a fringe edge is encountered a value of $N*2\pi$ is added to or subtracted from the values of the successive pixels, depending on the direction of phase change prior to reaching the fringe edge.
If the values of the pixels where increasing, then 2π is added, otherwise it is subtracted. N is the count of the fringe edges that where encountered on that particular scan line.
The value of N is incremented or decremented each time a fringe edge is found depending on the direction of the phase change.

When the horizontal scan is completed, the individual horizontal rows of pixels are independent of each other.
They are then adjusted to their correct respective position by a single vertical scan (in the same previous fashion).
By doing so the fringe pattern is completely unwrapped.

There are many variations on the above theme, and particularly on the strategy for choosing the best candidate for the vertical scan.
This algorithm relies significantly on the direction of the scanning.
This should not have any consequence if the fringes are clean and consistent.

But in practice this is rarely the case, even after image filtration. Thus, such algorithms are termed "path dependent".

The major advantage of this algorithm is its simplicity and speed. Some of the more powerful (noise immune) algorithms, such as the tile-based method, utilise at a lower level a fringe counting algorithm.

3.4.2. The minimum spanning tree and tile-based method

Introduction

The tile-based method, utilising the minimum spanning tree approach [56, 57, 58, 59, 60, 62], has been shown to be a powerful and reliable phase unwrapping technique [86].
This is partly due to combining the speed and efficiency of path dependent unwrapping methods (fringe counting), at its lowest level, with noise immunity by the process of tiling (dividing the image into tile like regions), at a higher level.

The method's noise immunity is in fact two-fold; it combats spike noise at a low level (i.e. within a tile), and minimises error propagation at a higher (i.e. inter-tile) level.

We give here a brief overview of this method, and we go on to describe it in more detail in the next chapter (Chapter 4).

Overview

The method is detailed and elaborate, but at its core it has a very intuitive concept:
The method takes advantage of the efficiency of path dependent unwrapping algorithms, but instead of using the almost ad hoc approach of scanning the image, as in the case of the fringe counting method, it seeks to select the unwrapping path more carefully.

The path dependent aspect

To improve on the method in which the unwrapping path is selected, the algorithm first observes that a path along which the phase differences are kept to a minimum would therefore minimise the chance of error.
This is further aided by unwrapping the pixels with the largest phase differences last, thus preventing potentially erroneous unwrapping information from propagating into the rest of the image.

The tiling aspect

Prevention of error propagation is taken a step further by dividing the image into tiles, which are unwrapped individually.
This prevents error from propagating from one tile region to another.

The tiles are finally adjusted to their correct positions with respect to the surrounding tiles. The tiles with the most problems are arranged last, thus preventing them from offsetting the rest of the image.

The minimum spanning tree aspect

The more rigorous selection of the unwrapping path is achieved by converting the image into a graph whose vertices are the individual pixels of the image, and edges are the grid produced by joining all the pixels vertically and horizontally.
Weights of the edges in each tile are then computed by evaluating the phase difference along these edges. The bigger the phase difference the higher the weight.

Finding the unwrapping route which minimises the total phase difference along its path becomes then the standard problem of finding a minimum spanning tree of a graph.

A tree is a path which joins all of the vertices in the graph without any cycles. A spanning tree is minimum if the sum of all the weights of its edges is less or equal to any other spanning tree that can be found in that graph.

Once the unwrapping path has been identified by finding a minimum spanning tree, phase discontinuities are removed (unwrapped) along the path by adding or subtracting the appropriate multiple of 2π, using the same technique employed by the fringe counting method.

For more noisy images, a dedicated edge detection algorithm may be employed to identify the precise location of phase discontinuities (or fringe locations), rather than relying on the potentially misleading local pixel phase information along the unwrapping path.

The tiles in the image are then converted to a graph, in the same manner, with the weights of the edges being obtained by the consideration of several factors, some of which may depend on the actual application.

A minimum spanning tree of the tiles' graph is then found, and the tiles are positioned at their relative height in the order dictated by its path, thus concluding the phase unwrapping process.

Masking off the inconsistent pixels in the image, as in the case of the cut methods, can be easily applied to this algorithm, which further improves its reliability.
Masked off pixels can contribute to the weight calculation of the tiles-graph's edges.

Aspects which could benefit from further research

Although this algorithm is powerful and versatile, it still has some complications.

These are mainly the calculation of the inter-tile weights and the selection of the tile size (and even its shape).

There is not a well-defined formula for calculating the contributions of the above-mentioned factors to the total weight of an inter-tile edge.

This is because one such formula would not be suitable to all types of application.
As for tile size, a 16 square pixel region (plus an overlap) is used as a rule of thumb. But again, this is not guaranteed to work equally well for all fringe densities.

3.5. Chapter conclusion

We have reviewed in this chapter the various approaches to phase unwrapping, and some of the methods and techniques employed by algorithm developers to minimise the likelihood of error and enhance the computational performance.

It is not always possible to objectively compare the performance of one unwrapping algorithm against another, as the various algorithms tend to be tailored for the applications for which they were devised, and as such are optimised differently.
Nevertheless, comparisons of different algorithms targeted at the same type of applications do exist in the literature for both temporal phase unwrapping [85] and spatial phase unwrapping [86].

Our research, at its preliminary stage, led us to the conclusion that the tile based method has a good potential for further research, for the following reasons:

- The method is acknowledged to have good noise immunity and computational performance [86].

- The method is detailed and has various aspects which have a good scope and potential for closer examination and research.
- The method is already in use both in industry and academia, thus further improvements to the method are likely to benefit from wide adoption and applicability.

In the next chapter we describe in more detail the various steps of the tile based algorithm, and identify one of its aspects as the primary object for our research.

Chapter 4 Tile-based phase unwrapping

4.1. Introduction

In the previous chapter, we briefly described some of the various phase unwrapping approaches, including the tile-based method.

In this chapter we expand on the overview of tile-based unwrapping [62], and give the various steps of the algorithm more detailed descriptions.
We also add some of our own insights into the algorithm, which were gained through a practical computer programming implementation of the algorithm.

Finally, we conclude by identifying the minimum spanning tree aspect as the chosen topic for our research.

4.2. Implementation of the tile-based method

Throughout the duration of our research, a practical insight into the various aspects of the tile-based phase unwrapping method was gained by a direct implementation of the algorithm in a computer program.

The analysis, design and implementation of the tile-based method were based only on its textual descriptions in Judge's thesis [62], without any inclusion or direct reference to his particular programming implementation. Our practical treatment of the algorithm is otherwise completely original, and its computer listings are presented in the appendix.

This approach was adopted for the following reasons:

- To gain a first hand general insight into the practical obstacles and considerations similar to those typically experienced by a hands-on practitioner in the field of automatic phase unwrapping.
- To gain a better appreciation of the more detailed aspects of the algorithm and its various intricacies.

- To practically explore the various operations of the algorithm in order to identify the performance bottlenecks and investigate how they may benefit from further development.

In the following sections, we make reference to our implementation of the algorithm to illustrate the output of its various stages, and to discus the related pertinent practical considerations.

4.3. Dividing the wrapped phase map into tiles

The interferogram image of the wrapped phase map, such as shown in Figure 11, is divided into tiles as illustrated in Figure 12.

Adjacent tiles overlap by a small band of pixels known as the overlap regions. These are typically two pixels deep and span the entire boundary of a tile.

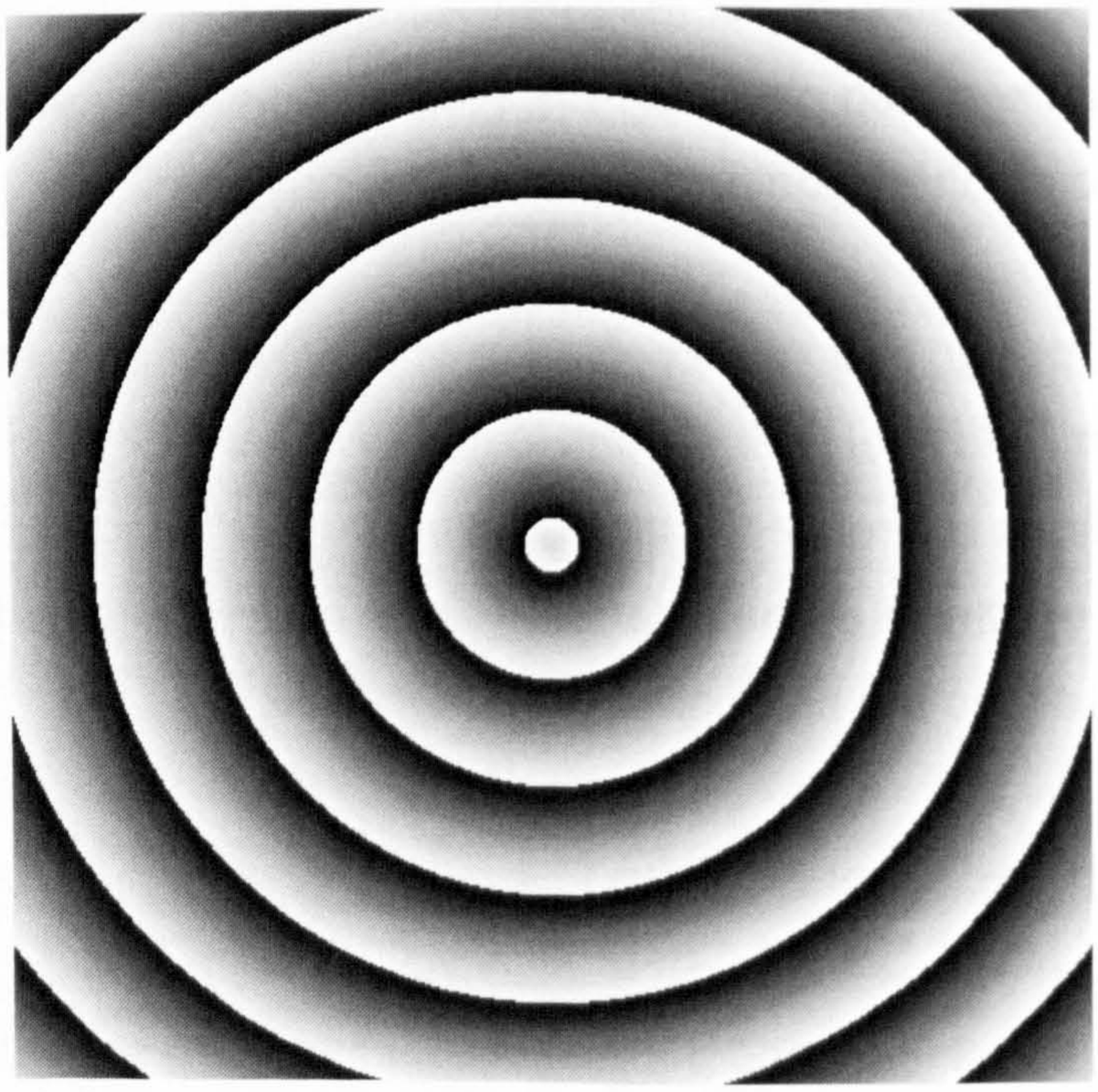


Figure 11 A computer-generated wrapped phase map

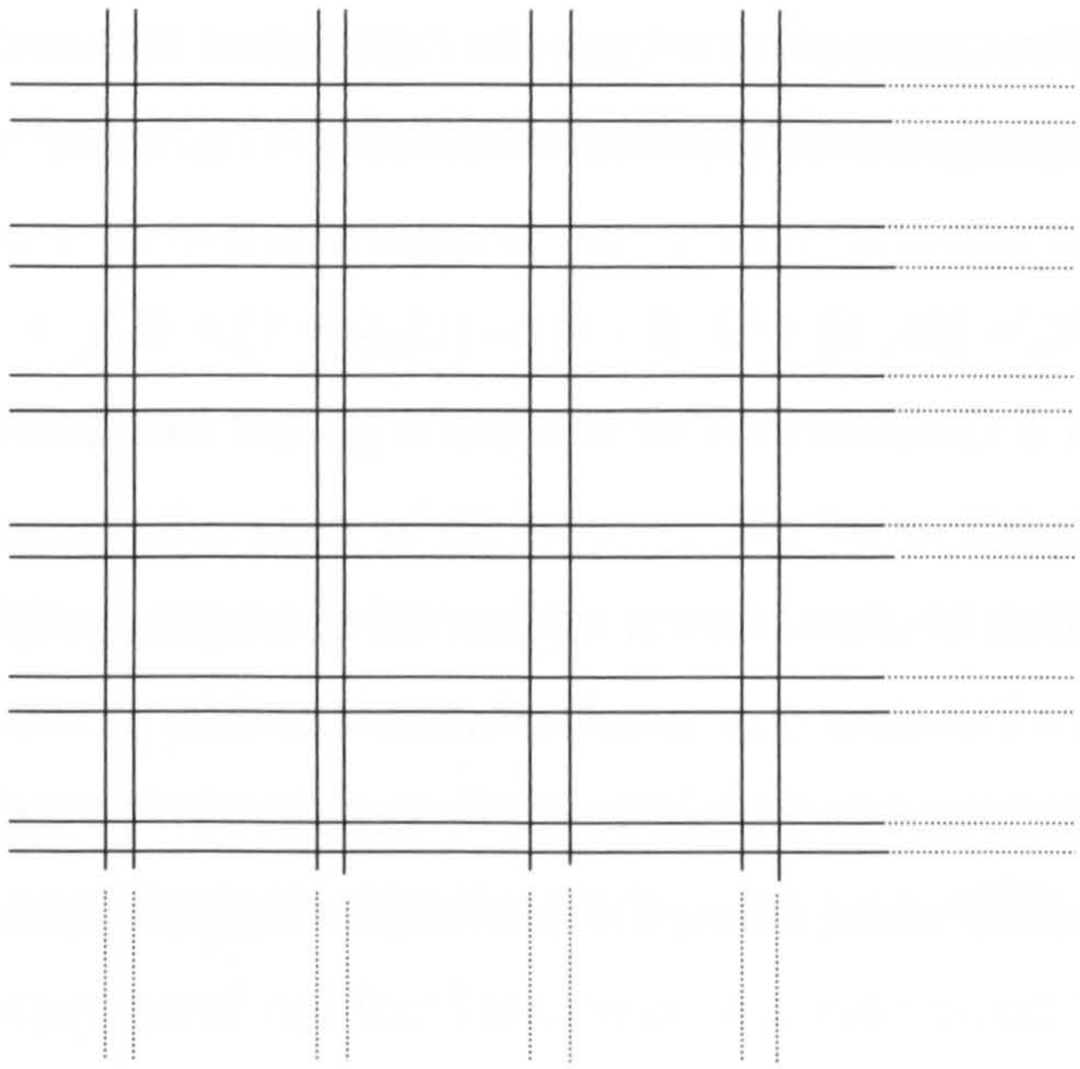


Figure 12 Tiles and their overlap regions

4.3.1. Impact of the tile size

The tile size is usually predetermined and is fixed for a particular application. There are two main criteria for selecting the tile-size:

- Anticipated fringe density. The tiles should not be so large that they often contain more than one fringe section (i.e. one phase discontinuity section).
- Anticipated size of brakes (or gaps) in fringe edges. The tiles should not be so small so that they often fall between gaps in fringe edges, causing in effect adjacent fringes to be merged.

In a typical configuration, a square tile with twelve pixels in each row, and an additional four-pixel deep overlap region is used.

4.4. Converting a tile into a weighted graph

Pixels in a tile are viewed as vertices in a weighted graph as shown in Figure 13.

The weights of the connecting edges are calculated based on the phase difference of the neighbouring pixels as follows:

$$W_x = |(U_x (i) + U_x (i - 1)) – (U_x (i + 1) + U_x (i + 2))|$$

Equation 3 Calculation of the tile's graph horizontal weights

where i is the index of a pixel on a horizontal scan (row) of the image, $U_{x(i)}$ is the interferogram intensity (i.e. pixel value) at index i.

W_x is therefore a calculated weight which represents the extent of the absolute phase difference along the particular edge path.

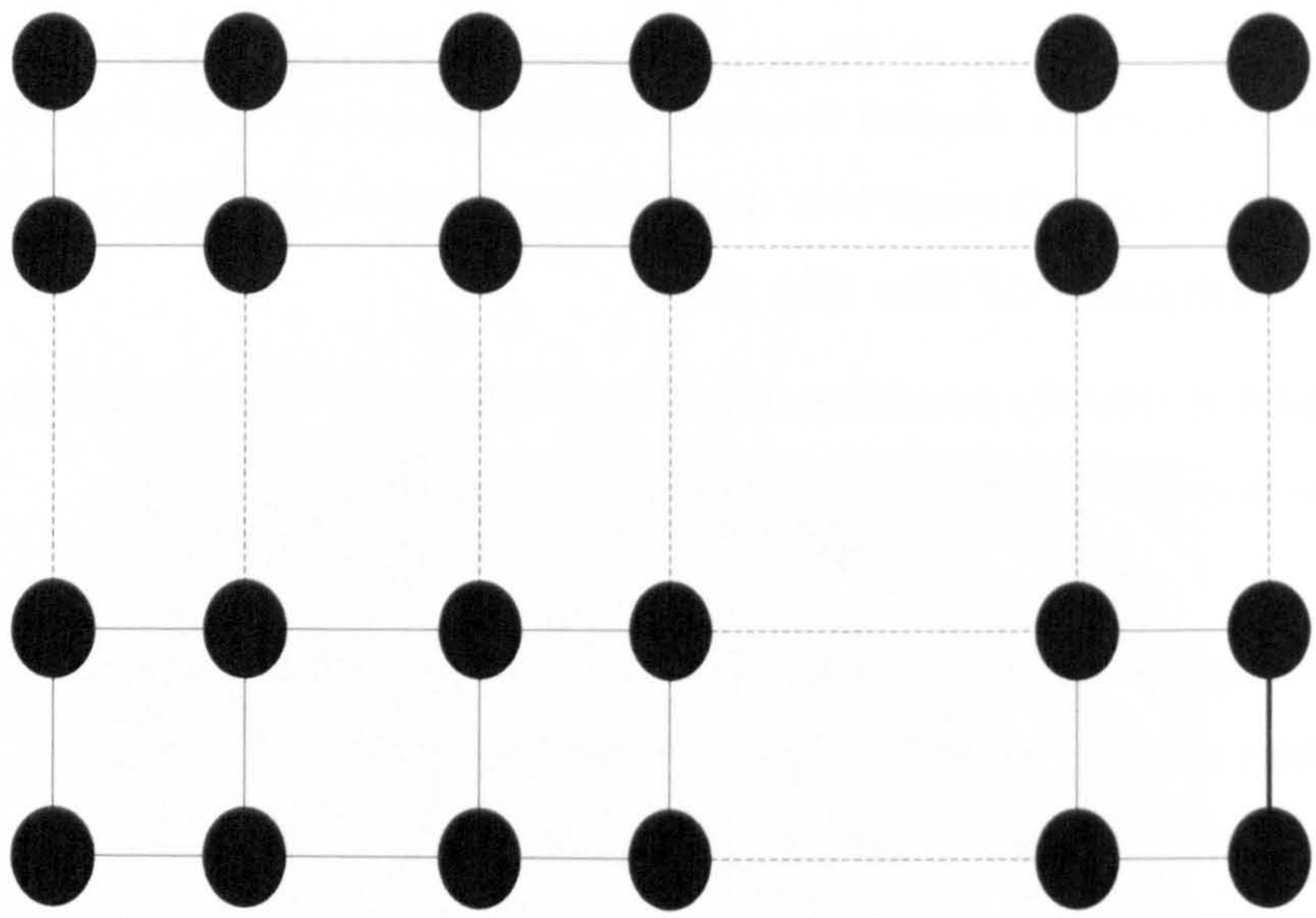


Figure 13 Each pixel in the tile is a vertex in the graph

Similarly, the weights of the vertical edges are calculated by:

$$W_y = |(U_y (i) + U_y (i - 1)) – (U_y (i + 1) + U_y (i + 2))|$$

Equation 4 Calculation of the tile's graph vertical weights

Each edge weight is calculated base on four pixels, rather than a minimum of two, which helps to reduce the effects of spike noise and to limit its influence on the overall unwrapping process.

This is because the effect of spike noise present in one pixel is spread to a two-pixel deep orthogonal neighbourhood, thus further delaying the processing of this area of the image, which in turn reduces the effects of error propagation.

The higher the assigned weight of an edge is, the lower the confidence in the phase unwrapping result along that edge. Thus, the best path is identified by the minimum spanning tree of the weighted graph.
It may be worth noting here that at the stage of phase unwrapping, the presence of a phase discontinuity is determined by examining the phase difference of only two (and not four) successive pixels along the unwrapping path dictated by the minimum spanning tree.
For particularly noisy images, Judge [62] proposes using a dedicated edge detection process to determine more reliably the locations of true phase discontinuities.

4.5. Unwrapping a tile

Now that the interferogram has been subdivided into tiles, each tile is unwrapped individually.

This is done in a straightforward way, by counting the phase discontinuities encountered along the unwrapping path.
A phase discontinuity is also referred to as a wrap-over point, and is defined when the absolute phase difference between two successive pixels along the unwrapping path is greater than 2π.

The unwrapping path is described by the minimum spanning tree of the weighted graph, Figure 14.

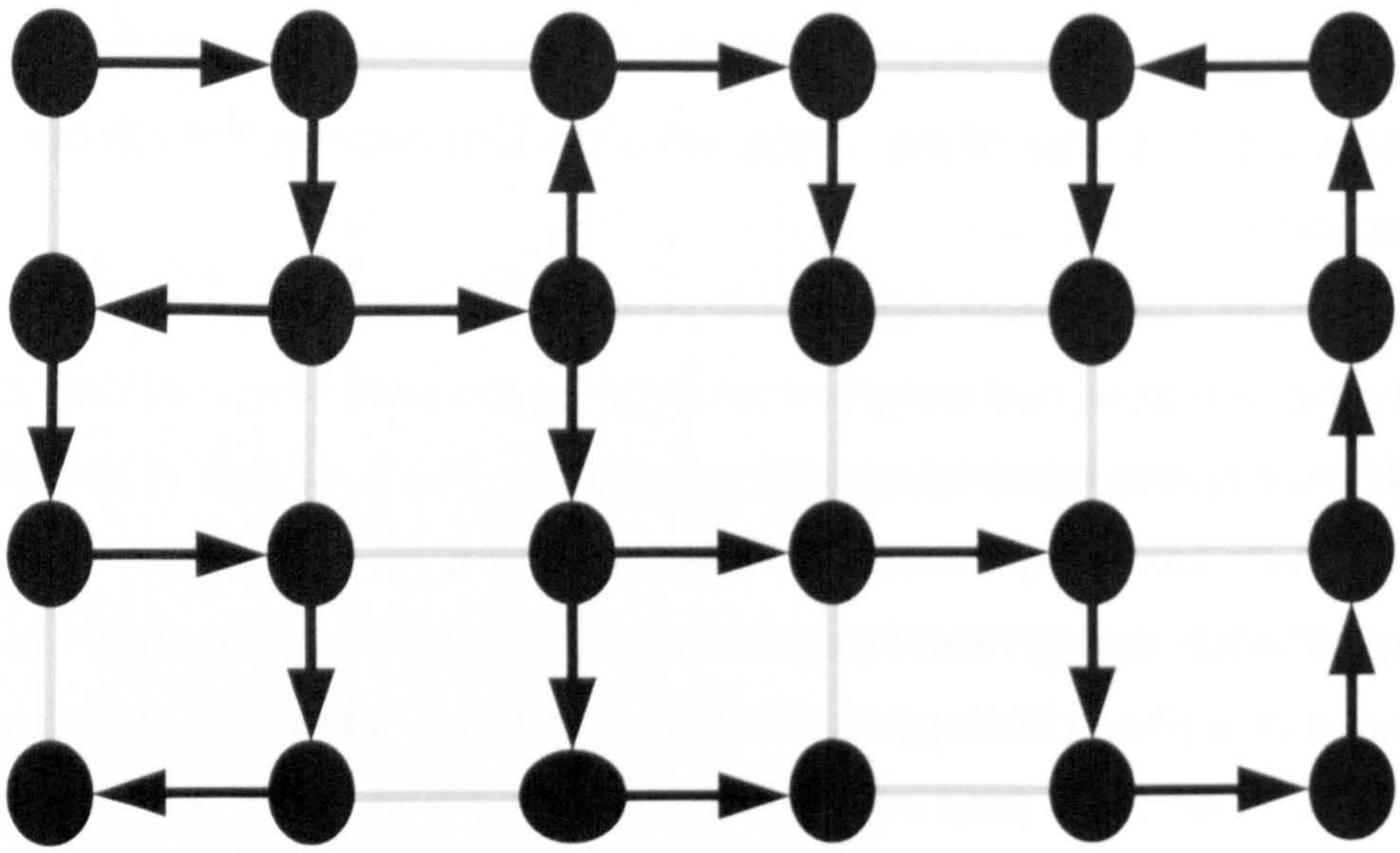


Figure 14 The unwrapping path follows that of the MST. MST edges are shown as dark arrows and the discarded graph edges are shown in faint lines.

A running total of the added and subtracted multiples of 2π is maintained, and is used to calculate the offset applied to remove the phase discontinuities.

This is essentially the same method as used by the path dependent fringe counting (3.4.1 The Fringe counting approach). The main difference being that the unwrapping path followed in this case is chosen more selectively.

Figure 15 shows the result of the phase unwrapping of the tiles of the computer generated image shown in Figure 11.

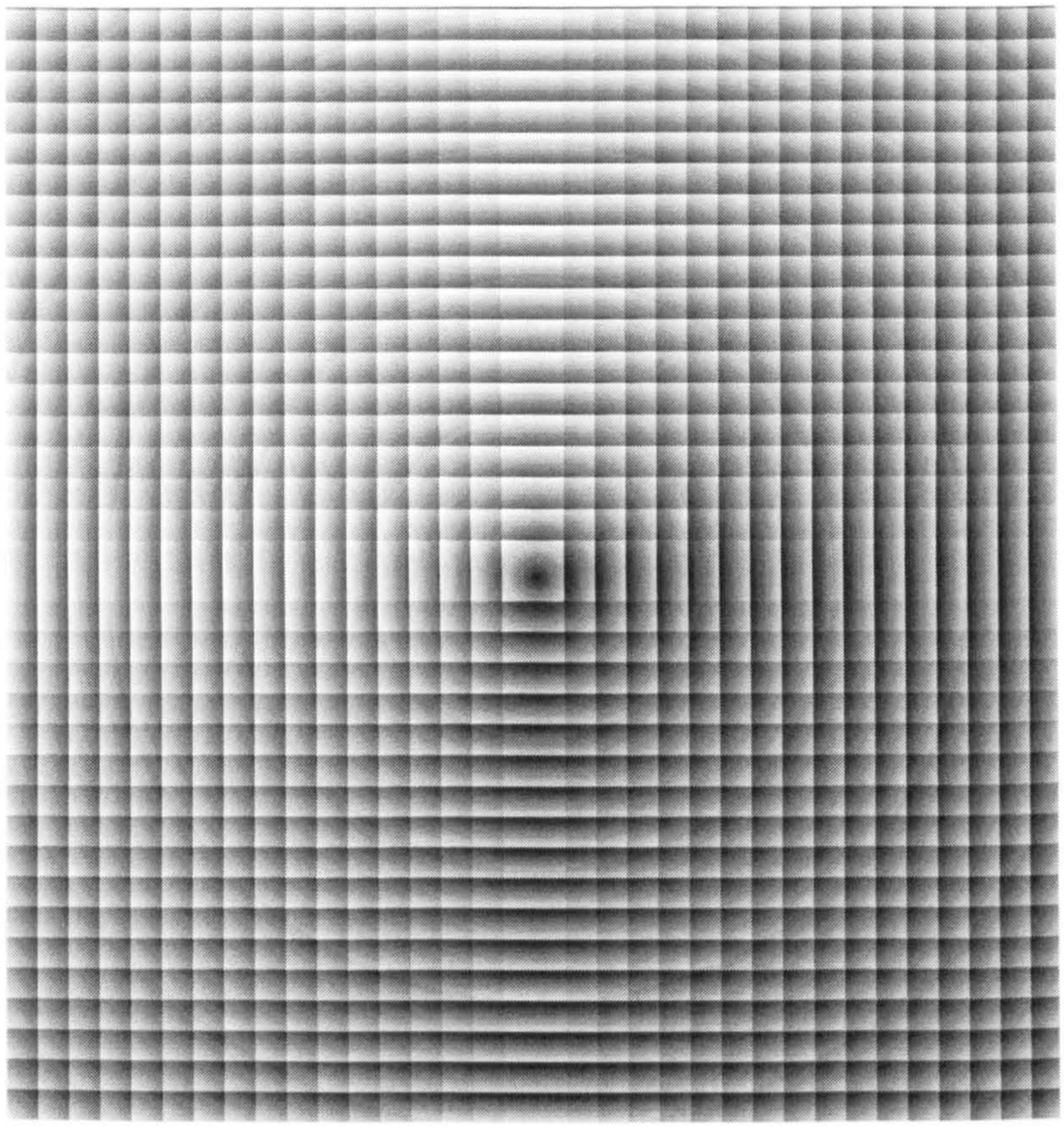


Figure 15 The tiled are individually unwrapped. This is the result of unwrapping the image in Figure 11. The unwrapped intensities in each tile are normalised to utilise the full range of the 8 bit grey scale used.

4.6. Assembling the tiles

The individual tiles are viewed as vertices in a weighted graph as shown in Figure 16.

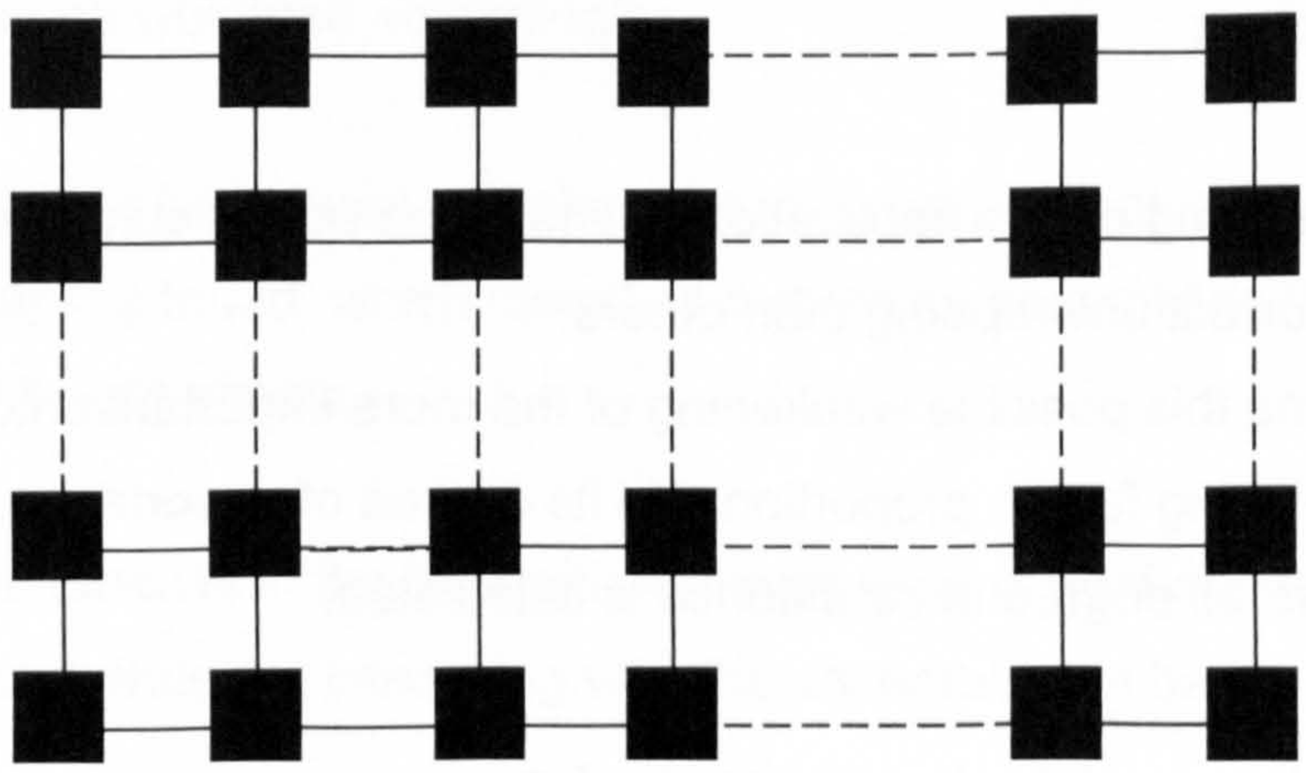


Figure 16 Tiles are vertices of a top level graph

The weights of the edges are calculated according to various criteria, which often depend on the particular application. These include:

- Fringe density. The higher the fringe density in a tile the higher the probability of an unwrapping error occurring in that tile.
- Presence of low modulation points in a tile. The higher the number of such pixels, the worse the quality of the tile's unwrapped information.
- Number of inconsistencies (fringe termination points). The higher the number of fringe breaks (i.e. edge end points) the more unreliable the unwrapped information becomes. Fringe termination points can be detected by testing the immediate orthogonal neighbourhood of each fringe pixel. If a fringe pixel does not have at least two adjacent fringe pixels it is marked as a fringe termination pixel. Fringe pixels located at the boundary of the tile would naturally be expected to have at least one adjacent fringe pixel rather than two.
- Agreement over the overlap region profile. The worse the agreement of the tiles over the unwrapped values of the overlapped region, the higher the probability of erroneous tile assembly along the edge is.

The higher the assigned weight of an edge is, the lower the confidence of the tile assembly along that edge.
Thus, the best path is identified by the minimum spanning tree of the weighted graph.

Often, depending on the application, some of the above criteria are more critical to correct unwrapping than others.
To overcome this possible weakening of the more important criteria, each is given a weighing factor, proportional to its degree of importance, before the average overall degree of confidence is calculated.

For example, two of the factors may be for:

- The total number of low modulation pixels in a tile
- The total number of fringe break points in a tile

Unwrapping ESPI phase maps typically suffers from both of the above problems, and therefore both of these factors are of relevance. However, it is possible to argue that the presence of low modulation pixels is less of an issue as these can be masked off (i.e. ignored) or replaced with the value of the average, or sometimes preferably the median, of good pixels in the immediate neighbourhood.
Fringe break points are usually more of a problem as the fringe interpolation algorithm needed to rectify such faults is likely to be complex as it typically needs to be customised to take into account the expected shapes of the fringes for the particular application.

The above example highlights the need to for assigning an appropriate weight to the factors employed. In this case, the factor for the total number of low modulation points would be less than that for the total number of fringe breaks.

For best results, it is advisable to record an experimental analysis history of similar interferograms obtained for the particular application.
This would help to identify the factors which play the most important role, and thus they can be weighted accordingly.

Once the weights have been calculated, a minimum spanning tree for the weighted graph is found, which identifies a route for assembling the tiles into their correct height offsets.

This approach results in the tiles with the least level of confidence to be assembled last, thus not interfering with the assembling of better quality tiles.

This in effect reduces unwrapping error propagation from the noisy areas of the interferogram to the rest of the unwrapped phase map.

Tile assembly proceeds from a chosen tile, called the root tile. Tiles adjacent to the root are added one at a time, in a jigsaw puzzle fashion, until the full unwrapped image is constructed, Figure 17.

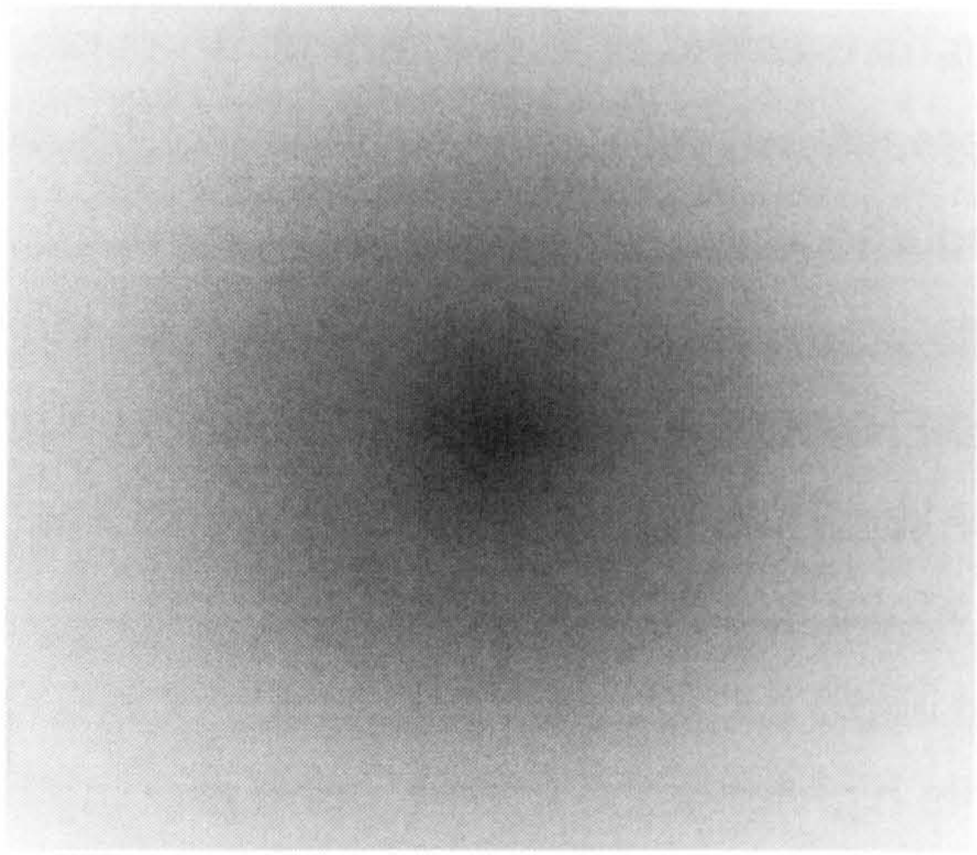


Figure 17 The unwrapped phase map is obtained by assembling the individually unwrapped tiles shown in Figure 15.

As each tile is added, its height is adjusted by an offset calculated by summing the heights from the root tile, across other assembled tiles, to the connecting edge along which the tile is to be assembled.

4.7. Practical considerations

In practice, there are other steps to the algorithm which are necessary to overcome the problem of identifying valid phase jumps in particularly noisy images, such as the wrapped phase map shown in Figure 18.

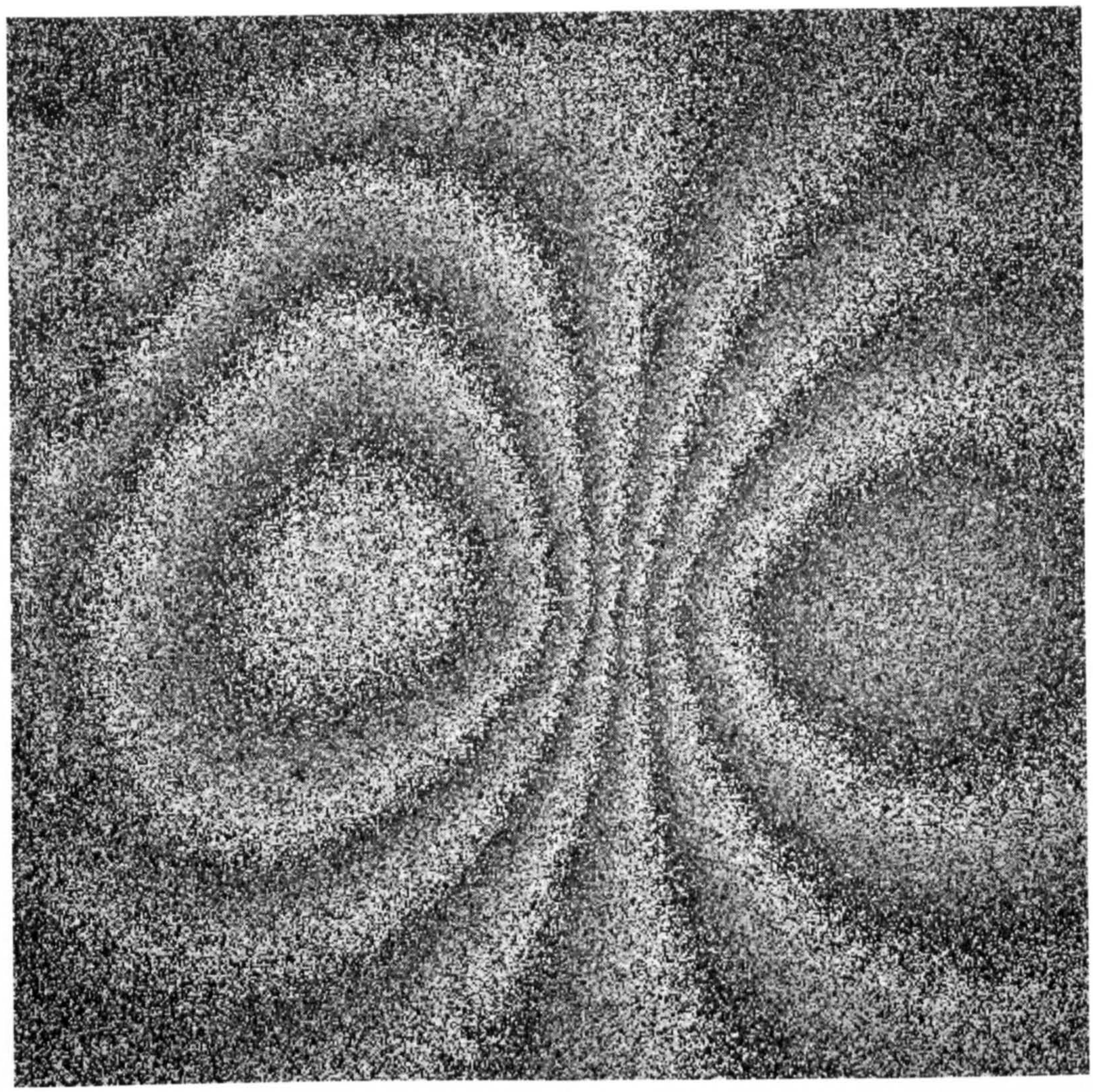


Figure 18 A speckle-interferometry wrapped phase map which is noisy by nature

4.7.1. Edge detection

Judge [62] proposes the use of the image processing technique of edge detection in order to identify the valid phase jumps.

Judge describes a two-threshold iterative method.

First, the wrapped phase map is subjected to edge detection using a high threshold, so that high quality edges are identified.

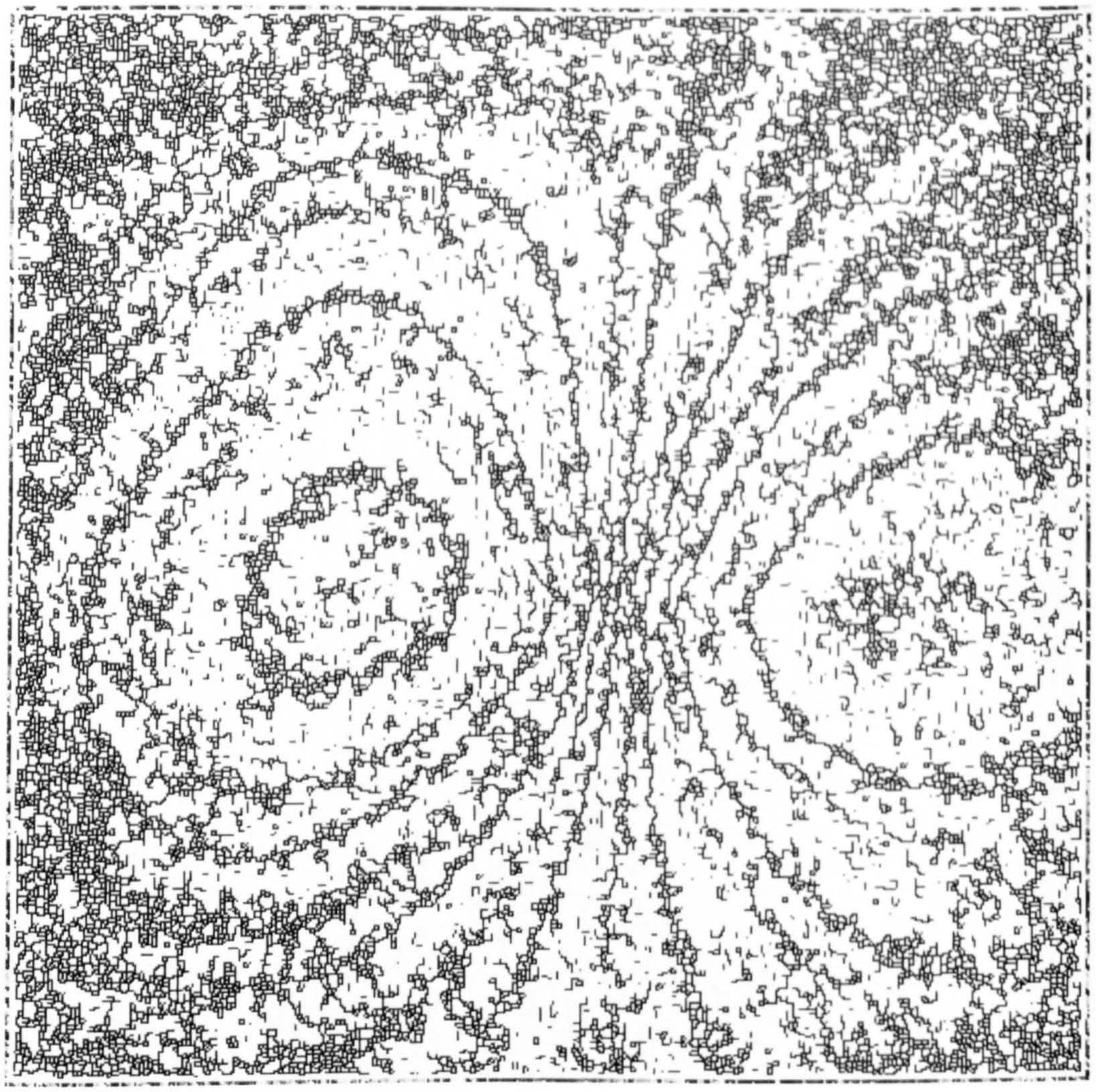


Figure 19 Iterative edge detection of a particularly noisy image does not always yield a useful output

The next iteration uses the low threshold for edge detection, which usually results in identifying lower edge quality candidate pixels, which are considered as potential candidates.

In the next phase, a candidate edge is promoted to a confirmed edge if it satisfies the condition of being adjacent to a confirmed edge.

The process is repeated, until no more candidate low-threshold edges can be confirmed.

Judge attributes the importance of using two thresholds, instead of the conventional use of only one threshold, to the fact that the value of the phase jump (i.e. fringe edge) corresponding to 2π fluctuates within the wrapped phase map, and thus may evade detection should only one threshold be used.

Judge does not prescribe any specific values for the two thresholds, although it is understood that some experimentation may be necessary to identify the threshold values which give good and consistent results for a given application.

Judge [62] examined the merits of various edge detection operators, such as the Sobel, Prewitt, Roberts and Hueckel.
Judge identified the Sobel operator, whose convolution kernels are shown in Figure 20, as the edge detector which offered a good compromise of computational speed and effective accurateness in relation to the two threshold edged detection described.

-1	0	1
-2	0	2
-1	0	1

Kernel used to obtain the horizontal gradient of the edge component

-1	-2	-1
0	0	0
1	2	1

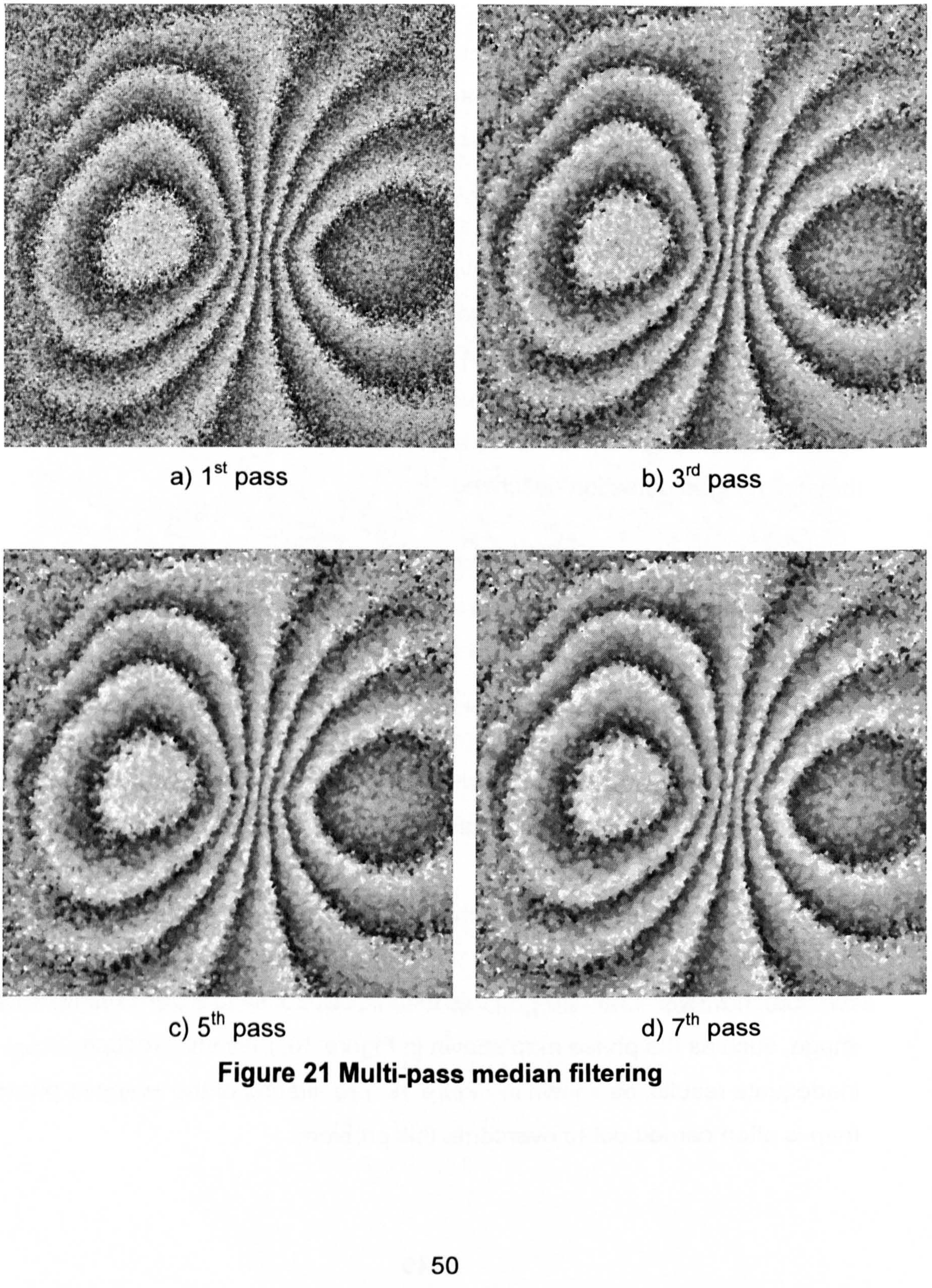
Kernel used to obtain the vertical gradient of the edge component

Figure 20 The convolution kernels of the gradient-based Sobel edge detection filter

Judge successfully demonstrated the effectiveness of this approach when processing ESPI images.
We note, however, that applying this technique directly to a particularly noisy image, such as the phase map shown in Figure 18, may still produce inadequate results, as shown in Figure 19. pre-filtering of the wrapped phase map is often carried out to overcome this problem.

4.7.2. Pre-filtering

To overcome the difficulty of correctly detecting valid phase jumps, or in other words the actual locations of the fringes, it is often the case that the wrapped phase map is subjected to low pass or median filtering beforehand.



a) 1st pass

b) 3rd pass

c) 5th pass

d) 7th pass

Figure 21 Multi-pass median filtering

Median filtering is particularly useful due to its ability to remove spike noise while preserving image edges.
Median filtering is usually applied multiple times to increase the amount of noise removed, hence further enhancing the image, Figure 21.

Edge detecting the filtered phase map using the two threshold procedure described by Judge produces a better result, Figure 22. However, the precise location of fringes remains ambiguous due to the effective width of the fringes thus detected.

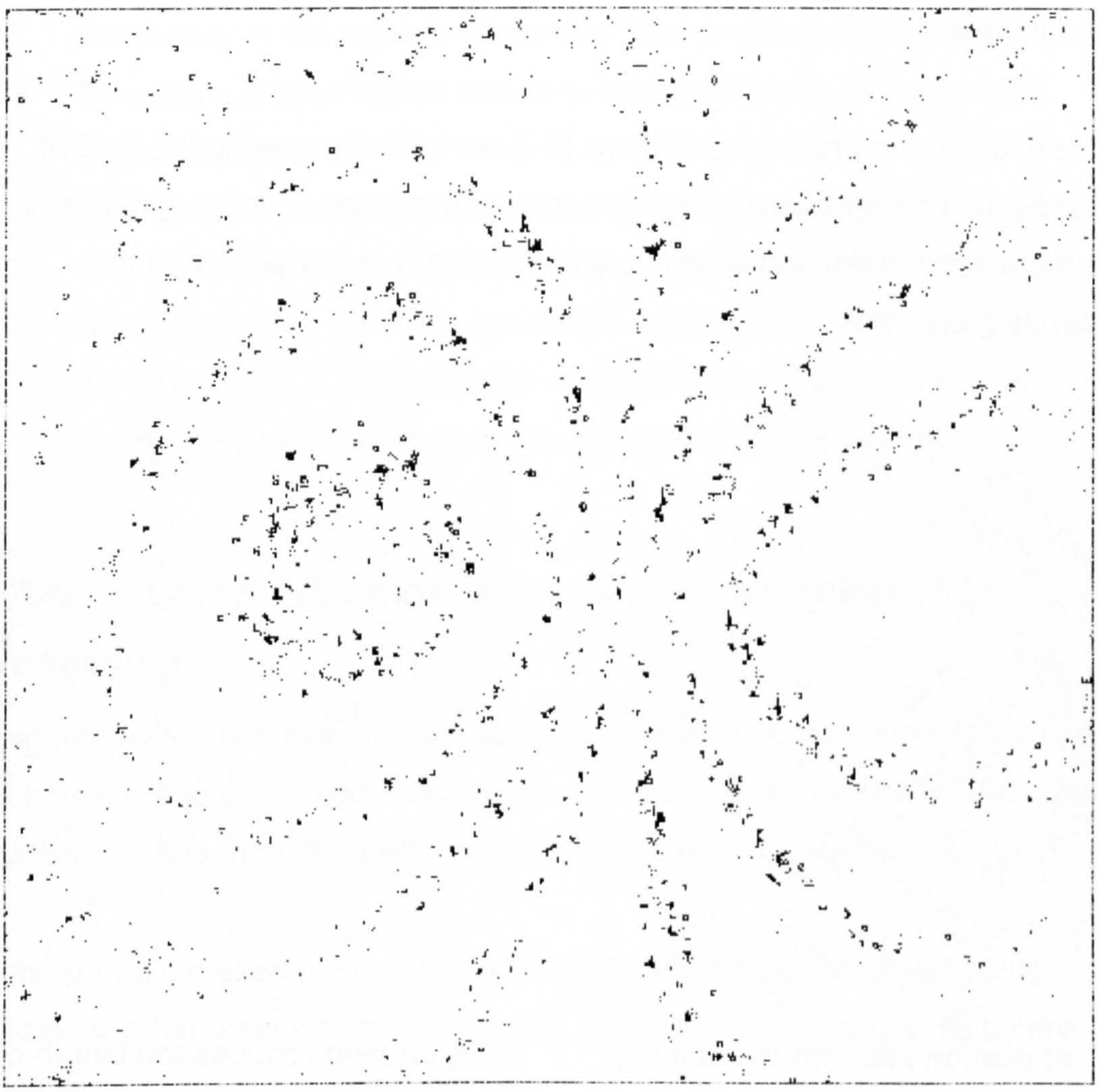


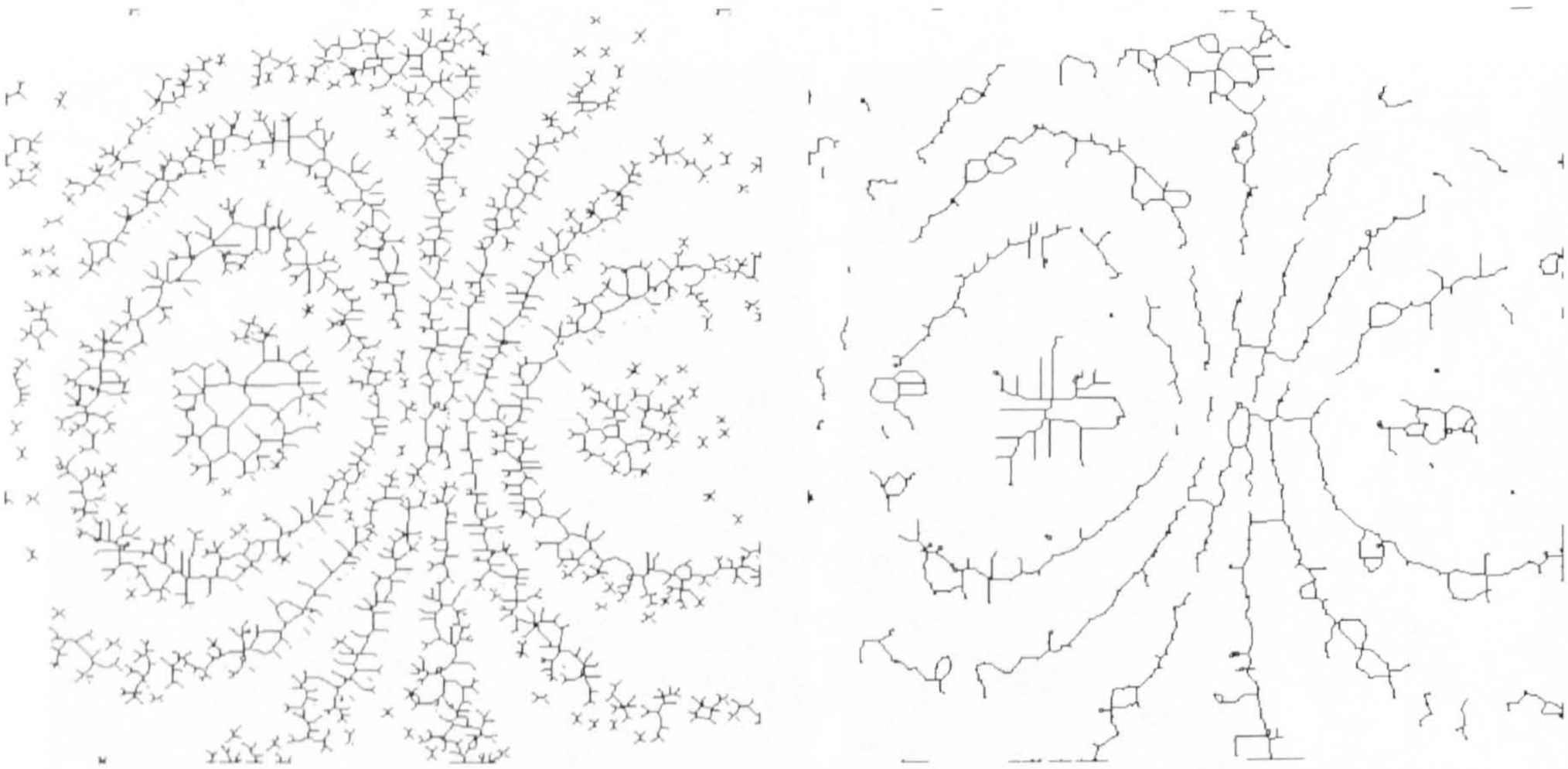
Figure 22 The result of edge detection after multi-pass median filtering (Figure 21-d) is less noisy, however the precise location of fringes remains ambiguous

We have identified a way of overcoming this fringe location ambiguity by combining Judge's two threshold edge detection approach with a thinning algorithm, which we present in section 8.3.2 on page 100.

4.7.3. Morphological operators

We also investigated the possibility of using morphological operators to improve on the quality of the obtained fringe edges. In particular, we identified the dilation and pruning operations, Figure 23, as suitable candidates.

The dilation and pruning operations [89] are typically used in the field of handwriting recognition to identify a textual character which may contain unexpected breaks and disjoints, such as due to a low quality image scanning process.



a) dilation followed by thinning produces "branches"

b) pruning reduces the length of the branches

Figure 23 Morphological dilation and pruning when applied to edged detection information (Figure 22)

The morphological operators have shown to have the potential for identifying more contiguous fringe edges.
Unfortunately however, the initial findings, Figure 23, were not particularly encouraging due to the following issues:

- Morphological operators are pattern matching processes, and are therefore inherently computationally intensive.
- Many repeated iterations of dilation, followed by thinning and then pruning are required to achieve a reasonable result.
- Dilation by its nature causes adjacent fringes to be erroneously joined, particularly in the regions where the fringes are of high density.
- Thinning a dilated region results in the appearance of unwanted branch features.
- The number of required subsequent pruning iterations is not always obvious and requires experimentations: A small number of pruning iterations leaves too many unwanted branches in the image. And, on the other hand, a large number of iterations results in some of the shorter fringes or fringe sections being removed altogether.

4.7.4. Low modulation points and partial-image processing

Low modulation points in the wrapped phase map are due to many factors, such as the angle of object illumination, the illumination (typically Gaussian) distribution, and the reflective properties of the object's surface.

In the particular example used throughout this chapter, the object being studied is a flat plate which is subjected to a mechanical load at its centre.

The plate is clamped in place along the boundaries of its four edges. Therefore, the boundary regions of the obtained image suffer from low modulation the most.

This is partly due to the fact that the object shape's deformation (or displacement) is of maximal extent around the central region, where the load is applied. And, conversely, the object deformation is of minimal extent around the boundary regions, where the object displacement is mostly resisted by the clamping frame. This is further aggravated by the typically Gaussian profile of the illuminating laser beam which maximal (brightest) centre is targeted at the centre of the plate, and its decaying extremities coincide with the edges of the plate. In addition, part of the clamping arrangement is in front of the plate, obscuring some of theses regions somewhat.

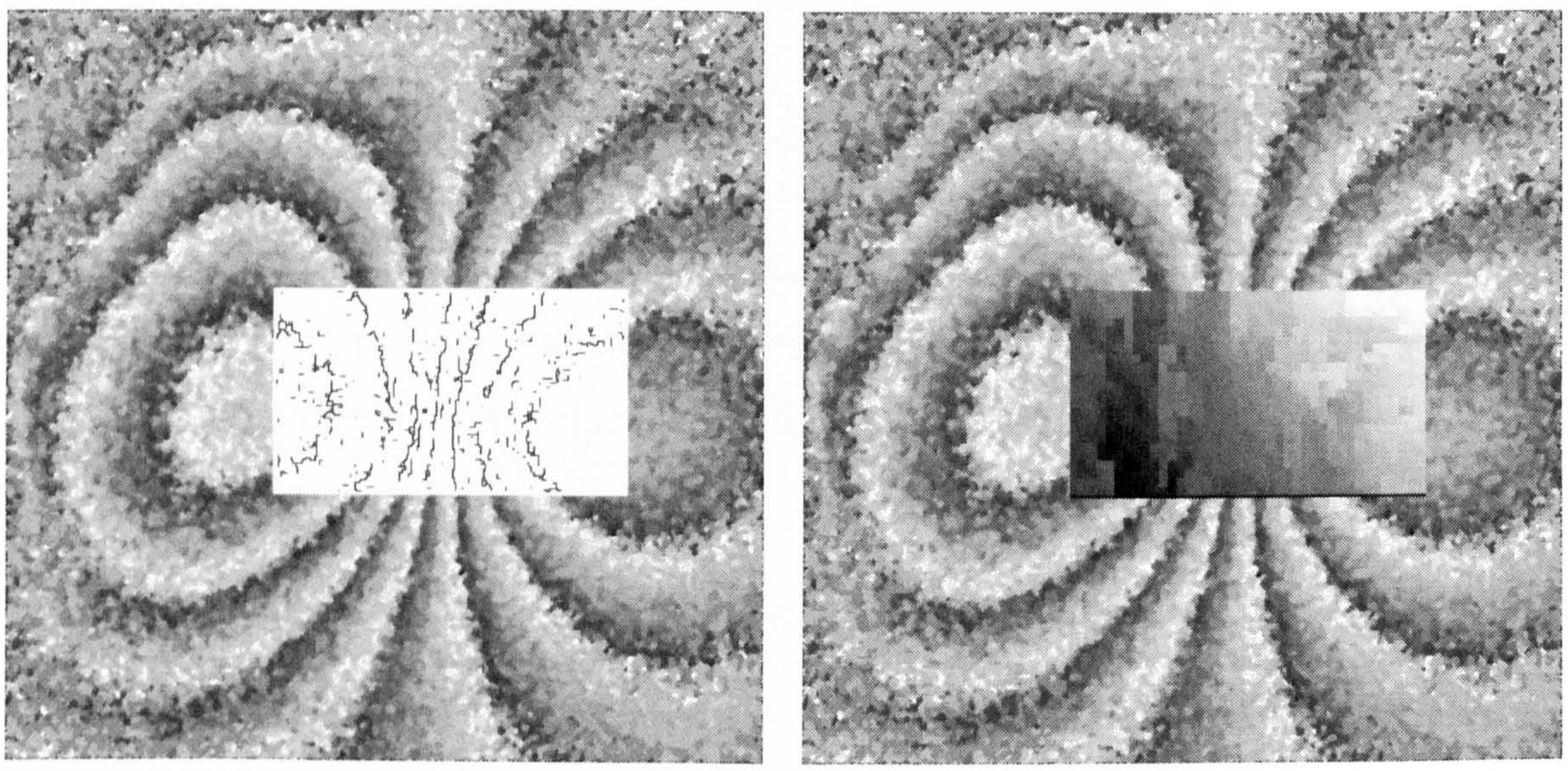


Figure 24 Targeting a specific rectangular region for processing

Processing only the central region of such a phase map avoids wasting the computational resources on unusable information. This also has the added benefit of generally obtaining a better quality unwrapped phase map, due to the decreased overall probability of errors and the associated risk of error propagation to the rest of the field.

In our implementation, we introduced the flexibility of specifying a rectangular region to which image processing and phase unwrapping is to be confined, as shown in Figure 24.

4.8. Discussion and identification of the research problem

Our practical implementation of the tile-based method of fringe unwrapping has allowed us to gain a better understanding of the more detailed aspects of the algorithm.

Working with wrapped phase maps obtained from optical experiments, as well as computer generated images, has allowed us to have a better appreciation of the various image processing issues encountered in practice.

In particular, we have identified the reliability of the methods used for locating fringe edges as a candidate area for further research consideration.
The reliable identification of fringe edges, however, is not confined to the tile-based method, but rather a fundamental problem in the more general domain of phase unwrapping.

More specific to the tile-based method is the impact of the chosen tile-size on the overall performance of the algorithm.
A restriction on the size of the tile is imposed by the performance of the minimum spanning tree (MST) algorithm.
Judge, the author of the tile-based approach, states that this is particularly an issue due to the fact that the performance of the MST algorithm is proportional to the square number of pixels in a tile.

Indeed, Judge sights this restriction as a motivation for dividing the image into tiles [63], thus reducing the computational complexity from $O(n^2)$ to $O([n/m]\, m^2)$, where m is the number of pixels in a tile, and n the number of pixels in the image. This can be illustrated by a simplified example; with $n=512$ and $m = 16$, the computational complexity is reduced from the order of

262144 operations to the order of 8192 operations. This does not include other computational overheads such the time taken by the tile assembly process.

Therefore, we have identified the improvement of the performance of the underlying minimum spanning tree algorithm as the primary area of our research, as it holds the key to both tile-size flexibility and the performance efficiency of the overall unwrapping algorithm.

In the following chapters we discuss the minimum spanning tree aspect from a graph theory perspective, and describe a novel algorithm to improve on its computational performance.

Chapter 5 Tile topology from a graph theory perspective

5.1. Introduction

We give here a brief summary of the relevant graph theory principles [98-101] which are relied upon in the descriptions given in the subsequent sections and chapters.

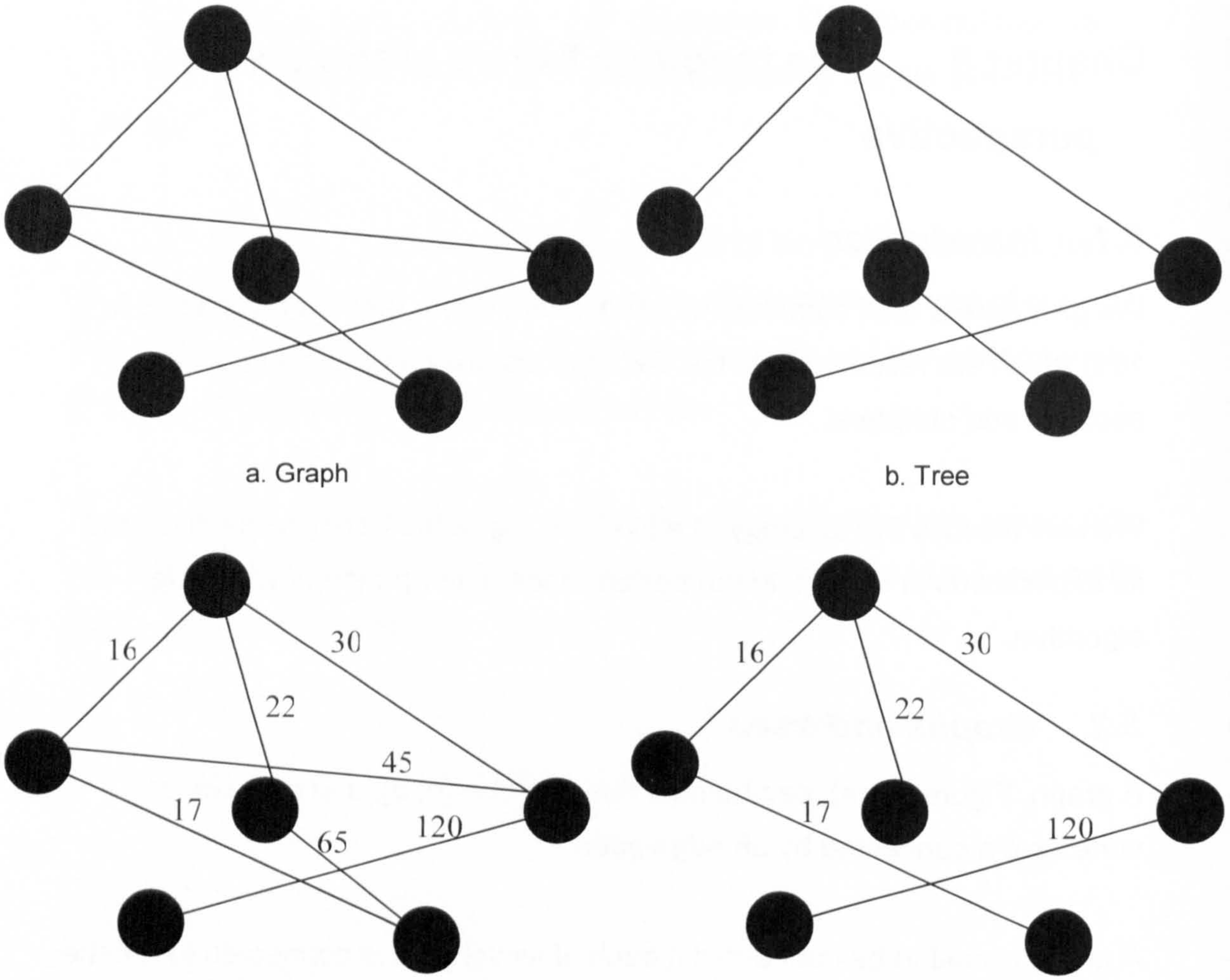
We also discuss the topology of a tile from a graph theory perspective, and its implications in relation to the performance of the phase unwrapping algorithm.

5.2. Graphs and trees

A graph, Figure 25 (a), can be described as G = [n, e], and consists of a vertex set n connected by an edge set e.

A graph is said to be connected if each of its vertices is connected to another vertex by at least one edge. A spanning tree, Figure 25 (b), is an acyclic connected graph. This means that each vertex in a tree is connected to any other vertex via one path only. Thus, an n vertex tree contains n -1 edges.

A weighted graph, Figure 25 (c), is a graph whose edges have a cost associated with each edge.
Finding a minimum spanning tree (MST) of such a graph is finding a tree whose total edge cost is minimal, Figure 25 (d).



a. Graph

b. Tree

c. Weighted graph

d. MST (minimum spanning tree)

Figure 25 Graph, trees and MST

5.2.1. Tree forests

Any selection, or a subset, of vertices in a graph, along with the edges which connects them, can be considered as a sub-graph, which is simply termed subgraph.

A subgraph may also be acyclic and connected, and therefore have a tree structure. Such a tree is similarly called a subtree of the graph.

A graph may contain more than one acyclic subgraph, i.e. more than one subtree. A collection of subtrees is often referred to as a tree forest of the graph.

5.3. *Tile graph topology*

The tiling method of phase unwrapping commences by dividing the wrapped phase map image into tiles.

The pixels of a tile become the vertices of the graph. These are connected by edges in a grid fashion, as shown in Figure 26.

Each edge is then assigned a weight equal to the absolute phase difference of the two pixels it connects.

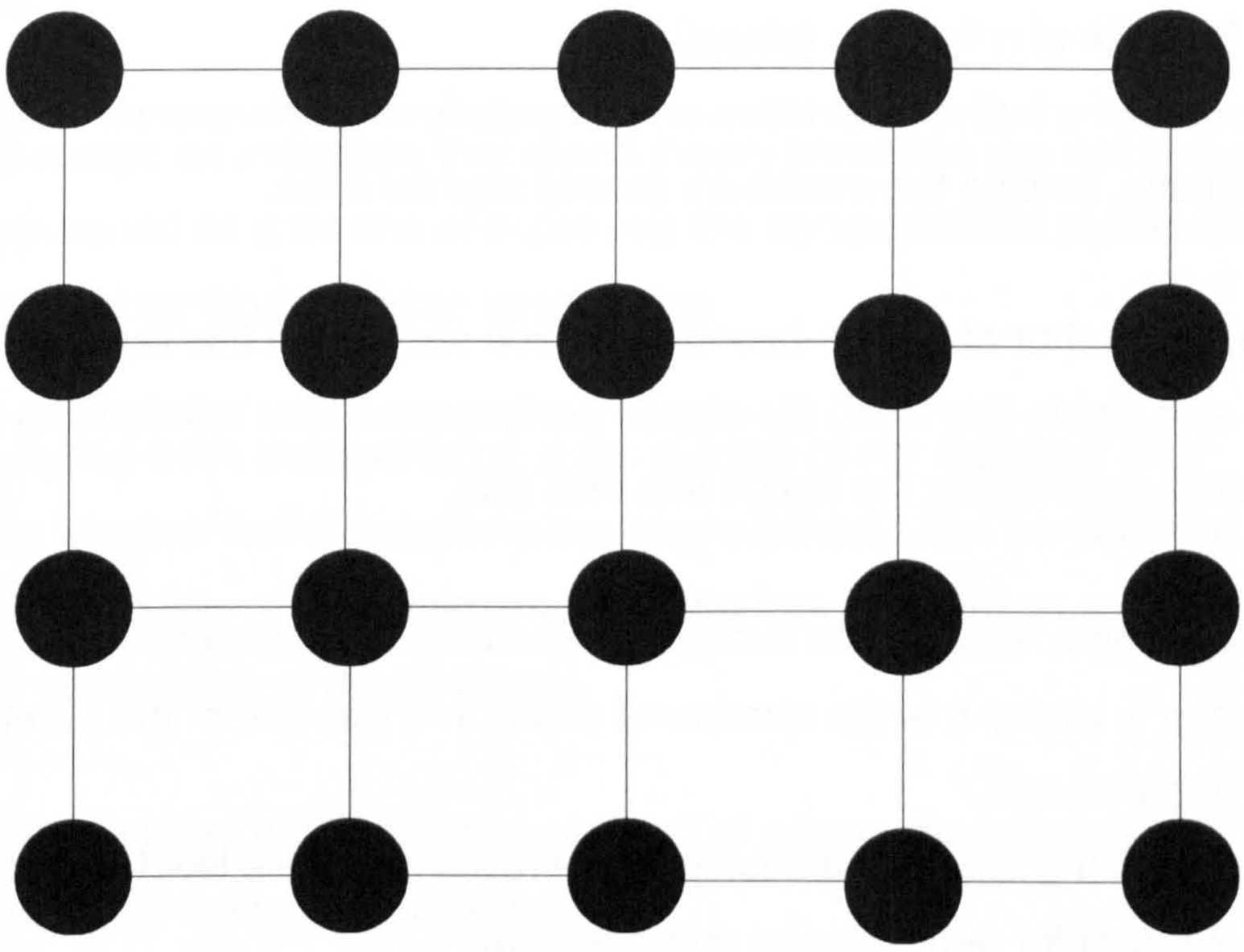


Figure 26 Topology of a tile's graph

Depending on the application, there are often minimum and maximum tile sizes between which the tiling method provides the best results.

There are various considerations taken into account when choosing the tiles' size, such as the expected fringe density and the size of fringe breaks. This is mainly to ensure that tiles are not so small that they often fall in between broken edges, and not so large so that they often contain sections of different fringes [62].

5.3.1. *Impact of tile-size on computational performance*

In addition to the limitations on tile-size stated in the previous section, we also note that there is another consideration in relation to the maximum tile size from a computational point of view.

Some MST algorithms, such as Prim [97], have a non-linear performance relationship with respect to the number of vertices (pixels) in the graph. Depending on the particular implementation, this could be $O(n^2)$ in the worst case, for a tile of n vertices (pixels).
This imposes a further restriction on computational performance sensitive applications, limiting the maximum usable size for a tile.

In fact, the author of the tile-based approach stated that this is particularly an issue, and sights this $O(n^2)$ tile-size to performance time relationship as a motivation for dividing the image into tiles [62].

The use of tiles reduces the overall computational complexity from $O(m^2)$ to $O([m/n] n^2)$, where n is the number of pixels in a tile, and m the number of pixels in the image.
The fact that the $O(n^2)$ relationship still persists at the tile level makes it an obvious target for performance improvement.

There exist implementations in the graph theory literature of Prim's algorithm which have a better time performance, O(e log n) [102, 103].

Indeed, an independent worker in the field of medical imaging has made use of an O(n log n) implementation, and published the improved overall phase unwrapping algorithm [65] after our research had already started.

5.3.2. *Tile graphs are planar*

A planar graph is a graph which can be drawn in a single plane, such that none of its edges intersect.

We observe that the actual topology of a tile's graph is planar. This allows for the use of deterministic and linear time MST algorithms [112, 113], which perform in O(n).

5.4. Chapter conclusion

To our knowledge, the fact that a tile's graph is planar, and the performance enhancement implications thereof, is yet to appear in the phase unwrapping literature.

This is perhaps an indication that graph theory principles are not being actively pursued as a means of improving the computational performance of the tile based method of phase unwrapping.

Therefore, we have determined that the subject of our research is to ascertain whether further performance improvements can be obtained by a closer examination of the minimum spanning tree problem on a tile's graph, from the perspective of graph theory.

In the next chapter, we discuss a selection of minimum spanning tree algorithms, and their various merits.

In subsequent chapters, we rely on graph theory to describe a novel algorithm which improves on the computational performance of any chosen minimum spanning tree method.

Blank

In

Original

Chapter 6 Minimum spanning tree algorithms

6.1. Introduction

The minimum spanning tree (MST) is one of the best known problems of graph theory, and perhaps the best studied optimization problem in computer science.
The minimum spanning tree problem is also generally accepted to be the first network optimisation problem to have been studied [90].
The problem can be stated as follows:

"Given a connected (and undirected) graph with e weighted edges and n vertices, the problem is to find a spanning tree of minimum total weight."

Clearly, identifying any spanning tree (regardless of its total weight) takes a search algorithm at best an O(e) time, as each of the edges will need to be examined at some stage.
It is quite remarkable that it is also possible to carry out an optimised search, to specifically find a spanning tree of minimum weight, in little worse than O(e), in the general case [105-109].

Furthermore, in the specific case of planar graphs, as eluded to in the previous chapter, it is possible to identify a minimum spanning tree in O(e) indeed [112, 113].

In this chapter, we first examine the classical MST algorithms of of Kruskal [95], Prim [97] and Boruvka [91, 92].
We then describe some of the more advanced algorithms which are aimed at the general case of the MST problem.
We finally examine two algorithms, which achieve O(n) on planar graphs.

6.2. *Classical MST algorithms*

The algorithms of Kruskal [95] and Prim [97] have been used repeatedly in many text books, on graph theory and computer science alike, to illustrate the MST problem and how it may be solved.

Boruvka's algorithm [91, 92] in fact predates both of the aforementioned, but only relatively recently has benefited from a more extensive coverage. This is probably due to its publication in Czech originally, and perhaps explains why it was unknowingly rediscovered by various independent workers in the field [93, 94].

6.2.1. Kruskal [95]

Kruskal first sorts all of the edges of the graph, Figure 27(a), with respect to each other and stores them in an ordered list.

It then initialises the tree to contain all of the graph's n vertices, and none of its e edges, Figure 27(b).

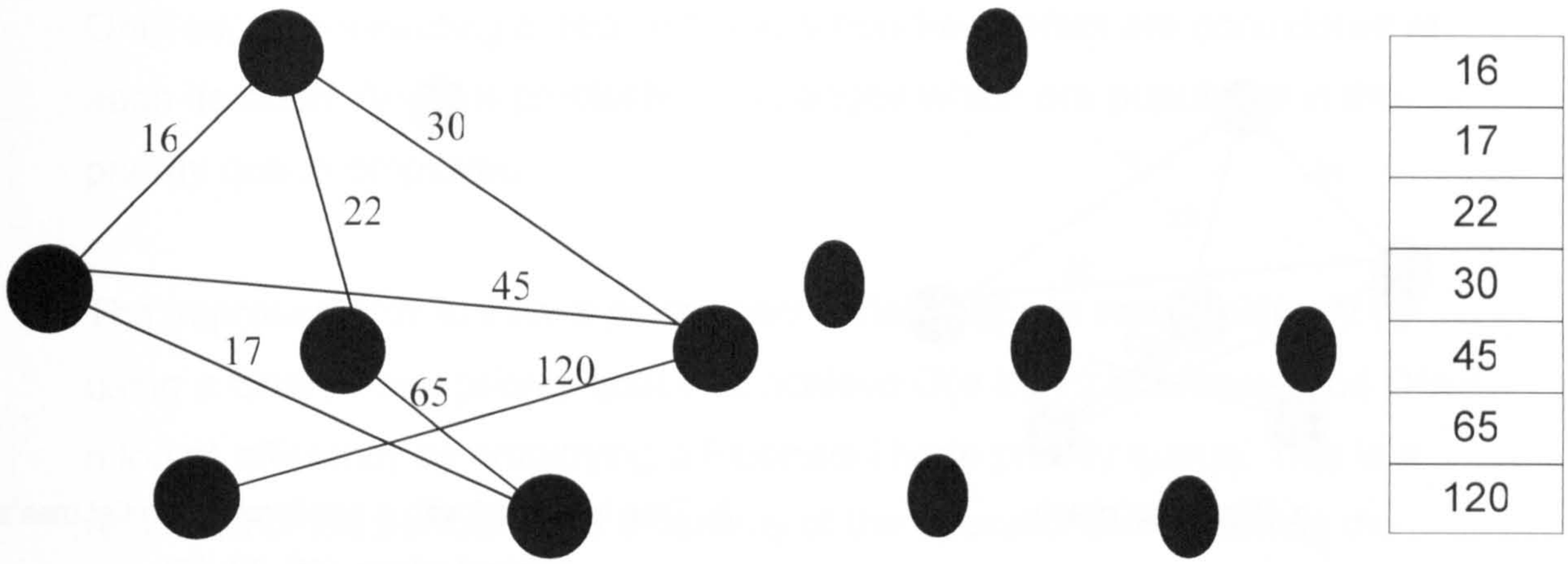
At each iteration, Figure 27(c), the minimum edge is removed from the sorted list and is added to the tree, only if it does not cause a cycle (i.e. more than one path between any two vertices in the tree). Determining whether an edge would cause a cycle can be accomplished by making use of the union-find algorithm [96].

The algorithm keeps iterating in the same fashion until all of the edges have been added to the tree or otherwise discarded.

The resultant tree is guaranteed to be a minimum spanning tree. It is possible to employ a union-find algorithm which has near linear time efficiency [96] for the purpose of testing for possible cycles. However, the overall MST algorithm is dominated by the cost of the initial edge-sort, which limits it to[9] O(e log e).

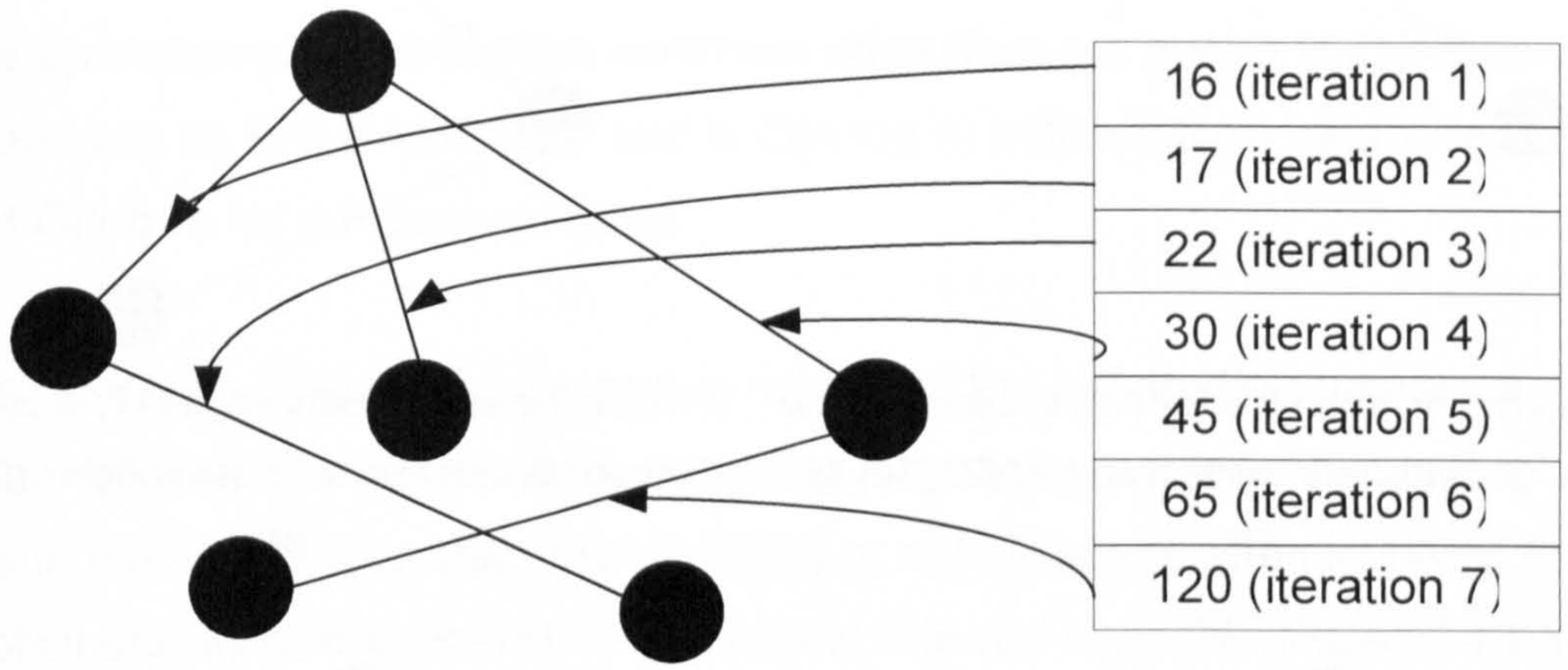
---

[9] In keeping with common practice in computer science, it is implicitly assumed that the computational time complexities cited have a log base of two, rather than ten.



a. Weighted graph

b. Tree initialised with vertices only, and edges are sorted.



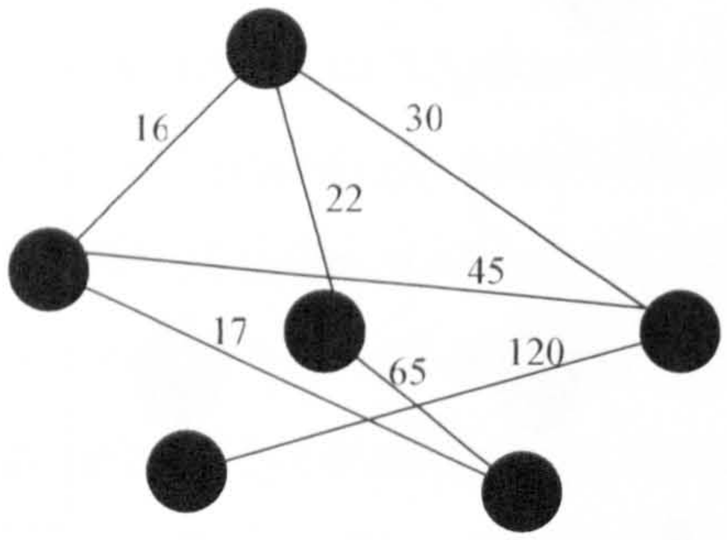
c. At each iteration edges in the sorted list are added to the tree in turn, unless they cause a cycle (iterations 5 and 6).

Figure 27 Kruskal's [95] algorithm

6.2.2. Prim [97]

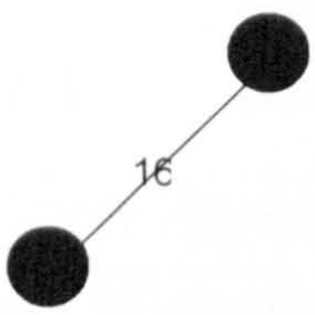
Prim $O(n^2)$ [97], can be implemented more efficiently with modification to achieve by the use of priority queues to O(e log n) [102, 103]. It can be further improved by the use of Fibonacci heap priority queues to O(e + n log n) [104].

These performance savings over Kruskal are mainly due to the fact it does not require the graph's edges, Figure 28(a), to be sorted beforehand. Instead, the algorithm starts from a tree with a single seed vertex, Figure 28(b).

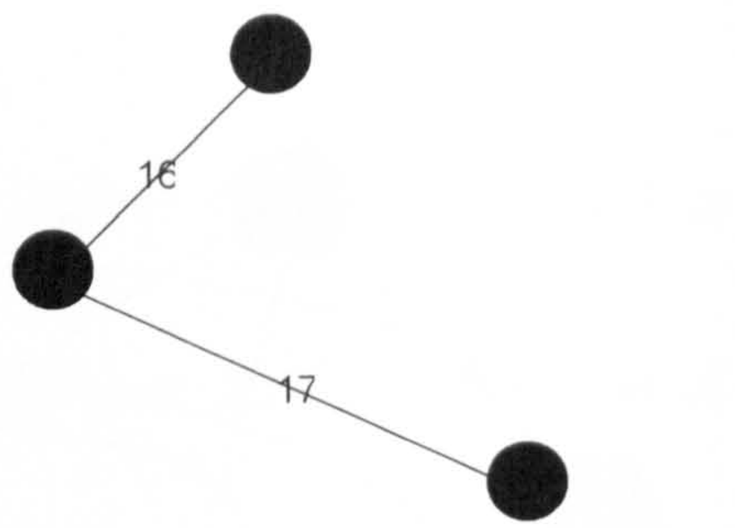


a. Weighted graph

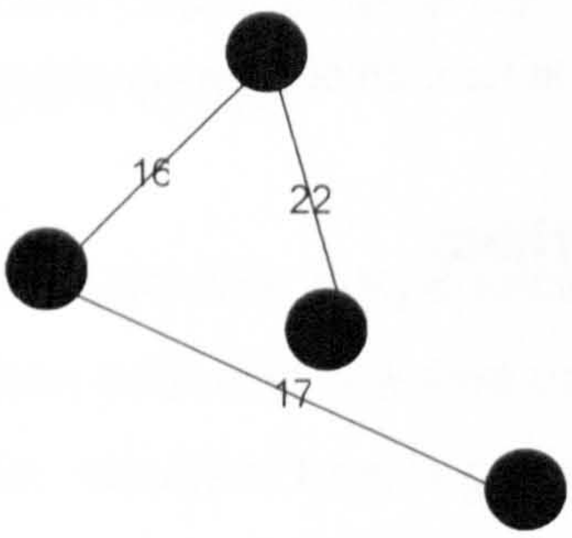
b. Tree initialised with a seed vertex. Initial tree's incident edges: (16, 22, 30).



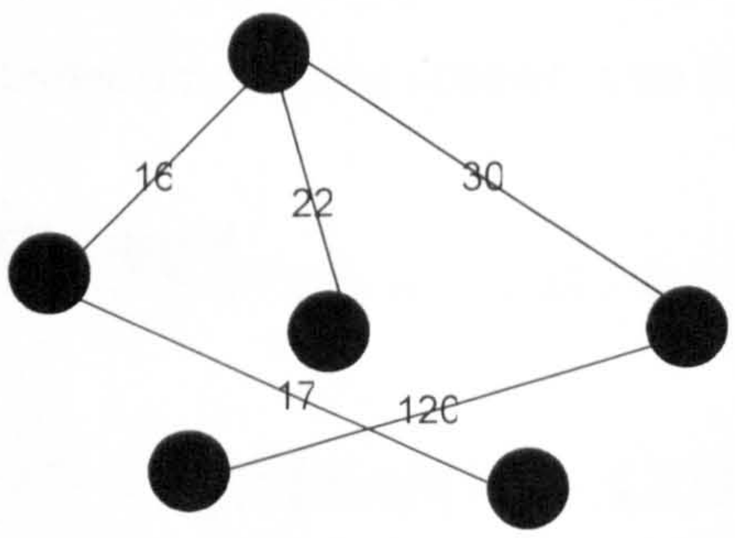
c. The minimum incident edge (16) is used to grow the tree. Grown tree's incident edges: (22, 30, 45, 17).



d. The minimum incident edge (17) is used to grow the tree. Grown tree's incident edges: (22, 30, 45, 65).



e. The minimum incident edge (22) is used to grow the tree. Grown tree's incident edges: (30, 45, 65).



f. The minimum incident edge (30) is used to grow the tree. Tree then has one incident edge (120), which is used last.

Figure 28 Prim's [97] algorithm

It then, at each iteration, Figure 28(c…f), searches all the edges incident to the tree and uses the lightest one to grow the tree by another vertex.

Only edges connecting a tree vertex to a non-tree vertex are considered at each iteration. And it is precisely these edges which are populated in the priority queue employed.

The improvements to Prim's performance stated above were achieved by using a binary heap priority queue to achieve O(e log n) efficiency, and O(e + n log n) efficiency by employing a Fibonacci heap priority queue. This is a reflection of the performance efficiency of the operations of obtaining the minimum edge from the priority queue, and then updating the queue once a new vertex has been added to the growing MST.

The operation of obtaining the minimum edge from the queue is commonly referred to as EXTRACT-MIN, and is O(n log n) efficient for a binary heap, and O(log n) for a Fibonacci heap.

The operation of updating the queue refreshes the information (usually a binary bit flag) of whether each remaining edge to be processed is in the queue (i.e. incident on the growing MST) or not. This operation involves an implicit priority queue operation commonly referred to as DECREASE-KEY, and is O(e log n) efficient for a binary heap, and O(1) constant time for a Fibonacci heap.

The priority queue operations of EXTRACT-MIN and DECREASE-KEY dominate the running time efficiency of Prim's algorithm hence implemented.

In the case of binary heap priority queues, the overall efficiency is O(n long n + e long n) which is in fact O(e log n), as there are at least as many edges as there are vertices in the graph being processed.

In the case of Fibonacci heap priority queues, the overall efficiency is O((n log n) + (e)), which is more commonly stated as O(e + n log n).

As the algorithm at each step considers a single edge connecting a single vertex which is not already in the tree to the tree-graph, the algorithm is guaranteed not to inadvertently create any cycles.

Once all of the n vertices have been added, the resultant tree is guaranteed to be a minimum spanning tree.

6.2.3. Boruvka [91, 92]

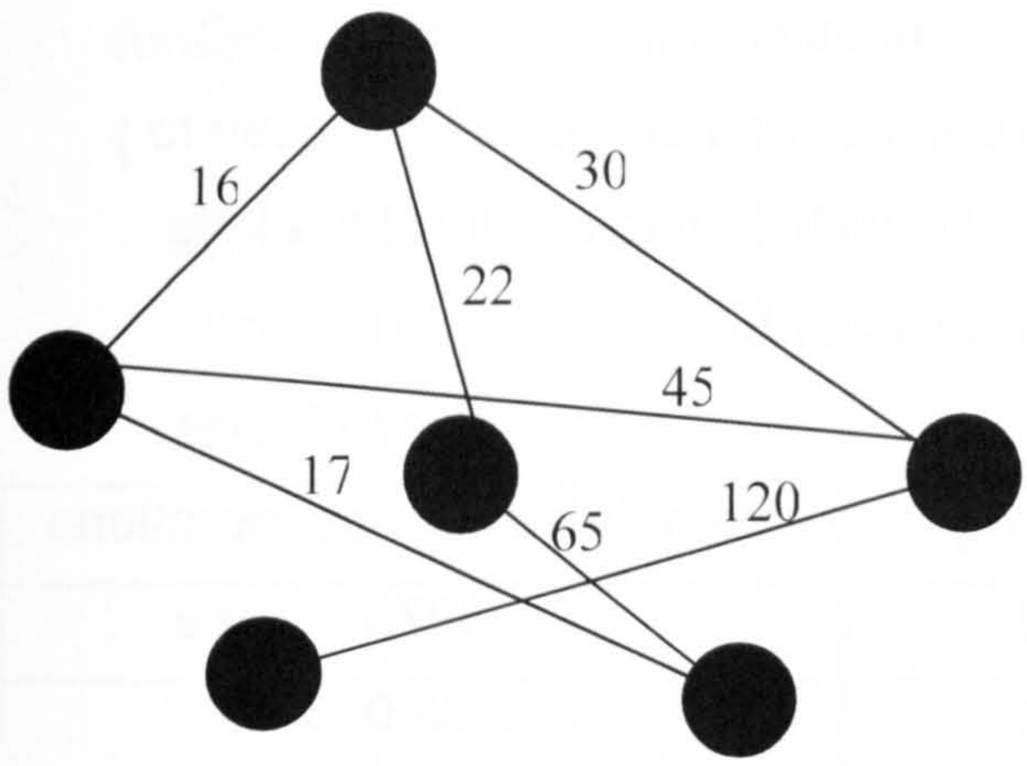
This algorithm starts its work from a copy of the actual graph, Figure 29(a), with all of its vertices and edges included.

Each iteration of the algorithm is two-fold: first, each vertex in the graph is considered, and its lightest incident edge is (notionally) coloured blue, Figure 29(b).
The second step of the iteration is to simultaneously contract all of the blue edges of the graph, thus reducing (or shrinking) the total number of vertices.

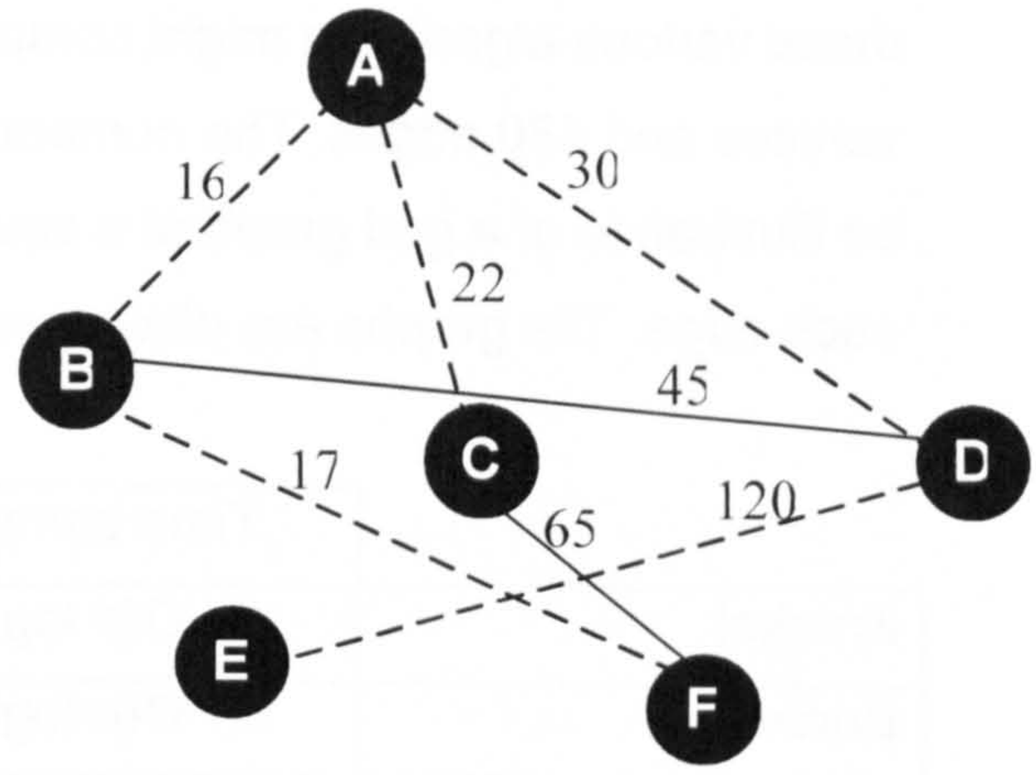
This means that each two vertices connected by a blue edge become a single vertex on which is now incident the union of the two sets of edges which previously belonged to the original two vertices, Figure 29(c). A blue edge itself becomes no longer part of the graph.

The algorithm keeps iterating until the graph has shrunk down to a single vertex, Figure 29(d).

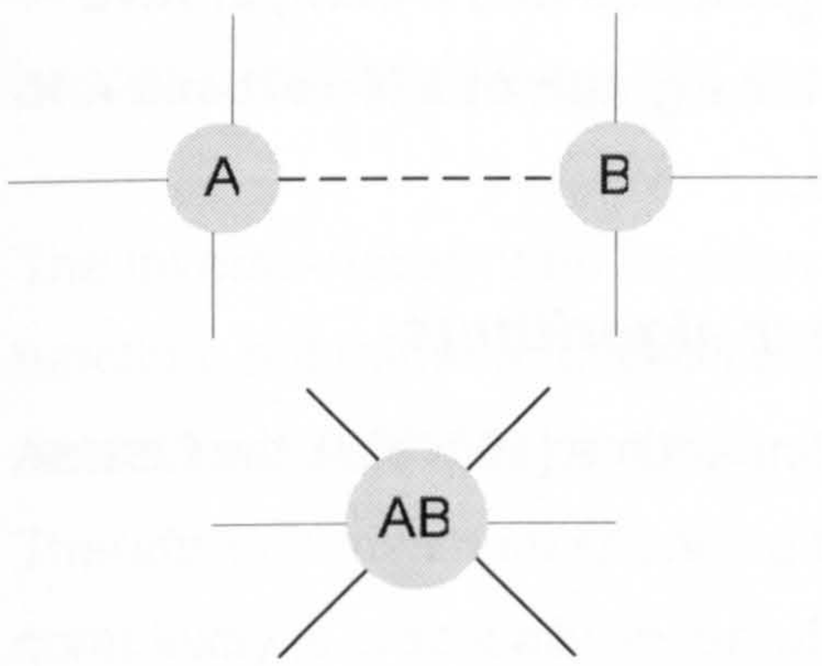
The algorithm also carries out a cleaning up operation after each iteration to remove self-loops (i.e. edges whose both ends become connected to a single vertex).



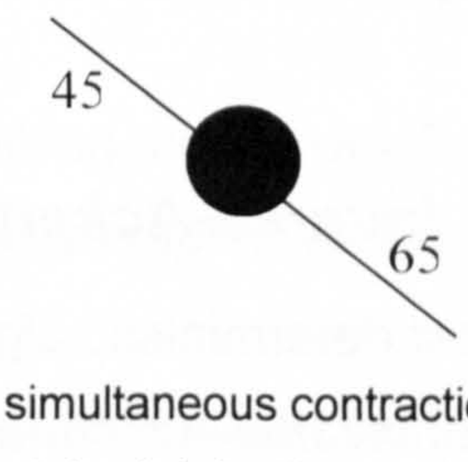
a. Weighted graph



b. Lightest (blue) edges shown in dashed lines. A's minimum edge is 16. Similarly, B→16, C→22, D→30, E→120, F→17.



c. Contracting a blue (dashed) edge. The union of non-blue edges incident onto A and B becomes incident on the merged vertex, AB.



d. Iterative simultaneous contraction of all blue edges ultimately shrinks the graph down to one vertex. In this example, only one iteration is required. Edges 45 and 65 are discarded

Figure 29 Boruvka's [91, 92] algorithm

Once the algorithm has completed, it is guaranteed to have found all of the minimum spanning tree edges by marking them blue.

As well as being the first known MST algorithm, it is also one of the most efficient of the classical methods and performs in $O(\min(e \log n, n^2))$.

6.2.4. Summary of classical MST algorithms

In the proceeding sections we gave brief descriptions of the most common classical MST algorithms of Kruskal [95], Prim [97] and Boruvka [91, 92], quoting their respective computational complexities. Table 1 shows how

these various algorithms might compare in practice, given a graph of 256 vertices and 480 edges. The number of vertices and edges were chosen to be illustrative of a grid graph of a square tile with 16 vertices (pixels) along each edge. Tile graphs are discussed in Chapter 5.

	Time complexity	Example number of operations
Kruskal	O(e log e)	4276
Boruvka	O(e log n)	3840
Prim [97]	$O(n^2)$	65536
Prim [102, 103]	O(e log n)	3840
Prim [104]	O(e + n log n)	2528

Table 1 Summary of classical MST algorithms and a comparative example of their number of operations for a graph of 256 vertices and 480 edges.

6.3. Inverse-Ackermann near linear algorithms

The best deterministic algorithms are near-linear time [105-109], the fastest being of inverse-Ackermann efficiency.

The Ackermann function can be defined in various ways depending on the preferred notation, be it mathematical or otherwise.
Typically, it can be defined in the following recursive fashion (n and m are non-negative integers):

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

In terms of algorithmic implementation, the following pseudo-computer program snippet shows how the function recurs by calling itself when m ≠ 0:

```
function Ackermann (m, n)
{
   if (m = 0)
      return n+1
   else if (n = 0)
      return Ackermann (m-1, 1)
   else
      return Ackermann (m-1, Ackermann (m, n-1))
}
```

The recursive implications of the function are that, even from small inputs such m = 4 and n = 3, the output values, as well as the number of iterations, become unfeasibly large and practically incomputable[10].

The inverse-Ackermann function is not strictly speaking the "inverse" of the function, but rather a function which shrinks in value as rapidly as the Ackermann functions grows.

Therefore, a minimum spanning tree algorithm of inverse-Ackermann type complexity is practically linear, although not strictly so.

The essential concept behind inverse-Ackermann MST algorithms, such as [106], is to delay the processing of as many edges as possible for as long as possible.

This follows the intuitive notion than many of the delayed edges may become discarded, such as due to ruling-in an alternative edge, without additional computational cost being incurred. This, clearly, results in conserving the computational effort.

---

[10] E.g. Ackermann (4, 2) = 2×10^{19728}, and Ackermann (4, 3) is far too large to be denoted in a conventional numeric representation.

Delaying the processing of edges is achieved by "hiding" the heavier edges behind lighter ones, and only making them "visible" again once the lighter ones have been processed (i.e. ruled-in or out of the MST).

Identifying which edges are heavy and which are lightest still requires a sorting effort, the computational impact of which is reduced by dividing the edges into subsets or packets.
Edges within a packet are sorted with respect to one another, and the lightest edge is identified in particular.

At each algorithm pass, only the lightest edge of each of the packets is considered. Once the lightest edge of a packet becomes no longer available (i.e. either ruled-in or out of the MST), the second lightest edge becomes visible, and so on.

Beyond this basic concept of the divide and conquer approach, the algorithm has more elaborate details. These include employing a specialised data structure to sort the edges in, and a mechanism for carefully choosing the packet size.

In addition, the algorithm carries out rudimentary chores, such as merging together some of the shrinking packets as they become undersized, during the progress of the algorithm.

By carefully selecting the packet size to match the number of algorithm iterations, along with the employment of efficient data structures (Fibonacci heaps) for sorting and merging the packets, the algorithm is able to achieve an inverse-Ackermann (near-linear) time performance.

6.4. *Randomised linear Karger et al [110]*

This non-deterministic (randomised) algorithm achieves linear time efficiency O(e) [110], by employing an O(e) minimum spanning tree verification method [111].
The basic concept of the algorithm is to start from a non-optimised spanning tree, and then continually improve it until it becomes a minimum spanning tree.

Selection of the tree edges at the various iterations of the algorithm are carried out using a pseudo-random number generator, giving candidate edges an even (or fair) chance of inclusion at each iteration.

Fundamental to the operation of this algorithm is the underlying minimum spanning tree verification method [111].
The verification method is not only able to ascertain whether a spanning tree of a graph is minimum or not in linear time, but is also able to identify, at no additional cost in time complexity, which of the spanning tree edges can be ruled out. Hence the O(e) efficiency of the algorithm.

The randomised algorithm, at each iteration, uses this additional information to substitute the ruled out edges with other candidates.
The new spanning tree is then passed on to the verification method for reassessment, and so on.

The algorithm's probabilistic analysis has shown it to have a linear time performance, O(e), of exponential likelihood.

6.5. *Planar graph algorithms*

One of the properties of any connected planar graph, G, is that it is possible to construct another planar graph, G*, so that each face of G corresponds to a vertex in G*.

Each pair of faces in G having an edge in common, have a corresponding edge in G* crossing it. G* is called a dual graph of G, Figure 30.

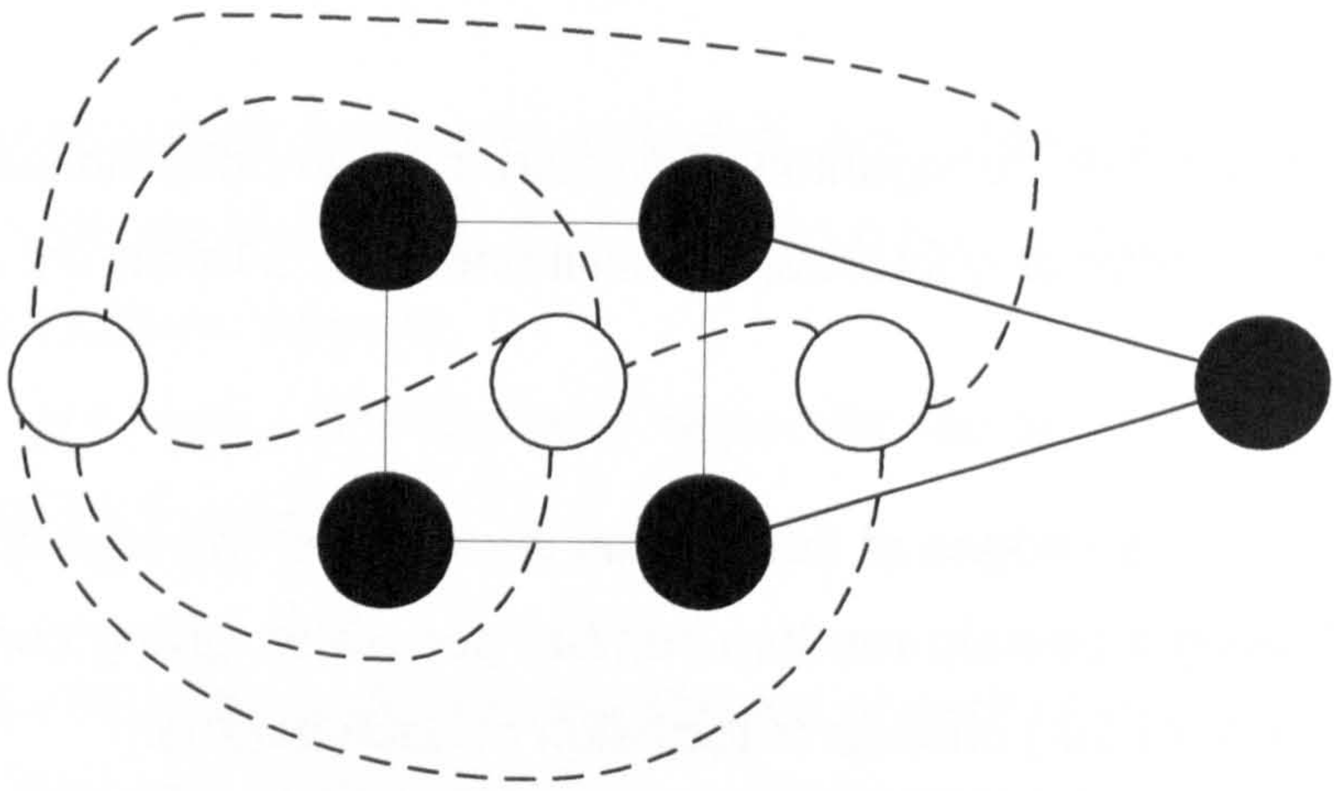


Figure 30 A graph G (solid edges and vertices) and it dual G*

The leftmost vertex in Figure 30 is a special vertex of G*, and corresponds to the outer face of G. A graph's outer face is also known as the unbounded or infinite face.

The construction of dual graph is relatively straightforward; if the graph G is a weighted graph, then its dual graph G* is also a weighted graph. The weights assigned to each edge of the constructed G* is equal to the weight of the edge it crosses in G.

A cycle-set of a graph is a set, or a group, of vertices which are connected by edges in such a way it is possible to traverse from any one vertex in the set to any other vertex, and then back again to the starting vertex in the set.

A cut-set of a graph is a set, or a group, of vertices which if where to be removed from the graph they would make the graph disconnected, i.e. the graph would have left in it one or more vertices which are no longer connected to any other vertex.

Dual graphs have interesting properties, stemming from the fact that each cycle-set of G's edges is also a cut-set of G*, and vice versa.

Matsui's algorithm [113], in particular, takes advantage of this property and relies upon it to construct an efficient minimum spanning tree algorithm.

6.5.1. Matsui [113]

Matsui [113], O(e + n), notes that a maximum edge of a vertex in a dual graph, G* can be ruled out of the graph's, G, minimum spanning tree. And, conversely, a minimum edge of a vertex in G can be ruled out of G*'s maximum spanning tree.

To take advantage of these facts, the algorithm begins by constructing G*, a fairly straightforward operation, in O(e).
It then relies upon another property of planar graphs, underpinned by Euler's formula[11], observing that at any one stage of the algorithm there is guaranteed to be at least one vertex in either G or G* which has less than four edges.
The algorithm maintains a record of such vertices over the course of its execution.

At each iteration, the algorithm processes a less than four-edge vertex, belonging to either G or G*. If it is a G vertex, the algorithm finds its minimum edge and rules it in the minimum spanning tree of G, and deletes (discards, or rules out) the corresponding edge from G*.
Otherwise (i.e. it is a G* vertex), the algorithm finds the vertex's maximum edge and rules it in the maximum spanning tree of G*, and deletes the corresponding edge from G.

---

[11] Euler's formula is perhaps one of the best known in graph theory. It is used here to establish that the mean number of edges for G and G* vertices is less than four.

A key factor to the algorithm's time linearity is the fact that a vertex which has less than four edges can always be identified in constant time. And the fact that such a vertex has, by definition, a limited number of edges (three or less) means that edge-weight comparisons can be conducted in constant time as well.

Therefore the algorithm is guaranteed to perform in linear time, O(e + n).

6.5.2. Cheriton and Tarjan [112]

Although his algorithm is occasionally cited in the literature during the discussions of planar graph algorithms, it is in essence a general case algorithm with a worst-case performance of O(e log log n).

The algorithm is shown to be O(e) for dense graphs. It is also shown to have an O(n) worst-case performance for planar graphs, although this is more of an artefact than a feature by design[12].

The algorithm, in its various suggested implementations, achieves all of the above by employing a specialised priority queue.

The priority queue employed is a data structure, which has been specifically designed to optimise the performance of operations required during the execution of this particular minimum spanning tree algorithm.

The performance of the priority queue operations, in fact, dominates the running time of the algorithm and gives it its various worst-case bounds.

Aside from the priority queue and its design, the algorithm can be viewed as a serialised implementation of the simultaneous edge contraction approach adopted by Boruvka [91, 92] (described above).

---

[12] Indeed, the property of linear time performance for planer graphs is acknowledged in the academic paper [112] to have been an observation made by another worker in the field during discussions of the work prior to its publication.

Boruvka's (blue) edge contraction can be viewed as simultaneously building minimum spanning tree forests.
As Boruvka's algorithm progresses, the forests inevitably grow into each other, forming lager and large frosts, until they become a single forest, which is in fact the minimum spanning tree.

Cheriton and Tarjan's algorithm also grows minimum spanning tree forests, although it does so more selectively and in a serial manner.

The algorithm's priority queue, in effect, selects the edges along which the minimum spanning tree forests are grown.
The algorithm also defers the deletion of edges causing cycles within forests for later processing.
This strategy is shown to reduce the computational cost of edge deletion.

The evident complexity of the algorithm is understandable, given the fact it primarily targets graphs in the general case, as opposed to specifically targeting planar graphs.

6.6. Discussion and chapter conclusion

We discussed in this chapter various algorithms for solving the minimum spanning tree problem.
We also discussed the properties of planar graphs, and how they can lead to the implementation of more efficient minimum spanning tree algorithms.

Matsui's [113] minimum spanning tree algorithm has linear time performance, and is relatively simple to implement, as it is specifically targeted at planar graphs.

For these reasons, it is our view that Matsui's algorithm is the best recommended algorithm for finding minimum spanning trees in the tile-based phase unwrapping method.

It is not possible to improve on the state of the art linear time complexity of the planar algorithms; such a possibility can only exist if some prior knowledge of the minimum spanning tree solution itself is available to the algorithm.
However, it may be possible to obtain further computational efficiency saving by utilising prior knowledge of the graph's specific topology.

The overall efficiency in this case would not be better than linear in terms of complexity, but rather, better in absolute terms than the performance that would be obtained from an algorithm which has no such prior knowledge to use to its advantage.

This very possibility of achieving improved computational efficiency is in fact the focal point of our research.
In the next chapter, we use graph theory principles to construct a novel algorithm which utilises the prior knowledge of a tile's graph to efficiently reduce the size of the minimum spanning tree problem.

Chapter 7 Reducing the minimum spanning tree problem

7.1. Introduction

Theoretically, it is only possible to improve on O(n) efficiency by having prior knowledge of the minimum spanning tree (MST) itself.
This is the case for MST maintenance algorithms which can carry out a single update, such as a single weight change, in O($\sqrt{e}$) [114] for a general-case graph, and in O(log n) for planar graphs [115, 116].

Although prior knowledge of the topology of a graph may not improve on O(n), it can help to improve the overall efficiency of the algorithm in empirical (or absolute) terms.
This could be by reducing the total number of times the edges and vertices are handled, or minimising the total number of edge weight comparisons necessary.

In this chapter, we describe a novel linear time O(n) algorithm which takes advantage of the prior knowledge of a tile's graph topology to reduce its size from G(n, e) to somewhere between zero, at best, and G[n/2, e/2] at the very most.
This has the effect of reducing the size of the problem needed to be solved by the underlying MST algorithm of choice.

Image processing algorithms often employ a sliding window technique to apply their various operations to the image in question. Similarly, we show how our algorithm can be applied to a tile using a sliding window.

For clarity, we start by describing the four steps of the algorithm separately. The discrete description is used subsequently for time-performance analysis, and for describing the sliding window operations.

7.2. Algorithm fundamentals and performance analysis

7.2.1. Initial state

The tile, Figure 31, can have any number of rows and columns of vertices. Each corner vertex has two edges, the minimum of which can be immediately ruled in the MST (if not, the resultant tree is either disconnected or is not minimum).

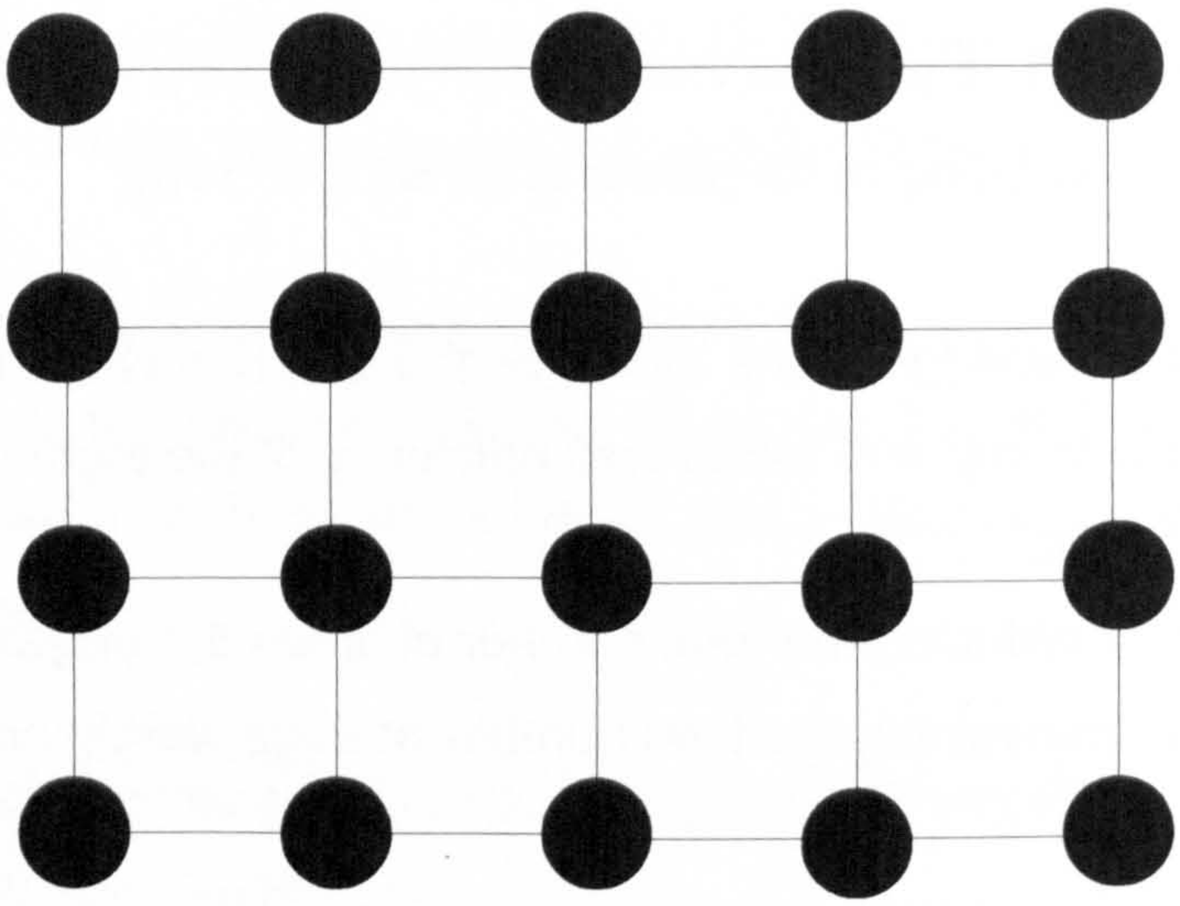


Figure 31 A tile may have any number of row and columns of vertices

Processing the four corners in this way can be carried out in constant time O(1), regardless of the number of vertices, n.
Figure 32 illustrates the representation of a tile's graph G, and its dual G*, once the corners have been processed.

The dual graph G* need not be constructed in practice, and is only shown here to help clarify the underlying interactions of the various steps of the algorithm.

We observe that at this initial state, due to prior knowledge of the topology, the graph fulfils the following criteria:

a Both G[n, e] and G*[n*, e*] are connected and naturally planar

b G and G* have n vertices each

c e = e*, n ≈ n* and the size of e approaches the size of 2n.

d Neither G or G* have two vertices connected to each other by more than one edge (this is secured by pre-processing the corners, which removes such edges from G*)

e Each vertex[13] in G and G*, has a maximum of 4 edges.

f Any G vertex can be picked from this initial state of G and processed independently, and its minimum edge can be ruled in the MST.

g Any G* vertex can be picked from this initial state of G* and processed independently. Its maximum edge can be ruled out of the MST.

h Any edge of G and G* can be processed in the same independent way as described in 1.1.1 or 1.1.1, unless one or more of its edges has already been ruled in or out by processing an adjacent vertex in the same graph.

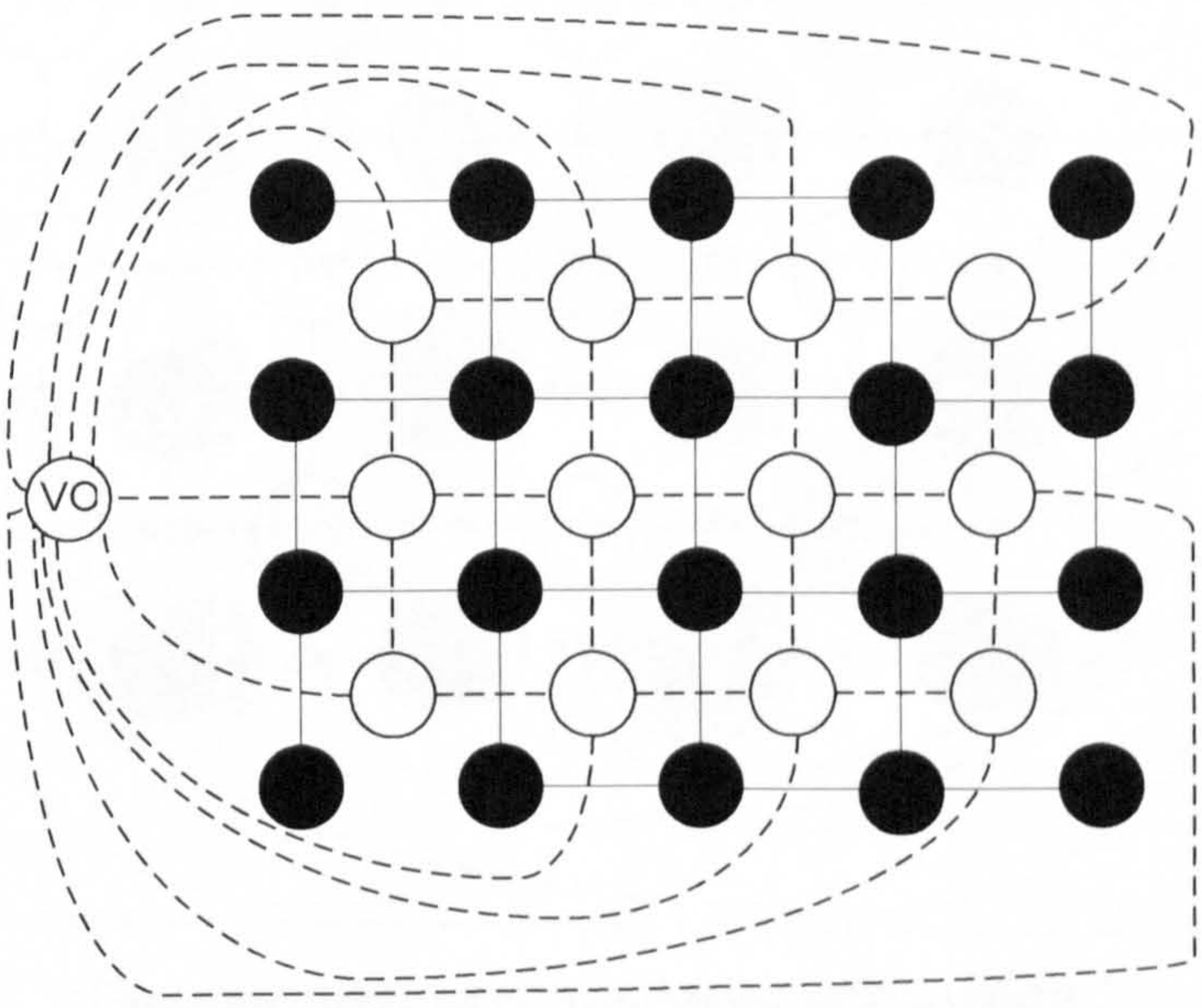


Figure 32 A tile's graph G (solid edges and vertices) and it dual G*. VO is a G* vertex corresponding to the outer face of G[14].

---

[13] As G*'s most outer vertex (VO) does not satisfy this criterion, it will not be relied upon for any of the steps of the algorithm. Although the edges of VO will still be processed by the algorithm, this will be done in the context of the other vertices in G* which are connected to VO. Therefore, all references to G* vertices from this point on shall implicitly exclude VO.

[14] A graph's outer face is also known as the unbounded or infinite face.

7.2.2. Step 1

Process G vertices alternately, ruling in one edge per vertex:

The alternate fashion in which the G vertices are targeted, Figure 33, ensures that they can be processed independently (criteria 1.1.1 and 1.1.1). In effect, the G vertices which are not being targeted act as an isolating barrier.

Each vertex has a maximum of four edges, therefore the minimum edge of each of these vertices can be ruled in the MST in O(1).

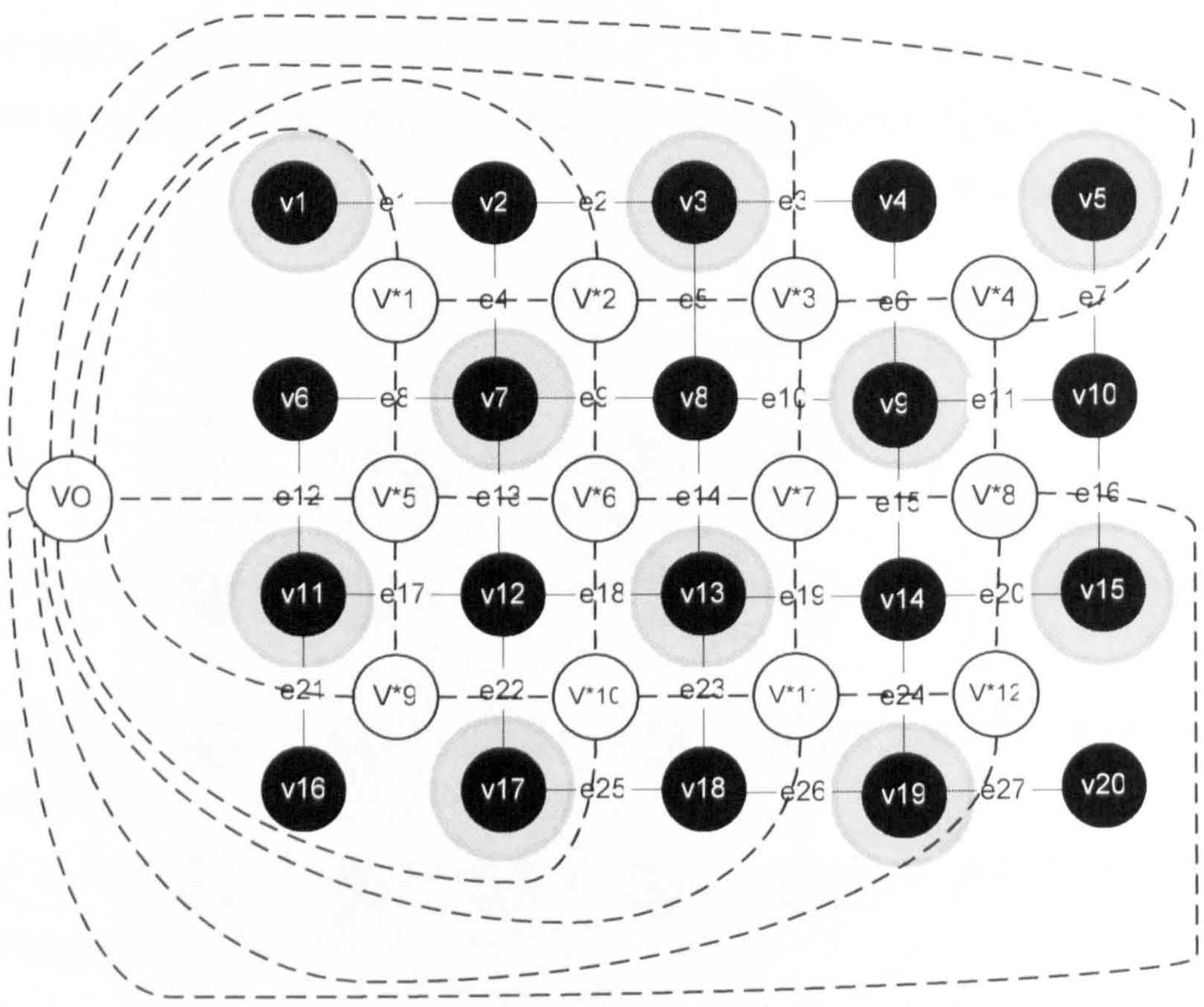


Figure 33 G vertices targeted by step 1

7.2.3. Step 2

Process G* vertices alternately, ruling out one edge per vertex:

Each targeted G* vertex, Figure 34, has a maximum of four edges, therefore the maximum edge of each of these vertices can be found in O(1).

This maximum edge of G* crosses an edge in G, which in turn can be ruled out from the MST.

In a similar way to step 1, G* vertices which are targeted can be processed independently.

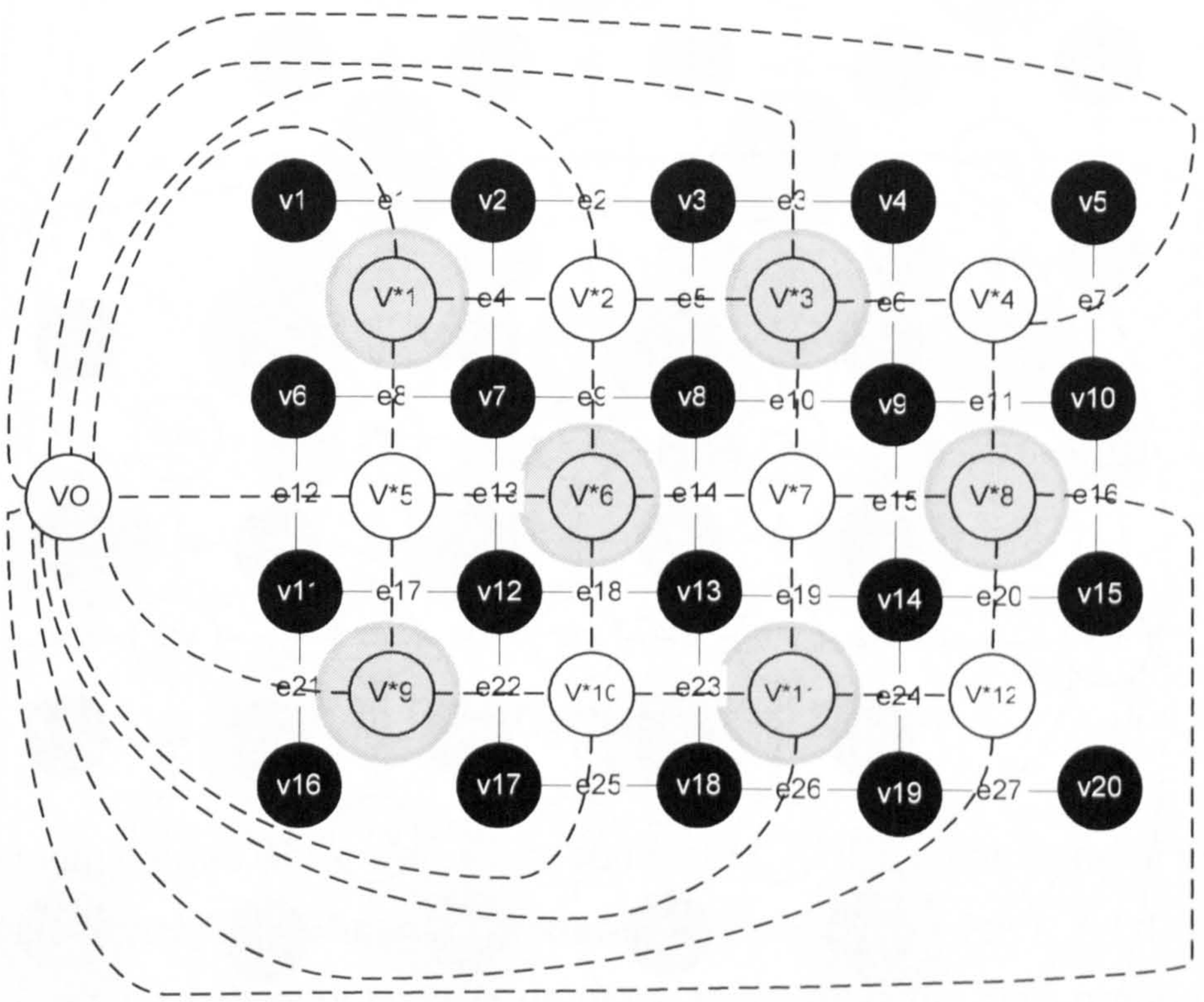


Figure 34 G* vertices targeted by step 2

7.2.4. Step 3

Ensure each remaining G vertex is connected to at least one ruled-in edge:

Each of the remaining G vertices, Figure 35, is examined in turn. If it is has an edge which has been already ruled in the MST, then no further processing is required. Otherwise, it must still have one or more edges which have not been processed, because step 2 could not possibly have deleted all of its edges.

This also means that such a vertex can still be processed independently (criteria 1.1.1 and 1.1.1), and therefore we can rule in its minimum edge in O(1).

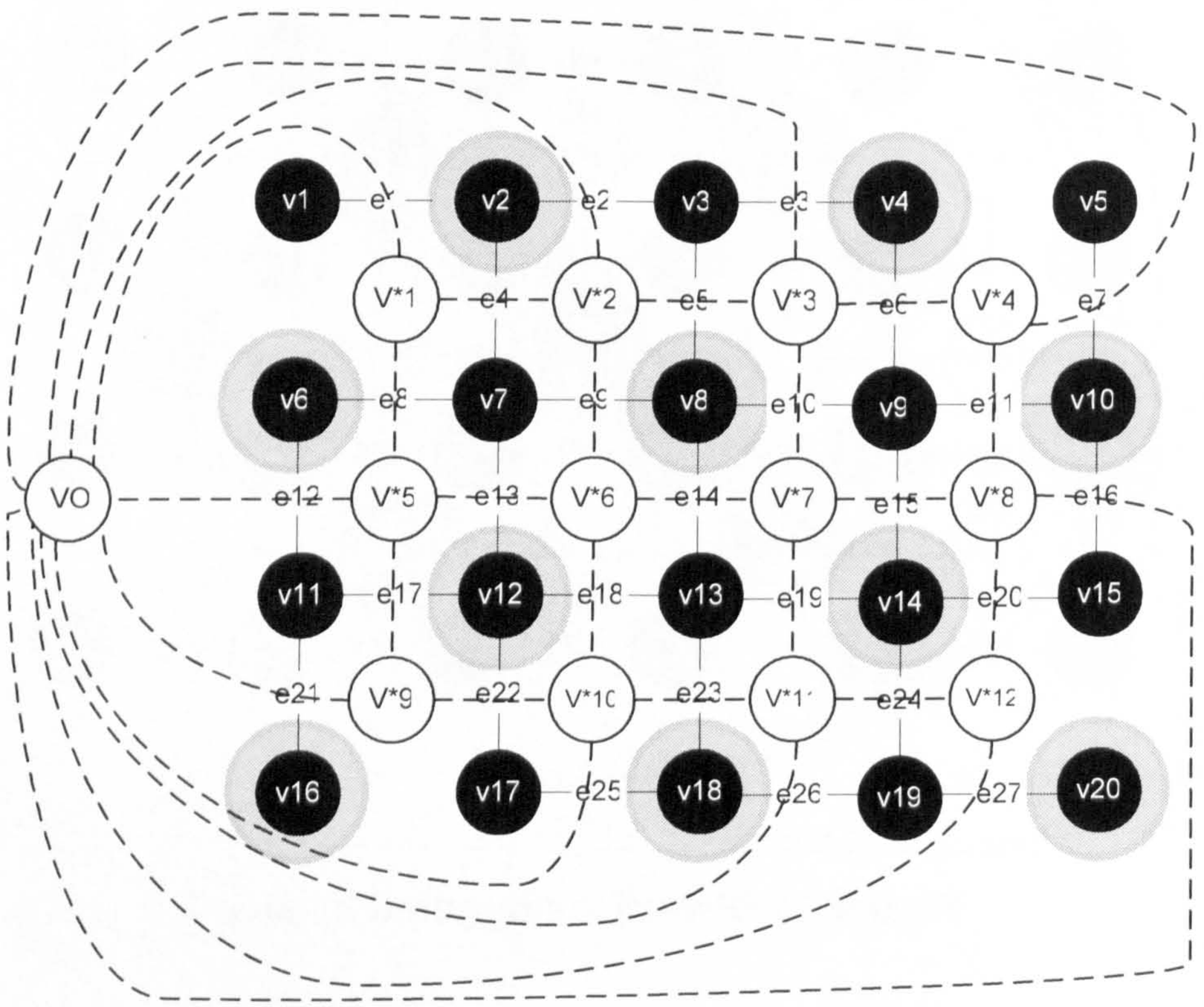


Figure 35 G vertices targeted by step 3

7.2.5. Step 4

Ensure each remaining G* vertex is connected to at least one ruled-out edge:

The remaining G* vertices, Figure 36, are targeted by this step.

It can be shown, using argument similar to those of step 3, that each of the targeted vertices in G* either has a ruled out edge (due to step 2), or, otherwise, its maximum edge can be found in O(1). This maximum edge of G* crosses an edge in G, which in turn can be ruled out from the MST.

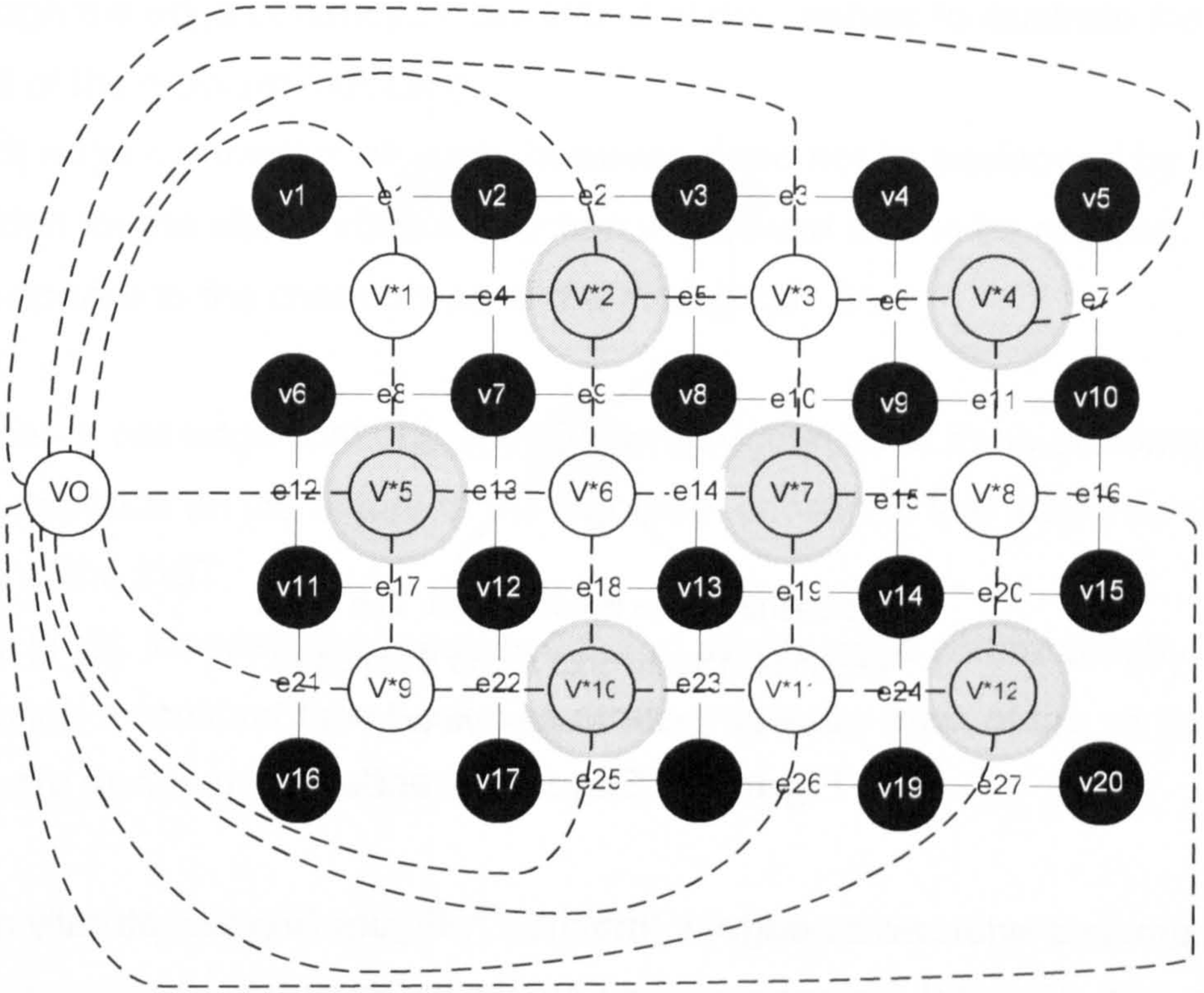


Figure 36 G* vertices targeted by step 4

The algorithm ends when step 4 has completed. At this point, none of the vertices in either G or G* can automatically satisfy criteria 1.1.1, 1.1.1 or 1.1.1 by relying on a prior knowledge of the topology of the tile's graph in its initial state before being processed by the reduction algorithm.

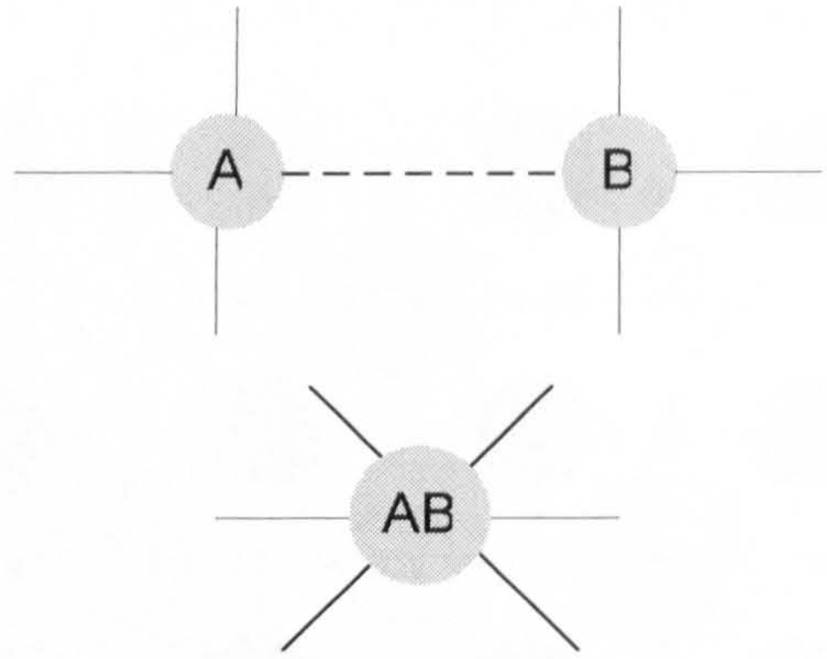
7.3. Problem reduction analysis

The consequence of steps 1 and 2 is that n/2 edges are ruled in the MST and n/2 edges are ruled out.

This means that the original G[n, e] graph is now reduced to G[n/2, e/2].

The ruled out edges can be clearly dismissed from the graph. Each ruled in edge, on the other hand, serves to fuse the two vertices it connects into one vertex.

The reaming (i.e. neither ruled in or out) edges of the original two vertices now become incident on the fused vertex.



The dashed edge is contracted. The union of edges incident onto A and B becomes incident on the merged vertex, AB.

Figure 37 Edge contraction

This process is known as edge contraction[16], Figure 37,and naturally results in the reduction of the number of the vertices in the graph.

If the algorithm were only to perform steps 1 and 2, it would still be guaranteed to reduce the original problem by half. This gives the algorithm the worst case reduction bound.

Steps 3 and 4 of the algorithm may improve on the above reduction, but are not guaranteed to do so; it is possible that by some chance every non-targeted vertex has one of its edges contracted or removed by an adjacent targeted vertex. In this case steps 3 and 4 would not perform any further edge contraction or deletion.

---

[16] Edge contraction was first discussed in Chapter 1 in section 6.2.3, on page 68.

At the other extreme, steps 3 and 4 could find n/2 edges to contract, therefore completely solving the MST problem. This is because an MST has exactly n-1 edges to be identified (i.e. ruled in or contracted).

7.4. Edge contraction and dual graph construction

Although the edge contraction, described above, serves to illustrate the extent of the problem reduction.
Explicit edge contraction as such, however, need not be performed by the algorithm for the above stated reduction in problem size to be realised. The same applies to the construction of the dual graph G*.

Whether or not edge contraction or G* construction need to be performed, solely depends on the nature of the algorithm chosen to find the remaining edges of the MST.
Matsui [113], for example, requires G*, and avoids edge contraction by employing a constant time bucketing strategy to keep track of the vertices which are less than four-connected.

By contrast, algorithms such as [106, 110] do depend on edge contraction, but do not employ a dual graph as they were not specifically aimed at planar graphs.
Such distinction is less obvious in the cases of some algorithms [91, 92, 105] which although make use of edge contraction, their respective asymptotic complexity does not rely on such use.

7.5. Time performance

Steps 1 and 2 visit n/2 G vertices, and carry out each operation in O(1) constant time.
Similarly, steps 3 and 4 visit n/2 G vertices, carrying out each operation in O(1).
Therefore, the algorithm performs all of its operations in O(n) linear time.

7.6. A windowing approach

We show here how the various steps of the algorithm are combined and applied to a tile via a sliding window.

In addition, we show how the algorithm can be performed without the construction of a dual graph G*, and how edges to be ruled out can be directly processed in the context of G.

The algorithm is carried out in two phases, processing n/2 vertices in each phase.

7.6.1. Phase 1 – Combining steps 1 and 2

Over all, phase 1 visits n/2 vertices. Figure 38 shows how each of the vertices and edges targeted by steps 1 and 2 can be processed using a sliding window.

When the window is in position “a” it performs step 1 on vertex a, ruling in the minimum of a1, a2, a3 and a4. It then performs step 2, ruling out the maximum of a2, a3, a5 and a6.

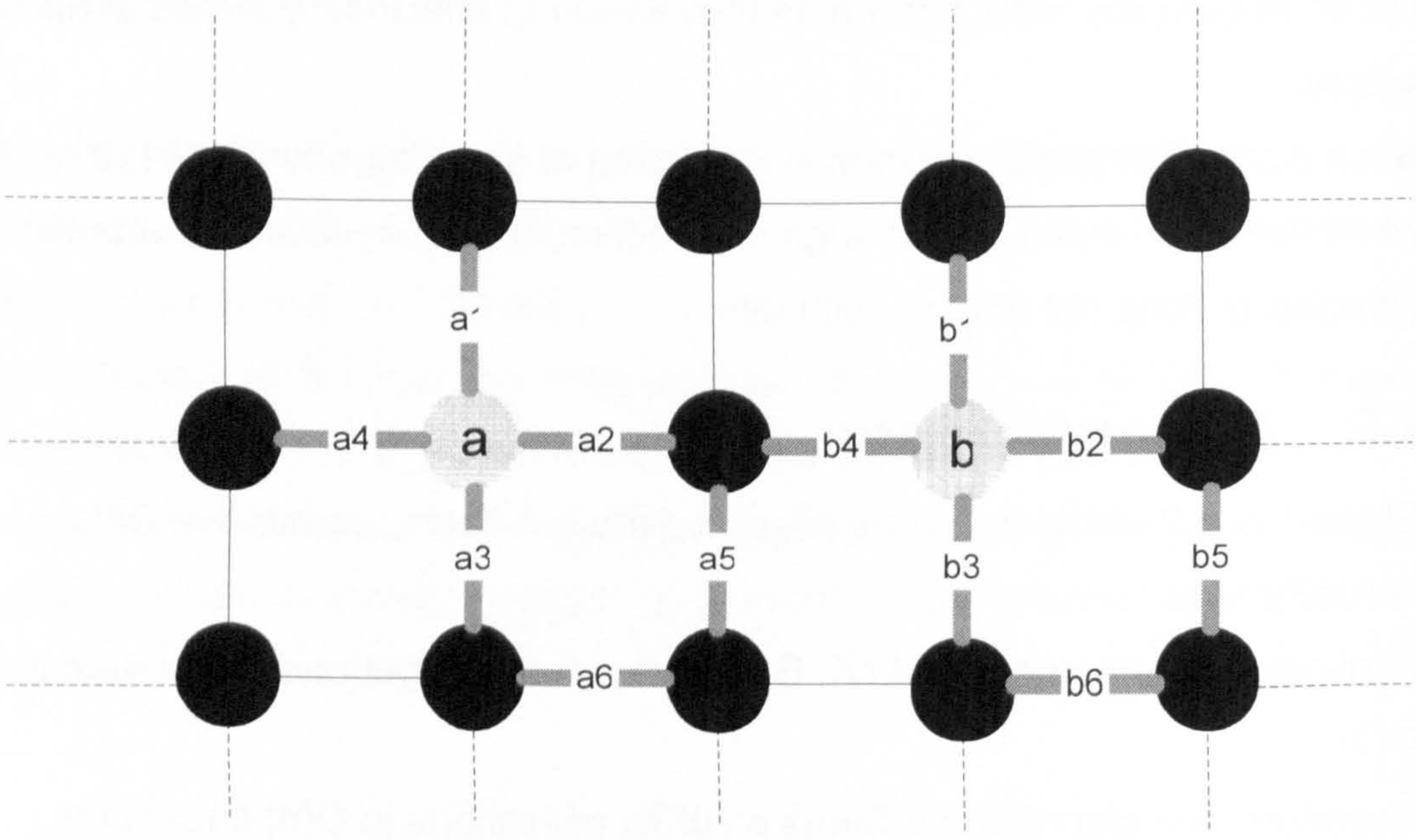


Figure 38 The sliding window visiting position “a” then sliding to position “b”.

The window is then moved to position “b” and the same process is similarly repeated, and so on.

Operations of Steps 1 and 2 can in fact be carried out in either order.
Each sliding window operation will process one vertex and a maximum of 6 edges, and is guaranteed to rule in one edge and rule out another.

7.6.2. Phase 2 – Combining steps 3, and 4

Phase 2 visits the remaining n/2 vertices using the same window shape, Figure 38.

The operations of steps 3 and 4 can be carried out, in either order, over the targeted six edges.
During this phase there are no guarantees as to the additional number of edges to be ruled in or out, as discussed in a pervious section.

7.7. Discussion and chapter conclusion

In this chapter, we described a novel algorithm which reduces the size of the minimum spanning tree problem for a tile’s graph G(n, e) to somewhere between zero, at best, and G[n/2, e/2] at the very most.

We have shown through further analysis that the algorithm performs in linear time O(n).
This is achieved by taking advantage of the prior knowledge of the graph’s topology, which is specifically used for tile-based phase unwrapping.

The algorithm’s approach is to preserve this known topology of the graph’s initial state, while processing as many of its edges as possible.

During this processing, some edges are ruled-out of, and others are ruled-into, the minimum spanning tree.

The topology of the graph allows for such decisions to be taken based on information obtained from a relatively small neighbourhood of vertices in the graph.

Processing edges, by its nature, alters the topology of the graph, and eventually there comes a point where the graph's initial topology can no longer be relied upon to make any further decisions based on localised information.
The algorithm's careful selection of the targeted vertices allows for the ability to make localised decisions to be sustained for as long as possible.

A (noisy) wrapped phase map is usually pre-processed before undergoing phase unwrapping.
Such pre-processing usually involves applying various image processing smoothing functions, such as low-pass and median filters.

Image processing filters are typically applied using sliding window techniques, and many image processing software and hardware components are specifically optimised for such applications.
Therefore, we have also shown how our algorithm can be applied using a sliding window in practice.
This is envisaged to assist in implementing the algorithm as an additional step at the wrapped phase map's per-processing stage.

Although our algorithm may in some circumstances completely solve the minimum spanning tree problem (i.e. reduce its size to zero), it is not guaranteed to do so.
The remainder of the problem can be naturally solved using any minimum spanning tree algorithm of choice.

In the next chapter we discuss how our algorithm can be combined with any minimum spanning tree algorithm, and how the resulting hybrid algorithm can be applied to obtain the complete solution.

Blank

In

Original

92

Chapter 8 Hybrid algorithms and practical implementation

8.1. Introduction

We have shown in the previous chapter how our algorithm can be used to reduce the size of the problem to be found by any minimum spanning tree (MST) algorithm, and how this can be applied to tile-based phase unwrapping.

In this chapter, we consider how a selection of minimum spanning tree algorithms, with no loss to generality, can be applied to find the remainder of the solution.

We examine a Prim-hybrid algorithm in particular, giving it a practical treatment, and show the results of its empirical analysis.

We then go on to describe a novel hybrid edge detection algorithm which we make use of for finding fringe locations more correctly in particularly noisy phase maps.

Finally, we combine both of the above hybrids with the tile based phase unwrapping method, and apply the resulting overall unwrapping algorithm to a selection of tiles containing up to four phase discontinuities (fringes).

8.2. Hybrid MST algorithms

8.2.1. Matsui [113]

Matsui is perhaps the best choice of algorithm for the problem; it is linear time efficient, relatively straightforward to implement, and is specifically targeted at planar graphs.

It is theoretically possible to implement Matsui's algorithm without constructing G*. On the other hand, G* provides for a more straightforward implementation.

Regardless of whether G* is constructed or not, the reduction algorithm needs to keep track of the number of edges incident on G and G* vertices, as Matsui relies on this information to pick the vertex to be processed next. This is an additional O(1) operation and does not impact the linearity of the reduction algorithm.

Both the reduction algorithm and Matsui's are linear time efficient, and the resulting hybrid is dominated by Matsui's O(e + n) efficiency.

8.2.2. Kruskal [95]

Kruskal normally performs worse than Prim [97], in O(e log e).
This is mainly due to its stipulation that all the edges need to be sorted at the start of the algorithm.
This means that even the edge which are eventually ruled out are also sorted, which results in efficiency loss.

The reduction algorithm decreases the number of edges in the graph from 2n to n or less.
This means that a Kruskal hybrid can be then applied in O(n log n), which is comparable to using Prim's algorithm without reduction.

8.2.3. Randomised Karger et al [110]

When Karger's algorithm is applied to the problem remaining after reduction, it randomly selects half of the remaining n (or less) edges (those neither ruled in or out) to connect the minimum spanning forests into a candidate minimum spanning tree.

The algorithm then uses a linear time O(e) verification algorithm [111] to rule some of these selected edges out of the MST.

The process is repeated iteratively until enough edges have been ruled out or a true MST has been identified.

The performance of Karger's algorithm is shown to be exponentially likely O(e), and thus the hybrid is equally exponentially likely O(n).

8.3. Practical implementation

We present here our practical implantations of:

- A hybrid MST algorithm combining our MST reduction algorithm with that of Prim's [97].
- A hybrid edge detection algorithm combining Judge's [62] two-threshold iterative approach with Yatagai's [87] employment of a Hilditch's [88] thinning algorithm.
- The tile based phase unwrapping algorithm, based on Judge's [62], showing how the tile level implementation performs for tiles containing up to four phase discontinuities.

8.3.1. MST reduction and Prim [97] hybrid

8.3.1.1. Introduction

Although Prim algorithm is not linear, it is certainly worth consideration as it is one of the simplest to understand and implement of the minimum spanning tree algorithms and is commonly available through most graph theory related computer programming libraries.

Significantly, Prim's is also the algorithm chosen for the tile-based method in its original implementation by Judge [62].
Therefore, we give Prim's algorithm an empirical treatment as well as theoretical.

8.3.1.2. Theoretical analysis

The reduction algorithm need not carry out any additional chores in addition to marking edges when it rules them in or out of the MST.

The ruled out edges help make Prim more efficient by reducing the number of edges it needs to search at each iteration to identify the next edge to connect the next vertex to the MST.

The edges already ruled in by the reduction algorithm effectively define minimum spanning forests (MSF) of vertices. Prim then adds one collection of vertices in a MSF in one of its iterations (as opposed to the usual single vertex), which improves its over all performance.
Prim can thus be used to connect the MSFs identified by the reduction algorithm, which amounts to the connected MST.

Prim's algorithm is not linear time efficient, therefore its time complexity dominates the overall performance of the hybrid algorithm.

8.3.1.3. Empirical results

For the empirical analysis of our reduction algorithm, we chose to use the Boost Graph Library (BGL) [124] for the following reasons:

- Its credibility. After the Standard Template Library which it extends, the Boost Library is perhaps the most respected in the C++ domain. The BGL in particular is very highly acclaimed.
- Its generality, wide adoption and applicability. The BGL is rapidly becoming the C++ graph library of choice for industry and academia alike.
- It supports an optimised version of Prim's algorithm with a well documented O(e log n) efficiency.
- Its data structures and algorithms have been designed, implemented, analysed and tested sufficiently rigorously for their respective time efficiency characteristics to be well understood and fully documented.

We implemented our reduction algorithm using the same components which the BGL uses for its native Prim's implementation.

Tile edge size	Vertex count	Time without reduction	Time with reduction	Time for reduction
64	4,096	31	16	0.016
128	16,384	47	16	0.016
256	65,536	156	110	0.094
512	262,144	843	453	0.438
1,024	1,048,576	3,547	2062	2.031
2,048	4,194,304	3,0781	2,0984	20.781
All times shown are in milliseconds. Data obtained using BoostPrim.cpp (page 286) and the Boost library version 1.31.0, compiled with Microsoft® Visual Studio .Net 2003, on a Mesh® Computers Plc personal computer: Intel® Pentium® 4 CPU 3.0 GHz, RAM 1 Gigabyte, Microsoft® XP Home Edition 2002 – service pack 2.				

Figure 39 Worst-case analysis of the enhanced time performance due to the reduction algorithm, in tabular format

Graphs with tile-topology were constructed and assigned random weights using a pseudo-random number generator. The range of edge weights used was 0 – 255, depicting the intensity range of an 8-bit digital camera.

The analysis results are shown in Figure 39 and Figure 40, and are discussed in the following section.

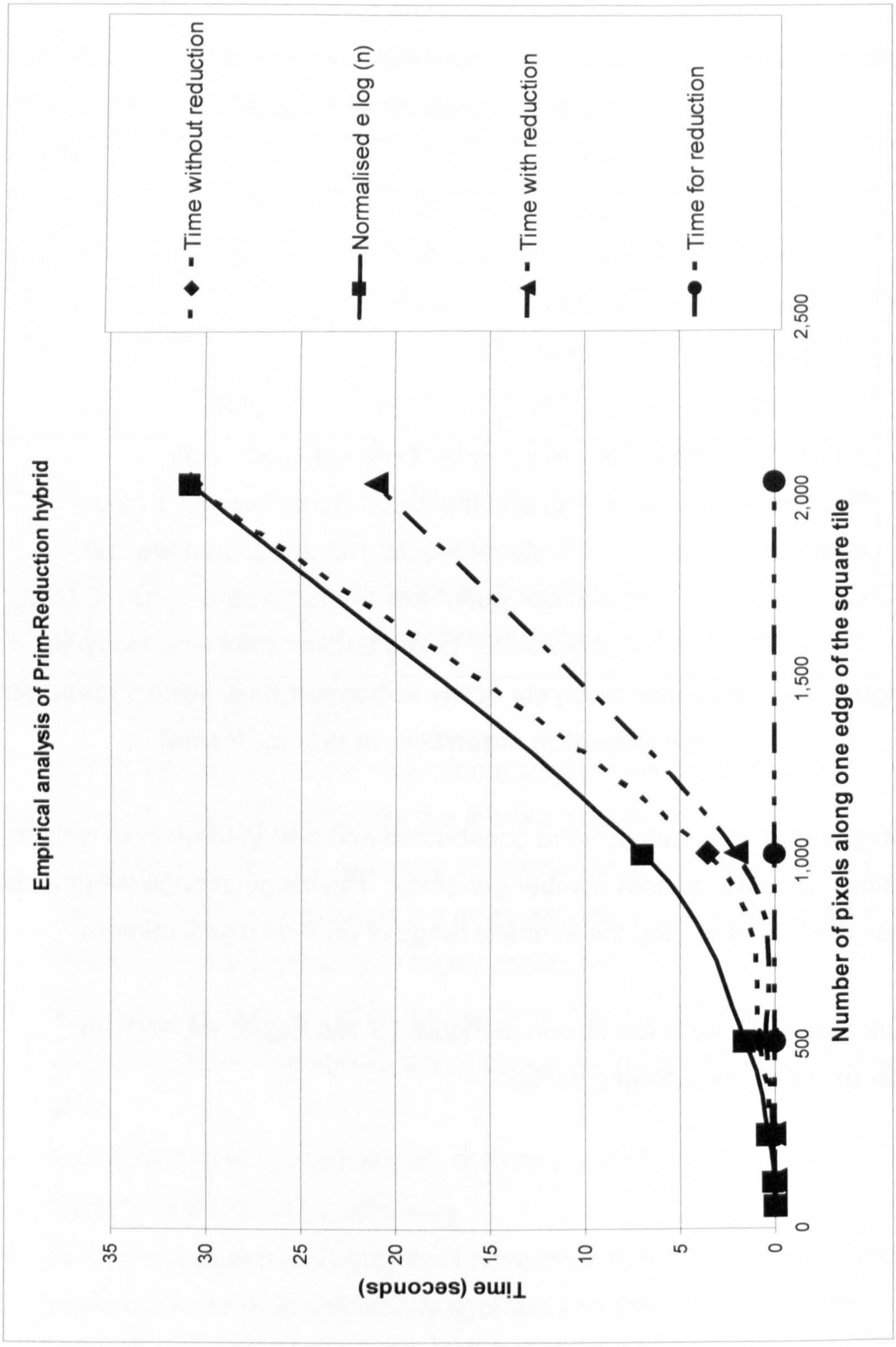


Figure 40 Worst-case analysis of the enhanced time performance due to the reduction algorithm, in chart format

8.3.1.4. Empirical worst-case analysis

Phase 1 of our reduction algorithm is guaranteed to reduce the problem's size by one half.

Further reduction by phase 2 is certainly possible, however it is not guaranteed, and its extent is likely to depend on the application in question.

Furthermore, the reduction algorithm is linear by nature. Hence, an estimate of the time performance of phase 2, for a given reduction percentage, can be reliably projected from the analysis results obtained for phase 1.

Therefore, we confined our empirical analysis to phase 1 of the reduction algorithm, and the results of which are shown in the previous section, Figure 39 and Figure 40.

Analysis of tile edge size of less than 64 did not produce meaningful time measurement: Most frequently, the time obtained was zero, indicating that the 1 millisecond resolution of the system clock does not provide sufficient resolution for such measurements.

At the other end of the spectrum, a tile edge size of 2048 (or 4,194,304 vertices) performed a less well than the general trend would have otherwise suggested. This is perhaps due to other factors coming into play, such as hard-disk cashing[17].

In general, the results obtained empirically confirm the theoretical performance analysis findings.

---

[17] It was observe that hard-disk activity become increasingly pronounced during the progress of this particular tile size.

It is also interesting to note that the reduction algorithm generally takes 0.1% of the total time taken by the overall hybrid, despite the fact that it reduces the problem's size by 50%.

Although the reduction algorithm halves the problem size, the overall time taken by the hybrid is more than half of that taken by the algorithm without reduction. This is may first appear to be contrary to the theoretical findings, however it is in fact due to the overhead of the computational processing required to make it possible for the output of the reduction algorithm to be seamlessly used by the third party (i.e. BGL) MST algorithm.

Hybrid algorithms by their nature often entail such overheads, although fortunately in this case the overhead does not detract from the significance of the overall efficiency enhancement achieved.

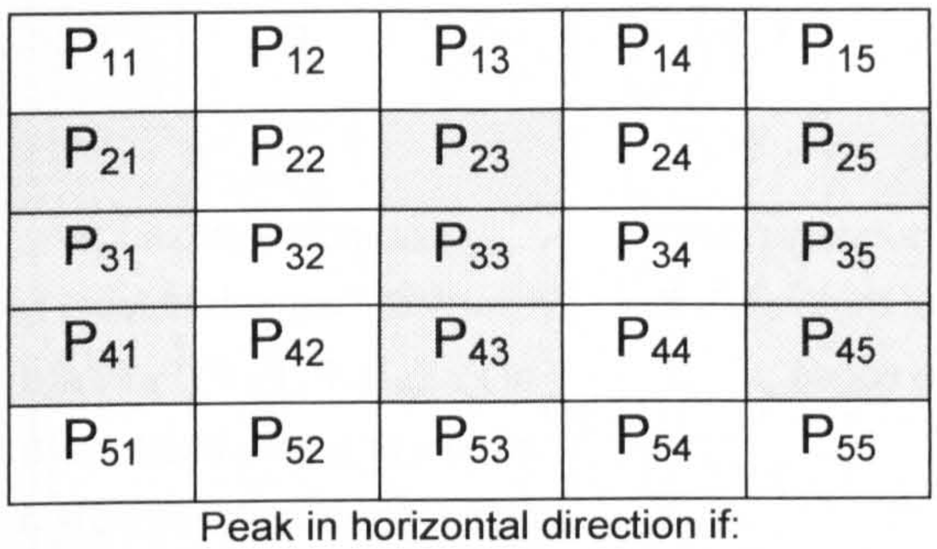
8.3.2. Judge's [62] edge detection with Hilditch's [88] thinning hybrid

It is interesting to note that Judge's iterative edged detection approach (described in section 4.7.1 on page 47) has similarities to that described by Yatagai [87].

Yatagai also uses an iterative approach, but rather than using a standard edge detection operator (such as the Sobel operator, Figure 20, used by Judge), Yatagai uses directional edge peak detection kernels, Figure 41.

Whereas Judge's iterative approach employs two thresholds to identify confirmed (high-threshold) and tentative (low-threshold) edges, Yatagai's iterative approach identifies a confirmed edge if a peak can be detected in two or more directions (Figure 41), and as a tentative edge if a peak can be detected in only one direction, at that location.

An interesting aspect of Yatagai's approach is the application of an edged thinning process, described by Hilditch [88], after the completion of each edge detection iteration.

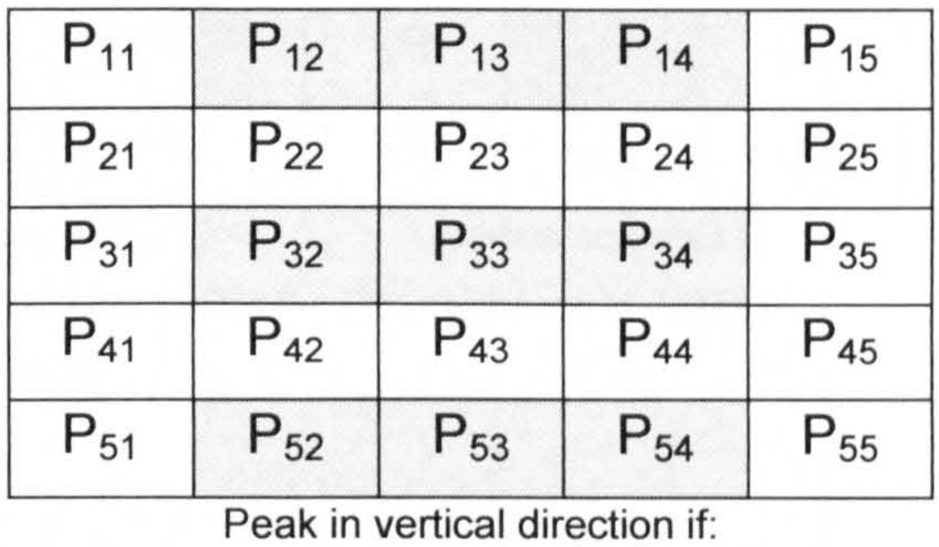


Peak in horizontal direction if:

left < centre and right < centre, where: centre = $P_{33} + P_{23} + P_{43}$

left = $P_{31} + P_{21} + P_{41}$

right = $P_{35} + P_{25} + P_{45}$

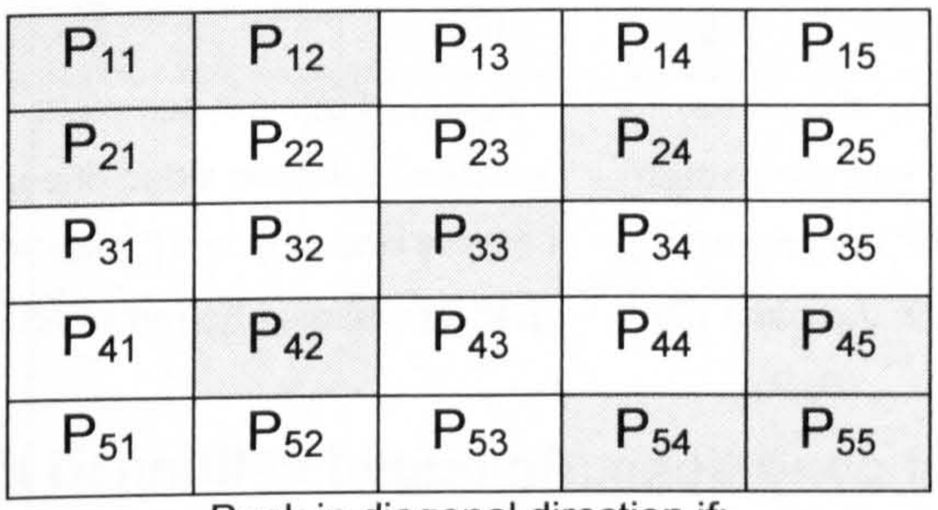


Peak in vertical direction if:

top < centre and bottom < centre, where: centre = $P_{32} + P_{33} + P_{34}$

top = $P_{12} + P_{13} + P_{14}$

bottom = $P_{52} + P_{53} + P_{54}$



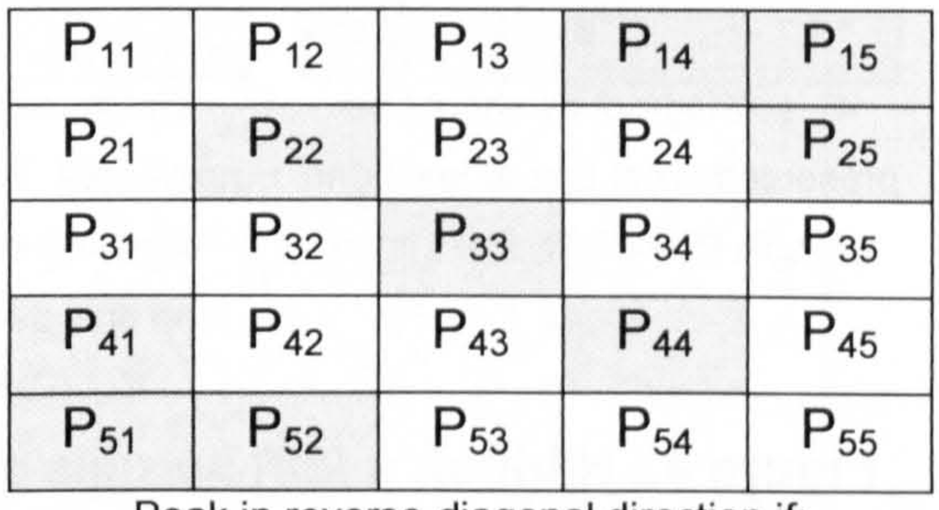
Peak in diagonal direction if:

top-left < centre and bottom-right < centre, where:

centre = $P_{24} + P_{33} + P_{42}$

top-left = $P_{12} + P_{12} + P_{21}$

bottom-right = $P_{45} + P_{55} + P_{54}$



Peak in reverse-diagonal direction if:

top-right < centre and bottom-left < centre, where:

centre = $P_{22} + P_{33} + P_{44}$

top-right = $P_{14} + P_{15} + P_{25}$

bottom-left = $P_{41} + P_{51} + P_{52}$

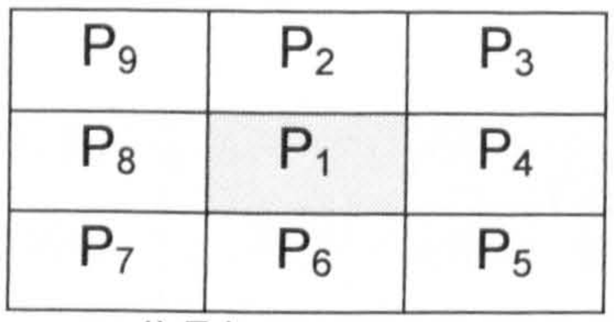
Figure 41 Peak detection kernels used by Yatagai [87].

Hilditch's thinning algorithm uses pattern matching kernels, Figure 42, to identify specific edge structures within the image which can be thinned.

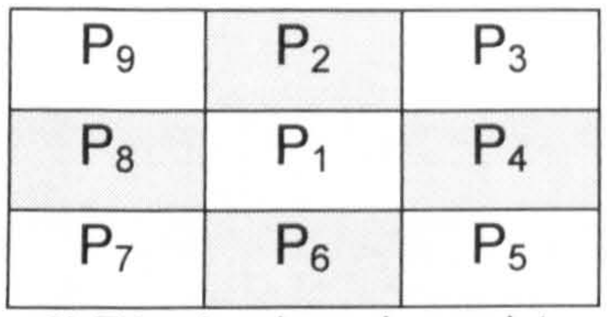
Yatagai's peak detection method does not give good results, Figure 43, when applied to particularly noisy images such as the wrapped phase maps shown in Figure 21.

Nevertheless, the concept of incorporating an edge thinning algorithm into the overall process was shown to be fundamentally an effective way of

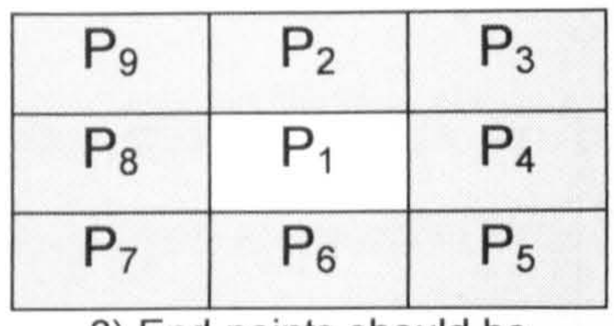
overcoming the ambiguity of fringe location, as compared to that obtained by edge detection alone.



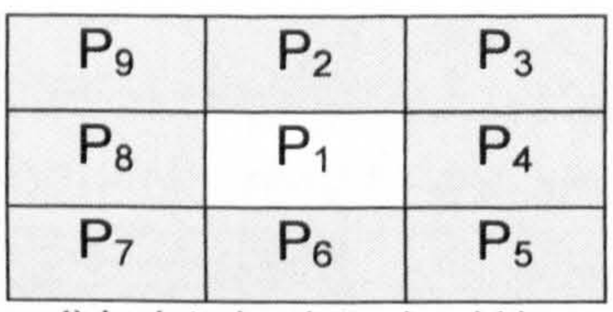
1) Edge presence: Test P_1 is an edge.



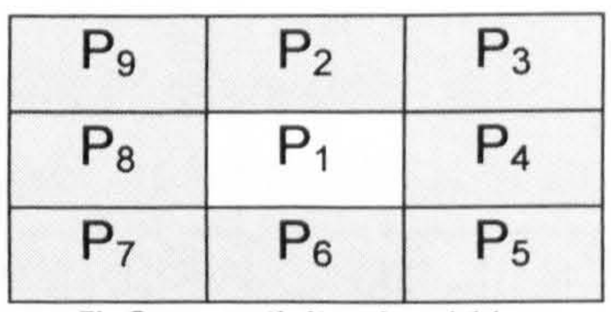
2) Edge is a boundary point: Test that at most three of P_2 P_4 P_6 P_8 are edges.



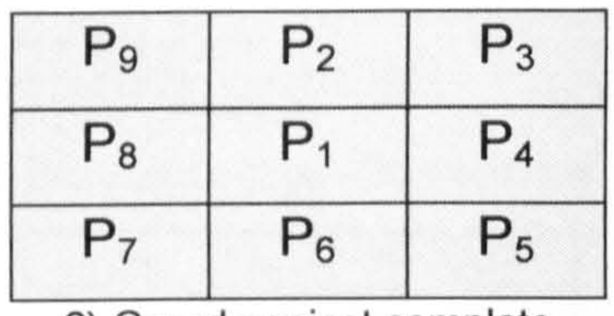
3) End-points should be preserved: Test that at least two of P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 are edges, or edges which were thinned by the algorithm in a previous iteration.



4) Isolated-points should be preserved: Test that at least one of P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 is an edge.



5) Connectivity should be preserved: Test that the pattern "edge, no-edge" occurs once and only once in the pairs (P_2 P_3), (P_3 P_4) … (P_9 P_2).



6) Guard against complete erosion of double sided lines. i.e. only one side in "||" and "=" patterns should be removed.

Figure 42 Hilditch's [88] kernels and conditions. An edged is thinned if it satisfies all six conditions.

To further investigate this potential, we combined Judge's iterative edge detection algorithm with the edge thinning method described by Hilditch.

We followed Yatagai's approach, by applying Hilditch's thinning algorithm after each edge detection iteration of Judge's (be it high or low threshold iteration).

This combined approach has yielded improved results, an example of which is shown in Figure 44.

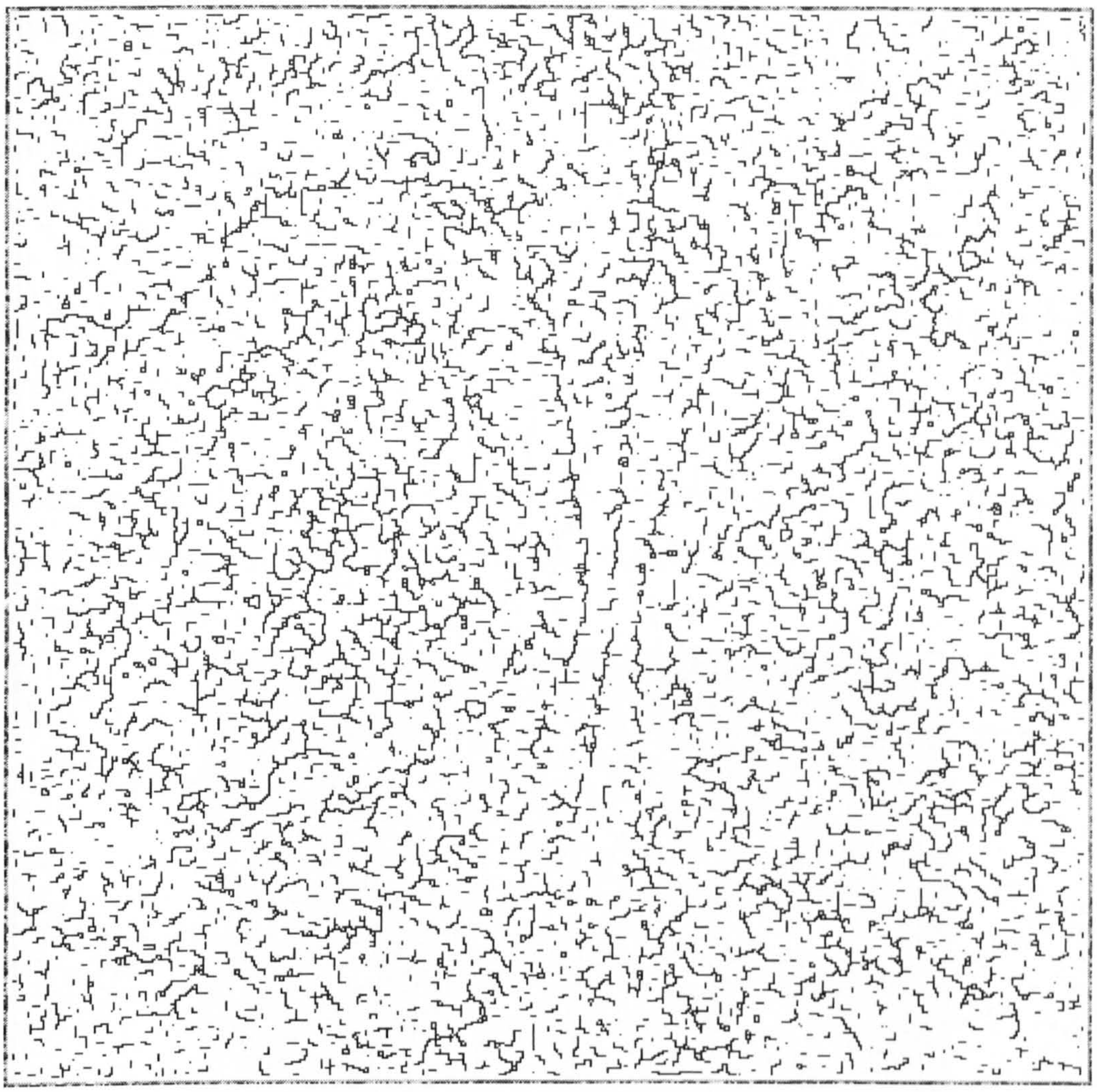


Figure 43 Yatagai's iterative peak detection and thinning approach does not work very well with noisy speckle phase maps

To further investigate this potential, we combined Judge's iterative edge detection algorithm with the edge thinning method described by Hilditch.

We followed Yatagai's approach, by applying Hilditch's thinning algorithm after each edge detection iteration of Judge's (be it high or low threshold iteration).

This combined approach has yielded improved results, an example of which is shown in Figure 44.

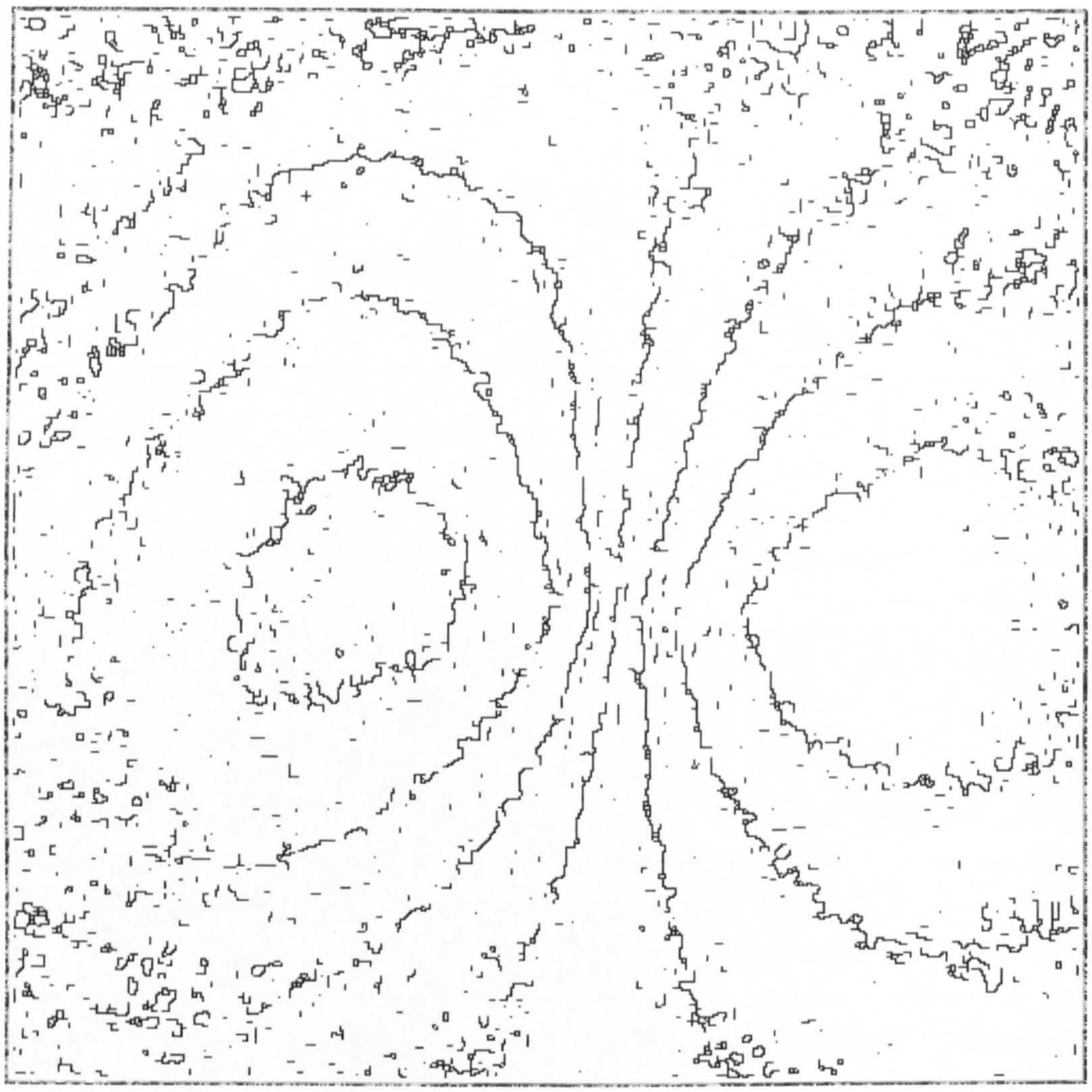


Figure 44 Combining iterative edge detection with edge thinning yields better identification of fringe locations

The edge detection process can be improved upon by further processing the edge information to remove localised and spurious short edges, especially those which appear in the inter-fringe regions.

Fringe interpolation techniques, such as those employed by Yatagai, would provide further improvements by overcoming the fringe-break problems.

8.3.3. Tile level unwrapping of up to four phase discontinuities

We show here the result of applying the various novel hybrids described above to a wrapped phase map obtained from a shearography system. Special thanks to Roger Groves for supplying the image [118].

The wrapped phase map, Figure 45, was obtained using a flat 150mm by 180mm aluminium plate, loaded at its centre by a micrometer to obtain a few μm out-of-plane displacement. 10mm of vertical shear was applied in the optical system.

Figure 46 shows the result of applying one pass of median filtering to the wrapped phase map.

A sub-region of the image is selected for processing. The fringe locations are obtained by applying the novel thinning algorithm described in section 8.3.2 page 100, Figure 48.

Figure 45 A shearography wrapped phase map from a mechanically loaded flat aluminium plate measuring 150mm by 180mm, with 10mm of vertical shear applied

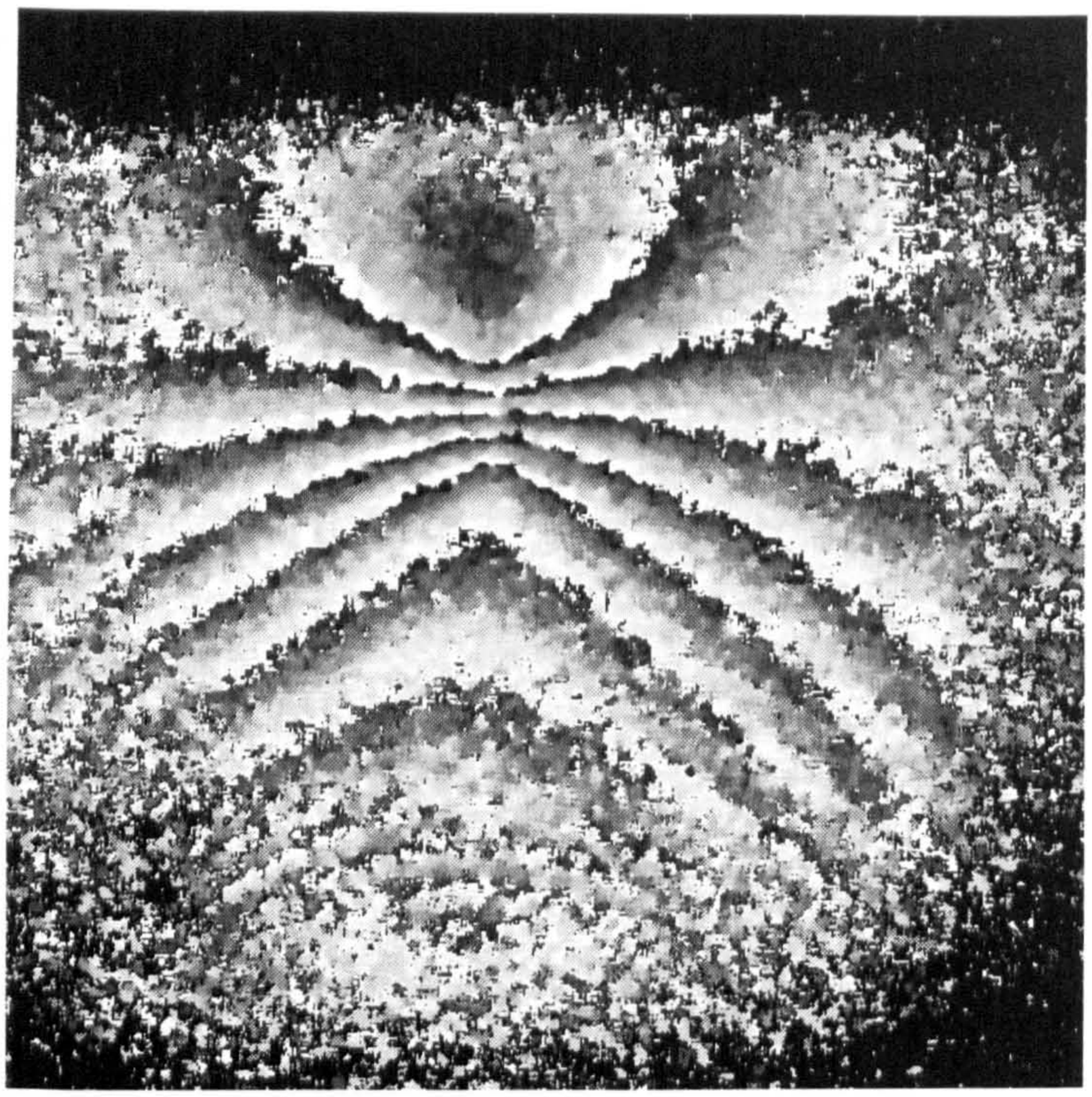


Figure 46 Filtered wrapped phase map obtained by applying one pass of median filtering to image shown in Figure 45

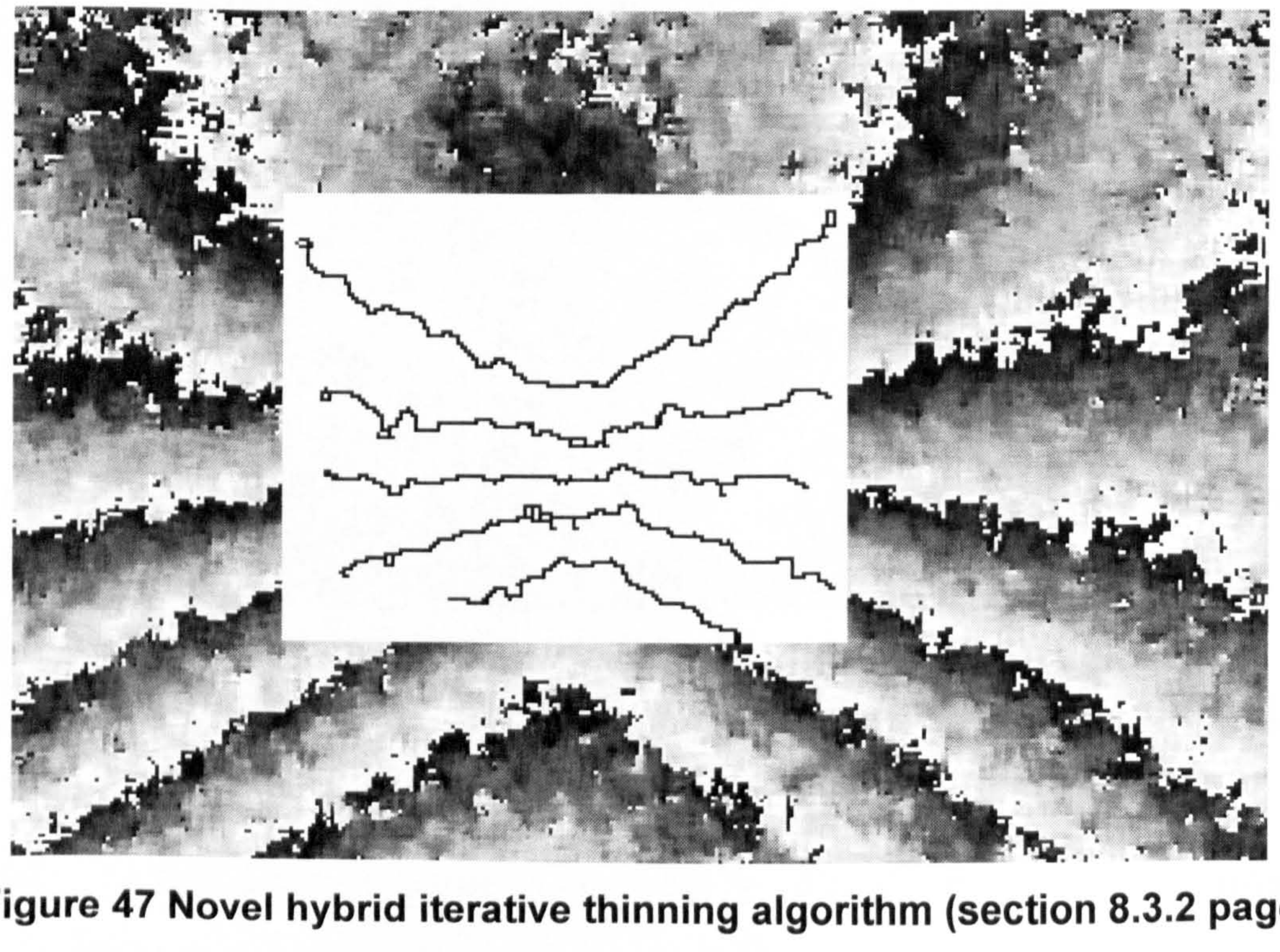


Figure 47 Novel hybrid iterative thinning algorithm (section 8.3.2 page 100) applied to a sub-region of the filtered wrapped phase map shown in Figure 46. Inter-fringe noise was eliminated by applying 10 iteration of morphological pruning (section 4.7.3 page 52)

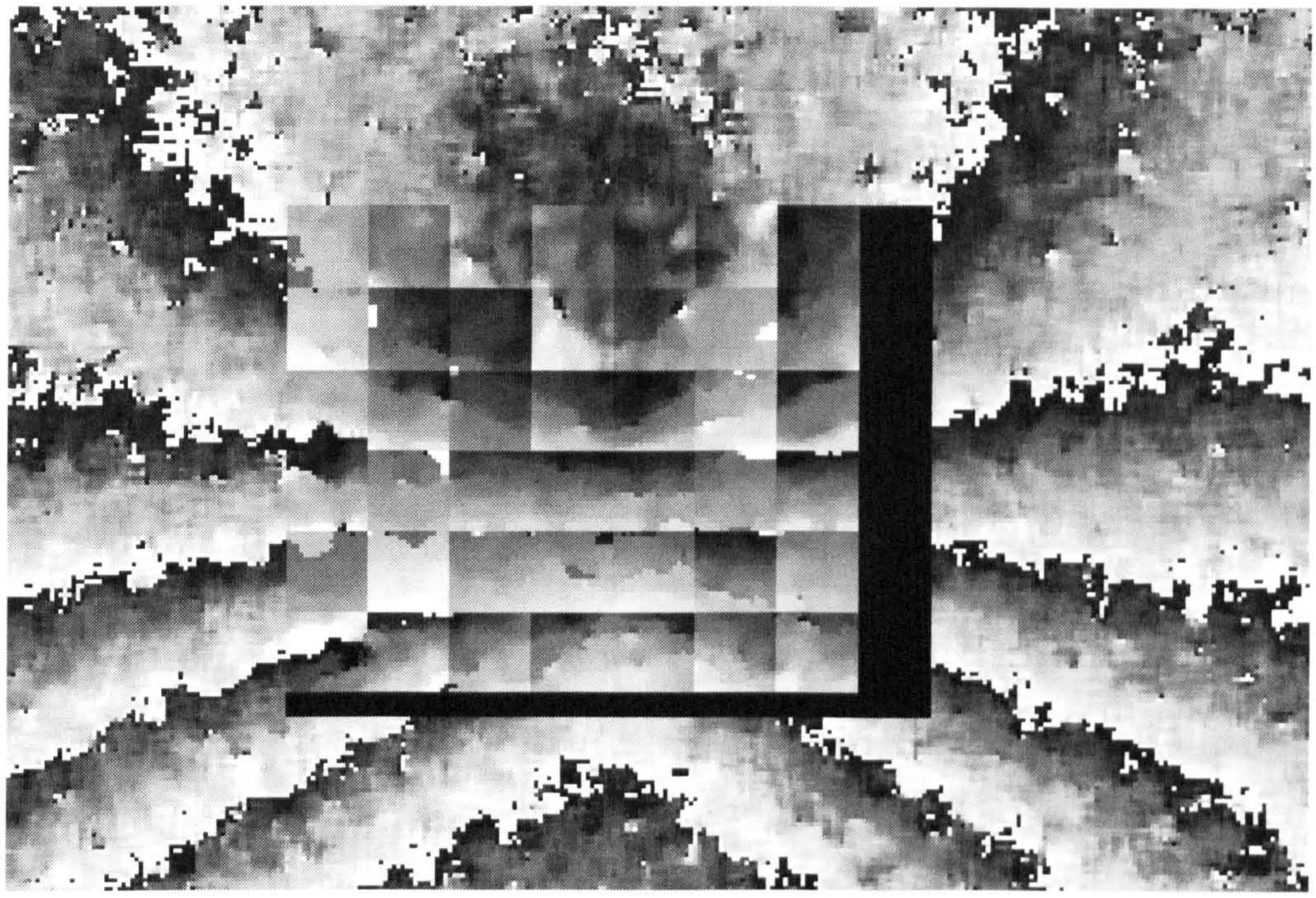


Figure 48 Tile level unwrapping, each tile containing one phase discontinuity (fringe) at most

The tiles were unwrapped individually and their respective contrasts stretched to make use of the full 8-bit range of the grey scale used.

It is interesting to note that some of the perimeter tiles appear to be unwrapped. This is due to the fact they contain fringe termination points. The MST algorithm effectively guides the unwrapping path around the fringe and through the gap in the fringe due to the termination point.

The fringe termination points in the perimeter tiles are in part an artefact of the operation of the thinning algorithm, which makes use of the pruning morphological operation (described in section 4.7.3 on page 52) to eliminate fringe-like noise which otherwise would appear in the inter-fringe regions, Figure 49.

Figure 50 shows tile level unwrapping of up to four phase discontinuities, and a summary of the break down of the time durations taken by the various steps of the unwrapping process are shown in Table 2.

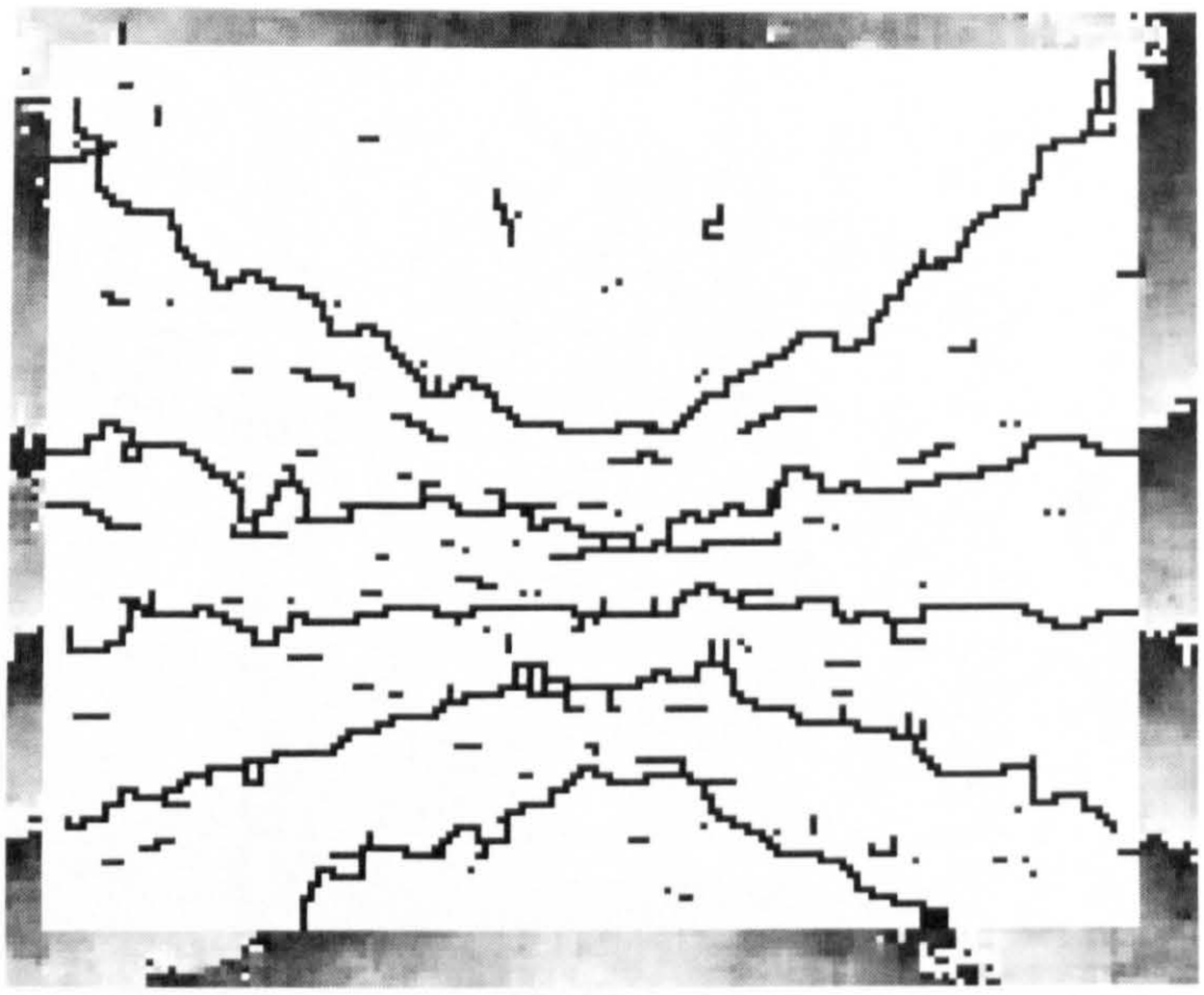


Figure 49 Inter-fringe noise discontinuities when not eliminated by the morphological pruning process, which by contrast was applied to the image shown in Figure 47

Data shown in Table 2 highlight the fact that aspects other than finding the MST could benefit from further optimisation for time performance purposes. Such optimisations are, however, outside the scope of this work.

Operation	Time Duration
Median filtering	14s for the entire 512x512 pixel image
Thinning	2.67s per tile
Finding MST	16ms per tile
Unwrapping and contrast stretching	15ms per tile
Tile edge length	54 pixels
Total number of pixel in tile	2616 pixels (vertices)
Data obtained for the operations illustrated in Figure 50 using a Mesh® Computers Plc personal computer: Intel® Pentium® 4 CPU 3.0 GHz, RAM 1 Gigabyte, Microsoft® XP Home Edition 2002 – service pack 2.	

Table 2 Summary of the break down of the time durations taken by the various steps of the unwrapping process.

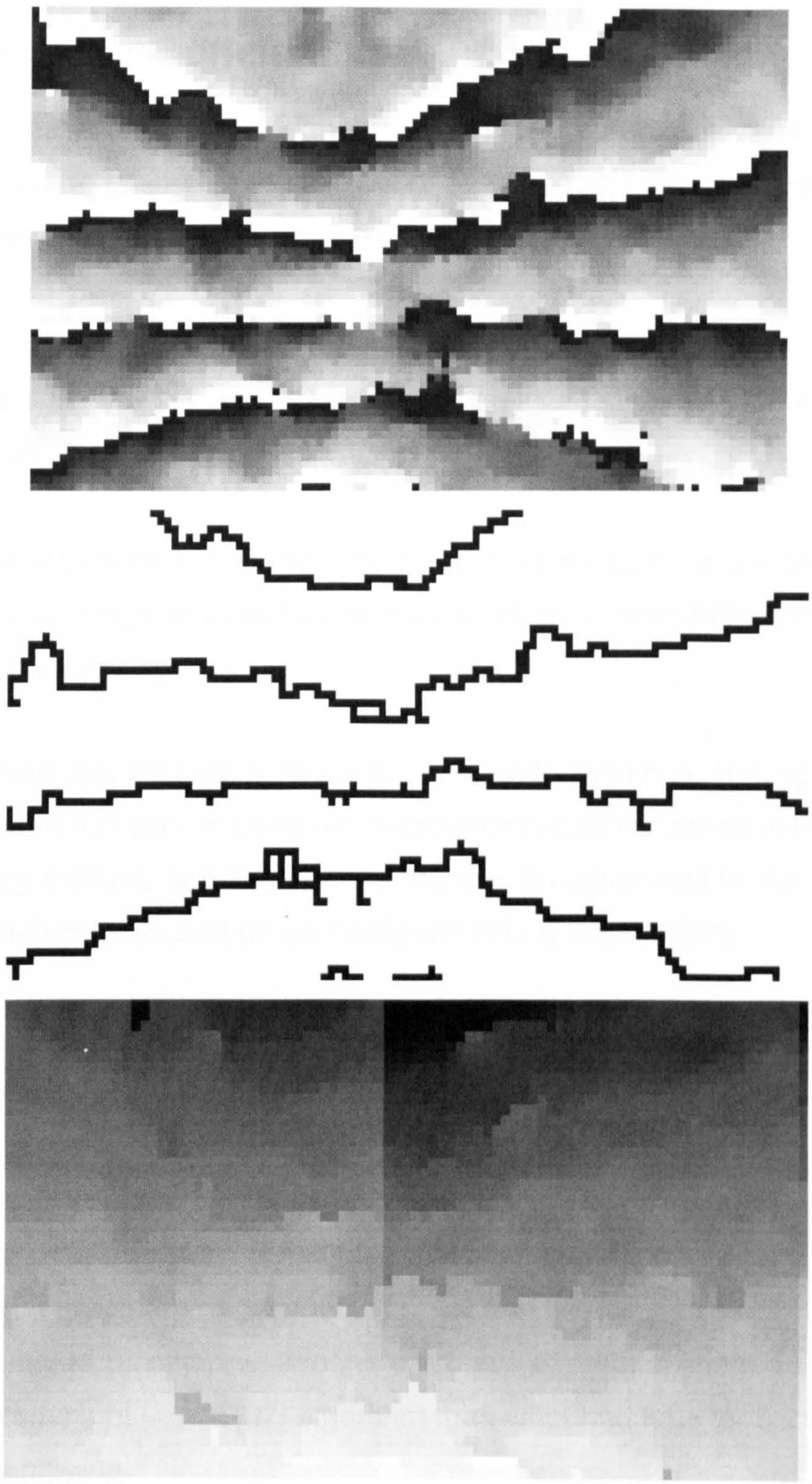


Figure 50 Tile level unwrapping, each tile containing four phase discontinuities (fringes) at least. Wrapped phase map subsection (top), detected fringe locations (centre) and unwrapped tiles (bottom)

Blank

In

Original

Chapter 9 Conclusion

Our preliminary study of interferometry and interferogram analysis has lead us to take up an interest in the subject of phase unwrapping and the role it plays in the field of automatic interferogram analysis.

We reviewed a selection of the great number of phase unwrapping algorithms which exist in the literature, and selected in particular the tile-based method for further investigation.

The tile-base method has many aspects, and central to its operation is the minimum spanning tree algorithm which it employs to select the path followed for phase unwrapping.

We observed that the time complexity of the underlying minimum spanning tree algorithm naturally impacts the computational performance of the overall unwrapping method, and furthermore imposes an upper limit on the size of the tile which can be used for performance-critical applications.

Therefore, we identified the improvement of the computational performance of the minimum spanning tree algorithm, within the context of phase unwrapping, as the focal point of our research.

We examined the problem of finding a minimum spanning tree for a tile's graph from a graph theory perspective.
Our initial investigations revealed the existence of more efficient implementations of Prim's [97] algorithm than what had thus far been used for tile unwrapping.
Shortly after we made this observation, an independent worker in the field of medical imaging published an algorithm with a similarly improve efficiency [65].

We had however by then have made a further observation showing that a tile's graph is indeed planar, and could therefore take advantage of a yet more efficient class of minimum spanning tree algorithms.

We built further on these early research findings by examining the properties of planar graphs more closely, and developing our understanding of how such properties can be taken advantage of to further enhance the time performance of finding a minimum spanning tree for a given tile.

Without having some prior knowledge of the solution, it is not possible to improve on the linear time complexity of minimum spanning tree algorithms for planar graphs.
Nevertheless, we noted that empirical performance gains could be achieved by taking advantage of the prior knowledge of the tile's graph topology.

We have shown how the prior knowledge of a tile's graph topology can help reduce the size of its minimum spanning tree problem by at least one half in a linear time. In the best case the problem is completely solved.

We have also shown how our novel reduction algorithm can be applied using a typical image processing sliding window technique. We believe that this particular aspect increases the potential for the applicability of our algorithm, as many ubiquitous software and hardware image processing components are specifically optimised for such operations.

Finally, an empirical treatment of Prim's algorithm, comparing its performance first without and then with reduction, has demonstrated the effectiveness of our algorithm and its practical applicability.

9.1. Future work

It would be interesting to learn the probability distribution of the size of reduction for a typical tile, and how this knowledge may further enhance the overall efficiency of the unwrapping algorithm.

It would be interesting to find out if further performance gains can be attained by the relaxation of the criterion of the identification of the unwrapping path from an “absolute minimum” to “minimal.
If this was the case, it would be then possible to merely connect any disjoints in the path identified by the problem reduction algorithm, without the need to apply an additional minimum spanning tree algorithm.

One could then consider the noise immunity vs. the computational cost achieved by such a partial approach, which ultimately leads to establishing a typical estimate for the computational performance cost vs. the noise immunity gain for a given application.

Blank

In

Original

References

1 M Takeda, H Ina and S Khobayashi, S Fourier-transform method of fringe pattern analysis for computer based topography and interferometry, J. Op Soc. Am., 72, (1981), 156-60.

2 W W Macy Jr., Two-dimensional fringe pattern analysis. Appl. Optics, 22 (23), (1983).

3 B Breuckmann and W Thieme, Computer-aided analysis of holographic interferograms using the phase shift method. Appl. Optics, 24 (14), (1985), 2145-49.

4 R Thalman and R Dandliker, High-resolution video processing for holographic interferometry applied to contouring and measuring deformations. SPIE ECOOSA, Vol. 429, Amsterdam, (1984).

5 B Breuckmann and W Thieme, Computer-aided analysis of holographic interferograms using the phase shift method. Appl. Optics, 24 (14), (1985), 2145-49.

6 R Dandliker and R Thalmann, Heterodyne and quasi-heterodyne holographic interferometry,Optical Engng, 24(5), (1989), 824-31.

7 J M Huntley and H Saldner, Temporal phase unwrapping algorithm for automated interferogram analysis, Applied Optics, 32(17), (1993), 3047-52

8 G T Reid, Image processing techniques for fringe pattern analysis, proc. Of the first international workshop on automatic processing of fringe patterns, Berlin, (1989), 12-20.

9 (Ed.) D W Robinson and G T Reid, Interferogram analysis – digital fringe pattern measurements techniques, Institute of physics publishing ltd 1993

10 (Ed.) P K Rastogi, Digital speckle pattern interferometry and related techniques, John Wiley and sons, 2001

11 E N Leith and J Upatnieks, Reconstructed wavefronts and communication theory, Journal of Optical Society of America, 52(10), (1962), 1123-30

12 M Idesawa, T Yatagai and T Soma, Scanning moiré method and automatic measurement of 3-D shapes, Applied Optics 16 (8), (1977), p2152-2162

13 M Takeda, Fringe formula for projection type moiré topography, Optics and Lasers in Engineering, 3(1), (1982), 45 – 52

14 J L Doty, Projection moiré for contour analysis, J. Opt. Soc. Am. 73, (1983), 366-372

15 T Young, Experimental Demonstration of the General Law of the Interference of Light, Philosophical Transactions of the Royal Society of London, 94, (1804), 1. Reprinted in Great Experiments in Physics, Morris Shamos ed., Holt Reinhart and Winston, New York, (1959), 96.

16 G T Reid, Moiré fringes in metrology, 5(2), (1984), 63-93

17 E Hecht, Optics, Addison-Wesley, 1987. ISBN 0 201 11611 1.

18 K Creath, Phase-shifting speckle interferometry, Applied Optics, 24 (18), (1985), 3053-58

19 J K Gasvik, Optical Metrology, Chichester, John Wiley & Sons, 1987

20 P Hariharan, optical holography, Cambridge university press, 1984.

21 M Reeves, A J Moore, D P Hand and J D C Jones, Dynamic shape measurement system for laser materials processing, Opt. Eng. 42, (2003), 2923

22 P S Theocaris, Moiré fringes in strain analysis, Pergamon press, 1984, ISNB 0 521 24348 3.

23 J A Leendertz, Interferometric displacement measurement on scattering surfaces utilizing speckle effect, J Phys E: Sci Instrum, 3, (1970), 214

24 J A Quiroga and E Bernabeu, Phase-unwrapping algorithm for noisy phase-map processing, Applied Optics, 33(29), (1994), 6725-31

25 K Creath, Phase-measurement interferometry techniques, Progress in Optics, 26, (1988), 349-93.

26 G T Reid, Automatic fringe analysis – a review, Optics and Lasers in Engineering, 7, (1986/7), 37-68.

27 C R Coggrave, Quantitative interferogram analysis, Phase vision ltd, 2004.

28 T R Judge and P J Bryanston-Cross, A review of phase unwrapping techniques in fringe analysis, Opt. Laser. Eng. 21 (4), (1994), 199-239

29 D C Ghiglia and M D Pritt, Two-dimensional phase unwrapping, John Wiley ans sons, 1998

30 T Yatagai, M Idesawa, Y Yamaashi and M Suzuki, Interactive fringe analysis system applications to Moiré contourgram and interferogram, Optical Engineering, 21, (1982), 091-6.

31 H O Saldner, J M Huntley, Temporal phase unwrapping: Application to surface profiling of discontinuous objects, Applied Optics 36(13), (1997), 2770-75

32 A D Nurse, Load-stepping photoelasticity: new developments using temporal phase unwrapping, Opt. Laser. Eng. 38 (1-2), (2002), 57-70

33 G Pedrini, I Alexeenko, W Osten et al, Temporal phase unwrapping of digital hologram sequences, Applied Optics 42(29), (2003), 5846-54

34 C R Coggrave and J M Huntley, Real-time visualisation of deformation fields using speckle interferometry and temporal phase unwrapping, Opt. Laser. Eng. 41(4), (2004), 601-620

35 A Davila, P D Ruiz, G H Kaufmann et al, Measurement of sub-surface delaminations in carbon fibre composites using high-speed phase-shifted speckle interferometry and temporal phase unwrapping, Opt. Laser. Eng. 40(5-6), (2003), 447-458

36 J M Huntley, Fringe analysis today and tomorrow, Proc. Of SPIE, 4933, (2003), 167-174

37 A B Suksmono and A Hirose, A fractal estimation method to reduce the distortion in phase unwrapping process, IEICE T Commun. E88B (1):, (2005), 364-371

38 A M Guarnieri, Using topography statistics to help phase unwrapping, IEE P-Radar Son. Nav 150 (3), (2003), 144-151

39 J Meneses, T Gharbi and P Humbert, Phase-unwrapping algorithm for images with high noise content based on a local histogram, Applied Optics 44 (7), (2005), 1207-1215

40 M A Herraez, JG Boticario, M J Lalor et al, Agglomerative clustering-

based approach for two-dimensional phase unwrapping, Applied Optics 44 (7), (2005), 1129-1140

41 J J Chyou, S J Chen and Y K Chen, Two-dimensional phase unwrapping with a multichannel least-mean-square algorithm, Applied Optics 43 (30), (2004) 5655-5661

42 C W Chen and H A Zebker, Two-dimensional phase unwrapping with use of statistical models for cost functions in nonlinear optimization, J Opt Soc. Am A, 18 (2), (2001), 338-351

43 G F Carballo and P W Fieguth, Probabilistic cost functions for network flow phase unwrapping, International Geoscience and Remote Sensing Symposium (IGARSS 99), 38 (5) part 1, (2000), 2192-2201

44 A Collaro, G Fornaro, G Franceschetti and et al, Local, global and unconventional phase unwrapping techniques,1997 International Geoscience and Remote Sensing Symposium (IGARSS 97) on Remote Sensing – A Scientific Vision for Sustainable Development, vol. I-IV, (1997), 433-435

45 A Collaro, G Franceschetti, F Palmieri et al, Phase unwrapping by means of genetic algorithms, J OPT Soc. Amer. A, 15(2), (1998), 407-418

46 L Guerriero, A Refice , S Stramaglia et al, Global approaches and local strategies for phase unwrapping, Nuovo Cimento c 24 (1), (2001), 205-222

47 ZJ Peng, F Qian, XF Wang et al, Phase unwrapping with regularized phase-tracking technique based on simulated annealing algorithm, Optik 114 (4), (2003) 175-180

48 M Minami and A Hirose, Phase singular points reduction by a layered complex-valued neural network in combination with constructive Fourier synthesis, Lect. Notes Comput. Sci. 2714, (2003), 943-950

49 U V Toussaint, S Gori and V Dose, Bayesian neural-networks-based evaluation of binary speckle data, Appl. Optics 43 (28), (2004) 5356-5363

50 W J Joo and S Y S Cha, Knowledge-based hybrid expert system for

automated interferometric data reduction, Opt. Laser Eng. 24 (1), (1996) 57-75

51 M Costantini, A novel phase unwrapping method based on network programming, IEEE T Geosci. Remot. 36 (3), (1998), 813-821

52 N Egidi, P Maponi and A comparative study of two fast phase unwrapping algorithms, Appl. Math. Comput. 148 (3), (2004), 599-629

53 M Hubig, S Suchandt and N Adam, Equivalence of cost generators for minimum cost flow phase unwrapping, J. Opt. Soc. Am. A 19 (1), (2002), 64-70

54 P Maponi and F Zirilli, A class of global optimization problems as models of the phase unwrapping problem, J. Global Optim., 21 (3), (2001), 289-316

55 M Hubig, S Suchandt, N Adam, A class of solution-invariant transformations of cost functions for minimum cost flow phase unwrapping, J. Opt. Soc. Am. A, 21 (10), (2004), 1975-1987

56 A Ettemeyer, U Neupert, H Rottenkolber and C Winter, Schnelle und robuste bildanalyse von streifenmustern – ein wichtiger schritt der automation von hlografischen profprozessen, Proc. 1st int. Workshop on automatic processing of fringe patterns. (1989), 23-31.

57 D P Tower, T R Judge and P J Bryanston-Cross, A quasi-heterodyne holograph technique and automatic algorithms for phase unwrapping, SPIE, (1989) 1163.

58 D P Tower, T R Judge and P J Bryanston-Cross, Analysis of holographic fringe data using the dual reference approach. Opt. Engng, 30 (4), (1991), 452-60.

59 D P Tower, T R Judge and P J Bryanston-Cross, Automatic interferogram analysis techniques applied to quasi-heterodyne holography and ESP. Optics and Lasers in Engng, 14, (1991), 239-81.

60 T R Judge, C Quan and P J Bryanston-Cross, Holographic deformation measurements by Fourier transform technique with automatic phase unwrapping. Opt Engng, 31 (3), (1992), 533-43

61 P J Bryanston-Cross, C QUAN, T R Judge, Application of the FFT

method for the quantitative extraction of information from high-resolution interferometric and photoelastic data, Opt. Laser Technol. 26 (3), (1994), 147-155

62 T R Judge, Quantitative digital image processing in fringe analysis and particle image velocimetry, PhD thesis, Warwick university (1992)

63 [62], Chapter 2, section 2.6.3.2, pp70-71.

64 N H Ching, D Rosenfeld and M Braun, Two-dimensional phase unwrapping using a minimum spanning tree algorithm, IEEE Trans. Img. Proc., 1 (3), (1992), 355-365

65 L An, Q S Xiang and S Chavez, A fast implementation of the minimum spanning tree method for phase unwrapping, IEEE T Med. Imaging,19 (8), (2000), 805-808

66 M Takeda and T Abe, Phase unwrapping by a maximum cross-amplitude spanning tree algorithm: A comparative study, Opt. Eng., 35 (8), (1996) 2345-2351

67 S Takao, S Yoneyama and M Takashi, Minute displacement and strain analysis using lens-less Fourier transformed holographic interferometry, Opt. Laser. Eng. 38 (5), (2002), 233-244

68 R Smythe and R Moore, Instantaneous phase measuring interferometry, Porc. Soc. Photo-Opt. Int., 429, (1983), 16-21

69 R Smythe and R Moore, Instantaneous phase measuring interferometry, Opt. Eng. 23 (4), (1984), 361-364

70 O Y Kwon and D M Shough, Multichannel grating phase shift interferometer, Proc. SPIE, 599, (1985), 273-9

71 K G Harding, M P Coletta and C H Vandommelen, Color-encoded □oiré contouring, Proc. SPIE 1005, (1988), 167-178

72 D C Ghiglia, G A Mastin and L A Romero, Cellular automata method for phase unwrapping, J. Opt. Soc. Am., 4A, (1987), 267-80.

73 A Spike and D W Robinson, Investigation of the cellular automata method of phase unwrapping and its implementation on an array processor, Optics and Lasers in Engineering, 14, (1992), 25-37.

74 C Buckberry and J Davies, Digital phase-shifting interferometry and its

application to automotive structures, Applied Optics, (1990), 275-6.

75 H Y Chang, C W Chen, C K Lee et al., The tapestry cellular automata phase unwrapping algorithm for interferogram analysis, Opt. Laser. Eng. 30 (6), (1998), 487-502

76 J R Buckland, J M Huntley and S R E Turner, Unwrapping noisy phase maps by use of a minimum-cost-matching algorithm, Appl. Opt., 34 (23), (1995), 5100-

77 A Baldi, Phase unwrapping by region growing, Applied optics, vol 42, no. 14, (2003), 2498-2505

78 R M Goldstein, H A Zebker and C L Werner, Satellite radar interferometry two-dimensional phase unwrapping, Radio Science, 23 (4), (1988), 713-720.

79 J M Huntley, Noise immune phase unwrapping algorithm, Applied Optics, 28 (15), (1989), 3268-70.

80 J M Huntley, Three-dimensional noise immune phase unwrapping algorithm, Applied Optics, 40, (2001), 3901-08,

81 D C Ghiglia and L A Romero, Minimum L^p-norm two-dimensional phase unwrapping, J Opt. Soc. Am, 13, (1996), 1999-2013

82 T J Flynn, Two-dimensional phase unwrapping with minimum weighted discontinuity, J Opt. Soc. Am, 14, (1997), 2692-2701

83 M Constantini, A novel phase unwrapping method based on network programming, IEEE Trans. Geosci. Remote Sens., 36, (1998), 813-821

84 C W Chen and H A Zebker, Network approaches to two-dimensional phase unwrapping: intractability and two new algorithms, J Opt. Soc. Am, 17, (2000), 401-414

85 J M Huntley and O Saldner, Shape measurement by temporal phase unwrapping: a comparison of unwrapping algorithms, Meas. Sci. Technol, 8 (9), (1997), 986-992

86 P Ettl and K Creath, Comparison of phase-unwrapping algorithms by using gradient of first failure. App Opt, 33 (25), (1996), 5108-5114

87 T Yatagai, M Idesawa, Y Yamaashi and M Suzuki, Interactive fringe analysis system: applications to □oiré contourogram and interferogram,

Opt. Eng., 21 (5), (1982), 901-906

88 C J Hilditch, Linear skeletons from square cupboards, Machine Intelligence 4, (1969), 403-420

89 J C Russ, The image processing handbook, CRC press, ISBN 084931142X, Chapter 7, (2002)

90 R L Graham and P Hell, On the history of the minimum spanning tree problem, Annals of the History of Computing, 7, (1985), 43-57

91 O Boruvka, O jistem problemu minimalnim, Praca Morvavke Prirodovedecke Spolecnosi, 3, (1926), 37-58 (Czech with summary in German, translated in [92])

92 J Nesetril, E Milkova and H Nesetrilova, Otakar Boruvka on minimum spanning tree problem, Discrete Mathematics, 233 (1-3), (2001), 3-36 (English translation of [91])

93 G Choquet, Etude de certains rseaux de routes, Comptes Rendus Acad. Sci., 206, (1938), 310-313

94 K Florek, L Lukasziewicz, J Perkal, H Steinhaus and S Zubrzycki, Sur la liaison et la division des points d'un ensemble fini, Colloq. Math., 2 (1951), 282-285

95 J B Kruskal, On the shortest spanning sub-tree of a graph and the travelling salesman problem, Proc. Amer. Math. Soc., 7, (1956), 48-50.

96 R E Tarjan, Efficiency of a good but not linear set union algorithm, Journal of the ACM, 22 (2), (1975), 215-25

97 R C Prim, Shortest connection networks and some generalisations, Bell Systems Tech. J., 36, (1957), 1389-1401

98 C Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, 1973.

99 J A Bondy and U S R Murty, Graph theory with applications, North-Holland, New York, 1976.

100 R G Busacker and T L Saaty, Finite graphs and networks – an introduction with applications, McGraw-Hill, New York, 1965.

101 F Harry, Graph theory, Addison-Wesley, Reading, MA, 1969.

102 R E Tarjan, Data structures and network algorithms, Soc Industrial App Math, Chap 6 (1993), 72-77

103 A Kerschenbaum and R Van Slyke, Computing minimum spanning trees efficiently, Proc 25th Ann Conf ACM, (1972), 518-527

104 B M E Moret and H D Shapiro, An empirical analysis of algorithms for constructing a minimum spanning tree, Lecture notes in computer science 519, (1991), 400-411

105 M L Fredman and R E Tarjan, Fibonacci heaps and their use in improved network optimisation algorithms, J ACM, 34 (3), (1987), 596-615

106 H N Gabow, Z Galil, T Spencer and R E Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, Combinatorica, 6, (1986), 109-122

107 A C Yao, An O(|E| log log|V|) algorithm for finding minimum spannig trees, Information Processing Letters, 4 (1), (1975), 21-3.

108 B Chazelle, A minimum spanning tree algorithm with inverse-Ackermann type complexity, J ACM, 47, (2000), 1028-1047.

109 S Pettie, Finding minimum spanning trees in O(mα(m, n)) time, Tech Rep TR99-23, Univ Texas at Austin, 1999.

110 D R Karger, P N Klein and R E Tarjan, A randomised linear-time algorithm to find minimum spanning trees, J. Assoc. Comput. Machinery, 42 (2), (1995), 321-28.

111 V King, A simpler minimum spanning tree verification algorithm, Proc. Workshop on algorithms and data structures, 1995

112 D Cheriton and R E Tarjan, Finding minimum spanning trees, SIAM J. Comput., 5 (4), (1976), 724-42.

113 T Matsui, The minimum spanning tree problem on a planar graph, Discrete Applied Mathematics, 58, (1995), 91-4.

114 G Frederickson, Data structures for on-line updating of minimum spanning tree, with applications, SIAM J Comp. 14 (4), (1985), 781-798

115 D Eppstine, G F Italiano, R Tamassia, R E Tarjan, J Westbrook and M Yung, Maintanace of minimum spanning forest in a dynamic planar graph, Proc 1st ACM/SIAM Symp. Discrete Algorithms, (1990), 1-11

116 H N Gabow and M Stallman, Efficient algorithms for graphic matroid

intersection and priority, Proc 12th Int. Conf Automata, Languages, and Programming, Springer-Verlag LNCS, 194, (1985), 210-220

117 J G Siek, L Q Lee and A Lumsdaine, The boost graph library, Addison-Wesley, 2002.

118 R M Groves, Development of Shearography for Surface Strain Measurement of Non-Planar Objects, PhD thesis, Cranfield University, 2001.

8 Appendix Program listings

A1. Introduction

The analysis, design, and implementation of our computer programming for the tile-based method were carried out based only on its textual descriptions in Judge's thesis [62]. There is no inclusion or direct reference to his particular programming implementation which he presents in the second volume of his thesis. Our practical treatment of the algorithm is otherwise completely original.

The reasons behind this approach were highlighted in Chapter 4, which also includes various examples of the practical results obtained, giving them descriptions in the context of the various steps of the algorithm.

We chose the C++ programming language for our implementation due to:

- Its inherent performance efficiency and wide-spread application, support, and optimisation for a large spectrum of computer platforms.
- The extensive availability of library extensions, for both computational functionality and user friendly interaction.
- Its support for both procedural and object oriented programming approaches, and the ability to combine both paradigms within the same application.

In the following sections of the appendix, we described the overall analysis and design of our implementation of the tile-based phase unwrapping method. We also present code listings of its computer programming implementation.

A1.1. Analysis

An initial analysis of the tile-based minimum spanning tree (MST) method has shown that its hierarchical nature lends it well to an object-oriented programming approach.

The class-responsibility-collaboration (CRC) method is often used during the analysis phase of the problem domain.
The method leads to the identification of the key objects of the solution by their problem domain names, Table 3.

C	Pixel
R	• An elementary type which has a grey-scale value and the concept of pixel-neighbourhoods.
C	• Vertex: A pixel can be treated as a vertex in a graph.
C	Image
R	• Provides access to pixels by a location index.
C	• Pixel: an image has a collection of pixels.
C	Tile
R	• Provides access to a subsection of an image. • Knows how to construct its weighted graph and its minimum spanning tree (MST).
C	• Pixel: a tile has a collection of pixels. • Vertex: A tile can be treated as a vertex in a graph. • Graph: a tile has a weighted graph. • MST: a tile has a MST spanning its weighted graph.
C	Tiled image
R	• Knows how to construct itself from an image. • Provides indexed access to its tiles. • Manages the assembly of tiles.
C	• Tile: a tiled image has a collection of tiles. • Vertex grid: a tiled image can be treated as a grid of tile-vertices. • Graph: a tiled image has a weighted graph
C	Vertex
R	• Is an abstraction defining the interface expected of a vertex in a graph

C	• Collaboration is realised by concrete objects which adopt the Vertex interface, such as Pixel and Tile.
C	Vertex grid
R	• Is an abstraction defining the interface expected of a grid of vertices
C	• Collaboration is realised by concrete objects which adopt the Vertex grid interface, such as Image and Tiled image.
C	Graph
R	• Knows how to construct itself from a Vertex grid • Provides indexed access to its vertices.
C	• Vertex: Graph has a collection of Vertex objects.
C	MST
R	• Knows how to construct itself from a weighted graph. • Provides indexed access to MST elements.
C	• Vertex: a MST has a collection of vertices.

Table 3 Class-Responsibility-Collaboration analysis of the problem domain, and the classes identified

A1.2. Design

The unified modelling language (UML) class diagram, Figure 51, shows the various relationships of the classes identified by the analysis phase.

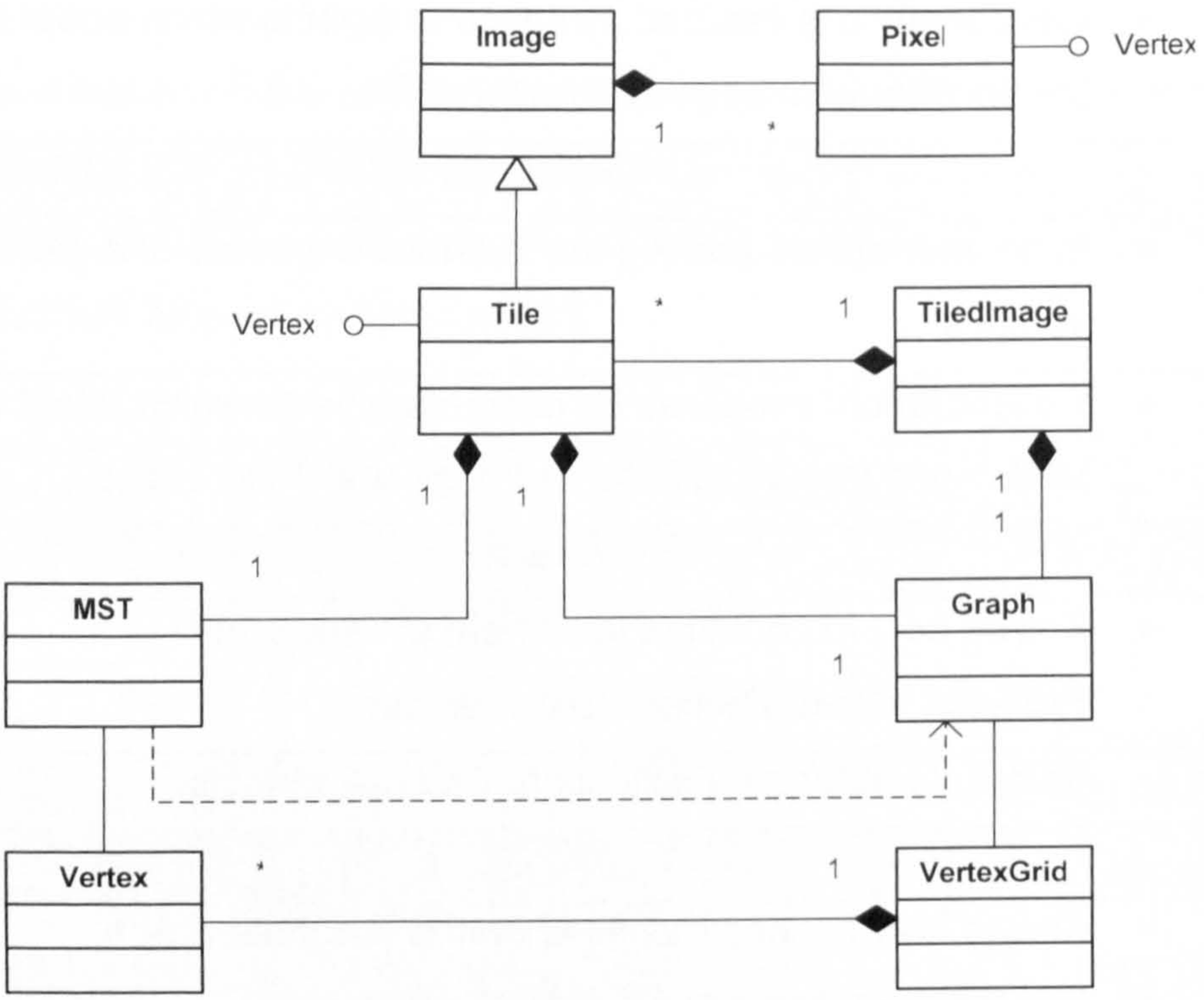


Figure 51 Class diagram

A1.3. Implementation listings

CommonTypes.h

```
#ifndef COMMON_TYPES_H
#define COMMON_TYPES_H

enum FilterKernelIndex
{
	TOP_LEFT =		0,
	NORTH =				1,
	TOP_RIGHT =		2,
	WEST =				3,
	CENTRE =		4,
	EAST =				5,
	BOTTOM_LEFT =	6,
	SOUTH =				7,
	BOTTOM_RIGHT =	8,
	KERNEL_SIZE =	9
};

struct Vertex_stc
{
	int	m_x;
	int	m_y;
	int	m_NeighbourCount;
	bool	m_HasNorthNeighbour;
	bool	m_HasWesthNeighbour;
	bool	m_HasEasthNeighbour;
	bool	m_HasSouthNeighbour;
	int	m_NorthWeight;
	int	m_EastWeight;
	int	m_WestWeight;
	int	m_SouthWeight;
};

enum NeighbourDirection_enum
{
	ND_UNDEFINED,
	ND_NORTH,
	ND_WEST,
	ND_EAST,
	ND_SOUTH
};

enum EdgeDirection_enm
{
	ED_UNDEFINED,
	ED_NORTH,
	ED_WEST,
	ED_EAST,
	ED_SOUTH,
    ED_NORTH_EAST,
    ED_NORTH_WEST,
    ED_SOUTH_EAST,
    ED_SOUTH_WEST
};


enum KeyCode_enum
{
    WRAPPED = 0,
    FILTERED,
    MST,
    WEIGHTS,
```

```
    UNWRAPPED
};

struct AttributeKeyCodeInfo_stc
{
    KeyCode_enum m_KeyCode;
    CString m_AttributeDesciption;
    BYTE m_Red;
    BYTE m_Green;
    BYTE m_Blue;

    AttributeKeyCodeInfo_stc(KeyCode_enum keyCode, CString
attributeDesciption, BYTE red, BYTE green, BYTE blue):
    m_KeyCode(keyCode),
    m_AttributeDesciption(attributeDesciption),
    m_Red(red),
    m_Green(green),
    m_Blue(blue)
    {
    }
};

static AttributeKeyCodeInfo_stc KeyCodeMAp[] =
{
    AttributeKeyCodeInfo_stc(WRAPPED, "abc", 0, 0, 0)
};

const int STREACHED_RANGE = 255;
const int TWO_PIE = 255;


#include "TileDataModel_cls.h"
#include "ImagePixel_cls.h"


class Graph_cls;
class Vertex_cls;

struct MstElement_stc
{
    MstElement_stc(Vertex_cls& vertex,
                   int weight,
                   EdgeDirection_enm direction):
    m_Vertex(vertex),
    m_Weight(weight),
    m_Direction(direction){}

    ~MstElement_stc() {}

    Vertex_cls&          m_Vertex;
    int                  m_Weight;
    EdgeDirection_enm    m_Direction;
};

#endif //COMMON_TYPES_H
```

Graph_cls.h

```
#ifndef GRAPH_CLS_H
#define GRAPH_CLS_H

class Vertex_cls;
class VertexGrid_cls;

class Graph_cls
{
public:
   Graph_cls(VertexGrid_cls& grid);
   Graph_cls(Graph_cls& grid);
   ~Graph_cls();

   int GetSize() const;
   int GetXSize() const;
   int GetYSize() const;
   Vertex_cls& operator[](int index) const;

private:
   VertexGrid_cls& m_Grid;
   const int m_GraphSize;
};

#endif //GRAPH_CLS_H
```

Graph_cls.cpp

```
#include "stdafx.h"
#include "VertexGrid_cls.h"
#include "Vertex_cls.h"
#include "Graph_cls.h"

Graph_cls::Graph_cls(VertexGrid_cls& image):
m_Grid(image),
m_GraphSize(m_Grid.GetVertexSize())
{
    const int xSize = m_Grid.GetXSize();
    const int ySize = m_Grid.GetYSize();
    ASSERT (m_GraphSize == (xSize * ySize));

    int maxWeight = INT_MIN;

    //calculate weights
    for(int counter=0; counter<m_GraphSize; counter++)
    {
        int y = counter / xSize;
        int x = counter - (y * xSize);
        int w = INT_MIN;

        if ((y < ySize -1) && y > 1)
        {
            w = abs((m_Grid[ counter]  + m_Grid[ counter+xSize] ) -
                        (m_Grid[ counter-xSize]  + m_Grid[ counter-
(2*xSize)] ));

            m_Grid[ counter] .SetSouthWeight(w);
            m_Grid[ counter+xSize] .SetNorthWeight(w);
        }
        else if ((y < ySize -1) && y)
        {
            w = abs((m_Grid[ counter]  - m_Grid[ counter+xSize] ));

            m_Grid[ counter] .SetSouthWeight(w);
            m_Grid[ counter+xSize] .SetNorthWeight(w);
        }

        if ((x < xSize -1) && x > 1)
        {
            w = abs((m_Grid[ counter]  + m_Grid[ counter+1] ) -
                        (m_Grid[ counter-1]  + m_Grid[ counter-2] ));

            m_Grid[ counter] .SetEastWeight(w);
            m_Grid[ counter-1] .SetWestWeight(w);
        }
        else if ((x < xSize -1) && x)
        {
            w = abs((m_Grid[ counter]  - m_Grid[ counter+1] ));

            m_Grid[ counter] .SetEastWeight(w);
            m_Grid[ counter-1] .SetWestWeight(w);
        }

        if(w > maxWeight)
        {
            ASSERT (w < INT_MAX);

            maxWeight = w + 1;
        }
```

```
}

//force undesirable vertices to be unwrapped last
//first do all vertices which are four-connected
for(int Ycount=1; Ycount<ySize-1; Ycount++)
{
   for(int Xcount=1; Xcount<xSize-1; Xcount++)
   {
      int index = (Ycount * xSize) + Xcount;

      if (m_Grid[ index] .IsToHaveMaxWeight())
      {
         m_Grid[ index - 1] .SetEastWeight(maxWeight);
         m_Grid[ index + 1] .SetWestWeight(maxWeight);
         m_Grid[ index - xSize] .SetSouthWeight(maxWeight);
         m_Grid[ index + xSize] .SetNorthWeight(maxWeight);
      }
   }
}

//now do the perimiter vertices
//starting with the north perimiter
for(int count=0; count<xSize; count++)
{
   if (m_Grid[ count] .IsToHaveMaxWeight())
   {
      m_Grid[ count + xSize] .SetNorthWeight(maxWeight);
   }
}

//south perimiter
for(int count=m_GraphSize-xSize; count<m_GraphSize; count++)
{
   if (m_Grid[ count] .IsToHaveMaxWeight())
   {
      m_Grid[ count - xSize] .SetSouthWeight(maxWeight);
   }
}

//West perimiter
for(int count=0; count<m_GraphSize-xSize; count+=xSize)
{
   if (m_Grid[ count] .IsToHaveMaxWeight())
   {
      m_Grid[ count + 1] .SetEastWeight(maxWeight);
   }
}

//East perimiter
for(int count=xSize-1; count<m_GraphSize; count+=xSize)
{
   if (m_Grid[ count] .IsToHaveMaxWeight())
   {
      m_Grid[ count - 1] .SetWestWeight(maxWeight);
      }
   }
}

Graph_cls::~Graph_cls()
{
}

int Graph_cls::GetSize() const
{
   return m_GraphSize;
```

```
}

int Graph_cls::GetXSize() const
{
    return m_Grid.GetXSize();
}

int Graph_cls::GetYSize() const
{
    return m_Grid.GetYSize();
}

Vertex_cls& Graph_cls::operator[](int index) const
{
    return m_Grid[index];
}
```

Image_cls.h

```
#ifndef IMAGE_CLS_H
#define IMAGE_CLS_H

class Vertex_cls;
#include "VertexGrid_cls.h"

class Pixel_cls;

class Image_cls :
   public VertexGrid_cls
{
public:
   Image_cls(int xSize = 0, int ySize = 0);
   virtual ~Image_cls();

   Pixel_cls& operator[](int index) const;
   void AssignImageData(int* pData, int size);
   int GetPixelSize();

   //VertexGrid_cls support
   int GetVertexSize() const;
   int GetXSize() const;
   int GetYSize() const;
   Vertex_cls& GetAt(int index) const;

private:
   const int m_xSize;
   const int m_ySize;
   const int m_PixelArraySize;
   Pixel_cls* m_PixelArray;
};

#endif //IMAGE_CLS_H
```

Image_cls.cpp

```
#include "stdafx.h"
#include "Pixel_cls.h"
#include "Image_cls.h"

Image_cls::Image_cls(int xSize, int ySize):
m_xSize(xSize),
m_ySize(ySize),
m_PixelArraySize(xSize * ySize)
{
    m_PixelArray = new Pixel_cls[ m_PixelArraySize] ;
}

Image_cls::~Image_cls()
{
    delete[ ] m_PixelArray;
}

Pixel_cls& Image_cls::operator[ ](int index) const
{
    ASSERT (index < (m_PixelArraySize));
    return m_PixelArray[ index] ;
}

void Image_cls::AssignImageData(int* pData, int size)
{
    ASSERT (size <= m_PixelArraySize);

    for(int counter=0; counter<size; counter++)
    {
        myself[ counter]  = pData[ counter] ;
    }
}

int Image_cls::GetPixelSize()
{
    return m_PixelArraySize;
}

//VertexGrid_cls support
int Image_cls::GetVertexSize() const
{
    return m_PixelArraySize;
}

int Image_cls::GetXSize() const
{
    return m_xSize;
}

int Image_cls::GetYSize() const
{
    return m_ySize;
}

Vertex_cls& Image_cls::GetAt(int index) const
{
    return myself[ index] ;
}
```

Imagepixel_cls.h

```
#ifndef IMAGE_PIXEL_CLS_H
#define IMAGE_PIXEL_CLS_H

class ImagePixel_cls
{
public:
    enum THINNED_EDGE_TYPE
    {
          NO_EDGE,
    EDGE
    };

    enum THINNED_EDGE_VERTICAL_SHAPE
    {
          NO_VERTICAL_SHAPE,
          NORTH_HIGH,
          SOUTH_HIGH
    };

    enum THINNED_EDGE_HORIZONTAL_SHAPE
    {
          NO_HORIZONTAL_SHAPE,
          WEST_HIGH,
          EAST_HIGH
    };

enum THINNED_EDGE_DIAGONAL_SHAPE
{
          NO_DIAGONAL_SHAPE,
    NORTH_WEST_HIGH,
    SOUTH_EAST_HIGH
};

enum THINNED_EDGE_REVERSE_DIAGONAL_SHAPE
{
          NO_REVERSE_DIAGONAL_SHAPE,
    NORTH_EAST_HIGH,
    SOUTH_WEST_HIGH
};

    ImagePixel_cls();
    ~ImagePixel_cls();

    void SetValue(int newValue);
    int GetValue() const;

    void SetNeighbour_NW(ImagePixel_cls* pNW);
    void SetNeighbour_NN(ImagePixel_cls* pNN);
    void SetNeighbour_NE(ImagePixel_cls* pNE);
    void SetNeighbour_WW(ImagePixel_cls* pWW);
    void SetNeighbour_EE(ImagePixel_cls* pEE);
    void SetNeighbour_SW(ImagePixel_cls* pSW);
    void SetNeighbour_SS(ImagePixel_cls* pSS);
    void SetNeighbour_SE(ImagePixel_cls* pSE);

    bool DoesHaveEightNeibours() const;
    bool DoesHaveEdge() const;

    THINNED_EDGE_TYPE GetThinnedEdgeType() const;
    THINNED_EDGE_HORIZONTAL_SHAPE GetThinnedEdgeHorizontalShape()
```

```
const;
      THINNED_EDGE_VERTICAL_SHAPE GetThinnedEdgeVerticalShape()
const;
      THINNED_EDGE_DIAGONAL_SHAPE GetThinnedEdgeDiagonalShape()
const;
      THINNED_EDGE_REVERSE_DIAGONAL_SHAPE
GetThinnedEdgeReverseDiagonalShape() const;

      void SetThinnedEdgeType(THINNED_EDGE_TYPE type);
      void
SetThinnedEdgeHorizontalShape(THINNED_EDGE_HORIZONTAL_SHAPE shape);
      void SetThinnedEdgeVerticalShape(THINNED_EDGE_VERTICAL_SHAPE
shape);
      void SetThinnedEdgeDiagonalShape(THINNED_EDGE_DIAGONAL_SHAPE
shape);
      void
SetThinnedEdgeReverseDiagonalShape(THINNED_EDGE_REVERSE_DIAGONAL_SHA
PE shape);

      void Reset();
      void CloneEdgeIfNotAlreadyEdge(ImagePixel_cls& pixelToClone);

private:
      int m_PixelValue;
      ImagePixel_cls* m_pNW;
      ImagePixel_cls* m_pNN;
      ImagePixel_cls* m_pNE;
      ImagePixel_cls* m_pWW;
      ImagePixel_cls* m_pEE;
      ImagePixel_cls* m_pSW;
      ImagePixel_cls* m_pSS;
      ImagePixel_cls* m_pSE;
      THINNED_EDGE_TYPE m_ThinnedEdgeType;
      struct EdgeShape_stc·
      {
            THINNED_EDGE_VERTICAL_SHAPE m_VerticalShape;
            THINNED_EDGE_HORIZONTAL_SHAPE m_HorizontalShape;
      THINNED_EDGE_REVERSE_DIAGONAL_SHAPE m_ReverseDiagonalShape;
      THINNED_EDGE_DIAGONAL_SHAPE m_DiagonalShape;
      };

      EdgeShape_stc m_ThinnedEdgeShape;
      bool m_EdgeHasBeenGravitated;
};

#endif //IMAGE_PIXEL_CLS_H
```

ImagePixel_cls.cpp

```
#include "stdafx.h"

#include "ImagePixel_cls.h"

ImagePixel_cls::ImagePixel_cls()
{
	Reset();
}

void ImagePixel_cls::Reset()
{
	m_EdgeHasBeenGravitated = false;
	m_ThinnedEdgeType = NO_EDGE;
	m_ThinnedEdgeShape.m_HorizontalShape = NO_HORIZONTAL_SHAPE;
	m_ThinnedEdgeShape.m_VerticalShape = NO_VERTICAL_SHAPE;
   m_ThinnedEdgeShape.m_DiagonalShape = NO_DIAGONAL_SHAPE;
   m_ThinnedEdgeShape.m_ReverseDiagonalShape =
NO_REVERSE_DIAGONAL_SHAPE;
	m_pNW = NULL;
	m_pNN = NULL;
	m_pNE = NULL;
	m_pWW = NULL;
	m_pEE = NULL;
	m_pSW = NULL;
	m_pSS = NULL;
	m_pSE = NULL;
}

ImagePixel_cls::~ImagePixel_cls()
{
}

void ImagePixel_cls::SetValue(int newValue)
{
	m_PixelValue = newValue;
}

int ImagePixel_cls::GetValue() const
{
	return m_PixelValue;
}

void ImagePixel_cls::SetNeighbour_NW(ImagePixel_cls* pNW)
{
	m_pNW = pNW;
}

void ImagePixel_cls::SetNeighbour_NN(ImagePixel_cls* pNN)
{
	m_pNN = pNN;
}

void ImagePixel_cls::SetNeighbour_NE(ImagePixel_cls* pNE)
{
	m_pNE = pNE;
}

void ImagePixel_cls::SetNeighbour_WW(ImagePixel_cls* pWW)
{
	m_pWW = pWW;
}
```

```
void ImagePixel_cls::SetNeighbour_EE(ImagePixel_cls* pEE)
{
	m_pEE = pEE;
}

void ImagePixel_cls::SetNeighbour_SW(ImagePixel_cls* pSW)
{
	m_pSW = pSW;
}

void ImagePixel_cls::SetNeighbour_SS(ImagePixel_cls* pSS)
{
	m_pSS = pSS;
}

void ImagePixel_cls::SetNeighbour_SE(ImagePixel_cls* pSE)
{
	m_pSE = pSE;
}

ImagePixel_cls::THINNED_EDGE_TYPE
ImagePixel_cls::GetThinnedEdgeType() const
{
	return m_ThinnedEdgeType;
}

bool ImagePixel_cls::DoesHaveEightNeibours() const
{
	return (m_pNW != NULL &&
			m_pNN != NULL &&
			m_pNE != NULL &&
			m_pWW != NULL &&
			m_pEE != NULL &&
			m_pSW != NULL &&
			m_pSS != NULL &&
			m_pSE != NULL);
}

bool ImagePixel_cls::DoesHaveEdge() const
{
	return (m_ThinnedEdgeType == ImagePixel_cls::EDGE);
}


ImagePixel_cls::THINNED_EDGE_HORIZONTAL_SHAPE
ImagePixel_cls::GetThinnedEdgeHorizontalShape() const
{
	return m_ThinnedEdgeShape.m_HorizontalShape;
}

ImagePixel_cls::THINNED_EDGE_VERTICAL_SHAPE
ImagePixel_cls::GetThinnedEdgeVerticalShape() const
{
	return m_ThinnedEdgeShape.m_VerticalShape;
}

ImagePixel_cls::THINNED_EDGE_DIAGONAL_SHAPE
ImagePixel_cls::GetThinnedEdgeDiagonalShape() const
{
	return m_ThinnedEdgeShape.m_DiagonalShape;
}

ImagePixel_cls::THINNED_EDGE_REVERSE_DIAGONAL_SHAPE
ImagePixel_cls::GetThinnedEdgeReverseDiagonalShape() const
```

```
{
        return m_ThinnedEdgeShape.m_ReverseDiagonalShape;
}

void ImagePixel_cls::SetThinnedEdgeType(THINNED_EDGE_TYPE type)
{
    m_ThinnedEdgeType = type;
}

void
ImagePixel_cls::SetThinnedEdgeHorizontalShape(THINNED_EDGE_HORIZONTA
L_SHAPE shape)
{
    m_ThinnedEdgeShape.m_HorizontalShape = shape;
}

void
ImagePixel_cls::SetThinnedEdgeVerticalShape(THINNED_EDGE_VERTICAL_SH
APE shape)
{
    m_ThinnedEdgeShape.m_VerticalShape = shape;
}

void
ImagePixel_cls::SetThinnedEdgeDiagonalShape(THINNED_EDGE_DIAGONAL_SH
APE shape)
{
    m_ThinnedEdgeShape.m_DiagonalShape = shape;
}

void
ImagePixel_cls::SetThinnedEdgeReverseDiagonalShape(THINNED_EDGE_REVE
RSE_DIAGONAL_SHAPE shape)
{
    m_ThinnedEdgeShape.m_ReverseDiagonalShape = shape;
}
```

Mst_cls.h

```
#ifndef MST_CLS_H
#define MST_CLS_H

#include "CommonTypes.h"

class Mst_cls
{
public:
    Mst_cls(Graph_cls& graph);
    ~Mst_cls();

    MstElement_stc& operator[](int index) const;

private:
    Graph_cls& m_Graph;

    const int m_GraphSize;
    const int m_TreeSize;

    MstElement_stc** m_pElements;

};

#endif //MST_CLS_H
```

Mst_cls.cpp

```
#include "stdafx.h"
#include "Graph_cls.h"
#include "Vertex_cls.h"
#include "Mst_cls.h"
#include "math.h"

Mst_cls::Mst_cls(Graph_cls& graph):
m_Graph(graph),
m_GraphSize(graph.GetSize()),
m_TreeSize(m_GraphSize)
{
   ASSERT (m_TreeSize == graph.GetSize());

   //construct MST
   m_pElements = new MstElement_stc*[ m_TreeSize];

   //add seed vertex
   const int startingVertex = 0;// m_GraphSize/2 +
(0.5*sqrt(m_GraphSize));
   Vertex_cls* vertexToBeAddedToMst = &graph[ startingVertex];
   Vertex_cls* previousVertexInMst = NULL;
   vertexToBeAddedToMst->AddToMst(ED_UNDEFINED, NULL);
   EdgeDirection_enm direction = vertexToBeAddedToMst-
>GetMinIncidentDirection();
   int minWeight = vertexToBeAddedToMst->GetMinIncidentEdgeWeight();

   m_pElements[ 0] = new MstElement_stc(*vertexToBeAddedToMst,
minWeight, direction);

   for (int counter=1; counter<m_TreeSize; counter++)
   {
      direction = ED_UNDEFINED;
      vertexToBeAddedToMst = NULL;
      minWeight = INT_MAX;
            int vertex;
      for (vertex=0; vertex<counter; vertex++)
      {
         if(minWeight > m_pElements[ vertex] -
>m_Vertex.GetMinIncidentEdgeWeight())
         {
            minWeight =  m_pElements[ vertex] -
>m_Vertex.GetMinIncidentEdgeWeight();
            vertexToBeAddedToMst =  m_pElements[ vertex] -
>m_Vertex.GetMinIncidentVertex();
            direction = m_pElements[ vertex] -
>m_Vertex.GetMinIncidentDirection();
            previousVertexInMst = &m_pElements[ vertex] ->m_Vertex;
         }
      }

      ASSERT (direction != ED_UNDEFINED);
      ASSERT (vertexToBeAddedToMst != NULL);
      ASSERT (!vertexToBeAddedToMst->IsAddedToMst());
      ASSERT ( minWeight != INT_MAX);

      m_pElements[ counter] = new
MstElement_stc(*vertexToBeAddedToMst, minWeight, direction);
      m_pElements[ counter] ->m_Vertex.AddToMst(direction,
previousVertexInMst);

      //now only maximum of four neighbours are updated with
```

```
OnVertexAddedToMst
        //this is done in the AddToMst function called above
        //for (vertex=0; vertex<counter; vertex++)
        //{
        //    m_pElements[ vertex] -
>m_Vertex.OnVertexAddedToMst(m_pElements[ counter] ->m_Vertex);
        //}
    }

}

Mst_cls::~Mst_cls()
{
    for (int counter=0; counter<m_TreeSize; counter++)
    {
        delete m_pElements[ counter] ;
    }

    delete[ ]  m_pElements;
}

MstElement_stc& Mst_cls::operator[ ] (int index) const
{
    return *(m_pElements[ index] );
}
```

NullPixel_cls.h

```
#ifndef NULLPIXEL_CLS_H
#define NULLPIXEL_CLS_H

#include "Pixel_cls.h"

class NullPixel_cls : public Pixel_cls
{
public:
   //constructors
   NullPixel_cls(const int value = 0) : Pixel_cls(value) {}
   NullPixel_cls(const Pixel_cls& pixel) : Pixel_cls(pixel) {}

   int GetAssembledValue() const
   {
      return 0;
   }
};

#endif //NULLPIXEL_CLS_H
```

Pixel_cls.h

```
#ifndef PIXEL_CLS
#define PIXEL_CLS

#include "Vertex_cls.h"

class Pixel_cls:
   public Vertex_cls
{
public:

   //constructors
   Pixel_cls(const int value = 0);
   Pixel_cls(const Pixel_cls& pixel);

   //destructors
   virtual ~Pixel_cls();

   //set/get commands
   void SetValue(int value);
   int GetValue() const;

   //operators, assignment
   Pixel_cls& operator=(const Pixel_cls& pixel);
   Pixel_cls& operator=(int value);

   //operators, comparison
   bool operator==(const Pixel_cls& pixel) const;
   bool operator==(int value) const;
   bool operator<(const Pixel_cls& pixel) const;
   bool operator<(int value) const;
   bool operator>(const Pixel_cls& pixel) const;
   bool operator>(int value) const;
   bool operator<=(const Pixel_cls& pixel) const;
   bool operator<=(int value) const;
   bool operator>=(const Pixel_cls& pixel) const;
   bool operator>=(int value) const;
   bool operator!=(const Pixel_cls& pixel) const;
   bool operator!=(int value) const;

   //arithmetic operators
   int operator+(int value) const;
   friend int operator+(const int value, const Pixel_cls& pixel);
   friend int operator+(const Pixel_cls& leftPixel, const Pixel_cls&
rightPixel);
   int operator-(int value) const;
   friend int operator-(const int value, const Pixel_cls& pixel);
   friend int operator-(const Pixel_cls& leftPixel, const Pixel_cls&
rightPixel);
   int operator*(int value) const;
   friend int operator*(const int value, const Pixel_cls& pixel);

   //cast operators
   operator int() const;

   //serialisation
   void Serialise(CArchive archive);
   void Deserialise(CArchive archive);

   //image porcessing operations
   void SetFourNeighbours(Pixel_cls *const pNorth,
                          Pixel_cls *const pWest,
```

```
                                        Pixel_cls *const pEast,
                                        Pixel_cls *const pSouth);

void SetNineNeigbours(Pixel_cls *const pNorthWest,
                      Pixel_cls *const pNorth,
                      Pixel_cls *const pNorthEast,
                      Pixel_cls *const pWest,
                      Pixel_cls *const pEast,
                      Pixel_cls *const pSouthWest,
                      Pixel_cls *const pSouth,
                      Pixel_cls *const pSouthEast);
int GetMedianValue() const;
Pixel_cls GetMedianPixel() const;
const Pixel_cls& GetMedianNeighbour() const;
int GetHighPassFilteredValue() const;
int GetLowPassFilteredValue() const;
int GetVerticalSobelValue() const;
int GetHorizontalSobelValue() const;
ImagePixel_cls* GetEdgeInfo() const;
void SetEdgeInfo(ImagePixel_cls* pEdge);
void SetHighEdge(bool isEdge);
bool IsHighEdge() const;
void SetLowEdge(bool isEdge);
bool IsLowEdge() const;
void SetHystarisisEdge(bool isEdge);
bool IsHystarisisEdge() const;

//Vertex_cls interface support and
//Minimum Spanning Tree operations
void SetNorthWeight(int weight);
void SetWestWeight(int weight);
void SetEastWeight(int weight);
void SetSouthWeight(int weight);
   int GetNorthWeight() const;
   int GetWestWeight() const;
   int GetEastWeight() const;
   int GetSouthWeight() const;
   int GetMinIncidentEdgeWeight() const;
   Vertex_cls* GetMinIncidentVertex() const;
   EdgeDirection_enm GetMinIncidentDirection() const;
   virtual int GetMinIncidentEdgeWeight_InMstNeighbourhood() const;
   virtual Vertex_cls* GetMinIncidentVertex_InMstNeighbourhood()
const;
   virtual EdgeDirection_enm
GetMinIncidentDirection_InMstNeighbourhood() const;
   bool IsAddedToMst() const;
   void AddToMst(EdgeDirection_enm direction, const Vertex_cls*
previousVertexInMst);
   void OnVertexAddedToMst(Vertex_cls& vertex);
   void SetUnwrappedValue(int value);
   int GetUnwrappedValue() const;
   const Vertex_cls* GetPreviousVertexInMst() const;
EdgeDirection_enm GetMstDirection() const;
int GetVertexValue() const;

void SetAssembledValue(int assembledValue);
virtual int GetAssembledValue() const;
void SetPhaseJumpsCount(int phaseJumpsCount);
int GetPhaseJumpsCount() const;
bool IsToHaveMaxWeight() const;

void RuleInMinEdge();
void RuleOutMaxDualEdge();
void EnsureConnectedToMinEdge();
void EnsureConnectedToDualMaxEdge();
```

```
private:
    void RefreshMinIncidentVertex();

    int m_Value;
    ImagePixel_cls* m_pEdgeInfo;
    bool m_IsHighEdge;
    bool m_IsLowEdge;
    bool m_IsHystarisisEdge;
    bool m_HasBeenUnwrapped;
    int m_PhaseJumpsCount;

    //Vertex_cls interface support
    bool m_IsAddedToMst;
    EdgeDirection_enm m_MstDirection;
    Vertex_cls* m_pNullVertex;
    Vertex_cls* m_pMinIncidentVertex;
    EdgeDirection_enm m_MinIncidentDirection;
    int m_MinIncidentEdgeWeight;
    Vertex_cls* m_pMinIncidentVertex_InMstNeighbourhood;
    EdgeDirection_enm m_MinIncidentDirection_InMstNeighbourhood;
    int m_MinIncidentEdgeWeight_InMstNeighbourhood;
    int m_UnwrappedValue;
    const Vertex_cls* m_pPreviousVertexInMst;

    int m_AssembledValue;

    int m_NorthWeight;
    int m_WestWeight;
    int m_EastWeight;
    int m_SouthWeight;

    Pixel_cls *const m_pNorthWestPixel;
    Pixel_cls *const m_pNorthPixel;
    Pixel_cls *const m_pNorthEastPixel;
    Pixel_cls *const m_pWestPixel;
    Pixel_cls *const m_pEastPixel;
    Pixel_cls *const m_pSouthWestPixel;
    Pixel_cls *const m_pSouthPixel;
    Pixel_cls *const m_pSouthEastPixel;

    enum EdgeType_enum
    {
        RULED_IN,
        RULED_OUT,
        INDETERMINATE
    };

    EdgeType_enum m_EdgeTpe;

};

#endif //PIXEL_CLS
```

Pixel_cls.cpp

```
#include <stdafx.h>
#include "Pixel_cls.h"

//constructors
Pixel_cls::Pixel_cls(const int value):
m_Value(value),
m_UnwrappedValue(value),
m_pEdgeInfo(NULL),
m_IsHighEdge(false),
m_IsLowEdge(false),
m_IsHystarisisEdge(false),
m_IsAddedToMst(false),
m_MstDirection(ED_UNDEFINED),
m_pNullVertex(NULL),
m_pMinIncidentVertex(m_pNullVertex),
m_MinIncidentDirection(ED_UNDEFINED),
m_MinIncidentEdgeWeight(INT_MAX),
m_pMinIncidentVertex_InMstNeighbourhood(m_pNullVertex),
m_MinIncidentDirection_InMstNeighbourhood(ED_UNDEFINED),
m_MinIncidentEdgeWeight_InMstNeighbourhood(INT_MAX),
m_pPreviousVertexInMst(NULL),
m_pNorthWestPixel(NULL),
m_pNorthPixel(NULL),
m_pNorthEastPixel(NULL),
m_pWestPixel(NULL),
m_pEastPixel(NULL),
m_pSouthWestPixel(NULL),
m_pSouthPixel(NULL),
m_pSouthEastPixel(NULL),
m_NorthWeight(0),
m_WestWeight(0),
m_EastWeight(0),
m_SouthWeight(0),
m_HasBeenUnwrapped(false),
m_AssembledValue(0),
m_PhaseJumpsCount(0),
m_EdgeTpe(INDETERMINATE)
{
}

Pixel_cls::Pixel_cls(const Pixel_cls& pixel):
m_Value(pixel.m_Value),
m_UnwrappedValue(pixel.m_UnwrappedValue),
m_pEdgeInfo(pixel.m_pEdgeInfo),
m_IsHighEdge(pixel.m_IsHighEdge),
m_IsLowEdge(pixel.m_IsLowEdge),
m_IsHystarisisEdge(pixel.m_IsHystarisisEdge),
m_IsAddedToMst(pixel.m_IsAddedToMst),
m_MstDirection(pixel.m_MstDirection),
m_pNullVertex(pixel.m_pNullVertex),
m_pMinIncidentVertex(pixel.m_pMinIncidentVertex),
m_MinIncidentDirection(pixel.m_MinIncidentDirection),
m_MinIncidentEdgeWeight(pixel.m_MinIncidentEdgeWeight),
m_pPreviousVertexInMst(pixel.m_pPreviousVertexInMst),
m_pNorthWestPixel(pixel.m_pNorthWestPixel),
m_pNorthPixel(pixel.m_pNorthPixel),
m_pNorthEastPixel(pixel.m_pNorthEastPixel),
m_pWestPixel(pixel.m_pWestPixel),
m_pEastPixel(pixel.m_pEastPixel),
m_pSouthWestPixel(pixel.m_pSouthWestPixel),
m_pSouthPixel(pixel.m_pSouthPixel),
```

```
m_pSouthEastPixel(pixel.m_pSouthEastPixel),
m_NorthWeight(pixel.m_NorthWeight),
m_WestWeight(pixel.m_WestWeight),
m_EastWeight(pixel.m_EastWeight),
m_SouthWeight(pixel.m_SouthWeight),
m_HasBeenUnwrapped(pixel.m_HasBeenUnwrapped),
m_AssembledValue(pixel.m_AssembledValue),
m_PhaseJumpsCount(pixel.m_PhaseJumpsCount),
m_EdgeTpe(pixel.m_EdgeTpe)
{
}

//destructors
Pixel_cls::~Pixel_cls()
{
}

//set/get commands
void Pixel_cls::SetValue(int value)
{
    m_Value = value;
}

int Pixel_cls::GetValue() const
{
    return m_Value;
}

//operators, assignment
Pixel_cls& Pixel_cls::operator=(const Pixel_cls& pixel)
{
    m_Value = pixel.m_Value;
    return myself;
}

Pixel_cls& Pixel_cls::operator=(int value)
{
    m_Value = value;
    return myself;
}


//operators, comparison
bool Pixel_cls::operator==(const Pixel_cls& pixel) const
{
    return m_Value == pixel.m_Value;
}

bool Pixel_cls::operator==(int value) const
{
    return m_Value == value;
}

bool Pixel_cls::operator<(const Pixel_cls& pixel) const
{
    return m_Value < pixel.m_Value;
}

bool Pixel_cls::operator<(int value) const
{
    return m_Value < value;
}

bool Pixel_cls::operator>(const Pixel_cls& pixel) const
{
```

```
    return m_Value > pixel.m_Value;
}

bool Pixel_cls::operator>(int value) const
{
    return m_Value > value;
}

bool Pixel_cls::operator<=(const Pixel_cls& pixel) const
{
    return m_Value <= pixel.m_Value;
}

bool Pixel_cls::operator<=(int value) const
{
    return m_Value <= value;
}

bool Pixel_cls::operator>=(const Pixel_cls& pixel) const
{
    return m_Value >= pixel.m_Value;
}

bool Pixel_cls::operator>=(int value) const
{
    return m_Value >= value;
}

bool Pixel_cls::operator!=(const Pixel_cls& pixel) const
{
    return m_Value != pixel.m_Value;
}

bool Pixel_cls::operator!=(int value) const
{
    return m_Value != value;
}


//arithmetic operators
int Pixel_cls::operator+(int value) const
{
    return (m_Value + value);
}

int operator+(const int value, const Pixel_cls& pixel)
{
    return (value + pixel.m_Value);
}

int operator+(const Pixel_cls& leftPixel, const Pixel_cls&
rightPixel)
{
    return (leftPixel.m_Value + rightPixel.m_Value);
}

int Pixel_cls::operator-(int value) const
{
    return (m_Value - value);
}

int operator-(const int value, const Pixel_cls& pixel)
{
    return (value - pixel.m_Value);
}
```

```
int operator-(const Pixel_cls& leftPixel, const Pixel_cls&
rightPixel)
{
    return (leftPixel.m_Value - rightPixel.m_Value);
}

int Pixel_cls::operator*(int value) const
{
    return (m_Value * value);
}

int operator*(const int value, const Pixel_cls& pixel)
{
    return (value * pixel.m_Value);
}


//cast operators
Pixel_cls::operator int() const
{
    return m_Value;
}


//serialisation
void Pixel_cls::Serialise(CArchive archive)
{
}

void Pixel_cls::Deserialise(CArchive archive)
{
}

//image porcessing operations
void Pixel_cls::SetFourNeighbours(Pixel_cls *const pNorth,
                                  Pixel_cls *const pWest,
                                  Pixel_cls *const pEast,
                                  Pixel_cls *const pSouth)
{
    (Pixel_cls*) m_pNorthPixel = pNorth;
    (Pixel_cls*) m_pWestPixel = pWest;
    (Pixel_cls*) m_pEastPixel = pEast;
    (Pixel_cls*) m_pSouthPixel = pSouth;
}

void Pixel_cls::SetNineNeigbours(Pixel_cls *const pNorthWest,
                                 Pixel_cls *const pNorth,
                                 Pixel_cls *const pNorthEast,
                                 Pixel_cls *const pWest,
                                 Pixel_cls *const pEast,
                                 Pixel_cls *const pSouthWest,
                                 Pixel_cls *const pSouth,
                                 Pixel_cls *const pSouthEast)
{
    (Pixel_cls*) m_pNorthPixel = pNorthWest;
    (Pixel_cls*) m_pNorthPixel = pNorth;
    (Pixel_cls*) m_pNorthPixel = pNorthEast;
    (Pixel_cls*) m_pWestPixel = pWest;
    (Pixel_cls*) m_pEastPixel = pEast;
    (Pixel_cls*) m_pSouthPixel = pSouthWest;
    (Pixel_cls*) m_pSouthPixel = pSouth;
    (Pixel_cls*) m_pSouthPixel = pSouthEast;
}
```

```
int Pixel_cls::GetMedianValue() const
{
   return GetMedianNeighbour().GetValue();
}

Pixel_cls Pixel_cls::GetMedianPixel() const
{
   return GetMedianNeighbour();
}

const Pixel_cls& Pixel_cls::GetMedianNeighbour() const
{
   const int count = 5;
   const Pixel_cls* neighbourBuff[ count]  = { this, m_pNorthPixel,
m_pWestPixel, m_pEastPixel, m_pSouthPixel} ;
   CList<const Pixel_cls*, const Pixel_cls*> orderedList;

      for (UINT counter=0; counter<count; counter++)
      {
            POSITION currentPosition =
orderedList.GetHeadPosition();
            while (currentPosition != NULL)
            {
                  if (orderedList.GetAt(currentPosition)->GetValue()
< neighbourBuff[ counter] ->GetValue())
                  {
                        orderedList.GetNext(currentPosition);
                  }
                  else
                  {
                        break; // out of the for loop
                  }
            }
            orderedList.InsertBefore(currentPosition,
neighbourBuff[ counter] );
      }

   return *orderedList.GetAt(orderedList.FindIndex(count/2));
}

int Pixel_cls::GetHighPassFilteredValue() const
{
   // 0  -1  0
   //-1   5 -1
   // 0  -1  0

   if (m_pNorthPixel && m_pWestPixel && m_pEastPixel &&
m_pSouthPixel)
   {
      return ((5 * m_Value) - *m_pNorthPixel - *m_pWestPixel -
*m_pEastPixel - *m_pSouthPixel);
   }
   else
   {
      return m_Value;
   }
}

int Pixel_cls::GetLowPassFilteredValue() const
{
   // 0  1  0    1
   // 1  1  1 *( - )
   // 0  1  0    5
   if (m_pNorthPixel && m_pWestPixel && m_pEastPixel &&
m_pSouthPixel)
```

```
    {
        return int((1.0/5.0) * (m_Value + *m_pNorthPixel +
*m_pWestPixel + *m_pEastPixel + *m_pSouthPixel));
    }
    else
    {
        return m_Value;
    }
}

int Pixel_cls::GetVerticalSobelValue() const
{
    // ySobel = (sW + (2 * sS) + sE) - (nW + (2 * nN) + nE)
    return ((*m_pSouthWestPixel + (2 * *m_pSouthPixel) +
*m_pSouthEastPixel) - (*m_pNorthWestPixel + (2 * *m_pNorthPixel) +
*m_pNorthEastPixel));
}

int Pixel_cls::GetHorizontalSobelValue() const
{
    // xSobel = (nW + (2 * wW) + sW) - (nE + (2 * eE) + sE)
    return ((*m_pNorthWestPixel + (2 * *m_pWestPixel) +
*m_pSouthWestPixel) - (*m_pNorthEastPixel + (2 * *m_pEastPixel) +
*m_pSouthEastPixel));
}

ImagePixel_cls* Pixel_cls::GetEdgeInfo() const
{
    ASSERT (m_pEdgeInfo != NULL);

    return m_pEdgeInfo;
}

void Pixel_cls::SetEdgeInfo(ImagePixel_cls* pEdge)
{
    m_pEdgeInfo = pEdge;
}

void Pixel_cls::SetHighEdge(bool isEdge)
{
    m_IsHighEdge = isEdge;
}

bool Pixel_cls::IsHighEdge() const
{
    return m_IsHighEdge;
}

void Pixel_cls::SetLowEdge(bool isEdge)
{
    m_IsLowEdge = isEdge;
}

bool Pixel_cls::IsLowEdge() const
{
    return m_IsLowEdge;
}

void Pixel_cls::SetHystarisisEdge(bool isEdge)
{
    m_IsHystarisisEdge = isEdge;
}

bool Pixel_cls::IsHystarisisEdge() const
{
```

```
    return m_IsHystarisisEdge;
}


//Minimum Spanning Tree operations
void Pixel_cls::SetNorthWeight(int weight)
{
    m_NorthWeight = weight;
}

void Pixel_cls::SetWestWeight(int weight)
{
    m_WestWeight = weight;
}

void Pixel_cls::SetEastWeight(int weight)
{
    m_EastWeight = weight;
}

void Pixel_cls::SetSouthWeight(int weight)
{
    m_SouthWeight = weight;
}

int Pixel_cls::GetNorthWeight() const
{
    return m_NorthWeight;
}

int Pixel_cls::GetWestWeight() const
{
    return m_WestWeight;
}

int Pixel_cls::GetEastWeight() const
{
    return m_EastWeight;
}

int Pixel_cls::GetSouthWeight() const
{
    return m_SouthWeight;
}

int Pixel_cls::GetMinIncidentEdgeWeight() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentEdgeWeight;
}

Vertex_cls* Pixel_cls::GetMinIncidentVertex() const
{
    ASSERT (m_IsAddedToMst);

    return m_pMinIncidentVertex;
}

EdgeDirection_enm Pixel_cls::GetMinIncidentDirection() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentDirection;
}
```

```
bool Pixel_cls::IsAddedToMst() const
{
    return m_IsAddedToMst;
}

int Pixel_cls::GetMinIncidentEdgeWeight_InMstNeighbourhood() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentEdgeWeight_InMstNeighbourhood;
}

Vertex_cls* Pixel_cls::GetMinIncidentVertex_InMstNeighbourhood()
const
{
    ASSERT (m_IsAddedToMst);

    return m_pMinIncidentVertex_InMstNeighbourhood;
}

EdgeDirection_enm
Pixel_cls::GetMinIncidentDirection_InMstNeighbourhood() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentDirection_InMstNeighbourhood;
}

void Pixel_cls::AddToMst(EdgeDirection_enm direction, const
Vertex_cls* previousVertexInMst)
{
/*    Precondition
(
    if (previousVertexInMst == NULL)
    {
        //must be seed vertex in MST
        ASSERT (direction == ED_UNDEFINED);
    }
    else
    {
        switch (direction)
        {
            case ED_NORTH:
            {
                ASSERT (previousVertexInMst == m_pSouthPixel);
                break;
            }
            case ED_WEST:
            {
                ASSERT (previousVertexInMst == m_pEastPixel);
                break;
            }
case ED_EAST:
{
    ASSERT (previousVertexInMst == m_pWestPixel);
    break;
}
case ED_SOUTH:
{
    ASSERT (previousVertexInMst == m_pNorthPixel);
    break;
}
default:
{
```

```
                    //must be one of the above
                    ASSERT (FALSE);
                }
            }
        }

    )
*/
    m_IsAddedToMst = true;
    m_MstDirection = direction;
    m_pPreviousVertexInMst = previousVertexInMst;

    //notify neighbouring vertices
    if (m_pNorthPixel)
    {
        m_pNorthPixel->OnVertexAddedToMst(myself);
    }
    if (m_pWestPixel)
    {
        m_pWestPixel->OnVertexAddedToMst(myself);
    }
    if (m_pEastPixel)
    {
        m_pEastPixel->OnVertexAddedToMst(myself);
    }
    if (m_pSouthPixel)
    {
        m_pSouthPixel->OnVertexAddedToMst(myself);
    }

    //RefreshMinIncidentVertex() must be called after neighbouring
    //vertices have been updated with OnVertexAddedToMst(), otherwise
    //m_pMinIncidentVertex_InMstNeighbourhood will not necessarely be
    //found correctly
    RefreshMinIncidentVertex();
}

EdgeDirection_enm Pixel_cls::GetMstDirection() const
{
    ASSERT (m_IsAddedToMst);

    return m_MstDirection;
}

void Pixel_cls::OnVertexAddedToMst(Vertex_cls& vertex)
{
    if (&vertex == m_pMinIncidentVertex)
    {
        RefreshMinIncidentVertex();
    }
}

void Pixel_cls::RefreshMinIncidentVertex()
{
ASSERT (m_IsAddedToMst);
m_MinIncidentEdgeWeight = INT_MAX;
m_pMinIncidentVertex = m_pNullVertex;
m_MinIncidentDirection = ED_UNDEFINED;

m_MinIncidentEdgeWeight_InMstNeighbourhood = INT_MAX;
m_pMinIncidentVertex_InMstNeighbourhood = m_pNullVertex;
m_MinIncidentDirection_InMstNeighbourhood = ED_UNDEFINED;

if (m_pNorthPixel)
{
```

```
        if (!m_pNorthPixel->IsAddedToMst())
        {
            m_MinIncidentEdgeWeight = m_NorthWeight;
            m_pMinIncidentVertex = (Pixel_cls*)m_pNorthPixel;
            m_MinIncidentDirection = ED_NORTH;
        }
        else
        {
            m_MinIncidentEdgeWeight_InMstNeighbourhood = m_pNorthPixel-
>GetMinIncidentEdgeWeight();
            m_pMinIncidentVertex_InMstNeighbourhood = m_pNorthPixel-
>GetMinIncidentVertex();
            m_MinIncidentDirection_InMstNeighbourhood = m_pNorthPixel-
>GetMinIncidentDirection();
        }
    }

    if (m_pWestPixel)
    {
        if (!m_pWestPixel->IsAddedToMst())
        {
            if (m_WestWeight < m_MinIncidentEdgeWeight)
            {
                m_MinIncidentEdgeWeight = m_WestWeight;
                m_pMinIncidentVertex = (Pixel_cls*)m_pWestPixel;
                m_MinIncidentDirection = ED_WEST;
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pWestPixel->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pWestPixel->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pWestPixel-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood =
m_pWestPixel->GetMinIncidentDirection();
            }
        }
    }

    if (m_pEastPixel)
    {
        if (!m_pEastPixel->IsAddedToMst())
        {
            if (m_EastWeight < m_MinIncidentEdgeWeight)
            {
                m_MinIncidentEdgeWeight = m_EastWeight;
                m_pMinIncidentVertex = (Pixel_cls*)m_pEastPixel;
                m_MinIncidentDirection = ED_EAST;
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pEastPixel->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pEastPixel->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pEastPixel-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood =
m_pEastPixel->GetMinIncidentDirection();
```

```
            }
        }
    }

    if (m_pSouthPixel)
    {
        if (!m_pSouthPixel->IsAddedToMst())
        {
            if (m_SouthWeight < m_MinIncidentEdgeWeight)
            {
                m_MinIncidentEdgeWeight = m_SouthWeight;
                m_pMinIncidentVertex = (Pixel_cls*)m_pSouthPixel;
                m_MinIncidentDirection = ED_SOUTH;
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pSouthPixel->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pSouthPixel->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pSouthPixel-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood =
m_pSouthPixel->GetMinIncidentDirection();
            }
        }
    }

    if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_MinIncidentEdgeWeight)
    {
        m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_MinIncidentEdgeWeight;
        m_pMinIncidentVertex_InMstNeighbourhood =
m_pMinIncidentVertex;
        m_MinIncidentDirection_InMstNeighbourhood =
m_MinIncidentDirection;
    }
}

void Pixel_cls::SetUnwrappedValue(int value)
{
//    ASSERT (m_HasBeenUnwrapped == false);

    m_HasBeenUnwrapped = true;

    m_UnwrappedValue = value;
}

int Pixel_cls::GetUnwrappedValue() const
{
    return m_UnwrappedValue;
}

const Vertex_cls* Pixel_cls::GetPreviousVertexInMst() const
{
    return m_pPreviousVertexInMst;
}

int operator+(const Vertex_cls& leftVertex, const Vertex_cls&
rightVertex)
{
    return leftVertex.GetVertexValue() +
```

```
rightVertex.GetVertexValue();
}

int operator-(const Vertex_cls& leftVertex, const Vertex_cls&
rightVertex)
{
   return leftVertex.GetVertexValue() -
rightVertex.GetVertexValue();
}

int Pixel_cls::GetVertexValue() const
{
   return m_Value;
}

void Pixel_cls::SetAssembledValue(int assembledValue)
{
   ASSERT(m_HasBeenUnwrapped);

   m_AssembledValue = assembledValue;
}

int Pixel_cls::GetAssembledValue() const
{
   ASSERT(m_HasBeenUnwrapped);

   return m_AssembledValue;
}

void Pixel_cls::SetPhaseJumpsCount(int phaseJumpsCount)
{
   m_PhaseJumpsCount = phaseJumpsCount;
}

int Pixel_cls::GetPhaseJumpsCount() const
{
   return m_PhaseJumpsCount;
}

bool Pixel_cls::IsToHaveMaxWeight() const
{
   return myself.GetEdgeInfo()->DoesHaveEdge();
}
```

Tile_cls.h

```
#ifndef TILE_CLS_H
#define TILE_CLS_H

#include "Image_cls.h"
#include "Vertex_cls.h"

class Graph_cls;
class Mst_cls;

class Tile_cls :
   private Image_cls,
   public Vertex_cls
{
public:
   Tile_cls(int size, int overlapSize);
   virtual ~Tile_cls();

   void ConstructWeightedGraph();
   void ConstructMst();
   void UnwrapAlongMstPath();
   void SetFourNeighbours(Tile_cls *const pNorth,
                          Tile_cls *const pWest,
                          Tile_cls *const pEast,
                          Tile_cls *const pSouth);

   Pixel_cls& operator[](int index) const;

   void NormaliseAssembledValues(int actualRange,
                                 int streachedRange,
                                 int minAssembledValue);

//Vertex_cls interface support and
//Minimum Spanning Tree operations
void SetNorthWeight(int weight);
void SetWestWeight(int weight);
void SetEastWeight(int weight);
void SetSouthWeight(int weight);
int GetNorthWeight() const;
int GetWestWeight() const;
int GetEastWeight() const;
int GetSouthWeight() const;
ImagePixel_cls* GetEdgeInfo() const;
void SetEdgeInfo(ImagePixel_cls* pEdge);
void SetPhaseJumpsCount(int phaseJumpsCount);
int GetPhaseJumpsCount() const;
int GetMinIncidentEdgeWeight() const;
   Vertex_cls* GetMinIncidentVertex() const;
   EdgeDirection_enm GetMinIncidentDirection() const;
   virtual int GetMinIncidentEdgeWeight_InMstNeighbourhood() const;
   virtual Vertex_cls* GetMinIncidentVertex_InMstNeighbourhood()
const;
   virtual EdgeDirection_enm
GetMinIncidentDirection_InMstNeighbourhood() const;
   bool IsAddedToMst() const;
   void AddToMst(EdgeDirection_enm direction, const Vertex_cls*
previousVertexInMst);
   void OnVertexAddedToMst(Vertex_cls& vertex);
   void SetUnwrappedValue(int assemblyOffset);
   int GetUnwrappedValue() const;
   const Vertex_cls* GetPreviousVertexInMst() const;
```

```
EdgeDirection_enm GetMstDirection() const;
int GetVertexValue() const;
int GetMinAssembledValue() const;
int GetMaxAssembledValue() const;
int GetNorthAssembelyOffset() const;
int GetWestAssembelyOffset() const;
int GetEastAssembelyOffset() const;
int GetSouthAssembelyOffset() const;
bool DoesContainEdges() const;
void SetDoesContainEdges(bool doesContainEdges);
bool IsToHaveMaxWeight() const;

BITMAPINFO* GetWrappedDib();
   BITMAPINFO* GetWeightsDib();
   BITMAPINFO* GetMstDib();
   BITMAPINFO* GetPhaseJumpsDib();
   BITMAPINFO* GetUnwrappedDib();

   CStdioFile& GetPhaseJumpsText(CString& fileNamePrefix);

private:

   void SetupTileNeighbourhoods();
   void NormaliseUnwrappedValues(int actualRange, int
streachedRange, int minAssembledValue);
   void CalculateTileValue();
   void RefreshMinIncidentVertex();
   void DestroyDib(BITMAPINFO* pDib);
   int GetPhaseJump(EdgeDirection_enm unwrapDirection, Vertex_cls&
pixelToUnwrap);

   const int m_Size;
   const int m_OverlapSize;
   const int m_PixelArraySize;

   const int& P; // )
   const int& T; // )> aliases, handy for equation-like
manipulations

   int m_MinUnwrappedValue;
   int m_MaxUnwrappedValue;

   Graph_cls* m_pGraph;
   Mst_cls* m_pMst;
   bool m_HasBeenUnwrapped;
   bool m_HasBeenAssembled;
   bool m_DoesContainEdges;

   //Vertex_cls interface support
   int m_Value;
   bool m_IsAddedToMst;
   EdgeDirection_enm m_MstDirection;
   Vertex_cls* m_pNullVertex;
   Vertex_cls* m_pMinIncidentVertex;
   EdgeDirection_enm m_MinIncidentDirection;
   Vertex_cls* m_pMinIncidentVertex_InMstNeighbourhood;
   EdgeDirection_enm m_MinIncidentDirection_InMstNeighbourhood;
   int m_MinIncidentEdgeWeight_InMstNeighbourhood;
   int m_MinIncidentEdgeWeight;
   int m_UnwrappedValue;
   const Vertex_cls* m_pPreviousVertexInMst;
   int m_MinAssembledValue;
   int m_MaxAssembledValue;
   mutable int m_NorthAssembelyOffset;
   mutable int m_WestAssembelyOffset;
```

```
mutable int m_EastAssembelyOffset;
mutable int m_SouthAssembelyOffset;

int m_NorthWeight;
int m_WestWeight;
int m_EastWeight;
int m_SouthWeight;

Tile_cls *const m_pNorthTile;
Tile_cls *const m_pWestTile;
Tile_cls *const m_pEastTile;
Tile_cls *const m_pSouthTile;

   BITMAPINFO* m_pWrappedDib;
   BITMAPINFO* m_pWeightDib;
   BITMAPINFO* m_pMstDib;
   BITMAPINFO* m_pPhaseJumpsDib;
   BITMAPINFO* m_pUnwrappedDib;

   bool m_DoesPhaseJumpsTextFileExist;
   CStdioFile m_PhaseJumpsTextFile;
};

#endif //TILE_CLS_H
```

Tile_cls.cpp

```
#include "stdafx.h"
#include "Pixel_cls.h"
#include "Graph_cls.h"
#include "Mst_cls.h"
#include "Tile_cls.h"

Tile_cls::Tile_cls(int size, int overlapSize):
Image_cls(size, size),
m_Size(size),
m_OverlapSize(overlapSize),
m_PixelArraySize(size * size),
P(m_OverlapSize),
T(m_Size),
m_pGraph(NULL),
m_pMst(NULL),
m_Value(0),
m_IsAddedToMst(false),
m_MstDirection(ED_UNDEFINED),
m_pNullVertex(NULL),
m_pMinIncidentVertex(m_pNullVertex),
m_MinIncidentDirection(ED_UNDEFINED),
m_MinIncidentEdgeWeight(INT_MAX),
m_pPreviousVertexInMst(NULL),
m_pNorthTile(NULL),
m_pWestTile(NULL),
m_pEastTile(NULL),
m_pSouthTile(NULL),
m_NorthWeight(0),
m_WestWeight(0),
m_EastWeight(0),
m_SouthWeight(0),
m_HasBeenUnwrapped(false),
m_HasBeenAssembled(false),
m_MinAssembledValue(0),
m_MaxAssembledValue(0),
m_NorthAssembelyOffset(0),
m_WestAssembelyOffset(0),
m_EastAssembelyOffset(0),
m_SouthAssembelyOffset(0),
m_pWrappedDib(NULL),
m_pWeightDib(NULL),
m_pMstDib(NULL),
m_pPhaseJumpsDib(NULL),
m_pUnwrappedDib(NULL),
m_DoesPhaseJumpsTextFileExist(false)
{
}

Tile_cls::~Tile_cls()
{
    if (m_pGraph)
    {
        delete m_pGraph;
        m_pGraph = NULL;
    }
    if (m_pMst)
    {
        delete m_pMst;
        m_pMst = NULL;
    }
}
```

```
void Tile_cls::ConstructWeightedGraph()
{
   ASSERT (m_pGraph == NULL);

   SetupTileNeighbourhoods();
   m_pGraph = new Graph_cls(myself);
}

void Tile_cls::ConstructMst()
{
   ASSERT (m_pMst == NULL);

   m_pMst = new Mst_cls(*m_pGraph);
}

void Tile_cls::SetupTileNeighbourhoods()
{
   for (int counter=0; counter<m_PixelArraySize; counter++)
   {
      int y = counter / m_Size;
      int x = counter - (y * m_Size);

      ASSERT ((x + (y * m_Size) < m_PixelArraySize));

      Pixel_cls* n = NULL;
      Pixel_cls* w = NULL;
      Pixel_cls* e = NULL;
      Pixel_cls* s = NULL;

      if (y > 0)
      {
         n = &myself[ counter - m_Size] ;
      }
      if (x > 0)
      {
         w = &myself[ counter - 1] ;
      }
      if (x < (m_Size - 1))
      {
         e = &myself[ counter + 1] ;
      }
      if (y < (m_Size - 1))
      {
         s = &myself[ counter + m_Size] ;
      }

      myself[ counter] .SetFourNeighbours(n, w, e, s);
   }
}
int Tile_cls::GetPhaseJump(EdgeDirection_enm unwrapDirection,
Vertex_cls& pixelToUnwrap)
{
   return -255;
}

void Tile_cls::UnwrapAlongMstPath()
{
   Precondition
   (
      for (int count=0; count<m_PixelArraySize; count++)
      {
         ASSERT ((*m_pMst)[ count] .m_Vertex.GetPhaseJumpsCount() ==
0);
      }
```

```
)

clock_t start, finish;
start = clock();

m_HasBeenUnwrapped = true;

(*m_pMst)[ 0] .m_Vertex.SetUnwrappedValue((*m_pMst)[ 0] .m_Vertex);
m_MinUnwrappedValue = (*m_pMst)[ 0] .m_Vertex;
m_MaxUnwrappedValue = (*m_pMst)[ 0] .m_Vertex;

//unwrap phase jumps
for (int counter=1; counter<m_PixelArraySize; counter++)
   {
      Vertex_cls& pixelToUnwrap = (*m_pMst)[ counter] .m_Vertex;
      EdgeDirection_enm unwrapDirection =
(*m_pMst)[ counter] .m_Direction;
      const Vertex_cls& previousUnwrappedPixel =
*pixelToUnwrap.GetPreviousVertexInMst();

      if (pixelToUnwrap.GetEdgeInfo()->DoesHaveEdge() &&
!previousUnwrappedPixel.GetEdgeInfo()->DoesHaveEdge())
      {
         //phase jump detected
         //int phaseDifference = pixelToUnwrap <
previousUnwrappedPixel ? 255 : -255;

//pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJu
mpsCount() + phaseDifference);
         int phaseDifference = 255;

         if (unwrapDirection == ED_NORTH)
         {
            if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeVerticalShape() == ImagePixel_cls::NORTH_HIGH)
            {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
            }
            else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeVerticalShape() == ImagePixel_cls::SOUTH_HIGH)
            {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
            }
            else if (previousUnwrappedPixel > pixelToUnwrap)
            {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
            }
            else
            {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
            }
         }
         else if (unwrapDirection == ED_SOUTH)
         {
            if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeVerticalShape() == ImagePixel_cls::SOUTH_HIGH)
            {
```

```
pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeVerticalShape() == ImagePixel_cls::NORTH_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_EAST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeHorizontalShape() == ImagePixel_cls::EAST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeHorizontalShape() == ImagePixel_cls::WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_WEST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeHorizontalShape() == ImagePixel_cls::WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeHorizontalShape() == ImagePixel_cls::EAST_HIGH)
                {
```

```
pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_NORTH_WEST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeDiagonalShape() == ImagePixel_cls::NORTH_WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeDiagonalShape() == ImagePixel_cls::SOUTH_EAST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_SOUTH_EAST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeDiagonalShape() == ImagePixel_cls::SOUTH_EAST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeDiagonalShape() == ImagePixel_cls::NORTH_WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {
```

```
pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_NORTH_EAST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeReverseDiagonalShape() ==
ImagePixel_cls::NORTH_EAST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeReverseDiagonalShape() ==
ImagePixel_cls::SOUTH_WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else if (unwrapDirection == ED_SOUTH_WEST)
            {
                if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeReverseDiagonalShape() ==
ImagePixel_cls::SOUTH_WEST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
                else if (pixelToUnwrap.GetEdgeInfo()-
>GetThinnedEdgeReverseDiagonalShape() ==
ImagePixel_cls::NORTH_EAST_HIGH)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
                else if (previousUnwrappedPixel > pixelToUnwrap)
                {

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() + phaseDifference);
                }
```

```
                else
                {
pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount() - phaseDifference);
                }
            }
            else
            {
                //must be one of the above
                ASSERT(FALSE);
            }

pixelToUnwrap.SetUnwrappedValue(previousUnwrappedPixel.GetUnwrappedV
alue());
            }
            else
            {
                pixelToUnwrap.SetUnwrappedValue(pixelToUnwrap +
previousUnwrappedPixel.GetPhaseJumpsCount());

pixelToUnwrap.SetPhaseJumpsCount(previousUnwrappedPixel.GetPhaseJump
sCount());
            }

            if (m_MinUnwrappedValue > pixelToUnwrap.GetUnwrappedValue())
            {
                m_MinUnwrappedValue = pixelToUnwrap.GetUnwrappedValue();
            }
            if (m_MaxUnwrappedValue < pixelToUnwrap.GetUnwrappedValue())
            {
                m_MaxUnwrappedValue = pixelToUnwrap.GetUnwrappedValue();
            }
        }

        //CalculateTileValue();

        NormaliseUnwrappedValues(m_MaxUnwrappedValue -
m_MinUnwrappedValue,
                                 STREACHED_RANGE,
                                 m_MinUnwrappedValue);
        finish = clock();
        CString message;
        message.Format("unwrapping time = %f", ((double)(finish - start)
/ CLOCKS_PER_SEC));
}

void Tile_cls::CalculateTileValue()
{
    Precondition
    (
        ASSERT (T > (2*P));
    )

    PostCondition
    (
        int accumaltedPixelsInOverlapRegion = 0;
    )

    m_Value = 0;
    const int TOTAL_PIXELS_IN_OVERLAP_REGION = (4*(P*T) - 4*(P*P));

    for (int counter=0; counter<m_PixelArraySize; counter++)
    {
        int y = counter / m_Size;
```

```
        int x = counter - (y * m_Size);

        ASSERT ((x + (y * m_Size) < m_PixelArraySize));

        if (y < P || y >= (T-P) || x < P || x >= (T-P))
        {
            m_Value += myself[ counter] ;

            PostCondition
            (
                accumaltedPixelsInOverlapRegion++;
            )
        }
    }

    m_Value = m_Value / TOTAL_PIXELS_IN_OVERLAP_REGION;

    PostCondition
    (
        ASSERT (accumaltedPixelsInOverlapRegion ==
TOTAL_PIXELS_IN_OVERLAP_REGION);
    )
}

void Tile_cls::NormaliseAssembledValues(int actualRange, int
streachedRange, int minAssembledValue)
{
    for (int counter=0; counter<m_PixelArraySize; counter++)
    {
        ASSERT (myself[ counter] .IsAddedToMst());

        myself[ counter] .SetAssembledValue(
            int(((myself[ counter] .GetAssembledValue() -
            minAssembledValue)*streachedRange) / (double)actualRange));
    }
}

void Tile_cls::NormaliseUnwrappedValues(int actualRange, int
streachedRange, int minAssembledValue)
{
    for (int counter=0; counter<m_PixelArraySize; counter++)
    {
        ASSERT (myself[ counter] .IsAddedToMst());

        myself[ counter] .SetUnwrappedValue(
            int((double(myself[ counter] .GetUnwrappedValue() -
            minAssembledValue)/(double)actualRange) *
(double)streachedRange));
    }
}

void Tile_cls::SetFourNeighbours(Tile_cls *const pNorth,
                                 Tile_cls *const pWest,
                                 Tile_cls *const pEast,
                                 Tile_cls *const pSouth)
{
    (Tile_cls*) m_pNorthTile = pNorth;
    (Tile_cls*) m_pWestTile = pWest;
    (Tile_cls*) m_pEastTile = pEast;
    (Tile_cls*) m_pSouthTile = pSouth;
}

Pixel_cls& Tile_cls::operator[](int index) const
{
    return Image_cls::operator[](index);
```

```
}


//Vertex_cls interface support
void Tile_cls::SetNorthWeight(int weight)
{
    m_NorthWeight = weight;
}

void Tile_cls::SetWestWeight(int weight)
{
    m_WestWeight = weight;
}

void Tile_cls::SetEastWeight(int weight)
{
    m_EastWeight = weight;
}

void Tile_cls::SetSouthWeight(int weight)
{
    m_SouthWeight = weight;
}

int Tile_cls::GetNorthWeight() const
{
    return m_NorthWeight;
}

int Tile_cls::GetWestWeight() const
{
    return m_WestWeight;
}

int Tile_cls::GetEastWeight() const
{
    return m_EastWeight;
}

int Tile_cls::GetSouthWeight() const
{
    return m_SouthWeight;
}

int Tile_cls::GetMinIncidentEdgeWeight() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentEdgeWeight;
}

Vertex_cls* Tile_cls::GetMinIncidentVertex() const
{
    ASSERT (m_IsAddedToMst);

    return m_pMinIncidentVertex;
}

EdgeDirection_enm Tile_cls::GetMinIncidentDirection() const
{
    ASSERT (m_IsAddedToMst);

    return m_MinIncidentDirection;
}
```

```
int Tile_cls::GetMinIncidentEdgeWeight_InMstNeighbourhood() const
{
   ASSERT (m_IsAddedToMst);

   return m_MinIncidentEdgeWeight_InMstNeighbourhood;
}

Vertex_cls* Tile_cls::GetMinIncidentVertex_InMstNeighbourhood()
const
{
   ASSERT (m_IsAddedToMst);

   return m_pMinIncidentVertex_InMstNeighbourhood;
}

EdgeDirection_enm
Tile_cls::GetMinIncidentDirection_InMstNeighbourhood() const
{
   ASSERT (m_IsAddedToMst);

   return m_MinIncidentDirection_InMstNeighbourhood;
}

bool Tile_cls::IsAddedToMst() const
{
   return m_IsAddedToMst;
}

void Tile_cls::AddToMst(EdgeDirection_enm direction, const
Vertex_cls* previousVertexInMst)
{
   Precondition
   (
   if (previousVertexInMst == NULL)
   {
      //must be seed vertex in MST
      ASSERT (direction == ED_UNDEFINED);
   }
   else
   {
      switch (direction)
      {
         case ED_NORTH:
         {
            ASSERT (previousVertexInMst == m_pSouthTile);
            break;
         }
         case ED_WEST:
         {
            ASSERT (previousVertexInMst == m_pEastTile);
            break;
         }
         case ED_EAST:
         {
            ASSERT (previousVertexInMst == m_pWestTile);
            break;
         }
         case ED_SOUTH:
         {
            ASSERT (previousVertexInMst == m_pNorthTile);
            break;
         }
         default:
         {
            //must be one of the above
```

```
                ASSERT (FALSE);
            }
        }
    }
)

m_IsAddedToMst = true;
m_MstDirection = direction;
m_pPreviousVertexInMst = previousVertexInMst;

//notify neighbouring vertices
if (m_pNorthTile)
{
    m_pNorthTile->OnVertexAddedToMst(myself);
}
if (m_pWestTile)
{
    m_pWestTile->OnVertexAddedToMst(myself);
}
if (m_pEastTile)
{
    m_pEastTile->OnVertexAddedToMst(myself);
}
if (m_pSouthTile)
{
    m_pSouthTile->OnVertexAddedToMst(myself);
}

//RefreshMinIncidentVertex() must be called after neighbouring
//vertices have been updated with OnVertexAddedToMst(), otherwise
//m_pMinIncidentVertex_InMstNeighbourhood will not necessarely be
//found correctly
RefreshMinIncidentVertex();
}

void Tile_cls::OnVertexAddedToMst(Vertex_cls& vertex)
{
    if (&vertex == m_pMinIncidentVertex)
    {
        RefreshMinIncidentVertex();
    }
}

void Tile_cls::RefreshMinIncidentVertex()
{
    ASSERT (m_IsAddedToMst);
    m_MinIncidentEdgeWeight = INT_MAX;
    m_pMinIncidentVertex = m_pNullVertex;
    m_MinIncidentDirection = ED_UNDEFINED;

    if (m_pNorthTile)
    {
        if (!m_pNorthTile->IsAddedToMst())
        {
            m_MinIncidentEdgeWeight = m_NorthWeight;
            m_pMinIncidentVertex = (Tile_cls*)m_pNorthTile;
            m_MinIncidentDirection = ED_NORTH;
        }
        else
        {
            m_MinIncidentEdgeWeight_InMstNeighbourhood = m_pNorthTile-
>GetMinIncidentEdgeWeight();
            m_pMinIncidentVertex_InMstNeighbourhood = m_pNorthTile-
>GetMinIncidentVertex();
            m_MinIncidentDirection_InMstNeighbourhood = m_pNorthTile-
```

```
>GetMinIncidentDirection();
        }
    }

    if (m_pWestTile)
    {
        if (!m_pWestTile->IsAddedToMst())
        {
            if (m_WestWeight < m_MinIncidentEdgeWeight)
            {
                m_MinIncidentEdgeWeight = m_WestWeight;
                m_pMinIncidentVertex = (Tile_cls*)m_pWestTile;
                m_MinIncidentDirection = ED_WEST;
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pWestTile->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pWestTile->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pWestTile-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood = m_pWestTile-
>GetMinIncidentDirection();
            }
        }
    }

    if (m_pEastTile)
    {
        if (!m_pEastTile->IsAddedToMst())
        {
            if (m_EastWeight < m_MinIncidentEdgeWeight)
            {
                m_MinIncidentEdgeWeight = m_EastWeight;
                m_pMinIncidentVertex = (Tile_cls*)m_pEastTile;
                m_MinIncidentDirection = ED_EAST;
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pEastTile->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pEastTile->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pEastTile-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood = m_pEastTile-
>GetMinIncidentDirection();
            }
        }
    }
}

if (m_pSouthTile)
{
    if (!m_pSouthTile->IsAddedToMst())
    {
        if (m_SouthWeight < m_MinIncidentEdgeWeight)
        {
            m_MinIncidentEdgeWeight = m_SouthWeight;
            m_pMinIncidentVertex = (Tile_cls*)m_pSouthTile;
            m_MinIncidentDirection = ED_SOUTH;
```

```
            }
        }
        else
        {
            if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_pSouthTile->GetMinIncidentEdgeWeight())
            {
                m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_pSouthTile->GetMinIncidentEdgeWeight();
                m_pMinIncidentVertex_InMstNeighbourhood = m_pSouthTile-
>GetMinIncidentVertex();
                m_MinIncidentDirection_InMstNeighbourhood =
m_pSouthTile->GetMinIncidentDirection();
            }
        }
    }

    if (m_MinIncidentEdgeWeight_InMstNeighbourhood >
m_MinIncidentEdgeWeight)
    {
        m_MinIncidentEdgeWeight_InMstNeighbourhood =
m_MinIncidentEdgeWeight;
        m_pMinIncidentVertex_InMstNeighbourhood =
m_pMinIncidentVertex;
        m_MinIncidentDirection_InMstNeighbourhood =
m_MinIncidentDirection;
    }
}

void Tile_cls::SetUnwrappedValue(int assemblyOffset)
{
    //the unwrapped value in this context is the assembely offset
    m_HasBeenAssembled = true;
    m_UnwrappedValue = assemblyOffset;

    m_MinAssembledValue = INT_MAX;
    m_MaxAssembledValue = 0;

    for (int counter=0; counter<m_PixelArraySize; counter++)
    {
        int assembledValue = myself[ counter] .GetUnwrappedValue() +
assemblyOffset;
        myself[ counter] .SetAssembledValue(assembledValue);

        if (assembledValue < m_MinAssembledValue)
        {
            m_MinAssembledValue = assembledValue;
        }
        if(assembledValue > m_MaxAssembledValue)
        {
            m_MaxAssembledValue = assembledValue;
        }
    }
}

int Tile_cls::GetUnwrappedValue() const
{
    return m_UnwrappedValue;
}

const Vertex_cls* Tile_cls::GetPreviousVertexInMst() const
{
    return m_pPreviousVertexInMst;
}
```

```
EdgeDirection_enm Tile_cls::GetMstDirection() const
{
   ASSERT (m_IsAddedToMst);

   return m_MstDirection;
}

int Tile_cls::GetVertexValue() const
{
   ASSERT (m_HasBeenUnwrapped);

   return m_Value;
}

int Tile_cls::GetMinAssembledValue() const
{
   ASSERT (m_HasBeenAssembled);

   return m_MinAssembledValue;
}

int Tile_cls::GetMaxAssembledValue() const
{
   ASSERT (m_HasBeenAssembled);

   return m_MaxAssembledValue;
}

int Tile_cls::GetNorthAssembelyOffset() const
{
   //commented out, Boolean should be changed to be a member
variable
   //Precondition
   //(
   //   static bool assertFunctionIsOnlyEverCalledOnce = true;
   //   ASSERT (assertFunctionIsOnlyEverCalledOnce);
   //   assertFunctionIsOnlyEverCalledOnce = false;
   //)
   ASSERT (m_HasBeenAssembled);
   ASSERT (m_pNorthTile);

   m_NorthAssembelyOffset = 0;
   int northTileCounterOffset = (T*(T-P));

   for(int counter=0; counter<(T*P); counter++)
   {
      int northTileCounter = counter + (T*(T-P));
      m_NorthAssembelyOffset += myself[ counter] .GetAssembledValue()
-

(*m_pNorthTile)[ northTileCounter] .GetUnwrappedValue();
   }
   m_NorthAssembelyOffset = m_NorthAssembelyOffset / (T*P);

   return m_NorthAssembelyOffset;
}

int Tile_cls::GetWestAssembelyOffset() const
{
   //commented out, Boolean should be changed to be a member
variable
   //Precondition
   //(
   //   static bool assertFunctionIsOnlyEverCalledOnce = true;
```

```
//    ASSERT (assertFunctionIsOnlyEverCalledOnce);
//    assertFunctionIsOnlyEverCalledOnce = false;
//)
ASSERT (m_HasBeenAssembled);
ASSERT (m_pWestTile);

m_WestAssembelyOffset = 0;
int westTileCounterOffset = (T*(T-P));

for(int y=0; y<T; y++)
{
    for(int x=0; x<P; x++)
    {
        int counter = (T * y) + x;
        int westTileCounter = (T * y) + (x + (T - P));
        m_WestAssembelyOffset +=
myself[ counter] .GetAssembledValue() -

(*m_pWestTile)[ westTileCounter] .GetUnwrappedValue();
    }
}

m_WestAssembelyOffset = m_WestAssembelyOffset / (T*P);

return m_WestAssembelyOffset;
}

int Tile_cls::GetEastAssembelyOffset() const
{
    //commented out, Boolean should be changed to be a member
variable
    //Precondition
    //(
    //   static bool assertFunctionIsOnlyEverCalledOnce = true;
    //   ASSERT (assertFunctionIsOnlyEverCalledOnce);
    //   assertFunctionIsOnlyEverCalledOnce = false;
    //)
    ASSERT (m_HasBeenAssembled);
    ASSERT (m_pEastTile);

    m_EastAssembelyOffset = 0;

    for(int y=0; y<T; y++)
    {
        for(int x=(T-P); x<T; x++)
        {
            int counter = (T * y) + x;
            int eastTileCounter = (T * y) + (x - (T - P));
            m_EastAssembelyOffset +=
myself[ counter] .GetAssembledValue() -

(*m_pEastTile)[ eastTileCounter] .GetUnwrappedValue();
        }
    }

    m_EastAssembelyOffset = m_EastAssembelyOffset / (T*P);

    return m_EastAssembelyOffset;
}

int Tile_cls::GetSouthAssembelyOffset() const
{
    //commented out, Boolean should be changed to be a member
variable
    //Precondition
```

```
    //(
    //    static bool assertFunctionIsOnlyEverCalledOnce = true;
    //    ASSERT (assertFunctionIsOnlyEverCalledOnce);
    //    assertFunctionIsOnlyEverCalledOnce = false;
    //)
    ASSERT (m_HasBeenAssembled);
    ASSERT (m_pSouthTile);

    m_SouthAssembelyOffset = 0;

    for(int counter=(T*(T-P)); counter<(T*T); counter++)
    {
        int southTileCounter = counter - (T*(T-P));
        m_SouthAssembelyOffset += myself[ counter] .GetAssembledValue()
-

(*m_pSouthTile)[ southTileCounter] .GetUnwrappedValue();
    }

    m_SouthAssembelyOffset = m_SouthAssembelyOffset / (T*P);

    return m_SouthAssembelyOffset;
}

bool Tile_cls::DoesContainEdges() const
{
    return m_DoesContainEdges;
}

void Tile_cls::SetDoesContainEdges(bool doesContainEdges)
{
    m_DoesContainEdges = doesContainEdges;
}

ImagePixel_cls* Tile_cls::GetEdgeInfo() const
{
    //not applicable to tiles
    ASSERT(FALSE);

    return NULL;
}

bool Tile_cls::IsToHaveMaxWeight() const
{
    return false;
}

void Tile_cls::SetEdgeInfo(ImagePixel_cls* pEdge)
{
    //not applicable to tiles
    ASSERT(FALSE);
}

void Tile_cls::SetPhaseJumpsCount(int phaseJumpsCount)
{
    //not applicable to tiles
    ASSERT(FALSE);
}

int Tile_cls::GetPhaseJumpsCount() const
{
    //not applicable to tiles
    ASSERT(FALSE);

    return 0;
```

```
}

BITMAPINFO* Tile_cls::GetWrappedDib()
{
   if (m_pWrappedDib != NULL)
   {
      DestroyDib(m_pWrappedDib);
   }

   BYTE* pByte = new BYTE[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD)*T*T)];

   m_pWrappedDib = (BITMAPINFO*) pByte;
   m_pWrappedDib->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
   m_pWrappedDib->bmiHeader.biWidth = T;
   m_pWrappedDib->bmiHeader.biHeight = T;
   m_pWrappedDib->bmiHeader.biPlanes = 1;                //must be 1
   m_pWrappedDib->bmiHeader.biBitCount = 32;             //8 bit RGB
   m_pWrappedDib->bmiHeader.biCompression = BI_RGB;      //no
compression
   m_pWrappedDib->bmiHeader.biSizeImage = 0;             //0 for RGB
bitmaps
   m_pWrappedDib->bmiHeader.biXPelsPerMeter = 0;
   m_pWrappedDib->bmiHeader.biYPelsPerMeter = 0;
   m_pWrappedDib->bmiHeader.biClrUsed = 0;
   m_pWrappedDib->bmiHeader.biClrImportant = 0;

   for (int yCounter=0; yCounter!=T; yCounter++)
   {
      for (int xCounter=0; xCounter!=T; xCounter++)
      {
         int topDownBmpIndex = ((T-1-yCounter)*T)+xCounter;
         int bottomUpBmpIndex = (yCounter*T)+xCounter;

         m_pWrappedDib->bmiColors[ topDownBmpIndex] .rgbRed =
myself[ bottomUpBmpIndex] ;
         m_pWrappedDib->bmiColors[ topDownBmpIndex] .rgbGreen =
myself[ bottomUpBmpIndex] ;
         m_pWrappedDib->bmiColors[ topDownBmpIndex] .rgbBlue =
myself[ bottomUpBmpIndex] ;
         m_pWrappedDib->bmiColors[ topDownBmpIndex] .rgbReserved = 0;
//padding
      }
   }

   return m_pWrappedDib;
}

void Tile_cls::DestroyDib(BITMAPINFO* pDib)
{
   delete [] ((BYTE*)pDib);
}

BITMAPINFO* Tile_cls::GetWeightsDib()
{
   if (m_pWeightDib != NULL)
   {
      DestroyDib(m_pWeightDib);
   }

   BYTE* pByte = new BYTE[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD)*T*T)];

   m_pWeightDib = (BITMAPINFO*) pByte;
```

```
   m_pWeightDib->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
   m_pWeightDib->bmiHeader.biWidth = T;
   m_pWeightDib->bmiHeader.biHeight = T;
   m_pWeightDib->bmiHeader.biPlanes = 1;                    //must be 1
   m_pWeightDib->bmiHeader.biBitCount = 32;                 //8 bit RGB
   m_pWeightDib->bmiHeader.biCompression = BI_RGB;          //no
compression
   m_pWeightDib->bmiHeader.biSizeImage = 0;                 //0 for RGB
bitmaps
   m_pWeightDib->bmiHeader.biXPelsPerMeter = 0;
   m_pWeightDib->bmiHeader.biYPelsPerMeter = 0;
   m_pWeightDib->bmiHeader.biClrUsed = 0;
   m_pWeightDib->bmiHeader.biClrImportant = 0;

   //find out range of weights
   int minWeight = INT_MAX;
   int maxWeight = INT_MIN;
   int* unwrapDirectionWeight = new int[ T*T] ;

   for(int count=0; count<T*T; count++)
   {
      int weight = INT_MAX;

      switch (myself[ count] .GetMstDirection())
      {
         case ED_NORTH:
         {
            weight = unwrapDirectionWeight[ count]  =
myself[ count] .GetSouthWeight();
            break;
         }
         case ED_WEST:
         {
            weight = unwrapDirectionWeight[ count]  =
myself[ count] .GetEastWeight();
            break;
         }
         case ED_EAST:
         {
            weight = unwrapDirectionWeight[ count]  =
myself[ count] .GetWestWeight();
            break;
         }
         case ED_SOUTH:
         {
            weight = unwrapDirectionWeight[ count]  =
myself[ count] .GetNorthWeight();
            break;
         }
         default:
         {
            ASSERT (count == 0);
            weight = unwrapDirectionWeight[ count]  = 0;
      }
   }

   if (weight < minWeight)
   {
      minWeight = weight;
   }

   if (weight > maxWeight)
   {
      maxWeight = weight;
```

```
    }
}

double range = maxWeight - minWeight;
ASSERT (range >= 0);

for (int yCounter=0; yCounter!=T; yCounter++)
{
    for (int xCounter=0; xCounter!=T; xCounter++)
    {
        int topDownBmpIndex = ((T-1-yCounter)*T)+xCounter;
        int index = (yCounter*T)+xCounter;

        const int GRAY_BAND = 256;
        const int BLUE_BAND = 512;
        const int GREEN_BAND = 768;
        const int RED_BAND = 1024;
        const int FULL_RANGE = 1024;

        int normalisedWeight =
int((abs(unwrapDirectionWeight[ index]  - minWeight) / range) *
FULL_RANGE);

        if(normalisedWeight < GRAY_BAND)
        {
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbRed =
normalisedWeight;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbGreen =
normalisedWeight;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbBlue =
normalisedWeight;
        }
        else if(normalisedWeight < BLUE_BAND)
        {
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbRed = 0;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbGreen = 0;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbBlue =
normalisedWeight;
        }
        else if(normalisedWeight < GREEN_BAND)
        {
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbRed = 0;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbGreen =
normalisedWeight;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbBlue = 0;
        }
        else
        {
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbRed =
normalisedWeight;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbGreen = 0;
            m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbBlue = 0;
        }

        m_pWeightDib->bmiColors[ topDownBmpIndex] .rgbReserved = 0;
//padding
    }
}

delete [] unwrapDirectionWeight;

return m_pWeightDib;
}

BITMAPINFO* Tile_cls::GetMstDib()
```

```
{
    if (m_pMstDib != NULL)
    {
        DestroyDib(m_pMstDib);
    }

    BYTE* pByte = new BYTE[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD)*T*T)];

    m_pMstDib = (BITMAPINFO*) pByte;

    m_pMstDib->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    m_pMstDib->bmiHeader.biWidth = T;
    m_pMstDib->bmiHeader.biHeight = T;
    m_pMstDib->bmiHeader.biPlanes = 1;                  //must be 1
    m_pMstDib->bmiHeader.biBitCount = 32;               //8 bit RGB
    m_pMstDib->bmiHeader.biCompression = BI_RGB;        //no compression
    m_pMstDib->bmiHeader.biSizeImage = 0;               //0 for RGB
bitmaps
    m_pMstDib->bmiHeader.biXPelsPerMeter = 0;
    m_pMstDib->bmiHeader.biYPelsPerMeter = 0;
    m_pMstDib->bmiHeader.biClrUsed = 0;
    m_pMstDib->bmiHeader.biClrImportant = 0;

    for (int yCounter=0; yCounter!=T; yCounter++)
    {
        for (int xCounter=0; xCounter!=T; xCounter++)
        {
            int topDownBmpIndex = ((T-1-yCounter)*T)+xCounter;
            int index = (yCounter*T)+xCounter;

            switch (myself[ index] .GetMstDirection())
            {
case ED_NORTH:
{
    //green for north
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbRed = 0;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbGreen = 255;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbBlue = 0;
    break;
}
case ED_WEST:
{
    //white for west
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbRed = 255;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbGreen = 255;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbBlue = 255;
    break;
}
case ED_EAST:
{
    //red for right
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbRed = 255;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbGreen = 0;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbBlue = 0;
    break;
}
case ED_SOUTH:
{
    //blue for bottom
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbRed = 0;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbGreen = 0;
    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbBlue = 255;
    break;
}
```

```
                default:
                {
                    //must be one of the above, unless it is the first
vertix in mst
                    ASSERT (yCounter == 0 && xCounter ==0);
                    //black for unknown/undefined direction
                    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbRed = 0;
                    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbGreen = 0;
                    m_pMstDib->bmiColors[ topDownBmpIndex] .rgbBlue = 0;
                }
            }
            m_pMstDib->bmiColors[ topDownBmpIndex] .rgbReserved = 0;
//padding
        }
    }

    return m_pMstDib;
}

BITMAPINFO* Tile_cls::GetPhaseJumpsDib()
{
    if (m_pPhaseJumpsDib != NULL)
    {
        DestroyDib(m_pPhaseJumpsDib);
    }

    BYTE* pByte = new BYTE[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD)*T*T)] ;

    m_pPhaseJumpsDib = (BITMAPINFO*) pByte;

    m_pPhaseJumpsDib->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    m_pPhaseJumpsDib->bmiHeader.biWidth = T;
    m_pPhaseJumpsDib->bmiHeader.biHeight = T;
    m_pPhaseJumpsDib->bmiHeader.biPlanes = 1;                    //must be 1
    m_pPhaseJumpsDib->bmiHeader.biBitCount = 32;                 //8 bit RGB
    m_pPhaseJumpsDib->bmiHeader.biCompression = BI_RGB;          //no
compression
    m_pPhaseJumpsDib->bmiHeader.biSizeImage = 0;                 //0 for RGB
bitmaps
    m_pPhaseJumpsDib->bmiHeader.biXPelsPerMeter = 0;
    m_pPhaseJumpsDib->bmiHeader.biYPelsPerMeter = 0;
    m_pPhaseJumpsDib->bmiHeader.biClrUsed = 0;
    m_pPhaseJumpsDib->bmiHeader.biClrImportant = 0;

    for (int yCounter=0; yCounter!=T; yCounter++)
    {
        for (int xCounter=0; xCounter!=T; xCounter++)
        {
            int topDownBmpIndex = ((T-1-yCounter)*T)+xCounter;
            int index = (yCounter*T)+xCounter;
            int phaseJumpCount = myself[ index] .GetPhaseJumpsCount() /
255;

            if (myself[ index] .GetEdgeInfo()->DoesHaveEdge())
            {
                //detected phase jumps are essentially the fringe data
                //these are colour coded: Blue is one phase juump
                //blue+green is zero phase jump
                //blue+red is two or more phase jumps

                if (phaseJumpCount == -1)
                {
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
```

```
255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 0;
                }
                else if (phaseJumpCount == 0 )
                {
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
0;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 255;
                }
                else if (phaseJumpCount == 1)
                {
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
0;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 0;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 255;
                }
                else
                {
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 0;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 255;
                }
                else
                {
                    //phase jump unwrapping factor is colour coded
                    //white means zero phase jump unwrapping
                    //Green means one phase jump, pale yellow is a -1 phase
jump
                    //red means more than one phase jump

                    if (phaseJumpCount == -1 )
                    {
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
128;
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 128;
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 0;
                    }
                    else if (phaseJumpCount == 0)
                    {
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
255;
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 255;
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 255;
                    }
                    else if (phaseJumpCount == 1)
                    {
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
0;
                        m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
```

```
= 255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 0;
                }
                else
                {
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbRed =
255;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbGreen
= 0;
                    m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbBlue
= 0;
                }
            }
            m_pPhaseJumpsDib->bmiColors[ topDownBmpIndex] .rgbReserved =
0; //padding
        }
    }

    return m_pPhaseJumpsDib;
}

BITMAPINFO* Tile_cls::GetUnwrappedDib()
{
    if (m_pUnwrappedDib != NULL)
    {
        DestroyDib(m_pUnwrappedDib);
    }

    BYTE* pByte = new BYTE[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD)*T*T)];

    m_pUnwrappedDib = (BITMAPINFO*) pByte;

    m_pUnwrappedDib->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    m_pUnwrappedDib->bmiHeader.biWidth = T;
    m_pUnwrappedDib->bmiHeader.biHeight = T;
    m_pUnwrappedDib->bmiHeader.biPlanes = 1;                    //must be 1
    m_pUnwrappedDib->bmiHeader.biBitCount = 32;                 //8 bit RGB
    m_pUnwrappedDib->bmiHeader.biCompression = BI_RGB;          //no
compression
    m_pUnwrappedDib->bmiHeader.biSizeImage = 0;                 //0 for RGB
bitmaps
    m_pUnwrappedDib->bmiHeader.biXPelsPerMeter = 0;
    m_pUnwrappedDib->bmiHeader.biYPelsPerMeter = 0;
    m_pUnwrappedDib->bmiHeader.biClrUsed = 0;
    m_pUnwrappedDib->bmiHeader.biClrImportant = 0;

    for (int yCounter=0; yCounter!=T; yCounter++)
    {
        for (int xCounter=0; xCounter!=T; xCounter++)
        {
            int topDownBmpIndex = ((T-1-yCounter)*T)+xCounter;
            int index = (yCounter*T)+xCounter;
            int value =
myself[ index] .GetAssembledValue();//GetUnwrappedValue();

            m_pUnwrappedDib->bmiColors[ topDownBmpIndex] .rgbRed = value;
            m_pUnwrappedDib->bmiColors[ topDownBmpIndex] .rgbGreen =
value;
            m_pUnwrappedDib->bmiColors[ topDownBmpIndex] .rgbBlue =
value;
            m_pUnwrappedDib->bmiColors[ topDownBmpIndex] .rgbReserved =
0; //padding
        }
```

```
    }

    return m_pUnwrappedDib;
}

CStdioFile& Tile_cls::GetPhaseJumpsText(CString& fileNamePrefix)
{
    if (!m_DoesPhaseJumpsTextFileExist)
    {
        m_DoesPhaseJumpsTextFileExist = true;
        m_PhaseJumpsTextFile.Open(fileNamePrefix+"phj_dbg.txt",
CFile::modeCreate | CFile::modeWrite | CFile::typeText);

        for (int yCounter=0; yCounter!=T; yCounter++)
        {
            CString line;
            CString temp;

            for (int xCounter=0; xCounter!=T; xCounter++)
            {
                int index = (yCounter*T)+xCounter;

                temp.Format("(%02d,%02d, ", xCounter, yCounter);
                line += temp;
                int phaseJumpCount = myself[ index] .GetPhaseJumpsCount()
/ 255;

                if (myself[ index] .GetEdgeInfo()->DoesHaveEdge())
                {
                    //detected phase jumps are essentially the fringe
data
                    //phase jump correction (unwrapping) factor is in
phaseJumpCount
                    temp.Format(" PJD, %d) ", phaseJumpCount);
                    line += temp;
                }
                else
                {
                    //phase jump correction (unwrapping) factor is in
phaseJumpCount
                    temp.Format(" NPJ, %d) ", phaseJumpCount);
                    line += temp;
                }
            }

            m_PhaseJumpsTextFile.WriteString(line + '\n');
        }

        m_PhaseJumpsTextFile.Close();
    }

    return m_PhaseJumpsTextFile;
}
```

TileDataModel_cls.h

```
#ifndef TILEDATAMODEL_CLS_H
#define TILEDATAMODEL_CLS_H

class TileDataModel_cls
{
public:
	TileDataModel_cls();
	~TileDataModel_cls();

	static TileDataModel_cls* GetUniqueInstance();

	//static int GetImageSize();
	//static int GetImageCoveredByTilesSize();
	static int GetTileSize();
	static int GetNumberOfTiles();
	static int GetTileOverLapSize();

	enum
	{
		//IMAGE     =
512,
		//IMAGE_COVERED_BY_TILES =
460,//504,//508,//508,//460,//510,
		TILE =
10,//16,//16,//40//16,//40,//80//26,
		NUMBER_OF_TILES =
24,//6,//42,//14,//42,//14,//6,//23,
		TILE_OVELAP =
	};

private:
		static TileDataModel_cls* m_pUniqueInstance;
		static int DestroyUniqeInstance();

		//one dimontional sizes
		//image, tiles, etc are assumed to be square
		//int m_Wrapped          [TILE][TILE];
		//int m_PreFiltered      [TILE][TILE];
		//int m_Thinned          [TILE][TILE];
		//int m_MST              [TILE][TILE];
		//int m_Weights          [TILE][TILE];
		//int m_Unwrapped [TILE][TILE];
};

#endif //TILEDATAMODEL_CLS_H
```

TileDataModel_cls.cpp

```
#include "stdafx.h"

#include "TileDataModel_cls.h"

TileDataModel_cls* TileDataModel_cls::m_pUniqueInstance = NULL;

TileDataModel_cls::TileDataModel_cls()
{
}

TileDataModel_cls::~TileDataModel_cls()
{
}

int TileDataModel_cls::DestroyUniqeInstance()
{
	ASSERT (m_pUniqueInstance != NULL);
	m_pUniqueInstance = NULL;
	delete m_pUniqueInstance;
	return 0;
}

TileDataModel_cls* TileDataModel_cls::GetUniqueInstance()
{
	if (m_pUniqueInstance == NULL)
	{
		m_pUniqueInstance = new TileDataModel_cls();
		onexit(DestroyUniqeInstance);
	}
	return m_pUniqueInstance;
}

int TileDataModel_cls::GetTileSize()
{
	return TILE;
}

int TileDataModel_cls::GetNumberOfTiles()
{
	return NUMBER_OF_TILES;
}

int TileDataModel_cls::GetTileOverLapSize()
{
	return TILE_OVELAP;
}
```

TiledImage_cls.h

```
#ifndef TILEDIMAGE_CLS_H
#define TILEDIMAGE_CLS_H

#include "VertexGrid_cls.h"
#include "NullPixel_cls.h"

class Tile_cls;
class Pixel_cls;
class ImagePixel_cls;
typedef Tile_cls* P_TILE_CLS;

class TiledImage_cls :
   public VertexGrid_cls
{
public:

   static TiledImage_cls* GetUniqueInstance();
   static void DestroyUniqueInstance();

   ~TiledImage_cls();
   TiledImage_cls();

   void SetUp(int tileWidth, int imageWidth, int imageHeight, BYTE*
pData, ImagePixel_cls imagePixelArray[ 512][ 512] );

   Tile_cls& GetTile(int xCoord, int yCoord);
   Pixel_cls& TiledImage_cls::GetPixel(int tileXCoord, int
tileYCoord,
                                       int PixelXCoord, int
PixelYCoord);
   Pixel_cls& GetPixel(int X, int Y);

   void UnwrapTiles();
   void AssembleTiles();

   //VertexGrid_cls support
   int GetVertexSize() const;
   int GetXSize() const;
   int GetYSize() const;
   Vertex_cls& GetAt(int index) const;

private:
   void ConstructInterTileWeightedGraph();
   void ConstructInterTileWeightedMst();
   void AssembleTilesAlongMstPath();
   void SetupTileNeighbourhoods();
   void NormaliseAssembledValues();
   void CleanUp();

   static TiledImage_cls* m_pUniqueInstance;

   Graph_cls* m_pGraph;
   Mst_cls* m_pMst;

   int m_MinAssembledValue;
   int m_MaxAssembledValue;

   const int& P; // )
   const int& W; // |
   const int& H; // |
   const int& T; // |> aliases, handy for equation-like
```

```
manipulations
    const int& Nx; // )
    const int& Ny; // )

    const int m_OverlapSize;
    int m_TileXCount;
    int m_TileYCount;
    int m_ImageWidth;
    int m_ImageHeight;
    int m_TileArraySize;
    int m_TilePixelSize;

    P_TILE_CLS* m_pTileBuff;
    CList<Tile_cls*, Tile_cls*> m_TileList;

    NullPixel_cls m_NullPixel;
};

#endif //TILEDIMAGE_CLS_H
```

TiledImage_cls.cpp

```
#include "stdafx.h"
#include "Tile_cls.h"
#include "Pixel_cls.h"
#include "Graph_cls.h"
#include "Mst_cls.h"
#include "ImagePixel_cls.h"

#include "TiledImage_cls.h"

typedef Tile_cls* P_TILE_CLS;

TiledImage_cls* TiledImage_cls::m_pUniqueInstance = NULL;

TiledImage_cls* TiledImage_cls::GetUniqueInstance()
{
    if (m_pUniqueInstance == NULL)
    {
        m_pUniqueInstance = new TiledImage_cls();
        atexit(DestroyUniqueInstance);
    }

    return m_pUniqueInstance;
}

void TiledImage_cls::DestroyUniqueInstance()
{
    ASSERT (m_pUniqueInstance != NULL);
    if (m_pUniqueInstance != NULL)
    {
        delete m_pUniqueInstance;
    }
}

TiledImage_cls::TiledImage_cls():
W(m_ImageWidth),
H(m_ImageHeight),
P(m_OverlapSize),
T(m_TilePixelSize),
Nx(m_TileXCount),
Ny(m_TileYCount),
m_OverlapSize(4),
m_TileXCount(0),
m_TileYCount(0),
m_ImageWidth(0),
m_ImageHeight(0),
m_TileArraySize(0),
m_TilePixelSize(0),
m_NullPixel(0),
m_pGraph(NULL),
m_pMst(NULL),
m_pTileBuff(NULL)
{
}

void TiledImage_cls::SetUp(int tileWidth, int imageWidth, int
imageHeight, BYTE* pData, ImagePixel_cls imagePixelArray[ 512][ 512] )
{
    //First of all, free up any memory previously allocated
    CleanUp();
```

```
m_TilePixelSize = tileWidth;
m_ImageHeight = imageHeight;
m_ImageWidth = imageWidth;
m_TileXCount = W / (T - P);
m_TileYCount = H / (T - P);
m_TileArraySize = m_TileXCount * m_TileYCount;

ASSERT (m_ImageHeight > m_TileYCount);
ASSERT (m_ImageWidth > m_TileXCount);
ASSERT (m_ImageHeight > 0);
ASSERT (m_ImageWidth > 0);
ASSERT (m_TileList.IsEmpty());

//          W - P                  W + P(N-1)
// N = ---------- <=> T = ------------
//          T - P                    N

//now in intialisation list
//   m_TilePixelSize = ((imageWidth + (m_OverlapSize * (tileCount -
1))) / tileCount);

   m_pTileBuff = new P_TILE_CLS[ m_TileArraySize] ;

   //create tiles
   for (int counter=0; counter<m_TileArraySize; counter++)
   {
      m_pTileBuff[ counter] = new Tile_cls(m_TilePixelSize,
m_OverlapSize);
      m_TileList.AddTail(m_pTileBuff[ counter] );
   }

   //copy image data into tiles
   int x = 0;                              //local x coordinate in tile
   int y = 0;                              //local y coordinate in tile
   int X = 0;                              //x coordinate in original image
   int Y = 0;                              //y coordinate in original image
   int xp = 0;                             //x coordinate of tile
   int yp = 0;                             //y coordinate of tile

      for(yp=0; yp<Ny; yp++)
      {
            for(xp=0; xp<Nx; xp++)
            {
                  for(y=0; y<T; y++)
                  {
                        for(x=0; x<T; x++)
                        {
                              X = x+(xp*(T-P));
                              Y = y+(yp*(T-P));
                              (*m_pTileBuff[ (yp*Nx)+xp] )[ (y*T)+x] =
pData[ (Y*W) + X] ;

(*m_pTileBuff[ (yp*Nx)+xp] )[ (y*T)+x] .SetEdgeInfo(&(imagePixelArray[ Y]
[ X] ));

//(*m_pTileBuff[ (yp*N)+xp] )[ (y*T)+x] .SetUnwrappedValue(imagePixelArr
ay[ Y][ X] .DoesHaveEdge()? 255 : 0);

//(*m_pTileBuff[ (yp*Nx)+xp] )[ (y*T)+x] .SetUnwrappedValue(imagePixelAr
ray[ Y][ X] .GetValue());
                        }
                  }
            }
      }
}
```

```
TiledImage_cls::~TiledImage_cls()
{
    CleanUp();
}

void TiledImage_cls::CleanUp()
{
    ASSERT (m_TileList.GetSize() == m_TileArraySize);

    if (!m_TileList.IsEmpty())
    {
        for (int counter=0; counter<m_TileArraySize; counter++)
        {
            ASSERT (!m_TileList.IsEmpty());

            Tile_cls* pTile = m_TileList.RemoveTail();
            delete pTile;
        }
    }

    if (m_pGraph != NULL)
    {
        delete m_pGraph;
        m_pGraph = NULL;
    }

    if (m_pMst != NULL)
    {
        delete m_pMst;
        m_pMst = NULL;
    }

    if (m_pTileBuff != NULL)
    {
        delete[] m_pTileBuff;
        m_pTileBuff = NULL;
    }
}

Tile_cls& TiledImage_cls::GetTile(int xCoord, int yCoord)
{
    ASSERT (xCoord < Nx);
    ASSERT (yCoord < Ny);

    return (*m_pTileBuff[ (yCoord*Ny)+xCoord] );
}

Pixel_cls& TiledImage_cls::GetPixel(int tileXCoord, int tileYCoord,
int PixelXCoord, int PixelYCoord)
{
    ASSERT (tileXCoord < Nx);
    ASSERT (tileYCoord < Ny);
    ASSERT (PixelXCoord < T);
    ASSERT (PixelYCoord < T);

    return
(*m_pTileBuff[ (tileYCoord*Nx)+tileXCoord] )[ (PixelYCoord*T)+PixelXCoo
rd] ;
}

Pixel_cls& TiledImage_cls::GetPixel(int X, int Y)
{
    ASSERT (X < W);
    ASSERT (Y < H);
```

```
//calculate the tile's coordinates
int tileXCoord = int(X/(T-P));
int tileYCoord = int(Y/(T-P));

//calculate the pixel's coordinates within the tile
   int PixelXCoord = X-((int(X/(T-P)))*(T-P));
   int PixelYCoord = Y-((int(Y/(T-P)))*(T-P));

if (tileXCoord >= Nx || tileYCoord >= Ny)
{
   return m_NullPixel;
}
else
{
   return GetPixel(tileXCoord, tileYCoord, PixelXCoord,
PixelYCoord);
   }
}

void TiledImage_cls::UnwrapTiles()
{
   for (int counter=0; counter<m_TileArraySize; counter++)
   {
      m_pTileBuff[ counter] ->ConstructWeightedGraph();
      m_pTileBuff[ counter] ->ConstructMst();
      m_pTileBuff[ counter] ->UnwrapAlongMstPath();
   }
}

void TiledImage_cls::AssembleTiles()
{
   ConstructInterTileWeightedGraph();
   ConstructInterTileWeightedMst();
   //AssembleTilesAlongMstPath();
}

void TiledImage_cls::ConstructInterTileWeightedGraph()
{
   SetupTileNeighbourhoods();

   if (m_pGraph != NULL)
   {
      delete m_pGraph;
   }
   m_pGraph = new Graph_cls(myself);
}

void TiledImage_cls::ConstructInterTileWeightedMst()
{
   if (m_pMst != NULL)
   {
      delete m_pMst;
   }
   m_pMst = new Mst_cls(*m_pGraph);
}

void TiledImage_cls::SetupTileNeighbourhoods()
{
   for (int counter=0; counter<m_TileArraySize; counter++)
   {
      int y = counter / m_TileXCount;
      int x = counter - (y * m_TileXCount);

      ASSERT ((x + (y * m_TileXCount) < m_TileArraySize));
```

```
            Tile_cls* n = NULL;
            Tile_cls* w = NULL;
            Tile_cls* e = NULL;
            Tile_cls* s = NULL;

            if (y > 0)
            {
               n = m_pTileBuff[ counter - m_TileXCount] ;
            }
            if (x > 0)
            {
               w = m_pTileBuff[ counter - 1] ;
            }
            if (x < (m_TileXCount - 1))
            {
               e = m_pTileBuff[ counter + 1] ;
            }
            if (y < (m_TileYCount - 1))
            {
               s = m_pTileBuff[ counter + m_TileXCount] ;
            }

            m_pTileBuff[ counter] ->SetFourNeighbours(n, w, e, s);
         }
      }

      void TiledImage_cls::AssembleTilesAlongMstPath()
      {
         // the unwrapped value in this context is the assembely offset

         //initialise the seed vertex
         (*m_pMst)[ 0] .m_Vertex.SetUnwrappedValue(0);
         m_MinAssembledValue = INT_MAX;
         m_MaxAssembledValue = 0;

         //calculate assembly offsets
         for (int counter=1; counter<m_TileArraySize; counter++)
         {
            Vertex_cls& vertexToAssemble = (*m_pMst)[ counter] .m_Vertex;
            const Vertex_cls* pPreviousAssembledVertex =
      vertexToAssemble.GetPreviousVertexInMst();
            EdgeDirection_enm unwrapDirection =
      (*m_pMst)[ counter] .m_Direction;

            switch (unwrapDirection)
            {
               case ED_NORTH:
               {

      vertexToAssemble.SetUnwrappedValue(pPreviousAssembledVertex-
      >GetNorthAssembelyOffset());
                  break;
               }
               case ED_WEST:
               {

      vertexToAssemble.SetUnwrappedValue(pPreviousAssembledVertex-
      >GetWestAssembelyOffset());
                  break;
               }
               case ED_EAST:
               {

      vertexToAssemble.SetUnwrappedValue(pPreviousAssembledVertex-
```

```
>GetEastAssembelyOffset());
                break;
            }
            case ED_SOUTH:
            {

vertexToAssemble.SetUnwrappedValue(pPreviousAssembledVertex-
>GetSouthAssembelyOffset());
                break;
            }
            default:
            {
                //has to be one of the above
                ASSERT (FALSE);
            }
        }

        if (m_MinAssembledValue >
vertexToAssemble.GetMinAssembledValue())
        {
            m_MinAssembledValue =
vertexToAssemble.GetMinAssembledValue();
        }
        if (m_MaxAssembledValue <
vertexToAssemble.GetMaxAssembledValue())
        {
            m_MaxAssembledValue =
vertexToAssemble.GetMaxAssembledValue();
        }
    }

    NormaliseAssembledValues();
}

void TiledImage_cls::NormaliseAssembledValues()
{
    int actualRange = m_MaxAssembledValue - m_MinAssembledValue;
    for (int counter=0; counter<m_TileArraySize; counter++)
    {
        ASSERT (myself[ counter] .IsAddedToMst());

        m_pTileBuff[ counter] ->NormaliseAssembledValues(actualRange,
STREACHED_RANGE, m_MinAssembledValue);
    }
}

//VertexGrid_cls support
int TiledImage_cls::GetVertexSize() const
{
    return m_TileArraySize;
}

int TiledImage_cls::GetXSize() const
{
    return m_TileXCount;
}

int TiledImage_cls::GetYSize() const
{
    return m_TileYCount;
}

Vertex_cls& TiledImage_cls::GetAt(int index) const
{
    return *m_pTileBuff[ index] ;
```

}

Blank

In

Original

Vertex_cls.h

```
#ifndef VERTEX_CLS_H
#define VERTEX_CLS_H

#include "CommonTypes.h"

class Vertex_cls
{
public:
   Vertex_cls(){}
   virtual ~Vertex_cls(){}

   virtual void SetNorthWeight(int weight) = 0;
   virtual void SetWestWeight(int weight) = 0;
   virtual void SetEastWeight(int weight) = 0;
   virtual void SetSouthWeight(int weight) = 0;
   virtual int GetNorthWeight() const = 0;
   virtual int GetWestWeight() const = 0;
   virtual int GetEastWeight() const = 0;
   virtual int GetSouthWeight() const = 0;
   virtual int GetMinIncidentEdgeWeight() const = 0;
   virtual Vertex_cls* GetMinIncidentVertex() const = 0;
   virtual EdgeDirection_enm GetMinIncidentDirection() const = 0;
   virtual int GetMinIncidentEdgeWeight_InMstNeighbourhood() const =
0;
   virtual Vertex_cls* GetMinIncidentVertex_InMstNeighbourhood()
const = 0;
   virtual EdgeDirection_enm
GetMinIncidentDirection_InMstNeighbourhood() const = 0;
   virtual bool IsAddedToMst() const = 0;
   virtual void AddToMst(EdgeDirection_enm direction, const
Vertex_cls* previousVertexInMst) = 0;
   virtual void OnVertexAddedToMst(Vertex_cls& vertex) = 0;
   virtual void SetUnwrappedValue(int value) = 0;
   virtual int GetUnwrappedValue() const = 0;
   virtual const Vertex_cls* GetPreviousVertexInMst() const = 0;
   virtual EdgeDirection_enm GetMstDirection() const = 0;
   virtual int GetVertexValue() const = 0;
   virtual int GetMinAssembledValue() const {ASSERT (FALSE); return
0;}
   virtual int GetMaxAssembledValue() const {ASSERT (FALSE); return
0;}
   virtual int GetNorthAssembelyOffset() const {ASSERT (FALSE);
return 0;}
   virtual int GetWestAssembelyOffset() const {ASSERT (FALSE);
return 0;}
   virtual int GetEastAssembelyOffset() const {ASSERT (FALSE);
return 0;}
   virtual int GetSouthAssembelyOffset() const {ASSERT (FALSE);
return 0;}

   virtual ImagePixel_cls* GetEdgeInfo() const = 0;
   virtual void SetEdgeInfo(ImagePixel_cls* pEdge) = 0;
   virtual void SetPhaseJumpsCount(int phaseJumpsCount) = 0;
   virtual int GetPhaseJumpsCount() const = 0;
   virtual bool IsToHaveMaxWeight() const = 0;

   //virtual void RuleInMinEdge() = 0;
   //virtual void RuleOutMaxDualEdge() = 0;
   //virtual void EnsureConnectedToMinEdge() = 0;
   //virtual void EnsureConnectedToDualMaxEdge() = 0;
```

```
   friend int operator+(const Vertex_cls& leftVertex, const
Vertex_cls& rightVertex);
   friend int operator-(const Vertex_cls& leftVertex, const
Vertex_cls& rightVertex);
   operator int() const {return GetVertexValue();}
};

#endif //VERTEX_CLS_H
```

VertexGrid_cls.h

```
#ifndef VERTEXGRID_CLS_H
#define VERTEXGRID_CLS_H

class Vertex_cls;

class VertexGrid_cls
{
public:
   VertexGrid_cls(){}
   virtual ~VertexGrid_cls(){}

   virtual int GetVertexSize() const = 0;
   virtual int GetXSize() const = 0;
   virtual int GetYSize() const = 0;
   virtual Vertex_cls& GetAt(int index) const = 0;

   Vertex_cls& operator[](int index) const {return GetAt(index);}
};

#endif //VERTEXGRID_CLS_H
```

common_defs.h

```
#ifndef COMMON_DEFS
#define COMMON_DEFS

typedef std::vector<std::vector<bool> >::iterator Y_IsEdgeIterator;
typedef std::vector<bool>::iterator X_IsEdgeIterator;


#endif //COMMON_DEFS
```

ProgrammingStyle.h

```
#ifndef PROGRAMMINGSTYLE_H
#define PROGRAMMINGSTYLE_H

#define myself (*this)

#if _DEBUG
      #define DEBUG_CHECK_INVARIANTS(CheckInvariantsCode)
CheckInvariantsCode
      #define PREPROS_COMMA ,
      #define Precondition(PreconditionCode) PreconditionCode
      #define PostCondition(PostconditionCode) PostconditionCode
      #define Invariant(InvariantCode) InvariantCode
      #define DebugLog(DebugLogCode) DebugLogCode
#else
      #define DEBUG_CHECK_INVARIANTS(CheckInvariantsCode)
      #define PREPROS_COMMA
      #define Precondition(PreconditionCode)
      #define PostCondition(PostconditionCode)
      #define Invariant(InvariantCode)
      #define DebugLog(DebugLogCode)
#endif //_DEBUG

#endif //PROGRAMMINGSTYLE_H
```

unwrapping.h

```
// unwrapping.h : main header file for the unwrapping DLL
//

#pragma once

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"          // main symbols
#include "CommonTypes.h"

// Unwrapping
// See unwrapping.cpp for the implementation of this class
//

class Unwrapping_Internal : public CWinApp
{
public:
      Unwrapping_Internal();

// Overrides
public:
      virtual BOOL InitInstance();
   const char* GetStatus();
   bool SetProcessingRegion(int xTopLeft, int yTopLeft, int
xBottomRight, int yBottomRight);
      bool ConfigurePhaseStepping(int steps);
   bool SetStep(int setpNumber, const char* fileName);
   bool GenerateWrappedMap(const char* wrappedFileName);
   bool SetWrappedMap(const char* wrappedFileName);
   bool FilterWrappedMap(int passes);
   bool SkeletonWrappedMap(const char* skeletonFileName);
   bool SetUnwrappingStartPoint(int x, int y);
   bool GenerateUnWrappedMap(const char* unwrappedFileName);
   void SetEdgeInfo(int x, int y, bool isEdge);
      DECLARE_MESSAGE_MAP()

private:
   enum ContrastStreach_enum
      {
            NO_STREACH,
            PERCENTAGE_STREACH
      };
   CString GetModifiedFileName(const char* modifierString, const
CString& fileName);

   bool GenerateWrappedPhaseMap();
   bool LoadFile(CFile& file, const CString& fileName);
   bool ContrastStreachImage (CString imageToBeStreachedFileName,
CString streachedImageFileName);
   bool FilterImage (CString imageToBeFilteredFileName, CString
filteredImageFileName, CString thinnedImageFileName);
   bool TileUnwrapImage(CString filteredWrappedPhaseMApImageName,
CString thinnedImageFileName, CString unwrappedFileName);
   bool IsVlidProcessingRegion(const CRect& )const;
   bool SkeletonImage();

   const CString m_DefaultStatus;
   const CString m_NotImplementedStatus;
   const CString m_SuccessText;
```

```
CString m_StatusText;
CString m_Step1FileName;
CString m_Step2FileName;
CString m_Step3FileName;
CString m_WrappedMapFileName;
CString m_EdgeFileName;
CString m_FilteredFileName;
CString m_SkeletonFileName;

enum AlgorithmState
{
    DEFAULT,
    PHASE_STEPPING_CONFIGURED,
    READY_TO_GENERATE_WRAPPED_MAP,
    READY_TO_UNWRAP,
} m_State;

enum
{
    MAX_IMAGE_SIZE = 512,
    PIE_INTENSITY = 185,
    CONTRAST_STREACH_PERCENTAGE = 1,
};

struct PrepareImagePixel_stc
{
    int m_ImageXcoord;
    int m_ImageYCoord;
    int m_ImageIndex;
    int m_ImageWidth;
    int m_ImagePixelInfo_xCoord;
    int m_ImagePixelInfo_yCoord;
    int m_ImagePixelInfoIndex;
    BYTE* m_pWrappedFileImage;
    BYTE* m_pSkeletonedFileImage;
};
    bool PrepareImagePixelArray();
    void PrepareImagePixelInfo(const PrepareImagePixel_stc&
prepareImagePixel);
    void LoadImagePixelInfo(const PrepareImagePixel_stc& stc);

    ImagePixel_cls m_ImagePixelArray[ MAX_IMAGE_SIZE][ MAX_IMAGE_SIZE];
    bool m_EdgeInfo[ MAX_IMAGE_SIZE * MAX_IMAGE_SIZE];
    bool m_LowModulationInfo[ MAX_IMAGE_SIZE * MAX_IMAGE_SIZE];
    const int m_LowModulationIntensityThreshold;
    CRect m_ProcessingRegion;
    CPoint m_UnwrappingStartPoint;
};

class DLLEXPORT Unwrapping
{
public:
      Unwrapping();
    const char* GetStatus();
    bool SetProcessingRegion(int xTopLeft, int yTopLeft, int
xBottomRight, int yBottomRight);
       bool ConfigurePhaseStepping(int steps);
    bool SetStep(int setpNumber, const char* fileName);
    bool GenerateWrappedMap(const char* wrappedFileName);
    bool SetWrappedMap(const char* wrappedFileName);
    bool FilterWrappedMap(int passes);
    bool SkeletonWrappedMap(const char* skeletonFileName);
    bool SetUnwrappingStartPoint(int x, int y);
    bool GenerateUnWrappedMap(const char* unwrappedFileName);
};
```

unwrapping.cpp

```
// unwrapping.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "unwrapping.h"
#include <shlwapi.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

//
//    Note!
//
//         If this DLL is dynamically linked against the MFC
//         DLLs, any functions exported from this DLL which
//         call into MFC must have the AFX_MANAGE_STATE macro
//         added at the very beginning of the function.
//
//         For example:
//
//         extern "C" BOOL PASCAL EXPORT ExportedFunction()
//         {
//               AFX_MANAGE_STATE(AfxGetStaticModuleState());
//               // normal function body here
//         }
//
//         It is very important that this macro appear in each
//         function, prior to any calls into MFC.  This means that
//         it must appear as the first statement within the
//         function, even before any object variable declarations
//         as their constructors may generate calls into the MFC
//         DLL.
//
//         Please see MFC Technical Notes 33 and 58 for additional
//         details.
//

// Unwrapping

BEGIN_MESSAGE_MAP(Unwrapping_Internal, CWinApp)
END_MESSAGE_MAP()

// Unwrapping construction

Unwrapping_Internal::Unwrapping_Internal():
m_DefaultStatus("Internal Error. Unknown Status"),
m_NotImplementedStatus("Function is not implemented yet"),
m_SuccessText("Operation completed successfully"),
m_State(DEFAULT),
m_StatusText(""),
m_Step1FileName(""),
m_Step2FileName(""),
m_Step3FileName(""),
m_WrappedMapFileName(""),
m_LowModulationIntensityThreshold(7),
m_ProcessingRegion(0, 0, 0, 0)
{
   for(int counter = 0; counter != MAX_IMAGE_SIZE * MAX_IMAGE_SIZE;
counter++)
   {
```

```
            m_EdgeInfo[ counter] = false;
            m_LowModulationInfo[ counter] = false;
        }
    }

    // The one and only Unwrapping object
    Unwrapping_Internal unwrappingInternal;


    // Unwrapping initialization

    BOOL Unwrapping_Internal::InitInstance()
    {
            CWinApp::InitInstance();

            return TRUE;
    }

    const char* Unwrapping_Internal::GetStatus()
    {
        return m_StatusText;
    }

    bool Unwrapping_Internal::ConfigurePhaseStepping(int steps)
    {
        bool result = false;
        m_StatusText = m_DefaultStatus;

        if (steps == 3)
        {
            m_State = PHASE_STEPPING_CONFIGURED;
            m_StatusText = m_SuccessText;
            result = true;
        }

        return result;
    }
    bool Unwrapping_Internal::SetStep(int setpNumber, const char*
    fileName)
    {
        bool result = false;
        m_StatusText = m_DefaultStatus;

        if (PHASE_STEPPING_CONFIGURED == m_State)
        {
            if (setpNumber > 0 && setpNumber <= 3)
            {
                switch (setpNumber)
                {
                    case 1:
                    {
                        m_Step1FileName = fileName;
                        break;
    }
    case 2:
    {
        m_Step2FileName = fileName;
        break;
    }
    case 3:
    {
        m_Step3FileName = fileName;
        break;
    }
    default:
```

```
    {
        ASSERT(FALSE); //has to be one of the above
        m_StatusText = "invalid setpNumber";
    }
}
if (PathFileExists(fileName))
{
    m_StatusText = m_SuccessText;
    result = true;
}
else
{
    m_StatusText.Format("the file \"%s\" does not exist",
fileName);
            }
        }
    }
    else
    {
        m_StatusText = "invalid phase stepping configuration";
    }

    if (PHASE_STEPPING_CONFIGURED == m_State &&
        PathFileExists(m_Step1FileName) &&
        PathFileExists(m_Step2FileName) &&
        PathFileExists(m_Step3FileName))
    {
        m_State = READY_TO_GENERATE_WRAPPED_MAP;
    }

    return result;
}

bool Unwrapping_Internal::GenerateWrappedMap(const char*
wrappedFileName)
{
    bool result = false;
    m_StatusText = m_DefaultStatus;

    if (READY_TO_GENERATE_WRAPPED_MAP == m_State)
    {
        m_WrappedMapFileName = wrappedFileName;
        if (GenerateWrappedPhaseMap())
        {
            m_State = READY_TO_UNWRAP;
            m_StatusText = m_SuccessText;
            result = true;
        }
    }
    else
    {
        m_StatusText = "Not ready for unwrapping, prior configuration
may be required";
    }

    return result;
}

CString Unwrapping_Internal::GetModifiedFileName(const char*
modifierString, const CString& fileName)
{
    CString pathlessFileName = PathFindFileName(fileName);
    CString pathName = fileName.Left(fileName.GetLength() -
pathlessFileName.GetLength());
    CString modifiedFileName;
```

```
    modifiedFileName.Format("%s%s%s", pathName, modifierString,
pathlessFileName);
    return modifiedFileName;
}

bool Unwrapping_Internal::GenerateUnWrappedMap(const char*
unwrappedFileName)
{
    bool result = false;
    m_StatusText = m_DefaultStatus;
    if(PrepareImagePixelArray() &&
        TileUnwrapImage(m_WrappedMapFileName, m_SkeletonFileName,
unwrappedFileName))
    {
        m_StatusText = m_SuccessText;
        result = true;
    }
    return result;
}

bool Unwrapping_Internal::FilterWrappedMap(int passes)
{
    bool result = false;
    m_StatusText = m_DefaultStatus;

    if (READY_TO_UNWRAP == m_State)
    {
        CString streachedImageName =
GetModifiedFileName("ContrastSteached", m_WrappedMapFileName);
        CString filteredIamgeName = GetModifiedFileName("Filtered",
m_WrappedMapFileName);
        CString thinnedImageName = GetModifiedFileName("Thinned",
m_WrappedMapFileName);
        if (ContrastStreachImage(m_WrappedMapFileName,
streachedImageName))
        {
            CString inputFileName = streachedImageName;
            bool errorEncounteredDuringFilteringPasses = false;
            for (int filteringPassesCount = 0; filteringPassesCount !=
passes; filteringPassesCount++)
            {
                CString sequentailFilteredName;
                sequentailFilteredName.Format("Filtered_%02d",
filteringPassesCount);
                CString sequentailThinnedName;
                sequentailThinnedName.Format("Thinned_%02d",
filteringPassesCount);
                CString sequentailStreachedName;
                sequentailStreachedName.Format("Streached_%02d",
filteringPassesCount);
                filteredIamgeName =
GetModifiedFileName(sequentailFilteredName, m_WrappedMapFileName);
                thinnedImageName =
GetModifiedFileName(sequentailThinnedName, m_WrappedMapFileName);
                streachedImageName =
GetModifiedFileName(sequentailStreachedName, m_WrappedMapFileName);
                if (!FilterImage(inputFileName, filteredIamgeName,
thinnedImageName) ||
                    !ContrastStreachImage(filteredIamgeName,
streachedImageName))
                {
                    errorEncounteredDuringFilteringPasses = true;
                    break;
                }
                inputFileName = filteredIamgeName;
```

```
            }
            if (!errorEncounteredDuringFilteringPasses)
            {
                m_WrappedMapFileName = filteredIamgeName;
                m_EdgeFileName = thinnedImageName;
                m_StatusText = m_SuccessText;
                result = true;
            }
        }
    }
    else
    {
        m_StatusText = "Wrapped map needs to be first generated or its
    file name needs to be specified.";
        }
        return result;
    }

    bool Unwrapping_Internal::SetWrappedMap(const char* wrappedFileName)
    {
        bool result = false;
        m_StatusText = m_DefaultStatus;

        if (PathFileExists(wrappedFileName))
        {
            m_WrappedMapFileName = wrappedFileName;
            m_State = READY_TO_UNWRAP;
            m_StatusText = m_SuccessText;
            result = true;
        }
        return result;
    }

    bool Unwrapping_Internal::IsVlidProcessingRegion(const CRect&
    rect)const
    {
        return (rect.TopLeft().x < rect.BottomRight().x &&
    rect.TopLeft().y < rect.BottomRight().y);
    }

    bool Unwrapping_Internal::SetProcessingRegion(int xTopLeft, int
    yTopLeft, int xBottomRight, int yBottomRight)
    {
        bool result = false;
        m_StatusText = "invalid processing region: ensure xTopLeft <
    xBottomRight and yTopLeft < yBottomRight";

        if (xTopLeft < xBottomRight && yTopLeft < yBottomRight)
        {
            m_ProcessingRegion.SetRect(xTopLeft, yTopLeft, xBottomRight,
    yBottomRight);
            m_StatusText = m_SuccessText;
            result = true;
        }
        return result;
    }

    bool Unwrapping_Internal::SetUnwrappingStartPoint(int x, int y)
    {
        bool result = false;
        m_StatusText = "invalid start point: ensure start point is in
    processing region";

        if (m_ProcessingRegion.PtInRect(CPoint(x, y)))
        {
```

```
        m_UnwrappingStartPoint.SetPoint(x, y);
        m_StatusText = m_SuccessText;
        result = true;
    }
    return result;
}

bool Unwrapping_Internal::SkeletonWrappedMap(const char*
skeletonFileName)
{
    bool result = false;
    m_SkeletonFileName = skeletonFileName;

    if (!IsVlidProcessingRegion(m_ProcessingRegion))
    {
        m_StatusText = "need to specify the processing region first";
    }
    else if (READY_TO_UNWRAP != m_State)
    {
        m_StatusText = "need to generate wrapped phase map first";
    }
    else if (SkeletonImage())
    {
        m_StatusText = m_SuccessText;
        result = true;
    }

    return result;
}

void Unwrapping_Internal::SetEdgeInfo(int x, int y, bool isEdge)
{
    int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
    int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
    int imageIndex = (imageYCoord * MAX_IMAGE_SIZE) + imageXcoord;

    m_EdgeInfo[ imageIndex] ;
}

////////////////////////////////////
//
// Exported Unwrapping
//
////////////////////////////////////
Unwrapping_Internal* pUnwrappingInternal = &unwrappingInternal;

Unwrapping::Unwrapping()
{
}

const char* Unwrapping::GetStatus()
{
    return pUnwrappingInternal->GetStatus();
}

bool Unwrapping::ConfigurePhaseStepping(int steps)
{
    return pUnwrappingInternal->ConfigurePhaseStepping(steps);
}

bool Unwrapping::SetStep(int setpNumber, const char* fileName)
{
    return pUnwrappingInternal->SetStep(setpNumber, fileName);
```

```
}


bool Unwrapping::GenerateWrappedMap(const char* wrappedFileName)
{
   return pUnwrappingInternal->GenerateWrappedMap(wrappedFileName);
}

bool Unwrapping::GenerateUnWrappedMap(const char* unWrappedFileName)
{
   return pUnwrappingInternal-
>GenerateUnWrappedMap(unWrappedFileName);
}


bool Unwrapping::SetWrappedMap(const char* wrappedFileName)
{
   return pUnwrappingInternal->SetWrappedMap(wrappedFileName);
}

bool Unwrapping::SetProcessingRegion(int xTopLeft, int yTopLeft, int
xBottomRight, int yBottomRight)
{
   return pUnwrappingInternal->SetProcessingRegion(xTopLeft,
yTopLeft, xBottomRight, yBottomRight);
}

bool Unwrapping::SetUnwrappingStartPoint(int x, int y)
{
   return pUnwrappingInternal->SetUnwrappingStartPoint(x, y);
}

bool Unwrapping::SkeletonWrappedMap(const char* skeletonFileName)
{
   return pUnwrappingInternal->SkeletonWrappedMap(skeletonFileName);
}

bool Unwrapping::FilterWrappedMap(int passes)
{
   return pUnwrappingInternal->FilterWrappedMap(passes);
}
```

GenerateWrappedPhaseMap.cpp

```
#include "stdafx.h"
#include "unwrapping.h"
#include "CDibApiWrapper.h"
#include <math.h>

bool Unwrapping_Internal::LoadFile(CFile& file, const CString&
fileName)
{
	if (file.Open(fileName, CFile::modeRead |
CFile::shareDenyWrite))
	{
		return true;
	}
	return false;
}

bool Unwrapping_Internal::GenerateWrappedPhaseMap()
{
	// establish that all the files specified for the steps can be
opened
	// and are in a correct format
	CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
	ASSERT(pDibApiWrapper != NULL);
	CFile step1File, step2File, step3File;
	HDIB hStep1DIB = NULL;
	HDIB hStep2DIB = NULL;
	HDIB hStep3DIB = NULL;
	if (LoadFile(step1File, m_Step1FileName))
	{
		hStep1DIB = pDibApiWrapper->ReadDIBFile(step1File);
		if (hStep1DIB == NULL)
		{
			m_StatusText = "the format of Step 1 image has not
been recognised";
		}
	}
	else
	{
		m_StatusText = "the file of Step 1 image could not be
openned";
	}

	if (LoadFile(step2File, m_Step2FileName))
	{
		hStep2DIB = pDibApiWrapper->ReadDIBFile(step2File);
		if (hStep2DIB == NULL)
		{
			m_StatusText = "the format of Step 2 image has not
been recognised";
		}
	}
	else
	{
		m_StatusText = "the file of Step 2 image could not be
openned";
	}

	if (LoadFile(step3File, m_Step3FileName))
	{
		hStep3DIB = pDibApiWrapper->ReadDIBFile(step3File);
```

```
            if (hStep3DIB == NULL)
            {
                  m_StatusText = "the format of Step 3 image has not
been recognised";
            }
      }
      else
      {
            m_StatusText = "the file of Step 3 image could not be
openned";
      }

      if (hStep1DIB && hStep2DIB && hStep3DIB)
      {
            // it looks like we have all we need to compute
            // the wrapped phse map
            LPSTR pStep1Hdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hStep1DIB);
            LPSTR pStep1Buffer = pDibApiWrapper-
>FindDIBBits(pStep1Hdr);

            LPSTR pStep2Hdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hStep2DIB);
            LPSTR pStep2Buffer = pDibApiWrapper-
>FindDIBBits(pStep2Hdr);

            LPSTR pStep3Hdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hStep3DIB);
            LPSTR pStep3Buffer = pDibApiWrapper-
>FindDIBBits(pStep3Hdr);

            // check that all image sizes are the same
            if (pDibApiWrapper->DIBHeight(pStep1Hdr) ==
pDibApiWrapper->DIBHeight(pStep2Hdr) &&
                  pDibApiWrapper->DIBHeight(pStep2Hdr) ==
pDibApiWrapper->DIBHeight(pStep3Hdr) &&
                  pDibApiWrapper->DIBWidth(pStep1Hdr) ==
pDibApiWrapper->DIBWidth(pStep2Hdr) &&
                  pDibApiWrapper->DIBWidth(pStep2Hdr) ==
pDibApiWrapper->DIBWidth(pStep3Hdr))
            {
                  // create a new DIB the same size of any of the
step images
                  HDIB hDIBWrapped = pDibApiWrapper-
>CloneDIB(hStep1DIB);
                  BYTE* pWrappedBuffer =
                        ((BYTE*)hDIBWrapped +
                        sizeof(BITMAPINFOHEADER) +
                        pDibApiWrapper->PaletteSize(pStep1Hdr));

                  long int dibSize = pDibApiWrapper-
>DIBHeight(pStep1Hdr) *
                                                pDibApiWrapper-
>DIBWidth(pStep1Hdr);
                  for (long int iCounter=0; iCounter<dibSize;
iCounter++)
                  {
                        double intensity1 = pStep1Buffer[ iCounter];
                        double intensity2 = pStep2Buffer[ iCounter];
                        double intensity3 = pStep3Buffer[ iCounter];
                        double arctan = atan2 (intensity3-
intensity2, intensity1-intensity2);
                        //double arctan = atan2
(sqrt(3.0)*(intensity1-intensity3), (2.0*intensity2)-intensity1-
intensity3);
```

```
                        // this is scaled to 8 bit grey scale i.e.
256/(2*pi) = 40.743665431 ...
                        ((BYTE*)pWrappedBuffer)[iCounter] =
char(int(arctan * 40.74366543152520595683424342336 4));
            if (((BYTE*)pWrappedBuffer)[iCounter] <=
m_LowModulationIntensityThreshold)
            {
                m_LowModulationInfo[iCounter] = true;
            }
                }

#if 0
//start computer generated wrapped phase map

                int heighst_grey = 0;
                int lowest_grey = 255;
                int x,y;
                div_t div_result;
                double
grey_level,grey_level_1,grey_level_2,grey_level_3, grey_level_trunc,
arctan;

                for (long i=0; i<512*512;i++)
                {
                    div_result = div (i, 512);
                    x = 255 - div_result.quot;
                                    //+255 <= y <= -255;
                    y = (i-(div_result.quot*512)) - 255;
                                //-255 <= x <= +255;

      grey_level_1=128*(1+sin((pow((pow(x,2)+pow(y,2)),0.5) *
/*pow(2,0.5)
*/5*3.14159265359/127.5)+(0.25*3.14159265359)));//(arctan)/0.0039062
5;///0.0245436926;
      //convert to a grey scale

      grey_level_2=128*(1+sin((pow((pow(x,2)+pow(y,2)),0.5) *
/*pow(2,0.5)
*/5*3.14159265359/127.5)+(0.75*3.14159265359)));//(arctan)/0.0039062
5;///0.0245436926;
      //convert to a grey scale

      grey_level_3=128*(1+sin((pow((pow(x,2)+pow(y,2)),0.5) *
/*pow(2,0.5)
*/5*3.14159265359/127.5)+(1.25*3.14159265359)));//(arctan)/0.0039062
5;///0.0245436926;
      //convert to a grey scale
                    y = int(grey_level_3 - grey_level_2);
                    x = int(grey_level_1 - grey_level_2);
                    arctan = atan2(double(abs(y)),abs(x));
                    if ((x<=0)&&(y>0)) arctan = (1 *
3.14159265359) - arctan;            //upper left quadrent
                    if ((x<=0)&&(y<=0)) arctan = arctan + (1   *
3.14159265359);   //lower left quadrent
                    if ((x>0)&&(y<=0)) arctan = (2 *
3.14159265359) - arctan;            //lower right quadrent
                    grey_level = arctan/0.0245436926;
                    modf(grey_level, &grey_level_trunc);
                                //truncate
                    ((BYTE*)pWrappedBuffer)[i] =
char(grey_level_trunc);                                 //assign the
pixle in the xy coordinate to grey level
                    if (heighst_grey < grey_level_trunc)
heighst_grey = int(grey_level_trunc);
                    if (lowest_grey > grey_level_trunc)
```

```
lowest_grey = int(grey_level_trunc);
                    }

                    for (i=0; i<512*512;i++)
                    {
                          grey_level = (((BYTE*)pWrappedBuffer)[ i] -
lowest_grey)*(255.0/(heighst_grey-lowest_grey));
                          modf(grey_level, &grey_level_trunc);
                                //truncate
                          ((BYTE*)pWrappedBuffer)[ i] =
char(grey_level_trunc);                                    //assign the
pixle in the xy coordinate to grey level
                    }

//end computer generated wrapped phase map
#endif //0
                    ::GlobalUnlock((HGLOBAL) pStep1Hdr);
                    ::GlobalUnlock((HGLOBAL) pStep2Hdr);
                    ::GlobalUnlock((HGLOBAL) pStep3Hdr);

                    // create new file (or use already created on) for
the
                    // wrapped phase map
                    pDibApiWrapper->SaveDIB(hDIBWrapped,
CFile(m_WrappedMapFileName, CFile::modeWrite|CFile::modeCreate));
          pDibApiWrapper->DestroyClonedDIB(hDIBWrapped);
                    return true;
               }
               else
               {
                    m_StatusText = "the size of all step images should
be equal";
               }
               return false;
          }
```

PrepareImagePixelArray.cpp

```
#include "stdafx.h"
#include "unwrapping.h"
#include "CDibApiWrapper.h"

bool Unwrapping_Internal::PrepareImagePixelArray()
{
    //open wrapped image file
       CFile wrappedImageFile;
    if (!LoadFile(wrappedImageFile, m_WrappedMapFileName))
    {
       m_StatusText = "Could not open wrapped phase map file";
       return false;
    }

       CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
       HDIB hWrppedImageDIB = pDibApiWrapper-
>ReadDIBFile(wrappedImageFile);

       if (hWrppedImageDIB == NULL)
       {
             m_StatusText = "The format of the wrapped phase map file
has not been recognised";
       return false;
       }

       LPSTR pWrappedImageHdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hWrppedImageDIB);
       BYTE* pWrappedImageBuffer =
             ((BYTE*)pWrappedImageHdr +
             sizeof(BITMAPINFOHEADER) +
             pDibApiWrapper->PaletteSize(pWrappedImageHdr));

    long int wrappedImage_DibHeight = pDibApiWrapper-
>DIBHeight(pWrappedImageHdr);
       long int wrappedImage_DibWidth = pDibApiWrapper-
>DIBWidth(pWrappedImageHdr);

    //open skeletoned image file
       CFile skeletonedImageFile;
    if (!LoadFile(skeletonedImageFile, m_SkeletonFileName))
    {
       m_StatusText = "Could not open skeletoned image file";
       return false;
    }

       HDIB hSkeletonedImageDIB = pDibApiWrapper-
>ReadDIBFile(skeletonedImageFile);
       if (hSkeletonedImageDIB == NULL)
       {
             m_StatusText = "The format of the skeletoned image file
has not been recognised";
       return false;
       }

       LPSTR pSkeletonedImageHdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hSkeletonedImageDIB);
       BYTE* pSkeletonedImageBuffer =
             ((BYTE*)pSkeletonedImageHdr +
             sizeof(BITMAPINFOHEADER) +
             pDibApiWrapper->PaletteSize(pSkeletonedImageHdr));
```

```
PrepareImagePixel_stc stc;
stc.m_pWrappedFileImage = pWrappedImageBuffer;
stc.m_pSkeletonedFileImage = pSkeletonedImageBuffer;
stc.m_ImageWidth = wrappedImage_DibWidth;

   for(int y = 0; y != m_ProcessingRegion.Height(); y++)
   {
         for(int x = 0; x != m_ProcessingRegion.Width(); x++)
         {
         stc.m_ImageXcoord = x + m_ProcessingRegion.TopLeft().x;
         stc.m_ImageYCoord = y + m_ProcessingRegion.TopLeft().y;
         stc.m_ImageIndex = (stc.m_ImageYCoord *
wrappedImage_DibWidth) + stc.m_ImageXcoord;
         stc.m_ImagePixelInfo_xCoord = x;
         stc.m_ImagePixelInfo_yCoord = m_ProcessingRegion.Height() -
1 - y;
         stc.m_ImagePixelInfoIndex = (stc.m_ImagePixelInfo_yCoord *
m_ProcessingRegion.Width()) + stc.m_ImagePixelInfo_xCoord;
         LoadImagePixelInfo(stc);
            }
   }

      for(int y = 0; y != m_ProcessingRegion.Height(); y++)
      {
            for(int x = 0; x != m_ProcessingRegion.Width(); x++)
            {
         stc.m_ImageXcoord = x + m_ProcessingRegion.TopLeft().x;
         stc.m_ImageYCoord = y + m_ProcessingRegion.TopLeft().y;
         stc.m_ImageIndex = (stc.m_ImageYCoord *
wrappedImage_DibWidth) + stc.m_ImageXcoord;
         stc.m_ImagePixelInfo_xCoord = x;
         stc.m_ImagePixelInfo_yCoord = m_ProcessingRegion.Height() -
1 - y;
         stc.m_ImagePixelInfoIndex = (stc.m_ImagePixelInfo_yCoord *
m_ProcessingRegion.Width()) + stc.m_ImagePixelInfo_xCoord;
         PrepareImagePixelInfo(stc);
      }
   }

   ::GlobalUnlock((HGLOBAL) hSkeletonedImageDIB);
   ::GlobalUnlock((HGLOBAL) hWrppedImageDIB);

   m_StatusText = m_SuccessText;
   return true;
}

void Unwrapping_Internal::LoadImagePixelInfo(const
PrepareImagePixel_stc& stc)
{

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .Reset();

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetValue(stc.m_pWrappedFileImage[ stc.m_ImageIndex] );
   //char message[ 100] ;
   //sprintf(message, "Pixel[ %d][ %d] =%d\n",
stc.m_ImagePixelInfo_yCoord, stc.m_ImagePixelInfo_xCoord,
m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .GetValue());
   //TRACE(message);
}

void Unwrapping_Internal::PrepareImagePixelInfo(const
```

```
PrepareImagePixel_stc& stc)
{

   const int& y = stc.m_ImagePixelInfo_yCoord;
   const int& x = stc.m_ImagePixelInfo_xCoord;

   if (y > 1 && y < MAX_IMAGE_SIZE - 1 && x > 1 && x <
MAX_IMAGE_SIZE - 1)
   {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_NW(&m_ImagePixelArray[ y-1][ x-1] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_NN(&m_ImagePixelArray[ y-1][ x] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_NE(&m_ImagePixelArray[ y-1][ x+1] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_WW(&m_ImagePixelArray[ y][ x-1] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_EE(&m_ImagePixelArray[ y][ x+1] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_SW(&m_ImagePixelArray[ y+1][ x-1] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_SS(&m_ImagePixelArray[ y+1][ x] );

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetNeighbour_SE(&m_ImagePixelArray[ y+1][ x+1] );

      if (stc.m_pSkeletonedFileImage[ stc.m_ImageIndex]  == 0)
      {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeType(ImagePixel_cls::EDGE);

         if
(m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo
_xCoord+1] .GetValue()/*east*/ >
/*west*/m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePi
xelInfo_xCoord-1] .GetValue())
         {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeHorizontalShape(ImagePixel_cls::EAST_HIGH);
         }
         else
         {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeHorizontalShape(ImagePixel_cls::WEST_HIGH);
         }

         if (m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord-
1][ stc.m_ImagePixelInfo_xCoord] .GetValue()/*north*/ >
/*south*/m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord+1][ stc.m_Imag
ePixelInfo_xCoord] .GetValue())
         {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeVerticalShape(ImagePixel_cls::NORTH_HIGH);
```

```
            }
            else
            {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeVerticalShape(ImagePixel_cls::SOUTH_HIGH);
            }

            if (m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord-
1][ stc.m_ImagePixelInfo_xCoord-1] .GetValue()/*north-west*/ >
/*south-
east*/m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord+1][ stc.m_ImagePi
xelInfo_xCoord+1] .GetValue())
            {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeDiagonalShape(ImagePixel_cls::NORTH_WEST_HIGH)
;
            }
            else
            {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo
xCoord] .SetThinnedEdgeDiagonalShape(ImagePixel_cls::SOUTH_EAST_HIGH)
;
            }

            if (m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord-
1][ stc.m_ImagePixelInfo_xCoord+1] .GetValue()/*north-east*/ >
/*south-
west*/m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord+1][ stc.m_ImagePi
xelInfo_xCoord-1] .GetValue())
            {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeReverseDiagonalShape(ImagePixel_cls::NORTH_EAS
T_HIGH);
            }
            else
            {

m_ImagePixelArray[ stc.m_ImagePixelInfo_yCoord][ stc.m_ImagePixelInfo_
xCoord] .SetThinnedEdgeReverseDiagonalShape(ImagePixel_cls::SOUTH_WES
T_HIGH);
            }
        }
    }
}
```

Skeleton_Judge.h

```
#ifndef SKELETON_JUDGE_H
#define SKELETON_JUDGE_H

class Unwrapping_Internal;

class Skeleton_Judge
{
public:
   Skeleton_Judge(Unwrapping_Internal& unwrapping_Internal, BYTE*
const pImageBytes, int imageWidth, const CRect& processingRegion);
   ~Skeleton_Judge();

   void GetSkeleton(std::vector<std::vector<BYTE> >& skeleton);

private:
   Unwrapping_Internal& m_Unwrapping_Internal;
   BYTE* m_pImageBytes;
   const CRect m_ProcessingRegion;
   std::vector<std::vector<BYTE> > I; //image 2-D array
   typedef std::vector<std::vector<BYTE> >::iterator
Y_ImageIterator;
   typedef std::vector<BYTE>::iterator X_ImageIterator;

   struct Edge
   {
      enum State
      {
         NONE,
         TENTATIVE,
         TENTATIVE_ADJACENT_TO_CONFIRMED,
         CONFIRMED,
         HILDITCH_EDGE,
         HILDITCH_REMOVED_EDGE,
         HILDITCH_NO_EDGE,
      };
      Edge (State state) : m_State(state) {}
      State m_State;
      State m_HilditchState;
      bool IsThinnable()const { return m_State == CONFIRMED ||
m_State == TENTATIVE_ADJACENT_TO_CONFIRMED;}
   };

   std::vector<std::vector<Edge> > E; //edges 2-D array
   typedef std::vector<std::vector<Edge> >::iterator Y_EdgeIterator;
   typedef std::vector<Edge>::iterator X_EdgeIterator;
   static const int m_HighThreshold = 255;//250
   static const int m_LowThreshold = 255;//240

   //edge detection
   //--------------
   void EdgeDetectImage();
   Edge::State GetEdgeDetected(int x, int y) const;
   bool HasValidNeighbourhood_5x5(int x, int y) const;
   bool HasValidNeighbourhood_3x3(int x, int y) const;

   //thinning
   //--------
   void ThinEdges_Hilditch();
   bool DoesSatisfyHilditchCondition_1(int x, int y) const;
   bool DoesSatisfyHilditchCondition_2(int x, int y) const;
   bool DoesSatisfyHilditchCondition_3(int x, int y) const;
```

```
    bool DoesSatisfyHilditchCondition_4(int x, int y) const;
    bool DoesSatisfyHilditchCondition_5(int x, int y) const;
    bool DoesSatisfyHilditchCondition_6(int x, int y) const;

    //skeletoning
    //-----------
    bool AddAnyAdjacentEdgesFound();
    bool PromoteAnyAdjacentEdgesFound();

    //Enhancement
    //-----------
    void PruneEdges_Morphological();
};

#endif //SKELETON_JUDGE_H
```

Skeleton_Judge.cpp

```
#include "stdafx.h"
#include "Skeleton_Judge.h"
#include "common_defs.h"
#include "unwrapping.h"

Skeleton_Judge::Skeleton_Judge(Unwrapping_Internal&
unwrapping_Internal, BYTE* pImageBytes, int imageWidth, const CRect&
processingRegion)
: m_Unwrapping_Internal(unwrapping_Internal)
, m_pImageBytes(pImageBytes)
, m_ProcessingRegion(processingRegion)
, I(processingRegion.Height())
, E(processingRegion.Height())
{
    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
            int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
            int imageIndex = (imageYCoord * imageWidth) + imageXcoord;
            I[ y] .push_back(m_pImageBytes[ imageIndex] );
        }
    }

    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            E[ y] .push_back(Edge(Edge::NONE));
        }
    }
}

Skeleton_Judge::~Skeleton_Judge(){}

void Skeleton_Judge::GetSkeleton(std::vector<std::vector<BYTE> >&
skeleton)
{
    EdgeDetectImage();
    ThinEdges_Hilditch();
    do
    {
        AddAnyAdjacentEdgesFound();
        ThinEdges_Hilditch();
    } while (PromoteAnyAdjacentEdgesFound());

    PruneEdges_Morphological();

    //patch perimeters to avoid artefact fringe termination points
    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        if (E[ y][ 2] .m_State == Edge::CONFIRMED)// ||
            //(y < m_ProcessingRegion.Height() - 2 && E[ y+1][ 1] .m_State
== Edge::CONFIRMED) ||
            //(y > 1 && E[ y-2][ 1] .m_State == Edge::CONFIRMED))

        {
            E[ y][ 0] .m_State = Edge::CONFIRMED;
            E[ y][ 1] .m_State = Edge::CONFIRMED;
        }
```

```
        if (E[ y][ m_ProcessingRegion.Width() - 3] .m_State ==
Edge::CONFIRMED)// ||
            //(y < m_ProcessingRegion.Height() - 1 &&
E[ y+1][ m_ProcessingRegion.Width() - 3] .m_State == Edge::CONFIRMED)
||
            //(y > 0 && E[ y-1][ m_ProcessingRegion.Width() - 2] .m_State
== Edge::CONFIRMED))

        {
            E[ y][ m_ProcessingRegion.Width() - 1] .m_State =
Edge::CONFIRMED;
            E[ y][ m_ProcessingRegion.Width() - 2] .m_State =
Edge::CONFIRMED;
        }
    }
    for (int x = 0; x != m_ProcessingRegion.Width(); x++)
    {
        if (E[ 2][ x] .m_State == Edge::CONFIRMED)// ||
            //(x < m_ProcessingRegion.Width() - 1 && E[ 1][ x+1] .m_State
== Edge::CONFIRMED) ||
            //(x > 0 && E[ 1][ x-1] .m_State == Edge::CONFIRMED))
        {
            E[ 0][ x] .m_State = Edge::CONFIRMED;
            E[ 1][ x] .m_State = Edge::CONFIRMED;
        }
        if (E[ m_ProcessingRegion.Height() - 3][ x] .m_State ==
Edge::CONFIRMED)// ||
            //(x < m_ProcessingRegion.Width() - 1 &&
E[ m_ProcessingRegion.Height() - 2][ x+1] .m_State == Edge::CONFIRMED)
||
            //(x > 0 && E[ m_ProcessingRegion.Height() - 2][ x-1] .m_State
== Edge::CONFIRMED))
        {
            E[ m_ProcessingRegion.Height() - 1][ x] .m_State =
Edge::CONFIRMED;
            E[ m_ProcessingRegion.Height() - 2][ x] .m_State =
Edge::CONFIRMED;
        }
    }

    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            E[ y][ x] .m_State == Edge::CONFIRMED ? skeleton[ y][ x]  = 255 :
skeleton[ y][ x]  = 0;
            m_Unwrapping_Internal.SetEdgeInfo(x, y, E[ y][ x] .m_State ==
Edge::CONFIRMED);
        }
    }
}

//edge detection
//---------------
void Skeleton_Judge::EdgeDetectImage()
{
    for(int y = 2; y != m_ProcessingRegion.Height() - 2; y++)
    {
        for (int x = 2; x != m_ProcessingRegion.Width() - 2; x++)
        {
            E[ y][ x] .m_State = GetEdgeDetected(x, y);
        }
    }
}
```

```
Skeleton_Judge::Edge::State Skeleton_Judge::GetEdgeDetected(int x,
int y) const
{
   ASSERT(HasValidNeighbourhood_3x3(x, y));
      int  nW = I[ y-1][ x-2] ;
      int  nN = I[ y-1][ x] ;
      int  nE = I[ y-1][ x+1] ;
      int  wW = I[ y][ x-1] ;
      int  eE = I[ y][ x+1] ;
      int  sW = I[ y+1][ x-1] ;
      int  sS = I[ y+1][ x] ;
      int  sE = I[ y+1][ x+1] ;

      int xSobel = (nW + (2 * wW) + sW) - (nE + (2 * eE) + sE);
      int ySobel = (sW + (2 * sS) + sE) - (nW + (2 * nN) + nE);

      int absXSobel = abs(xSobel);
      int absYSobel = abs(ySobel);

      if (absXSobel >=  m_HighThreshold || absYSobel >=
m_HighThreshold)
      {
      return Edge::CONFIRMED;
      }
      else if (absXSobel > m_LowThreshold || absYSobel >=
m_LowThreshold)
      {
      return Edge::TENTATIVE;
      }
   else
   {
      return Edge::NONE;
   }
}

bool Skeleton_Judge::HasValidNeighbourhood_5x5(int x, int y) const
{
   return (x >= 2) && (x <= (m_ProcessingRegion.Width() - 2)) && (y
>= 2) && (y <= (m_ProcessingRegion.Height() - 2));
}

bool Skeleton_Judge::HasValidNeighbourhood_3x3(int x, int y) const
{
   return (x > 0 && y > 0 && x < m_ProcessingRegion.Width() - 1 && y
<  m_ProcessingRegion.Height() - 1);
}

//skeletoning
//-----------
bool Skeleton_Judge::AddAnyAdjacentEdgesFound()
{
   //A TENTATIVE edge adjacent to a CONFIRMED edge
   //is promoted to a TENTATIVE_ADJACENT_TO_CONFIRMED edge.
   //Returns false if none are found

   bool anyFound = false;
   for(int y = 0; y != m_ProcessingRegion.Height(); y++)
   {
      for (int x = 0; x != m_ProcessingRegion.Width(); x++)
      {
         if (E[ y][ x] .m_State == Edge::TENTATIVE)
         {
            if ((y > 0 && E[ y-1][ x] .m_State == Edge::CONFIRMED) ||
                (y < m_ProcessingRegion.Height() - 1 &&
E[ y+1][ x] .m_State == Edge::CONFIRMED) ||
```

```
                    (x > 0 && E[ y][ x-1] .m_State == Edge::CONFIRMED) ||
                    (x < m_ProcessingRegion.Width() - 1 &&
E[ y][ x+1] .m_State == Edge::CONFIRMED))
                {
                    E[ y][ x] .m_State =
Edge::TENTATIVE_ADJACENT_TO_CONFIRMED;
                    anyFound = true;
                }
            }
        }
    }
    return anyFound;
}

bool Skeleton_Judge::PromoteAnyAdjacentEdgesFound()
{
    //A TENTATIVE_ADJACENT_TO_CONFIRMED edge is promoted to a
    //CONFIRMED edge unconditionally.
    //Returns false if none are found
    bool anyFound = false;
    for(Y_EdgeIterator row = E.begin(); row != E.end(); row++)
    {
        for (X_EdgeIterator col = row->begin(); col != row->end();
col++)
        {
            if (col->m_State == Edge::TENTATIVE_ADJACENT_TO_CONFIRMED)
            {
                col->m_State = Edge::CONFIRMED;
                anyFound = true;
            }
        }
    }
    return anyFound;
}

void Skeleton_Judge::ThinEdges_Hilditch()
{
    //initialise
    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (E[ y][ x] .IsThinnable())
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_EDGE;
            }
            else
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_NO_EDGE;
            }
        }
    }

    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (DoesSatisfyHilditchCondition_1(x, y) &&
                DoesSatisfyHilditchCondition_2(x, y) &&
                DoesSatisfyHilditchCondition_3(x, y) &&
                DoesSatisfyHilditchCondition_4(x, y) &&
                DoesSatisfyHilditchCondition_5(x, y) &&
                DoesSatisfyHilditchCondition_6(x, y))
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_REMOVED_EDGE;
```

```
//all conditions are satisfied so thin edge...
            }
        }
    }

    //remove all edges thinnied by Hilditch
    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (E[ y][ x] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE)
            {
                E[ y][ x] .m_State = Edge::NONE;
            }
        }
    }
}

bool Skeleton_Judge::DoesSatisfyHilditchCondition_1(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));
    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 1 - Edge Presence:
    //ensure P1 is an edge
    return (E[ y][ x] .m_HilditchState == Edge::HILDITCH_EDGE);
}

bool Skeleton_Judge::DoesSatisfyHilditchCondition_2(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 2 - Edge is a boundary point:
    //ensure P2 + P4 + P6 + P8 <= 3(edges)
    int edgeCount = 0;
    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0; //0 is a third operand dummy
    E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
```

```
edgeCount++ : 0;

    return edgeCount <= 3;
}

bool Skeleton_Judge::DoesSatisfyHilditchCondition_3(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 3 - End points should be preserved:
    //ensure P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9 >= 2 (edges)
    int edgeCount = 0;
    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0; //0 is a third operand dummy
    E[ y-1][ x+1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x+1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x+1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x-1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x-1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    E[ y-1][ x-1].m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y-1][ x+1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x+1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y+1][ x+1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y+1][ x-1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x-1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y-1][ x-1].m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;

    return edgeCount >= 2;
}

bool Skeleton_Judge::DoesSatisfyHilditchCondition_4(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
```

```
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 4 - Isolated points should be preserved:
    //ensure P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9 >= 1 (edges)
    int edgeCount = 0;
    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0; //0 is a third operand dummy
    E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    return edgeCount >= 1;
}

bool Skeleton_Judge::DoesSatisfyHilditchCondition_5(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 5 - Connectivity should be preserved:
    int P2 = E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P3 = E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P4 = E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P5 = E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P6 = E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P7 = E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P8 = E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;
    int P9 = E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ||
E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE;

    int patternCount = 0;
    patternCount += P2 - (P2 * P3 * P4);
    patternCount += P4 - (P4 * P5 * P6);
    patternCount += P6 - (P6 * P7 * P8);
    patternCount += P8 - (P8 * P9 * P2);

    return patternCount == 1;
}
```

```
bool Skeleton_Judge::DoesSatisfyHilditchCondition_6(int x, int y)
const
{
   ASSERT(HasValidNeighbourhood_3x3(x, y));

   // +----+----+----+
   // | P9 | P2 | P3 |
   // +----+----+----+
   // | P8 | P1 | P4 |
   // +----+----+----+
   // | P7 | P6 | P5 |
   // +----+----+----+
   //Condition 6 - Guard against complete erosion of double sided
lines
   // i.e. a || and = patterns are eroded to a | and - patterns
respectively
   int patternCount = 0;
   E[ y-1][ x]   .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0; //0 is a third operand dummy
   E[ y-1][ x+1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y]  [ x+1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y+1][ x+1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y+1][ x]   .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y+1][ x-1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y]  [ x-1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
   E[ y-1][ x-1] .m_HilditchState != Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;

   return patternCount == 8;
}

void Skeleton_Judge::PruneEdges_Morphological()
{
      enum
      {
            NO_EDGE, //"has no edge"
            IS_EDGE, //"has edge"
            XX_CARE  //"don't care"
      };
   enum
   {
         TOP_LEFT =          0,
         NORTH =             1,
         TOP_RIGHT =         2,
         WEST =              3,
         CENTRE =          4,
         EAST =              5,
         BOTTOM_LEFT =   6,
         SOUTH =               7,
         BOTTOM_RIGHT = 8,
         KERNEL_SIZE =  9
   };

      int strcutreKernel[ KERNEL_SIZE] ;
   std::vector<std::vector<Edge> > E_temp = E; //temporary edges 2-D
array to hold "previous" morphological pass values
   std::vector<std::vector<Edge> >* pImageEdgesBefore = &E;
   std::vector<std::vector<Edge> >* pImageEdgesAfter = &E_temp;
```

```
      for(int prunings=0; prunings<10; prunings++)
      {
            for (int strcutreKernelCount=0; strcutreKernelCount < 8;
strcutreKernelCount++)
            {
                  if (strcutreKernelCount == 0)
                  {
                        strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                        strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                        strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       XX_CARE; strcutreKernel[ BOTTOM_RIGHT] =
XX_CARE;
                  }
                  else if (strcutreKernelCount == 1)
                  {
                        strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                        strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                        strcutreKernel[ BOTTOM_LEFT] = XX_CARE;
strcutreKernel[ SOUTH] =       XX_CARE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                  }
                  else if (strcutreKernelCount == 2)
                  {
                        strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   XX_CARE;
                        strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            XX_CARE;
                        strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                  }
                  else if (strcutreKernelCount == 3)
                  {
                        strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                        strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            XX_CARE;
                        strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
XX_CARE;
                  }
                  else if (strcutreKernelCount == 4)
                  {
                        strcutreKernel[ TOP_LEFT] =          XX_CARE;
      strcutreKernel[ NORTH] =       XX_CARE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                        strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                        strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                  }
```

```
                    else if (strcutreKernelCount == 5)
                    {
                            strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       XX_CARE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                            strcutreKernel[ WEST] =              NO_EDGE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                            strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                    }
                    else if (strcutreKernelCount == 6)
                    {
                            strcutreKernel[ TOP_LEFT] =          NO_EDGE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                            strcutreKernel[ WEST] =              XX_CARE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                            strcutreKernel[ BOTTOM_LEFT] = XX_CARE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                    }
                    else if (strcutreKernelCount == 7)
                    {
                            strcutreKernel[ TOP_LEFT] =          XX_CARE;
      strcutreKernel[ NORTH] =       NO_EDGE;
strcutreKernel[ TOP_RIGHT] =   NO_EDGE;
                            strcutreKernel[ WEST] =              XX_CARE;
strcutreKernel[ CENTRE] =      IS_EDGE; strcutreKernel[ EAST] =
            NO_EDGE;
                            strcutreKernel[ BOTTOM_LEFT] = NO_EDGE;
strcutreKernel[ SOUTH] =       NO_EDGE; strcutreKernel[ BOTTOM_RIGHT] =
NO_EDGE;
                    }

                    for (int iCounter=1;
iCounter<m_ProcessingRegion.Height()-1; iCounter++)
                    {
                            for (int jCounter=1;
jCounter<m_ProcessingRegion.Width()-1; jCounter++)
                            {

      (*pImageEdgesAfter)[ iCounter][ jCounter] .m_State =
(*pImageEdgesBefore)[ iCounter][ jCounter] .m_State;
                bool centre =
(*pImageEdgesAfter)[ iCounter][ jCounter] .m_State == Edge::CONFIRMED;
                                if (centre)
                                {
                    bool top_left = (*pImageEdgesBefore)[ iCounter-
1][ jCounter-1] .m_State == Edge::CONFIRMED;
                                        bool north =
(*pImageEdgesBefore)[ iCounter][ jCounter-1] .m_State ==
Edge::CONFIRMED;
                                        bool top_right =
(*pImageEdgesBefore)[ iCounter+1][ jCounter-1] .m_State ==
Edge::CONFIRMED;
                                        bool west =
(*pImageEdgesBefore)[ iCounter-1][ jCounter] .m_State ==
Edge::CONFIRMED;
                                        bool east =
(*pImageEdgesBefore)[ iCounter+1][ jCounter] .m_State ==
Edge::CONFIRMED;
                                        bool bottom_left =
```

```
(*pImageEdgesBefore)[ iCounter-1][ jCounter+1] .m_State ==
Edge::CONFIRMED;
                                        bool south =
(*pImageEdgesBefore)[ iCounter][ jCounter+1] .m_State ==
Edge::CONFIRMED;
                                        bool bottom_right =
(*pImageEdgesBefore)[ iCounter+1][ jCounter+1] .m_State ==
Edge::CONFIRMED;

                                        if (
                                            ((strcutreKernel[ TOP_LEFT]
== XX_CARE) ||     (strcutreKernel[ TOP_LEFT]  == IS_EDGE && top_left)
|| (strcutreKernel[ TOP_LEFT]  == NO_EDGE && !top_left)) &&
                                            ((strcutreKernel[ NORTH]  ==
XX_CARE) || (strcutreKernel[ NORTH]  == IS_EDGE && north) ||
      (strcutreKernel[ NORTH]  == NO_EDGE && !north)) &&

      ((strcutreKernel[ TOP_RIGHT]  == XX_CARE) ||
      (strcutreKernel[ TOP_RIGHT]  == IS_EDGE && top_right) ||
(strcutreKernel[ TOP_RIGHT]  == NO_EDGE && !top_right)) &&
                                            ((strcutreKernel[ WEST]  ==
XX_CARE) || (strcutreKernel[ WEST]  == IS_EDGE && west) ||
      (strcutreKernel[ WEST]  == NO_EDGE && !west)) &&
                                            ((strcutreKernel[ EAST]  ==
XX_CARE) || (strcutreKernel[ EAST]  == IS_EDGE && east) ||
      (strcutreKernel[ EAST]  == NO_EDGE && !east)) &&

      ((strcutreKernel[ BOTTOM_LEFT]  == XX_CARE) ||
      (strcutreKernel[ BOTTOM_LEFT]  == IS_EDGE && bottom_left) ||
(strcutreKernel[ BOTTOM_LEFT]  == NO_EDGE && !bottom_left)) &&
                                            ((strcutreKernel[ SOUTH]  ==
XX_CARE) || (strcutreKernel[ SOUTH]  == IS_EDGE && south) ||
(strcutreKernel[ SOUTH]  == NO_EDGE && !south)) &&

      ((strcutreKernel[ BOTTOM_RIGHT]  == XX_CARE) ||
      (strcutreKernel[ BOTTOM_RIGHT]  == IS_EDGE && bottom_right) ||
(strcutreKernel[ BOTTOM_RIGHT]  == NO_EDGE && !bottom_right))
                                            )
                                        {
                                    (*pImageEdgesAfter)[ iCounter][ jCounter] .m_State
= Edge::NONE;
                                        }

                                    }
                                }
                            }
                            std::vector<std::vector<Edge> >* temp =
pImageEdgesBefore;
                            pImageEdgesBefore = pImageEdgesAfter;
                            pImageEdgesAfter = temp;
                    }
            }
}
```

Skeleton_Yatagai.h

```
#ifndef SKELETON_YATAGAI_H
#define SKELETON_YATAGAI_H

//Ref Yatagai et al, 1982 Opt. Eng. (21) 901-6
class Skeleton_Yatagai
{
public:
   Skeleton_Yatagai(BYTE* const pImageBytes, int imageWidth, const
CRect& processingRegion);
   ~Skeleton_Yatagai();

   void GetSkeleton(std::vector<std::vector<BYTE> >& skeleton);

private:
   BYTE* m_pImageBytes;
   const CRect m_ProcessingRegion;
   std::vector<std::vector<BYTE> > I; //image 2-D array
   typedef std::vector<std::vector<BYTE> >::iterator
Y_ImageIterator;
   typedef std::vector<BYTE>::iterator X_ImageIterator;

   //X -> don't care,
   //T -> edge is present(true),
   //F -> edge not present (false)
   enum StructuringElement{X, T, F};

   struct Edge
   {
      enum State
      {
         NONE,
         TENTATIVE,
         TENTATIVE_ADJACENT_TO_CONFIRMED,
         CONFIRMED,
         HILDITCH_EDGE,
         HILDITCH_REMOVED_EDGE,
         HILDITCH_NO_EDGE,
      };
      Edge (State state) : m_State(state) {}
      State m_State;
      State m_HilditchState;
      bool IsThinnable()const {return m_State == CONFIRMED ||
m_State == TENTATIVE_ADJACENT_TO_CONFIRMED;}
      bool operator ==(StructuringElement rhs) const
      {
         return (rhs == X ||
                 IsThinnable() && rhs == T ||
                 !IsThinnable() && rhs == F);
      }
   };
   std::vector<std::vector<Edge> > E; //edges 2-D array
   typedef std::vector<std::vector<Edge> >::iterator Y_EdgeIterator;
   typedef std::vector<Edge>::iterator X_EdgeIterator;

   //peak detection
   //--------------
   void PeakDetectImage();
   int GetTotalPeaksDetected(int x, int y) const;
   bool HasValidNeighbourhood_5x5(int x, int y) const;
   bool HasValidNeighbourhood_3x3(int x, int y) const;
   bool CanDetectPeakInDirection_X(int x, int y) const;
```

```
bool CanDetectPeakInDirection_Y(int x, int y) const;
bool CanDetectPeakInDirection_XY(int x, int y) const;
bool CanDetectPeakInDirection_nXY(int x, int y) const;

//thinning
//--------
void ThinEdges_Morphological();
void ThinEdges_Hilditch();
bool DoesSatisfyHilditchCondition_1(int x, int y) const;
bool DoesSatisfyHilditchCondition_2(int x, int y) const;
bool DoesSatisfyHilditchCondition_3(int x, int y) const;
bool DoesSatisfyHilditchCondition_4(int x, int y) const;
bool DoesSatisfyHilditchCondition_5(int x, int y) const;
bool DoesSatisfyHilditchCondition_6(int x, int y) const;

    //skeletoning
    //-----------
    bool AddAnyAdjacentEdgesFound();
    bool PromoteAnyAdjacentEdgesFound();
};

#endif //SKELETON_YATAGAI_H
```

Skeleton_Yatagai.cpp

```
#include "stdafx.h"
#include "Skeleton_Yatagai.h"

Skeleton_Yatagai::Skeleton_Yatagai(BYTE* pImageBytes, int
imageWidth, const CRect& processingRegion)
: m_pImageBytes(pImageBytes)
, m_ProcessingRegion(processingRegion)
, I(processingRegion.Height())
, E(processingRegion.Height())
{
    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
            int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
            int imageIndex = (imageYCoord * imageWidth) + imageXcoord;
            I[ y] .push_back(m_pImageBytes[ imageIndex] );
        }
    }

    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            E[ y] .push_back(Edge(Edge::NONE));
        }
    }
}

Skeleton_Yatagai::~Skeleton_Yatagai()
{
}

void Skeleton_Yatagai::GetSkeleton(std::vector<std::vector<BYTE> >&
skeleton)
{
    PeakDetectImage();
    //ThinEdges_Morphological();
    ThinEdges_Hilditch();
    do
    {
        AddAnyAdjacentEdgesFound();
        //ThinEdges_Morphological();
        ThinEdges_Hilditch();
    } while (PromoteAnyAdjacentEdgesFound());

    for(int y = 0; y != m_ProcessingRegion.Height(); y++)
    {
        for (int x = 0; x != m_ProcessingRegion.Width(); x++)
        {
            E[ y][ x] .m_State == Edge::CONFIRMED ? skeleton[ y][ x]  = 255 :
skeleton[ y][ x]  = 0;
        }
    }
}

//skeletoning
//-----------
bool Skeleton_Yatagai::AddAnyAdjacentEdgesFound()
{
```

```
//A TENTATIVE edge adjacent to a CONFIRMED edge
//is promoted to a TENTATIVE_ADJACENT_TO_CONFIRMED edge.
//Returns false if none are found

bool anyFound = false;
for(int y = 0; y != m_ProcessingRegion.Height(); y++)
{
   for (int x = 0; x != m_ProcessingRegion.Width(); x++)
   {
      if (E[ y][ x] .m_State == Edge::TENTATIVE)
      {
         if ((y > 0 && E[ y-1][ x] .m_State == Edge::CONFIRMED) ||
             (y < m_ProcessingRegion.Height() - 1 &&
E[ y+1][ x] .m_State == Edge::CONFIRMED) ||
             (x > 0 && E[ y][ x-1] .m_State == Edge::CONFIRMED) ||
             (x < m_ProcessingRegion.Width() - 1 &&
E[ y][ x+1] .m_State == Edge::CONFIRMED))
            {
               E[ y][ x] .m_State =
Edge::TENTATIVE_ADJACENT_TO_CONFIRMED;
               anyFound = true;
            }
         }
      }
   }
   return anyFound;
}

bool Skeleton_Yatagai::PromoteAnyAdjacentEdgesFound()
{
   //A TENTATIVE_ADJACENT_TO_CONFIRMED edge is promoted to a
   //CONFIRMED edge unconditionally.
   //Returns false if none are found
   bool anyFound = false;
   for(Y_EdgeIterator row = E.begin(); row != E.end(); row++)
   {
      for (X_EdgeIterator col = row->begin(); col != row->end();
col++)
      {
         if (col->m_State == Edge::TENTATIVE_ADJACENT_TO_CONFIRMED)
         {
            col->m_State = Edge::CONFIRMED;
            anyFound = true;
         }
      }
   }
   return anyFound;
}

//peak detection
//--------------
void Skeleton_Yatagai::PeakDetectImage()
{
   for(int y = 2; y != m_ProcessingRegion.Height() - 2; y++)
   {
for (int x = 2; x != m_ProcessingRegion.Width() - 2; x++)
{
   int totalPeaks = GetTotalPeaksDetected(x, y);
   if (totalPeaks >= 2)
   {
      E[ y][ x] .m_State = Edge::CONFIRMED;
   }
   else if (totalPeaks == 1)
   {
       E[ y][ x] .m_State = Edge::TENTATIVE;
```

```
        }
      }
   }
}

int Skeleton_Yatagai::GetTotalPeaksDetected(int x, int y) const
{
   int totalPeaksDetected = 0;
   CanDetectPeakInDirection_X(x, y) ? totalPeaksDetected++ : 0; //0
is a third operand dummy
   CanDetectPeakInDirection_Y(x, y) ? totalPeaksDetected++ : 0;
   CanDetectPeakInDirection_XY(x, y) ? totalPeaksDetected++ : 0;
   CanDetectPeakInDirection_nXY(x, y) ? totalPeaksDetected++ : 0;
   return totalPeaksDetected;
}

bool Skeleton_Yatagai::HasValidNeighbourhood_5x5(int x, int y) const
{
   return (x >= 2) && (x <= (m_ProcessingRegion.Width() - 2)) && (y
>= 2) && (y <= (m_ProcessingRegion.Height() - 2));
}

bool Skeleton_Yatagai::CanDetectPeakInDirection_X(int x, int y)
const
{
   ASSERT (HasValidNeighbourhood_5x5(x, y));
   // left < centre > right
   int left = I[ y][ x-2]  + I[ y-1][ x-2]  + I[ y+1][ x-2] ;
   int centre = I[ y][ x]  + I[ y-1][ x]  + I[ y+1][ x] ;
   int right = I[ y][ x+2]  + I[ y-1][ x+2]  + I[ y+1][ x+2] ;
   return (centre > left && centre > right);
}

bool Skeleton_Yatagai::CanDetectPeakInDirection_Y(int x, int y)
const
{
   ASSERT (HasValidNeighbourhood_5x5(x, y));
   // top < middle > bottom
   int top = I[ y-2][ x-1]  + I[ y-2][ x]  + I[ y-2][ x+1] ;
   int middle = I[ y][ x-1]  + I[ y][ x]  + I[ y][ x+1] ;
   int bottom = I[ y+2][ x-1]  + I[ y+2][ x]  + I[ y+2][ x+1] ;
   return (middle > top && middle > bottom);
}

bool Skeleton_Yatagai::CanDetectPeakInDirection_XY(int x, int y)
const
{
   ASSERT (HasValidNeighbourhood_5x5(x, y));
   // topLeft < centreDiag > bottomRight
   int topLeft = I[ y-2][ x-2]  + I[ y-2][ x-1]  + I[ y-1][ x-2] ;
   int centreDiag = I[ y-1][ x+1]  + I[ y][ x]  + I[ y+1][ x-1] ;
   int bottomRight = I[ y+2][ x+2]  + I[ y+2][ x+1]  + I[ y+1][ x+2] ;
   return (centreDiag > topLeft && centreDiag > bottomRight);
}

bool Skeleton_Yatagai::CanDetectPeakInDirection_nXY(int x, int y)
const
{
   ASSERT (HasValidNeighbourhood_5x5(x, y));
   // topRight < centreDiag > bottomLeft
   int topRight = I[ y-2][ x+2]  + I[ y-2][ x+1]  + I[ y-1][ x+2] ;
   int centreDiag = I[ y-1][ x-1]  + I[ y][ x]  + I[ y+1][ x+1] ;
   int bottomLeft = I[ y+2][ x-2]  + I[ y+2][ x-1]  + I[ y+1][ x-2] ;
   return (centreDiag > topRight && centreDiag > bottomLeft);
```

```
}

//thinning
//--------
void Skeleton_Yatagai::ThinEdges_Morphological()
{
    //there are 8  3X3 structuring elements used
    StructuringElement SE[ 8][ 3][ 3] =
    {
        //element 1
        F, F, F,
        X, T, X,
        T, T, T,
        //element 2
        X, F, F,
        T, T, F,
        X, T, X,
        //element 3, 90 degrees rotation of element 1
        T, X, F,
        T, T, F,
        T, X, F,
        //element 4, 90 degrees rotation of element 2
        X, T, X,
        T, T, F,
        X, F, F,
        //element 5, 90 degrees rotation of element 3
        T, T, T,
        X, T, X,
        F, F, F,
        //element 6, 90 degrees rotation of element 4
        X, T, X,
        F, T, T,
        F, F, X,
        //element 7, 90 degrees rotation of element 5
        F, X, T,
        F, T, T,
        F, X, T,
        //element 8, 90 degrees rotation of element 6
        F, F, X,
        F, T, T,
        X, T, X,
};

bool hasConnverged = true;
do
{
    hasConnverged = true;
    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            for (int i = 0; i != 8; i++)
            {
                if (E[ y-1][ x-1]  == SE[ i][ 0][ 0]  &&
                    E[ y-1][ x]    == SE[ i][ 0][ 1]  &&
                    E[ y-1][ x+1]  == SE[ i][ 0][ 2]  &&
                    E[ y]   [ x-1] == SE[ i][ 1][ 0]  &&
                    E[ y]   [ x]   == SE[ i][ 1][ 1]  &&
                    E[ y]   [ x+1] == SE[ i][ 1][ 2]  &&
                    E[ y+1][ x-1]  == SE[ i][ 2][ 0]  &&
                    E[ y+1][ x]    == SE[ i][ 2][ 1]  &&
                    E[ y+1][ x+1]  == SE[ i][ 2][ 2] )
                {
                    E[ y][ x] .m_State = Edge::NONE; //pattern has
matched so thin edge...
```

```
                    hasConnverged = false; //...and we have not
converged yet
                }
            }
        }
    }
  } while (hasConnverged == false);
}

void Skeleton_Yatagai::ThinEdges_Hilditch()
{
    //initialise
    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (E[ y][ x] .IsThinnable())
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_EDGE;
            }
            else
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_NO_EDGE;
            }
        }
    }

    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (DoesSatisfyHilditchCondition_1(x, y) &&
                DoesSatisfyHilditchCondition_2(x, y) &&
                DoesSatisfyHilditchCondition_3(x, y) &&
                DoesSatisfyHilditchCondition_4(x, y) &&
                DoesSatisfyHilditchCondition_5(x, y) &&
                DoesSatisfyHilditchCondition_6(x, y))
            {
                E[ y][ x] .m_HilditchState = Edge::HILDITCH_REMOVED_EDGE;
//all conditions are satisfied so thin edge...
            }
        }
    }

    //remove all edges thinnied by Hilditch
    for(int y = 1; y != m_ProcessingRegion.Height() - 1; y++)
    {
        for (int x = 1; x != m_ProcessingRegion.Width() - 1; x++)
        {
            if (E[ y][ x] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE)
            {
                E[ y][ x] .m_State = Edge::NONE;
            }
        }
    }
}
bool Skeleton_Yatagai::HasValidNeighbourhood_3x3(int x, int y) const
{
    return (x > 0 && y > 0 && x < m_ProcessingRegion.Width() - 1 && y
<   m_ProcessingRegion.Height() - 1);
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_1(int x, int y)
const
{
```

```
    ASSERT(HasValidNeighbourhood_3x3(x, y));
    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 1 - Edge Presence:
    //ensure P1 is an edge
    return (E[ y][ x] .m_HilditchState == Edge::HILDITCH_EDGE);
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_2(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 2 - Edge is a boundary point:
    //ensure P2 + P4 + P6 + P8 <= 3(edges)
    int edgeCount = 0;
    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0; //0 is a third operand dummy
    E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
    E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
    E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;

    return edgeCount <= 3;
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_3(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));
    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 3 - End points should be preserved:
    //ensure P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9 >= 2 (edges)
    int edgeCount = 0;
    E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
```

```
: 0; //0 is a third operand dummy
   E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

   E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;
   E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
edgeCount++ : 0;

   return edgeCount >= 2;
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_4(int x, int y)
const
{
   ASSERT(HasValidNeighbourhood_3x3(x, y));

   // +----+----+----+
   // | P9 | P2 | P3 |
   // +----+----+----+
   // | P8 | P1 | P4 |
   // +----+----+----+
   // | P7 | P6 | P5 |
   // +----+----+----+
   //Condition 4 - Isolated points should be preserved:
   //ensure P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9 >= 1 (edges)
   int edgeCount = 0;
   E[ y-1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0; //0 is a third operand dummy
   E[ y-1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y]  [ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x]   .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y+1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
   E[ y]  [ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;
```

```
    E[ y-1][ x-1] .m_HilditchState == Edge::HILDITCH_EDGE ? edgeCount++
: 0;

    return edgeCount >= 1;
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_5(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 5 - Connectivity should be preserved:
    //ensure sum(zero-one patterns in sequence
(P2,P3)(P3,P4)...(P9,P2)) == 1
    int patternCount = 0;
    E[ y-1][ x]   .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-
1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
//0 is a third operand dummy
    E[ y-1][ x+1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y]
[ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
    E[ y]   [ x+1] .m_HilditchState != Edge::HILDITCH_EDGE &&
E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++
: 0;
    E[ y+1][ x+1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y+1][ x]
.m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
    E[ y+1][ x]   .m_HilditchState != Edge::HILDITCH_EDGE && E[ y+1][ x-
1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
    E[ y+1][ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y]   [ x-
1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
    E[ y]   [ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-1][ x-
1] .m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;
    E[ y-1][ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-1][ x]
.m_HilditchState == Edge::HILDITCH_EDGE ? patternCount++ : 0;

    E[ y-1][ x]   .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-
1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0; //0 is a third operand dummy
    E[ y-1][ x+1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y]
[ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
    E[ y]   [ x+1] .m_HilditchState != Edge::HILDITCH_EDGE &&
E[ y+1][ x+1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ?
patternCount++ : 0;
    E[ y+1][ x+1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y+1][ x]
.m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ? patternCount++ :
0;
    E[ y+1][ x]   .m_HilditchState != Edge::HILDITCH_EDGE && E[ y+1][ x-
1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ? patternCount++ :
0;
    E[ y+1][ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y]   [ x-
1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ? patternCount++ :
0;
    E[ y]   [ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-1][ x-
1] .m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ? patternCount++ :
0;
    E[ y-1][ x-1] .m_HilditchState != Edge::HILDITCH_EDGE && E[ y-1][ x]
.m_HilditchState == Edge::HILDITCH_REMOVED_EDGE ? patternCount++ :
0;
```

```
    return patternCount == 1;
}

bool Skeleton_Yatagai::DoesSatisfyHilditchCondition_6(int x, int y)
const
{
    ASSERT(HasValidNeighbourhood_3x3(x, y));

    // +----+----+----+
    // | P9 | P2 | P3 |
    // +----+----+----+
    // | P8 | P1 | P4 |
    // +----+----+----+
    // | P7 | P6 | P5 |
    // +----+----+----+
    //Condition 6 - Guard against complete erosion of double sided
lines
    // i.e. a || and = patterns are eroded to a | and - patterns
respectively

//   ASSERT(FALSE); //not needed
    return 1;
}
```

SkeletonImage.cpp

```
#include "stdafx.h"
#include "unwrapping.h"
#include "CDibApiWrapper.h"
#include "Skeleton_Yatagai.h"
#include "Skeleton_Judge.h"

bool Unwrapping_Internal::SkeletonImage()
{
      CFile imageToBeSkeletonedFile;
   if (!LoadFile(imageToBeSkeletonedFile, m_WrappedMapFileName))
   {
      m_StatusText = "Could not open wrapped phase map file";
      return false;
   }

      CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
      HDIB hImageToBeSkeletonedDIB = pDibApiWrapper-
>ReadDIBFile(imageToBeSkeletonedFile);

      if (hImageToBeSkeletonedDIB == NULL)
      {
            m_StatusText = "The format of the image to be skeletoned
has not been recognised";
      return false;
      }

      LPSTR pImageToBeSkeletonedHdr  = (LPSTR)
::GlobalLock((HGLOBAL) hImageToBeSkeletonedDIB);
      BYTE* pImageToBeSkeletonedBuffer =
            ((BYTE*)pImageToBeSkeletonedHdr +
            sizeof(BITMAPINFOHEADER) +
            pDibApiWrapper->PaletteSize(pImageToBeSkeletonedHdr));

      // create a new DIB for the Skeletoned image
      HDIB hSkeletonedDIB = pDibApiWrapper-
>CloneDIB(hImageToBeSkeletonedDIB);
      LPSTR pSkeletonedHdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hSkeletonedDIB);
      BYTE* pSkeletonedBuffer =
            ((BYTE*)pSkeletonedHdr +
            sizeof(BITMAPINFOHEADER) +
            pDibApiWrapper->PaletteSize(pSkeletonedHdr));

      long int dibHeight = pDibApiWrapper-
>DIBHeight(pSkeletonedHdr);
      long int dibWidth = pDibApiWrapper->DIBWidth(pSkeletonedHdr);

   //create skeleton data structure and setup with correct
dimensions
   std::vector<std::vector<BYTE> >
skeleton(m_ProcessingRegion.Height());
   for (int counter = 0 ; counter != m_ProcessingRegion.Height();
counter++)
   {
      skeleton[ counter] =
std::vector<BYTE>(m_ProcessingRegion.Width());
   }

   //Skeleton_Yatagai skeletonAlgorithm(pImageToBeSkeletonedBuffer,
dibWidth, m_ProcessingRegion);
```

```
   Skeleton_Judge skeletonAlgorithm(myself,
pImageToBeSkeletonedBuffer, dibWidth, m_ProcessingRegion);
   skeletonAlgorithm.GetSkeleton(skeleton);

      for (int yCounter=0; yCounter != dibHeight; yCounter++)
      {
            for (int xCounter=0; xCounter != dibWidth; xCounter++)
            {
         int index = (yCounter * dibWidth) + xCounter;
         if (m_ProcessingRegion.PtInRect(CPoint(xCounter,
yCounter)))
         {
            int y = yCounter - m_ProcessingRegion.TopLeft().y;
            int x = xCounter - m_ProcessingRegion.TopLeft().x;
            if (skeleton[ y][ x] == 255)
            {
               pSkeletonedBuffer[ index] = 0;//255;
            }
            else
            {
               if (0)//pImageToBeSkeletonedBuffer[ index] > 60)
               {
                  pSkeletonedBuffer[ index] =
pImageToBeSkeletonedBuffer[ index] - 60;
               }
               else
               {
                  pSkeletonedBuffer[ index] = 255;//0;
               }
            }
         }
         else
         {
            pSkeletonedBuffer[ index] =
pImageToBeSkeletonedBuffer[ index] ;
         }
      }
   }

   ::GlobalUnlock((HGLOBAL) hSkeletonedDIB);
   ::GlobalUnlock((HGLOBAL) hImageToBeSkeletonedDIB);
   pDibApiWrapper->SaveDIB(
         hSkeletonedDIB,
         CFile(m_SkeletonFileName,
CFile::modeWrite|CFile::modeCreate));
   pDibApiWrapper->DestroyClonedDIB(hSkeletonedDIB);

   m_StatusText = m_SuccessText;
   return true;
}
```

TwoTierMedianFilter.h

```
#ifndef TWOTIERMEDIANFILTER_H
#define TWOTIERMEDIANFILTER_H

class TwoTierMedianFilter
{
private:
   int const m_PieIntensity;
   std::vector<int>& m_AllElements;
   std::vector<int> m_LowElements;
   std::vector<int> m_HighElements;
   bool m_IsEdge;
public:
   TwoTierMedianFilter(int pieIntensity, std::vector<int>&
elements):
      m_PieIntensity(pieIntensity),
      m_AllElements(elements),
      m_IsEdge(false)
   {
      std::sort(m_AllElements.begin(), m_AllElements.end());
      for (int counter = 0; counter != m_AllElements.size();
counter++)
      {
         if (m_AllElements[counter] < m_PieIntensity)
         {
            //as m_AllElements is already sorted, m_LowElements will
also be sorted
            m_LowElements.push_back(m_AllElements[counter]);
         }
         else
         {
            //as m_AllElements is already sorted, m_HighElements
will also be sorted
            m_HighElements.push_back(m_AllElements[counter]);
         }
      }
      m_IsEdge = (GetHighMedian() - GetLowMedian()) >
m_PieIntensity;
   }
   bool IsEdge() const
   {
      return m_IsEdge;
   }
   int GetOverallMedian() const
   {
      if (m_IsEdge)
      {
         //At a fringe edge, the value returned is the minimum of
the low group
         //or the maximum in the high group, depending on which
group contains
         //the most members
         if (m_LowElements.size() > m_HighElements.size())
         {
            return *(m_LowElements.begin());
         }
         else
         {
            return m_HighElements[m_HighElements.size()-1];
         }
      }
      else
```

```
        {
            return m_AllElements.size() ?
m_AllElements[ m_AllElements.size()/2]  : 0;
        }
    }
private:
    int GetHighMedian() const
    {
        return m_HighElements.size() ?
m_HighElements[ m_HighElements.size()/2]  : 0;
    }
    int GetLowMedian() const
    {
        return m_LowElements.size() ?
m_LowElements[ m_LowElements.size()/2]  : 0;
    }
};


#endif //TWOTIERMEDIANFILTER_H
```

UnwrapPhaseMap.cpp

```
#include "stdafx.h"
#include "unwrapping.h"
#include "CDibApiWrapper.h"
#include "Tile_cls.h"
#include "Pixel_cls.h"
#include "TiledImage_cls.h"
#include "TwoTierMedianFilter.h"

bool Unwrapping_Internal::
      ContrastStreachImage (CString imageToBeStreachedFileName,
                            CString streachedImageFileName)
{
      CFile imageToBeStreachedFile;
      if (LoadFile(imageToBeStreachedFile,
imageToBeStreachedFileName))
      {
         CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
         HDIB hImageToBeStreachedDIB = pDibApiWrapper-
>ReadDIBFile(imageToBeStreachedFile);
         if (hImageToBeStreachedDIB == NULL)
         {
               m_StatusText = "The format of the image to be
streached has not been recognised";
         }
         else
         {
               LPSTR pImageToBeStreachedHdr  = (LPSTR)
::GlobalLock((HGLOBAL) hImageToBeStreachedDIB);
               BYTE* pImageToBeStreachedBuffer =
                     ((BYTE*)pImageToBeStreachedHdr +
                     sizeof(BITMAPINFOHEADER) +
                     pDibApiWrapper-
>PaletteSize(pImageToBeStreachedHdr));

               // create a new DIB for the thinned image
               HDIB hStreachedDIB = pDibApiWrapper-
>CloneDIB(hImageToBeStreachedDIB);
               BYTE* pStreachedBuffer =
                     ((BYTE*)hStreachedDIB +
                     sizeof(BITMAPINFOHEADER) +
                     pDibApiWrapper-
>PaletteSize(pImageToBeStreachedHdr));

                  long int dibHeight = pDibApiWrapper-
>DIBHeight(pImageToBeStreachedHdr);
                  long int dibWidth = pDibApiWrapper-
>DIBWidth(pImageToBeStreachedHdr);

                  int histogram[ 256] ;
                  for (int i = 0; i != 256; i++)
                  {
                        histogram[ i]  = 0;
                  }

            //construct histogram of image
                     int maxValue = pImageToBeStreachedBuffer[ 0] ;
                     int minValue = pImageToBeStreachedBuffer[ 0] ;
                     for (int y=1; y != m_ProcessingRegion.Height();
y++)
                     {
```

```
                        for (int x=1; x !=
m_ProcessingRegion.Width(); x++)
                        {
                    int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
                    int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
                    int imageIndex = (imageYCoord * dibWidth) +
imageXcoord;

                    int value = pImageToBeStreachedBuffer[ imageIndex] ;
                    if (value < minValue)
                    {
                        minValue = value;
                    }
                    if (value > maxValue)
                    {
                        maxValue = value;
                    }
                    histogram[ value] ++;
                }
            }

                //  //Calculate contrast streached values of image
                    //int pixelCount = 0;
                    //for (i=0; i<256; i++)
                    //{
                    //    pixelCount += histogram[ i] ;
                    //    if (pixelCount > ((dibHeight * dibWidth) /
100) * CONTRAST_STREACH_PERCENTAGE)
                    //    {
                    //            minValue = i;
                    //            break;      //found the grey level
which more than given percentage
                    //    }
                    //}
                    //pixelCount = 0;
                    //for (i = 255; i !=0 ; i--)
                    //{
                    //    pixelCount += histogram[ i] ;
                    //    if (pixelCount > ((dibHeight * dibWidth) /
100) * CONTRAST_STREACH_PERCENTAGE)
                    //    {
                    //            maxValue = i;
                    //            break;      //found the grey level
which more than given percentage
                    //    }
                    //}

                    double contrast = maxValue - minValue;
                    if (contrast == 0)
                    {
                            contrast = 255;
                    }

                        for (int y=1; y != m_ProcessingRegion.Height();
y++)
                        {
                                for (int x=1; x !=
m_ProcessingRegion.Width(); x++)
                                {
                    int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
                    int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
                    int imageIndex = (imageYCoord * dibWidth) +
imageXcoord;

                                    int temp =
```

```
pImageToBeStreachedBuffer[ imageIndex] ;
                                    if (temp > maxValue)
                                    {
                                          pStreachedBuffer[ imageIndex]  = 255;
                                    }
                                    else if (temp < minValue)
                                    {
                                          pStreachedBuffer[ imageIndex]  = 0;
                                    }
                                    else
                                    {
                                       //pStreachedBuffer[ imageIndex]  =
int((double(temp - contrast)) *  (255.0/contrast));
                                       pStreachedBuffer[ imageIndex]  =
int((double(temp *  255.0)  /  contrast));
                                    }
                              }
                        }
                              pDibApiWrapper->SaveDIB(
                                    hStreachedDIB,
                                    CFile(streachedImageFileName,
CFile::modeWrite|CFile::modeCreate));
                        pDibApiWrapper->DestroyClonedDIB(hStreachedDIB);
                        return true;
                  }
            }
            else
            {
                  m_StatusText = "The file of image to be contrast streached
could not be open";
            }
            return false;
}

bool Unwrapping_Internal::
            FilterImage (CString imageToBeFilteredFileName,
                              CString filteredImageFileName,
                              CString thinnedImageFileName)
{
            CFile imageToBeFilteredFile;
            if (LoadFile(imageToBeFilteredFile,
imageToBeFilteredFileName))
            {
                  CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
                  HDIB hImageToBeFilteredDIB = pDibApiWrapper-
>ReadDIBFile(imageToBeFilteredFile);
                  if (hImageToBeFilteredDIB == NULL)
                  {
                        m_StatusText = "The format of the image to be
filtered has not been recognised";
                  }
                  else
                  {
                        LPSTR pImageToBeFilteredHdr  = (LPSTR)
::GlobalLock((HGLOBAL) hImageToBeFilteredDIB);
                        BYTE* pImageToBeFilteredBuffer =
                              ((BYTE*)pImageToBeFilteredHdr +
                              sizeof(BITMAPINFOHEADER) +
                              pDibApiWrapper-
>PaletteSize(pImageToBeFilteredHdr));

                        // create a new DIB for the thinned image
                        HDIB hThinnedDIB = pDibApiWrapper-
>CloneDIB(hImageToBeFilteredDIB);
```

```
                        BYTE* pThinnedBuffer =
                              ((BYTE*)hThinnedDIB +
                              sizeof(BITMAPINFOHEADER) +
                              pDibApiWrapper-
>PaletteSize(pImageToBeFilteredHdr));

                        // create a new DIB for the filtered image
                        HDIB hFilteredDIB = pDibApiWrapper-
>CloneDIB(hImageToBeFilteredDIB);
                        BYTE* pFilteredBuffer =
                              ((BYTE*)hFilteredDIB +
                              sizeof(BITMAPINFOHEADER) +
                              pDibApiWrapper-
>PaletteSize(pImageToBeFilteredHdr));

                        long int dibHeight = pDibApiWrapper-
>DIBHeight(pImageToBeFilteredHdr);
                        long int dibWidth = pDibApiWrapper-
>DIBWidth(pImageToBeFilteredHdr);

                        for (int y=1; y != m_ProcessingRegion.Height();
y++)
                        {
                              for (int x=1; x !=
m_ProcessingRegion.Width(); x++)
                              {
                    int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
                    int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
                    int imageIndex = (imageYCoord * dibWidth) +
imageXcoord;

                                          int centreIndex = imageIndex;
                                          int topLeftIndex = centreIndex -
dibWidth - 1;
                                          int northIndex = topLeftIndex + 1;
                                          int topRightIndex = northIndex + 1;
                                          int westIndex = centreIndex - 1;
                                          int eastIndex = centreIndex + 1;
                                          int bottom1LeftIndex = centreIndex +
dibWidth - 1;
                                          int southIndex = bottom1LeftIndex + 1;
                                          int bottomRightIndex = southIndex + 1;

                                          ASSERT (centreIndex >=0 && centreIndex
< dibWidth * dibHeight);
                                          ASSERT (topLeftIndex >=0 &&
topLeftIndex < dibWidth * dibHeight);
                                          ASSERT (northIndex >=0 && northIndex <
dibWidth * dibHeight);
                                          ASSERT (topRightIndex >=0 &&
topRightIndex < dibWidth * dibHeight);
                                          ASSERT (westIndex >=0 && westIndex <
dibWidth * dibHeight);
                                          ASSERT (eastIndex >=0 && eastIndex <
dibWidth * dibHeight);
                                          ASSERT (bottom1LeftIndex >=0 &&
bottom1LeftIndex < dibWidth * dibHeight);
                                          ASSERT (southIndex >=0 && southIndex <
dibWidth * dibHeight);
                                          ASSERT (bottomRightIndex >=0 &&
bottomRightIndex < dibWidth * dibHeight);

                              std::vector<int> neighborhoodPixels;
                              m_LowModulationInfo[ topLeftIndex]       ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ topLeftIndex] )
```

```
;
                    m_LowModulationInfo[ northIndex]          ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ northIndex] );
                    m_LowModulationInfo[ topRightIndex]       ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ topRightIndex]
);
                    m_LowModulationInfo[ westIndex]           ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ westIndex] );
                    m_LowModulationInfo[ eastIndex]           ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ eastIndex] );
                    m_LowModulationInfo[ bottomlLeftIndex]    ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ bottomlLeftInd
ex] );
                    m_LowModulationInfo[ southIndex]          ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ southIndex] );
                    m_LowModulationInfo[ bottomRightIndex]    ? 0 :
neighborhoodPixels.push_back(pImageToBeFilteredBuffer[ bottomRightInd
ex] );

                    TwoTierMedianFilter
twoTierMedianFilter(PIE_INTENSITY, neighborhoodPixels);
                    m_EdgeInfo[ centreIndex] =
twoTierMedianFilter.IsEdge();
                    m_EdgeInfo[ centreIndex] ? pThinnedBuffer[ centreIndex]
= 0 : pThinnedBuffer[ centreIndex] = 255;
                    pFilteredBuffer[ centreIndex] =
twoTierMedianFilter.GetOverallMedian();
                              }
                         }

               ::GlobalUnlock((HGLOBAL) hImageToBeFilteredDIB);
                    pDibApiWrapper->SaveDIB(
                         hThinnedDIB,
                              CFile(thinnedImageFileName,
CFile::modeWrite|CFile::modeCreate));
                    pDibApiWrapper->SaveDIB(
                         hFilteredDIB,
                         CFile(filteredImageFileName,
CFile::modeWrite|CFile::modeCreate));
               pDibApiWrapper->DestroyClonedDIB(hThinnedDIB);
               pDibApiWrapper->DestroyClonedDIB(hFilteredDIB);
          }
               return true;
     }
     else
     {
          m_StatusText = "The file of image to be filtered could not
be open";
     }
          return false;
}

bool Unwrapping_Internal::
          TileUnwrapImage (CString filteredWrappedPhaseMApImageName,
                         CString thinnedImageFileName,
                         CString unwrappedFileName)
{
          CFile filteredFile;
          CFile thinnedFile;
     HDIB hFilteredImageDIB = NULL;
     HDIB hThinnedImageDIB = NULL;

          CDibApiWrapper* pDibApiWrapper =
CDibApiWrapper::GetUniqueInstance();
```

```
      if (LoadFile(filteredFile, filteredWrappedPhaseMApImageName)
&& LoadFile(thinnedFile, thinnedImageFileName))
      {
            hFilteredImageDIB = pDibApiWrapper-
>ReadDIBFile(filteredFile);
            hThinnedImageDIB = pDibApiWrapper-
>ReadDIBFile(thinnedFile);
            if (hFilteredImageDIB == NULL || hThinnedImageDIB ==
NULL)
            {
                  if (hFilteredImageDIB == NULL)
                  {
                        m_StatusText = "The format of the filtered
image to be tile unwrapped has not been recognised";
                  }
                  if (hThinnedImageDIB == NULL)
                  {
                        m_StatusText = "The format of the thinned
image has not been recognised";
                  }
            return false;
               }
   }
   else
   {
            m_StatusText = "Could not load filtered image, or could
not load thinned image";
      return false;
   }

   LPSTR pFilteredImageHdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hFilteredImageDIB);
      HDIB hTileddDIB = pDibApiWrapper->CloneDIB(hFilteredImageDIB);
      LPSTR pTiledHdr  = (LPSTR) ::GlobalLock((HGLOBAL) hTileddDIB);
      BYTE* pImageToBeTiledBuffer =
            ((BYTE*)hTileddDIB +
            sizeof(BITMAPINFOHEADER) +
            pDibApiWrapper->PaletteSize(pTiledHdr));

      long int dibHeight = pDibApiWrapper-
>DIBHeight(pFilteredImageHdr);
      long int dibWidth = pDibApiWrapper-
>DIBWidth(pFilteredImageHdr);

      ASSERT(dibHeight == 512 && dibWidth == 512);

      const int T = TileDataModel_cls::GetTileSize();
   const int W = m_ProcessingRegion.Width();
   const int H = m_ProcessingRegion.Height();

   BYTE* pImage = new BYTE[ m_ProcessingRegion.Height() *
m_ProcessingRegion.Width()];
      for(int y = 0; y != m_ProcessingRegion.Height(); y++)
      {
            for(int x = 0; x != m_ProcessingRegion.Width(); x++)
            {
         int imageXcoord = x + m_ProcessingRegion.TopLeft().x;
         int imageYCoord = y + m_ProcessingRegion.TopLeft().y;
         int imageIndex = (imageYCoord * dibWidth) + imageXcoord;
         int bufferIndex = ((m_ProcessingRegion.Height() - 1 - y) *
m_ProcessingRegion.Width()) + x;
         pImage[ bufferIndex]  = pImageToBeTiledBuffer[ imageIndex];
            }
      }
```

```
   TiledImage_cls::GetUniqueInstance()->SetUp(T, W, H, pImage,
m_ImagePixelArray);
   TiledImage_cls::GetUniqueInstance()->UnwrapTiles();
   TiledImage_cls::GetUniqueInstance()->AssembleTiles();

      for (int yCounter=0; yCounter != dibHeight; yCounter++)
      {
            for (int xCounter=0; xCounter != dibWidth; xCounter++)
            {
         if (m_ProcessingRegion.PtInRect(CPoint(xCounter,
yCounter)))
         {
            int y = yCounter - m_ProcessingRegion.TopLeft().y;
            int x = xCounter - m_ProcessingRegion.TopLeft().x;
            int inversed_y = m_ProcessingRegion.Height() - 1 - y;
            int imageIndex = (dibWidth * yCounter) + xCounter;
            //int bufferIndex = ((m_ProcessingRegion.Height() - 1 -
y) * m_ProcessingRegion.Height()) + x;

            //pImageToBeTiledBuffer[ imageIndex] =
pImage[ bufferIndex] ;
            pImageToBeTiledBuffer[ imageIndex] =
               TiledImage_cls::GetUniqueInstance()->
                  GetPixel(x,
inversed_y).GetUnwrappedValue();//*/GetAssembledValue();
                  }
            }
   }

      ::GlobalUnlock((HGLOBAL) hTileddDIB);
      ::GlobalUnlock((HGLOBAL) hFilteredImageDIB);
      pDibApiWrapper->SaveDIB(
            hTileddDIB,
            CFile(unwrappedFileName,
CFile::modeWrite|CFile::modeCreate));

   pDibApiWrapper->DestroyClonedDIB(hTileddDIB);

   delete [] pImage;
      return true;
}
```

CDibApiWrapper.h

```
// see dibapi.h in MFC DIBLook sample

#ifndef CDIBAPIWRAPPER_H
#define CDIBAPIWRAPPER_H

/* Handle to a DIB */
DECLARE_HANDLE(HDIB);

/* DIB constants */
#define PALVERSION   0x300

/* DIB Macros*/

#define IS_WIN30_DIB(lpbi)  ((*(LPDWORD)(lpbi)) ==
sizeof(BITMAPINFOHEADER))
#define RECTWIDTH(lpRect)     ((lpRect)->right - (lpRect)->left)
#define RECTHEIGHT(lpRect)    ((lpRect)->bottom - (lpRect)->top)

// WIDTHBYTES performs DWORD-aligning of DIB scanlines.  The "bits"
// parameter is the bit count for the scanline (biWidth *
biBitCount),
// and this macro returns the number of DWORD-aligned bytes needed
// to hold those bits.

#define WIDTHBYTES(bits)    (((bits) + 31) / 32 * 4)


class CDibApiWrapper {
//oparations
public:

      static CDibApiWrapper* GetUniqueInstance();
      BOOL PaintDIB (HDC, LPRECT, HDIB, LPRECT, CPalette* pPal);
      BOOL CreateDIBPalette(HDIB hDIB, CPalette* cPal);
      LPSTR FindDIBBits (LPSTR lpbi);
      DWORD DIBWidth (LPSTR lpDIB);
      DWORD DIBHeight (LPSTR lpDIB);
      WORD PaletteSize (LPSTR lpbi);
      WORD DIBNumColors (LPSTR lpbi);
      HGLOBAL CopyHandle (HGLOBAL h);

      BOOL SaveDIB (HDIB hDib, CFile& file);
      HDIB ReadDIBFile(CFile& file);

      HDIB CloneDIB(HDIB hDibToBeCloned);
   void DestroyClonedDIB(HDIB hDibToBeDestroyed);
//attributes
private:
      static CDibApiWrapper* pUniqueInstance;
};

#endif //CDIBAPIWRAPPER_H
```

CDibApiWrapper.cpp

```
#include "stdafx.h"
#include "CDibApiWrapper.h"

//  see myfile.cpp in MFC DIBLook sample
//  SaveDIB()            - Saves the specified dib in a file
//  ReadDIBFile()        - Loads a DIB from a file
//
#include <math.h>
#include <io.h>

/*
 * Dib Header Marker - used in writing DIBs to files
 */
#define DIB_HEADER_MARKER   ((WORD) ('M' << 8) | 'B')



//  see dibapi.cpp in MFC DIBLook sample
//
//  Source file for Device-Independent Bitmap (DIB) API.  Provides
//  the following functions:
//
//  PaintDIB()           - Painting routine for a DIB
//  CreateDIBPalette()   - Creates a palette from a DIB
//  FindDIBBits()        - Returns a pointer to the DIB bits
//  DIBWidth()           - Gets the width of the DIB
//  DIBHeight()          - Gets the height of the DIB
//  PaletteSize()        - Gets the size required to store the DIB's
palette
//  DIBNumColors()       - Calculates the number of colors
//                         in the DIB's color table
//  CopyHandle()         - Makes a copy of the given global memory
block
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1992-1998 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and related
// electronic documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

#include "stdafx.h"
#include <io.h>
#include <errno.h>

//initialise static attribute (Singlton Pattern Implelmentation)
CDibApiWrapper* CDibApiWrapper::pUniqueInstance = NULL;

//return unique instance (Singlton Pattern Implelmentation)
CDibApiWrapper* CDibApiWrapper::GetUniqueInstance()
{
      if (pUniqueInstance == NULL)
   {
            pUniqueInstance = new CDibApiWrapper;
   }
      return pUniqueInstance;
}
```

```
/*************************************************************************
******
 *
 * PaintDIB()
 *
 * Parameters:
 *
 * HDC hDC          - DC to do output to
 *
 * LPRECT lpDCRect  - rectangle on DC to do output to
 *
 * HDIB hDIB        - handle to global memory with a DIB spec
 *                    in it followed by the DIB bits
 *
 * LPRECT lpDIBRect - rectangle of DIB to output into lpDCRect
 *
 * CPalette* pPal   - pointer to CPalette containing DIB's palette
 *
 * Return Value:
 *
 * BOOL             - TRUE if DIB was drawn, FALSE otherwise
 *
 * Description:
 *   Painting routine for a DIB.  Calls StretchDIBits() or
 *   SetDIBitsToDevice() to paint the DIB.  The DIB is
 *   output to the specified DC, at the coordinates given
 *   in lpDCRect.  The area of the DIB to be output is
 *   given by lpDIBRect.
 *

*************************************************************************
****/

BOOL CDibApiWrapper::PaintDIB(HDC     hDC,
                              LPRECT  lpDCRect,
                              HDIB    hDIB,
                              LPRECT  lpDIBRect,
                              CPalette* pPal)
{
      LPSTR    lpDIBHdr;            // Pointer to BITMAPINFOHEADER
      LPSTR    lpDIBBits;           // Pointer to DIB bits
      BOOL     bSuccess=FALSE;      // Success/fail flag
      HPALETTE hPal=NULL;           // Our DIB's palette
      HPALETTE hOldPal=NULL;        // Previous palette

      /* Check for valid DIB handle */
      if (hDIB == NULL)
         return FALSE;

      /* Lock down the DIB, and get a pointer to the beginning of
the bit
       *  buffer
       */
lpDIBHdr  = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);
lpDIBBits = FindDIBBits(lpDIBHdr);

// Get the DIB's palette, then select it into DC
if (pPal != NULL)
{
      hPal = (HPALETTE) pPal->m_hObject;

      // Select as background since we have
      // already realized in forground if needed
      hOldPal = ::SelectPalette(hDC, hPal, TRUE);
}
```

```
      /* Make sure to use the stretching mode best for color
pictures */
      ::SetStretchBltMode(hDC, COLORONCOLOR);

      /* Determine whether to call StretchDIBits() or
SetDIBitsToDevice() */
      if ((RECTWIDTH(lpDCRect)  == RECTWIDTH(lpDIBRect)) &&
         (RECTHEIGHT(lpDCRect) == RECTHEIGHT(lpDIBRect)))
            bSuccess = ::SetDIBitsToDevice(hDC,
// hDC
                                                lpDCRect->left,
// DestX
                                                lpDCRect->top,
// DestY
RECTWIDTH(lpDCRect),        // nDestWidth
RECTHEIGHT(lpDCRect),       // nDestHeight
                                                lpDIBRect->left,
// SrcX
(int)DIBHeight(lpDIBHdr) -
                                                     lpDIBRect-
>top -
RECTHEIGHT(lpDIBRect),   // SrcY
                                                0,
// nStartScan
(WORD)DIBHeight(lpDIBHdr),  // nNumScans
                                                lpDIBBits,
// lpBits
(LPBITMAPINFO)lpDIBHdr,     // lpBitsInfo
                                                     DIB_RGB_COLORS);
// wUsage
   else
        bSuccess = ::StretchDIBits(hDC,                          //
hDC
                                   lpDCRect->left,
// DestX
                                   lpDCRect->top,
// DestY
                                   RECTWIDTH(lpDCRect),
// nDestWidth
                                   RECTHEIGHT(lpDCRect),
// nDestHeight
                                   lpDIBRect->left,
// SrcX
                                   lpDIBRect->top,
// SrcY
                                   RECTWIDTH(lpDIBRect),
// wSrcWidth
                                   RECTHEIGHT(lpDIBRect),
// wSrcHeight
                                   lpDIBBits,
// lpBits
                                   (LPBITMAPINFO)lpDIBHdr,
// lpBitsInfo
                                   DIB_RGB_COLORS,
// wUsage
                                   SRCCOPY);
// dwROP
```

```
    ::GlobalUnlock((HGLOBAL) hDIB);

       /* Reselect old palette */
       if (hOldPal != NULL)
       {
             ::SelectPalette(hDC, hOldPal, TRUE);
       }

    return bSuccess;
}

/************************************************************************
******
 *
 * CreateDIBPalette()
 *
 * Parameter:
 *
 * HDIB hDIB        - specifies the DIB
 *
 * Return Value:
 *
 * HPALETTE         - specifies the palette
 *
 * Description:
 *
 * This function creates a palette from a DIB by allocating memory
for the
 * logical palette, reading and storing the colors from the DIB's
color table
 * into the logical palette, creating a palette from this logical
palette,
 * and then returning the palette's handle. This allows the DIB to
be
 * displayed using the best possible colors (important for DIBs with
256 or
 * more colors).
 *
************************************************************************
****/


BOOL CDibApiWrapper::CreateDIBPalette(HDIB hDIB, CPalette* pPal)
{
      LPLOGPALETTE lpPal;      // pointer to a logical palette
      HANDLE hLogPal;          // handle to a logical palette
      HPALETTE hPal = NULL;    // handle to a palette
      int i;                   // loop index
      WORD wNumColors;         // number of colors in color table
      LPSTR lpbi;              // pointer to packed-DIB
      LPBITMAPINFO lpbmi;      // pointer to BITMAPINFO structure
(Win3.0)
      LPBITMAPCOREINFO lpbmc;  // pointer to BITMAPCOREINFO
structure (old)
      BOOL bWinStyleDIB;       // flag which signifies whether this
is a Win3.0 DIB
      BOOL bResult = FALSE;

      /* if handle to DIB is invalid, return FALSE */

      if (hDIB == NULL)
        return FALSE;

   lpbi = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);
```

```
/* get pointer to BITMAPINFO (Win 3.0) */
lpbmi = (LPBITMAPINFO)lpbi;

/* get pointer to BITMAPCOREINFO (old 1.x) */
lpbmc = (LPBITMAPCOREINFO)lpbi;

/* get the number of colors in the DIB */
wNumColors = DIBNumColors(lpbi);

if (wNumColors != 0)
{
        /* allocate memory block for logical palette */
        hLogPal = ::GlobalAlloc(GHND, sizeof(LOGPALETTE)
                                                +
sizeof(PALETTEENTRY)
                                                * wNumColors);

        /* if not enough memory, clean up and return NULL */
        if (hLogPal == 0)
        {
            ::GlobalUnlock((HGLOBAL) hDIB);
            return FALSE;
        }

        lpPal = (LPLOGPALETTE) ::GlobalLock((HGLOBAL) hLogPal);

        /* set version and number of palette entries */
        lpPal->palVersion = PALVERSION;
        lpPal->palNumEntries = (WORD)wNumColors;

        /* is this a Win 3.0 DIB? */
        bWinStyleDIB = IS_WIN30_DIB(lpbi);
        for (i = 0; i < (int)wNumColors; i++)
        {
            if (bWinStyleDIB)
            {
                lpPal->palPalEntry[ i] .peRed = lpbmi-
>bmiColors[ i] .rgbRed;
                lpPal->palPalEntry[ i] .peGreen = lpbmi-
>bmiColors[ i] .rgbGreen;
                lpPal->palPalEntry[ i] .peBlue = lpbmi-
>bmiColors[ i] .rgbBlue;
                lpPal->palPalEntry[ i] .peFlags = 0;
            }
            else
            {
                lpPal->palPalEntry[ i] .peRed = lpbmc-
>bmciColors[ i] .rgbtRed;
                lpPal->palPalEntry[ i] .peGreen = lpbmc-
>bmciColors[ i] .rgbtGreen;
                lpPal->palPalEntry[ i] .peBlue = lpbmc-
>bmciColors[ i] .rgbtBlue;
                lpPal->palPalEntry[ i] .peFlags = 0;
            }
        }

        /* create the palette and get handle to it */
        bResult = pPal->CreatePalette(lpPal);
        ::GlobalUnlock((HGLOBAL) hLogPal);
        ::GlobalFree((HGLOBAL) hLogPal);
}

::GlobalUnlock((HGLOBAL) hDIB);
```

```
        return bResult;
}

/************************************************************************
******
 *
 * FindDIBBits()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * LPSTR            - pointer to the DIB bits
 *
 * Description:
 *
 * This function calculates the address of the DIB's bits and
returns a
 * pointer to the DIB bits.
 *

************************************************************************
****/


LPSTR CDibApiWrapper::FindDIBBits(LPSTR lpbi)
{
        return (lpbi + *(LPDWORD)lpbi + PaletteSize(lpbi));
}

/************************************************************************
******
 *
 * DIBWidth()
 *
 * Parameter:
 *
 * LPSTR lpbi       - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * DWORD            - width of the DIB
 *
 * Description:
 *
 * This function gets the width of the DIB from the BITMAPINFOHEADER
 * width field if it is a Windows 3.0-style DIB or from the
BITMAPCOREHEADER
 * width field if it is an other-style DIB.
 *

************************************************************************
****/


DWORD CDibApiWrapper::DIBWidth(LPSTR lpDIB)
{
        LPBITMAPINFOHEADER lpbmi;  // pointer to a Win 3.0-style DIB
        LPBITMAPCOREHEADER lpbmc;  // pointer to an other-style DIB

        /* point to the header (whether Win 3.0 and old) */
```

```
        lpbmi = (LPBITMAPINFOHEADER)lpDIB;
        lpbmc = (LPBITMAPCOREHEADER)lpDIB;

        /* return the DIB width if it is a Win 3.0 DIB */
        if (IS_WIN30_DIB(lpDIB))
              return lpbmi->biWidth;
        else  /* it is an other-style DIB, so return its width */
              return (DWORD)lpbmc->bcWidth;
}


/************************************************************************
******
 *
 * DIBHeight()
 *
 * Parameter:
 *
 * LPSTR lpbi         - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * DWORD              - height of the DIB
 *
 * Description:
 *
 * This function gets the height of the DIB from the
BITMAPINFOHEADER
 * height field if it is a Windows 3.0-style DIB or from the
BITMAPCOREHEADER
 * height field if it is an other-style DIB.
 *
************************************************************************
****/


DWORD CDibApiWrapper::DIBHeight(LPSTR lpDIB)
{
        LPBITMAPINFOHEADER lpbmi;  // pointer to a Win 3.0-style DIB
        LPBITMAPCOREHEADER lpbmc;  // pointer to an other-style DIB

        /* point to the header (whether old or Win 3.0 */

        lpbmi = (LPBITMAPINFOHEADER)lpDIB;
        lpbmc = (LPBITMAPCOREHEADER)lpDIB;

        /* return the DIB height if it is a Win 3.0 DIB */
        if (IS_WIN30_DIB(lpDIB))
              return lpbmi->biHeight;
        else  /* it is an other-style DIB, so return its height */
              return (DWORD)lpbmc->bcHeight;
}


/************************************************************************
******
 *
 * PaletteSize()
 *
 * Parameter:
 *
 * LPSTR lpbi         - pointer to packed-DIB memory block
 *
 * Return Value:
```

```
 *
 * WORD                - size of the color palette of the DIB
 *
 * Description:
 *
 * This function gets the size required to store the DIB's palette
by
 * multiplying the number of colors by the size of an RGBQUAD (for a
 * Windows 3.0-style DIB) or by the size of an RGBTRIPLE (for an
other-
 * style DIB).
 *
************************************************************************
****/


WORD CDibApiWrapper::PaletteSize(LPSTR lpbi)
{
    /* calculate the size required by the palette */
    if (IS_WIN30_DIB (lpbi))
         return (WORD)(DIBNumColors(lpbi) * sizeof(RGBQUAD));
    else
         return (WORD)(DIBNumColors(lpbi) * sizeof(RGBTRIPLE));
}


/************************************************************************
******
 *
 * DIBNumColors()
 *
 * Parameter:
 *
 * LPSTR lpbi          - pointer to packed-DIB memory block
 *
 * Return Value:
 *
 * WORD                - number of colors in the color table
 *
 * Description:
 *
 * This function calculates the number of colors in the DIB's color
table
 * by finding the bits per pixel for the DIB (whether Win3.0 or
other-style
 * DIB). If bits per pixel is 1: colors=2, if 4: colors=16, if 8:
colors=256,
 * if 24, no colors in color table.
 *

************************************************************************
****/

WORD CDibApiWrapper::DIBNumColors(LPSTR lpbi)
{
       WORD wBitCount;  // DIB bit count

       /*   If this is a Windows-style DIB, the number of colors in
the
        *   color table can be less than the number of bits per pixel
        *   allows for (i.e. lpbi->biClrUsed can be set to some
value).
        *   If this is the case, return the appropriate value.
```

```
 */

if (IS_WIN30_DIB(lpbi))
{
        DWORD dwClrUsed;

        dwClrUsed = ((LPBITMAPINFOHEADER)lpbi)->biClrUsed;
        if (dwClrUsed != 0)
                return (WORD)dwClrUsed;
}

/*  Calculate the number of colors in the color table based on
 *  the number of bits per pixel for the DIB.
 */
if (IS_WIN30_DIB(lpbi))
        wBitCount = ((LPBITMAPINFOHEADER)lpbi)->biBitCount;
else
        wBitCount = ((LPBITMAPCOREHEADER)lpbi)->bcBitCount;

/* return number of colors based on bits per pixel */
switch (wBitCount)
{
        case 1:
                return 2;

        case 4:
                return 16;

        case 8:
                return 256;

        default:
                return 0;
        }
}

////////////////////////////////////////////////////////////////////////
//////
//// Clipboard support

//--------------------------------------------------------------------
---
//
// Function:   CopyHandle (from SDK DibView sample clipbrd.c)
//
// Purpose:    Makes a copy of the given global memory block.
Returns
//             a handle to the new memory block (NULL on error).
//
//             Routine stolen verbatim out of ShowDIB.
//
// Parms:      h == Handle to global memory to duplicate.
//
// Returns:    Handle to new global memory block.
//
//--------------------------------------------------------------------
---

HGLOBAL CDibApiWrapper::CopyHandle (HGLOBAL h)
{
        if (h == NULL)
                return NULL;

        DWORD dwLen = ::GlobalSize((HGLOBAL) h);
```

```
        HGLOBAL hCopy = ::GlobalAlloc(GHND, dwLen);

        if (hCopy != NULL)
        {
              void* lpCopy = ::GlobalLock((HGLOBAL) hCopy);
              void* lp     = ::GlobalLock((HGLOBAL) h);
              memcpy(lpCopy, lp, dwLen);
              ::GlobalUnlock(hCopy);
              ::GlobalUnlock(h);
        }

        return hCopy;
}

/*************************************************************************
******
 *
 * SaveDIB()
 *
 * Saves the specified DIB into the specified CFile.  The CFile
 * is opened and closed by the caller.
 *
 * Parameters:
 *
 * HDIB hDib - Handle to the dib to save
 *
 * CFile& file - open CFile used to save DIB
 *
 * Return value: TRUE if successful, else FALSE or CFileException
 *

*************************************************************************
*****/


BOOL CDibApiWrapper::SaveDIB(HDIB hDib, CFile& file)
{
        BITMAPFILEHEADER bmfHdr; // Header for Bitmap file
        LPBITMAPINFOHEADER lpBI;   // Pointer to DIB info structure
        DWORD dwDIBSize;

        if (hDib == NULL)
              return FALSE;

        /*
         * Get a pointer to the DIB memory, the first of which
contains
         * a BITMAPINFO structure
         */
        lpBI = (LPBITMAPINFOHEADER) ::GlobalLock((HGLOBAL) hDib);
        if (lpBI == NULL)
              return FALSE;

        if (!IS_WIN30_DIB(lpBI))
        {
              ::GlobalUnlock((HGLOBAL) hDib);
              return FALSE;       // It's an other-style DIB (save not
supported)
        }

        /*
         * Fill in the fields of the file header
         */
```

```
      /* Fill in file type (first 2 bytes must be "BM" for a bitmap)
*/
      bmfHdr.bfType = DIB_HEADER_MARKER;  // "BM"

      // Calculating the size of the DIB is a bit tricky (if we want
to
      // do it right).  The easiest way to do this is to call
GlobalSize()
      // on our global handle, but since the size of our global
memory may have
      // been padded a few bytes, we may end up writing out a few
too
      // many bytes to the file (which may cause problems with some
apps).
      //
      // So, instead let's calculate the size manually (if we can)
      //
      // First, find size of header plus size of color table.  Since
the
      // first DWORD in both BITMAPINFOHEADER and BITMAPCOREHEADER
conains
      // the size of the structure, let's use this.

      dwDIBSize = *(LPDWORD)lpBI + PaletteSize((LPSTR)lpBI);  //
Partial Calculation

      // Now calculate the size of the image

      if ((lpBI->biCompression == BI_RLE8) || (lpBI->biCompression
== BI_RLE4))
      {
            // It's an RLE bitmap, we can't calculate size, so trust
the
            // biSizeImage field

            dwDIBSize += lpBI->biSizeImage;
      }
      else
      {
            DWORD dwBmBitsSize;  // Size of Bitmap Bits only

            // It's not RLE, so size is Width (DWORD aligned) *
Height

            dwBmBitsSize = WIDTHBYTES((lpBI->biWidth)*((DWORD)lpBI-
>biBitCount)) * lpBI->biHeight;

            dwDIBSize += dwBmBitsSize;

            // Now, since we have calculated the correct size, why
don't we
            // fill in the biSizeImage field (this will fix any .BMP
files which
            // have this field incorrect).

            lpBI->biSizeImage = dwBmBitsSize;
      }


      // Calculate the file size by adding the DIB size to
sizeof(BITMAPFILEHEADER)

      bmfHdr.bfSize = dwDIBSize + sizeof(BITMAPFILEHEADER);
      bmfHdr.bfReserved1 = 0;
      bmfHdr.bfReserved2 = 0;
```

```
        /*
         * Now, calculate the offset the actual bitmap bits will be in
         * the file -- It's the Bitmap file header plus the DIB
header,
         * plus the size of the color table.
         */
        bmfHdr.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER) + lpBI-
>biSize

+ PaletteSize((LPSTR)lpBI);
        TRY
        {
                // Write the file header
                file.Write((LPSTR)&bmfHdr, sizeof(BITMAPFILEHEADER));
                //
                // Write the DIB header and the bits
                //
                file.Write(lpBI, dwDIBSize);
        }
        CATCH (CFileException, e)
        {
                ::GlobalUnlock((HGLOBAL) hDib);
                THROW_LAST();
        }
        END_CATCH

        ::GlobalUnlock((HGLOBAL) hDib);
        return TRUE;
}

/*************************************************************************
******

  Function:  ReadDIBFile (CFile&)

   Purpose:  Reads in the specified DIB file into a global chunk of
             memory.

   Returns:  A handle to a dib (hDIB) if successful.
             NULL if an error occurs.

  Comments:  BITMAPFILEHEADER is stripped off of the DIB.
Everything
             from the end of the BITMAPFILEHEADER structure on
is
             returned in the global memory handle.

*************************************************************************
*****/

HDIB CDibApiWrapper::ReadDIBFile(CFile& file)
{
        BITMAPFILEHEADER bmfHeader;
        DWORD dwBitsSize;
        HDIB hDIB;
        LPSTR pDIB;

        /*
         * get length of DIB in bytes for use when reading
         */

        dwBitsSize = (DWORD)file.GetLength();
```

```
        /*
         * Go read the DIB file header and check if it's valid.
         */
        if (file.Read((LPSTR)&bmfHeader, sizeof(bmfHeader)) !=
sizeof(bmfHeader))
              return NULL;

        if (bmfHeader.bfType != DIB_HEADER_MARKER)
              return NULL;

        /*
         * Allocate memory for DIB
         */
        hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
dwBitsSize);
        if (hDIB == 0)
        {
              return NULL;
        }
        pDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

        /*
         * Go read the bits.
         */
        if (file.Read(pDIB, dwBitsSize - sizeof(BITMAPFILEHEADER)) !=
              dwBitsSize - sizeof(BITMAPFILEHEADER) )
        {
              ::GlobalUnlock((HGLOBAL) hDIB);
              ::GlobalFree((HGLOBAL) hDIB);
              return NULL;
        }
        ::GlobalUnlock((HGLOBAL) hDIB);
        return hDIB;
}

HDIB CDibApiWrapper::CloneDIB(HDIB hDibToBeCloned)
{
        LPSTR pDibToBeClonedHdr  = (LPSTR) ::GlobalLock((HGLOBAL)
hDibToBeCloned);
        HDIB hDIBCloned = NULL;

        unsigned long infoHeaderSize = sizeof(BITMAPINFOHEADER);
        unsigned long paletteSize = PaletteSize(pDibToBeClonedHdr);
        unsigned long iamgeSize =
DIBHeight(pDibToBeClonedHdr)*DIBWidth(pDibToBeClonedHdr);
        unsigned long totalSize = infoHeaderSize + paletteSize +
iamgeSize;

        ::GlobalUnlock((HGLOBAL) hDibToBeCloned);

        hDIBCloned = (HDIB) new BYTE[ totalSize];

        BYTE* pbDIB = (BYTE*) hDIBCloned;
        BYTE* pbDIB_original = (BYTE*) ::GlobalLock((HGLOBAL)
hDibToBeCloned);

        for (unsigned long i=0; i<totalSize; i++)
        {
              pbDIB[ i] = pbDIB_original[ i];
        }

        ::GlobalUnlock((HGLOBAL) hDibToBeCloned);
```

```
        return hDIBCloned;
}

void CDibApiWrapper::DestroyClonedDIB(HDIB hDibToBeDestroyed)
{
        LPSTR pDibToBeDestroyedHdr = (LPSTR) ::GlobalLock((HGLOBAL)
hDibToBeDestroyed);

    delete [] ((BYTE*)pDibToBeDestroyedHdr);

        ::GlobalUnlock((HGLOBAL) hDibToBeDestroyed);
}
```

CFrinWizBitmap.h

```
#ifndef CFRINWIZBITMAP_H
#define CFRINWIZBITMAP_H

#include <afxtempl.h>
#include "CDibApiWrapper.h"
#include "CFrinWizBitmapOrganiser.h"

class CFrinWizView;
class CFrinWizDoc;

class CFrinWizBitmap {

//operations
public:
	static CFrinWizBitmap* GetUniqueInstance();
	CFrinWizBitmap();
	~CFrinWizBitmap();
	void AttatchView(CFrinWizDoc* pDoc, CView* pSubscriberView);
	void New(CFrinWizDoc* pNewDoc);
	void Close(CFrinWizDoc* pDoc);

// Attributes
public:
	HDIB GetHDIB(CView* pSubscriberView);
	CPalette* GetDocPalette(CFrinWizDoc*) const;
	CSize GetDocSize(CFrinWizDoc* pDoc);

private:
	/**@shapeType AggregationLink
	@supplierCardinality 1 */
	CFrinWizBitmapOrganiser * pUniqueOrganiser;
	void InitDIBData(CFrinWizDoc* pDoc);

private:
	static CFrinWizBitmap* pUniqueInstance;

};

#endif //CFRINWIZBITMAP_H
```

CFrinWizBitmap.cpp

```
#include "stdafx.h"

#include "CFrinWizBitmap.h"

#define IDR_DIBTYPE                     3
#define IDS_DIB_TOO_BIG                 4
#define IDS_CANNOT_LOAD_DIB             5
#define IDS_CANNOT_SAVE_DIB             6

 //initialise static attribute (Singlton Pattern Implelmentation)
 CFrinWizBitmap* CFrinWizBitmap::pUniqueInstance = NULL;

 //return unique instance (Singlton Pattern Implelmentation)
 CFrinWizBitmap* CFrinWizBitmap::GetUniqueInstance()
 {
       if (pUniqueInstance == NULL)
             pUniqueInstance = new CFrinWizBitmap;
       return pUniqueInstance;
 }

 CFrinWizBitmap::CFrinWizBitmap()
 {
       pUniqueOrganiser = new CFrinWizBitmapOrganiser;
 }

 CFrinWizBitmap::~CFrinWizBitmap()
 {
       delete pUniqueOrganiser;
 }

 void CFrinWizBitmap::InitDIBData(CFrinWizDoc* pDoc)
 {
       CDibApiWrapper* pDibApi = CDibApiWrapper::GetUniqueInstance();
       ASSERT(pDibApi != NULL);

       HDIB temp_hDIB = pUniqueOrganiser->GetHDIB(pDoc);
       CPalette* temp_palette = pUniqueOrganiser-
>GetDocumentPalette(pDoc);

       if (temp_palette != NULL)
       {
             delete temp_palette;
             pUniqueOrganiser->SetDocumentPalette(pDoc, NULL);
       }
       if (temp_hDIB == NULL)
       {
             return;
       }
       // Set up document size
       LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) temp_hDIB);
       if (pDibApi->DIBWidth(lpDIB) > INT_MAX ||pDibApi-
>DIBHeight(lpDIB) > INT_MAX)
       {
             ::GlobalUnlock((HGLOBAL) temp_hDIB);
             ::GlobalFree((HGLOBAL) temp_hDIB);
             temp_hDIB = NULL;
             CString strMsg;
             strMsg.LoadString(IDS_DIB_TOO_BIG);
             MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION |
MB_OK);
```

```
            return;
        }
        pUniqueOrganiser->SetDocumentSize(pDoc, CSize((int) pDibApi-
>DIBWidth(lpDIB), (int) pDibApi->DIBHeight(lpDIB)));
        ::GlobalUnlock((HGLOBAL) temp_hDIB);
        // Create copy of palette
        temp_palette = new CPalette;
        pUniqueOrganiser->SetDocumentPalette(pDoc, temp_palette);
        if (temp_palette == NULL)
        {
            // we must be really low on memory
            ::GlobalFree((HGLOBAL) temp_hDIB);
            temp_hDIB = NULL;
            return;
        }
        if (pDibApi->CreateDIBPalette(temp_hDIB, temp_palette) ==
NULL)
        {
            // DIB may not have a palette
            delete temp_palette;
            pUniqueOrganiser->SetDocumentPalette(pDoc, NULL);
            return;
        }
}

void CFrinWizBitmap::AttatchView(CFrinWizDoc* pDoc, CView*
pSubscriberView)
{
        pUniqueOrganiser->AttatchView(pDoc, pSubscriberView);
}

void CFrinWizBitmap::New(CFrinWizDoc* pNewDoc)
{
        pUniqueOrganiser->AddEntry(pNewDoc, NULL);
}

HDIB CFrinWizBitmap::GetHDIB(CView* pSubscriberView)
{
        return pUniqueOrganiser->GetHDIB(pSubscriberView);
}

CSize CFrinWizBitmap::GetDocSize(CFrinWizDoc* pDoc)
{
        return (pUniqueOrganiser->GetDocumentSize(pDoc));
}
CPalette* CFrinWizBitmap::GetDocPalette(CFrinWizDoc* pDoc) const
{
        return (pUniqueOrganiser->GetDocumentPalette(pDoc));
}
void CFrinWizBitmap::Close(CFrinWizDoc* pDoc)
{
        pUniqueOrganiser->CloseDocument(pDoc);
}
```

unwrapping_import.h

```
#ifndef UNWRAPPING_IMPORT
#define UNWRAPPING_IMPORT

#define DLLIMPORT __declspec(dllimport)

class DLLIMPORT Unwrapping
{
public:
      Unwrapping();
   const char* GetStatus();
   bool SetProcessingRegion(int xTopLeft, int yTopLeft, int
xBottomRight, int yBottomRight);
      bool ConfigurePhaseStepping(int steps);
   bool SetStep(int setpNumber, const char* fileName);
   bool GenerateWrappedMap(const char* wrappedFileName);
   bool SetWrappedMap(const char* wrappedFileName);
   bool FilterWrappedMap(int passes);
   bool SkeletonWrappedMap(const char* skeletonFileName);
   bool SetUnwrappingStartPoint(int x, int y);
   bool GenerateUnWrappedMap(const char* unwrappedFileName);
};

#endif //UNWRAPPING_IMPORT
```

Test_Unwrapping.h

```
// Test_Unwrapping.h : main header file for the PROJECT_NAME
application
//

#pragma once

#ifndef   AFXWIN_H
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"          // main symbols


// CTest_UnwrappingApp:
// See Test_Unwrapping.cpp for the implementation of this class
//

class CTest_UnwrappingApp : public CWinApp
{
public:
      CTest_UnwrappingApp();

// Overrides
      public:
      virtual BOOL InitInstance();

// Implementation

      DECLARE_MESSAGE_MAP()
};

extern CTest_UnwrappingApp theApp;
```

Test_Unwrapping.cpp

```
// Test_Unwrapping.cpp : Defines the class behaviors for the
application.
//

#include "stdafx.h"
#include "Test_Unwrapping.h"
#include "Test_UnwrappingDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CTest_UnwrappingApp

BEGIN_MESSAGE_MAP(CTest_UnwrappingApp, CWinApp)
      ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()


// CTest_UnwrappingApp construction

CTest_UnwrappingApp::CTest_UnwrappingApp()
{
      // TODO: add construction code here,
      // Place all significant initialization in InitInstance
}

// The one and only CTest_UnwrappingApp object

CTest_UnwrappingApp theApp;


// CTest_UnwrappingApp initialization

BOOL CTest_UnwrappingApp::InitInstance()
{
      // InitCommonControls() is required on Windows XP if an
application
      // manifest specifies use of ComCtl32.dll version 6 or later
to enable
      // visual styles.  Otherwise, any window creation will fail.
      InitCommonControls();

      CWinApp::InitInstance();

      AfxEnableControlContainer();
      // Standard initialization
      // If you are not using these features and wish to reduce the
size
      // of your final executable, you should remove from the
following
      // the specific initialization routines you do not need
      // Change the registry key under which our settings are stored
      // TODO: You should modify this string to be something
appropriate
      // such as the name of your company or organization
      SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

```
CTest_UnwrappingDlg dlg;
m_pMainWnd = &dlg;
INT_PTR nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
	// TODO: Place code here to handle when the dialog is
	//  dismissed with OK
}
else if (nResponse == IDCANCEL)
{
	// TODO: Place code here to handle when the dialog is
	//  dismissed with Cancel
	}

	// Since the dialog has been closed, return FALSE so that we
exit the
	//  application, rather than start the application's message
pump.
	return FALSE;
}
```

Test_UnwrappingDlg.h

```
// Test_UnwrappingDlg.h : header file
//

#pragma once
#include "afxwin.h"


// CTest_UnwrappingDlg dialog
class CTest_UnwrappingDlg : public CDialog
{
// Construction
public:
	CTest_UnwrappingDlg(CWnd* pParent = NULL);	// standard
constructor

// Dialog Data
	enum { IDD = IDD_TEST_UNWRAPPING_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	//
DDX/DDV support

private:
   void PerformTest();
   void Log(const CString& text);
   void LogSeparator();

// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
private:
   CEdit m_LogEdit;
   CString m_LogEditText;
   int m_LineCount;
public:
   afx_msg void OnBnClickedSave();
   afx_msg void OnBnClickedTest();
};
```

Test_UnwrappingDlg.cpp

```
// Test_UnwrappingDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Test_Unwrapping.h"
#include "Test_UnwrappingDlg.h"
#include "unwrapping_import.h"
#include ".\test_unwrappingdlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	enum { IDD = IDD_ABOUTBOX };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

// Implementation
protected:
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()


// CTest_UnwrappingDlg dialog

CTest_UnwrappingDlg::CTest_UnwrappingDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CTest_UnwrappingDlg::IDD, pParent)
   , m_LogEditText(_T(""))
   , m_LineCount(0)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CTest_UnwrappingDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
```

```
    DDX_Control(pDX, IDC_LOG_EDIT, m_LogEdit);
    DDX_Text(pDX, IDC_LOG_EDIT, m_LogEditText);
}

BEGIN_MESSAGE_MAP(CTest_UnwrappingDlg, CDialog)
        ON_WM_SYSCOMMAND()
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        //}}AFX_MSG_MAP
    ON_BN_CLICKED(IDC_SAVE, OnBnClickedSave)
    ON_BN_CLICKED(IDC_TEST, OnBnClickedTest)
END_MESSAGE_MAP()


// CTest_UnwrappingDlg message handlers

BOOL CTest_UnwrappingDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // Add "About..." menu item to system menu.

        // IDM_ABOUTBOX must be in the system command range.
        ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
        ASSERT(IDM_ABOUTBOX < 0xF000);

        CMenu* pSysMenu = GetSystemMenu(FALSE);
        if (pSysMenu != NULL)
        {
                CString strAboutMenu;
                strAboutMenu.LoadString(IDS_ABOUTBOX);
                if (!strAboutMenu.IsEmpty())
                {
                        pSysMenu->AppendMenu(MF_SEPARATOR);
                        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
                }
        }

        // Set the icon for this dialog.  The framework does this
automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                 // Set big icon
        SetIcon(m_hIcon, FALSE);                // Set small icon

        return TRUE;  // return TRUE  unless you set the focus to a
control
}

void CTest_UnwrappingDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
        if ((nID & 0xFFF0) == IDM_ABOUTBOX)
        {
                CAboutDlg dlgAbout;
                dlgAbout.DoModal();
        }
        else
        {
                CDialog::OnSysCommand(nID, lParam);
        }
}

// If you add a minimize button to your dialog, you will need the
code below
//  to draw the icon.  For MFC applications using the document/view
```

```
model,
//  this is automatically done for you by the framework.

void CTest_UnwrappingDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}

// The system calls this function to obtain the cursor to display
while the user drags
//  the minimized window.
HCURSOR CTest_UnwrappingDlg::OnQueryDragIcon()
{
	return static_cast<HCURSOR>(m_hIcon);
}

void CTest_UnwrappingDlg::Log(const CString& text)
{
   UpdateData(TRUE);
   m_LogEditText += text;
   m_LogEditText += "\r\n";
   UpdateData(FALSE);
   m_LogEdit.LineScroll(++m_LineCount);
}

void CTest_UnwrappingDlg::LogSeparator()
{
   UpdateData(TRUE);
   m_LogEditText += "_____\r\n";
   UpdateData(FALSE);
   m_LogEdit.LineScroll(++m_LineCount);
}

CString g_LineNumberString;
#define TEST(code)                                          \
   BeginWaitCursor();                                       \
   g_LineNumberString.Format("%s Line:%03d: ",
CTime::GetCurrentTime().Format("%H:%M:%S"), __LINE__); \
   Log(g_LineNumberString + #code);                         \
   if (code)                                                \
   {                                                        \
      Log(CTime::GetCurrentTime().Format("%H:%M:%S") + " OK"); \
   }                                                        \
   else                                                     \
```

```
    {                                                    \
        Log("ERROR");                                    \
        Log(unwrappping.GetStatus());                    \
        LogSeparator();                                  \
        EndWaitCursor();                                 \
        return;                                          \
    }                                                    \
    LogSeparator();                                      \
    EndWaitCursor();                                     \

void CTest_UnwrappingDlg::PerformTest()
{
    Unwrapping unwrappping;

    //generate wrapped phase map first then unwrap it

    TEST(unwrappping.SetProcessingRegion(200, 300, 300, 400));
    TEST(unwrappping.ConfigurePhaseStepping(3)); //3-step phase
stepping was used to generate input images
    TEST(unwrappping.SetStep(1,
"c:\\unwrapping\\in_images\\group1\\step1.bmp"));
    TEST(unwrappping.SetStep(2,
"c:\\unwrapping\\in_images\\group1\\step2.bmp"));
    TEST(unwrappping.SetStep(3,
"c:\\unwrapping\\in_images\\group1\\step3.bmp"));

TEST(unwrappping.GenerateWrappedMap("c:\\unwrapping\\out_images\\gro
up1\\wrapped.bmp"));
    TEST(unwrappping.FilterWrappedMap(5));

    // File name specified in GenerateWrappedMap will be used in
supsequent steps unless
    //   overriden by using SetWrappedMap.
    // Also, when using SetWrappedMap, previous steps can be skipped
//TEST(unwrappping.SetWrappedMap("C:\\unwrapping\\out_images\\group1
\\Filtered_04wrapped.bmp"));

TEST(unwrappping.SkeletonWrappedMap("c:\\unwrapping\\out_images\\gro
up1\\skeleton_f4.bmp"));


TEST(unwrappping.GenerateUnWrappedMap("c:\\unwrapping\\out_images\\g
roup1\\unwrapped_f4.bmp"));

    //unwrap a phase map directly
//TEST(unwrappping.SetWrappedMap("c:\\unwrapping\\out_images\\wrappe
d.bmp"));
//TEST(unwrappping.GenerateUnWrappedMap("c:\\unwrapping\\out_images\
\unwrapped2.bmp"));
}

void CTest_UnwrappingDlg::OnBnClickedSave()
{
    CFile logFile("TestLog.txt", CFile::modeWrite|CFile::modeCreate);
    Log(CTime::GetCurrentTime().Format("Saved: %c ") +
logFile.GetFilePath());
    LogSeparator();
    logFile.Write(m_LogEditText, m_LogEditText.GetLength());
}

void CTest_UnwrappingDlg::OnBnClickedTest()
{
      PerformTest();
}
```

BoostPrim.h

```
#ifndef BOOST_PRIM_H
#define BOOST_PRIM_H

#include <boost/config.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/prim_minimum_spanning_tree.hpp>

using namespace boost;

typedef adjacency_list <
          vecS,
          vecS,
          undirectedS,
          property<vertex_distance_t, int>,
          property < edge_weight_t, int > > Graph;

typedef std::pair < int, int > Edge;

class BoostPrim
{
public:
   BoostPrim(int tileEdgeLength);
   ~BoostPrim();
   void FindMst();
   void EnableReduction() {mIsReductionEnabled = true;}

private:
   const int mTileEdgeLength;
   const int mEdgeCount;
   const int mVertexCount;
   int __nogc* mWeights;
   Edge __nogc* mEdges;
   Graph mGraph;
   bool mHasBeenReduced;
   bool mIsReductionEnabled;

   struct VertexEdgeWeights
   {
      int mNorth;
      int mEast;
      int mSouth;
      int mWest;
   };
   std::map<int, VertexEdgeWeights> mVerticesEdgeWeights;

   enum EdgeDirection
   {
      UNKNOWN,
      NORTH,
      EAST,
      SOUTH,
      WEST
   };

   //reduction algorithm phases
   void Reduce();
   void Reduction_Phase1();
   void Step1(int x, int y);
   void Step2(int x, int y);

   void Reduction_Phase2();
```

```
   void Step3(int x, int y);
   void Step4(int x, int y);

   //edge deletion and contraction
   void RuleIn(int x, int y, EdgeDirection direction);
   void RuleOut(int x, int y, EdgeDirection direction);

   //neighbourhood tests
   inline bool DoesHaveNorthNeighbour(const int& x, const int&
y)const { return y > 0;}
   inline bool DoesHaveEastNeighbour(const int& x, const int&
y)const { return x < mTileEdgeLength - 1;}
   inline bool DoesHaveSouthNeighbour(const int& x, const int&
y)const { return y < mTileEdgeLength - 1;}
   inline bool DoesHaveWestNeighbour(const int& x, const int&
y)const { return x > 0;}
   inline bool DoesHaveSuothEastVertex(const int& x, const int&
y)const { return y < mTileEdgeLength - 1 && x < mTileEdgeLength - 1;}

   //obtain a vertex sequential index from an x,y coordinates
   inline int VertexIndex(const int& x, const int& y)const { return
(y * mTileEdgeLength) + x;}
   inline int VertexIndex(const int& x, const int& y, const
EdgeDirection& direction)const;
   inline int SuothEastVertexIndex(const int& x, const int& y)const
{ return ((y + 1) * mTileEdgeLength) + x + 1;}
   inline void VertexCoordinate(int& x, int& y, const EdgeDirection&
direction)const;
};

template <class Directed, class Vertex,
   class OutEdgeListS,
   class VertexListS,
   class DirectedS,
   class VertexProperty,
   class EdgeProperty,
   class GraphProperty, class EdgeListS>
inline Vertex&
get_target(detail::edge_base<Directed,Vertex>& e,
      const adjacency_list<OutEdgeListS, VertexListS, DirectedS,
                  VertexProperty, EdgeProperty, GraphProperty,
EdgeListS>&)
{
   return e.m_target;
}

template <class Directed, class Vertex,
   class OutEdgeListS,
   class VertexListS,
   class DirectedS,
   class VertexProperty,
   class EdgeProperty,
   class GraphProperty, class EdgeListS>
inline Vertex&
get_source(detail::edge_base<Directed,Vertex>& e,
      const adjacency_list<OutEdgeListS, VertexListS, DirectedS,
                  VertexProperty, EdgeProperty, GraphProperty,
EdgeListS>&)
{
   return e.m_source;
}

#endif //BOOST_PRIM_H
```

BoostPrim.cpp

```
#include "StdAfx.h"
#include "BoostPrim.h"
#include <iostream>
#include <time.h>

using namespace std;
using namespace boost;
using namespace System;

const int g_MaxWeight = 255;

#if _DEBUG
const int g_AveragingLoopCounters[] = {1, 1, 1, 1};
#else
const int g_AveragingLoopCounters[] = {1, 1, 1, 1}; //use with
reduction
//const int g_AveragingLoopCounters[] = {1000, 100, 10, 1}; //use
when testing without reduction
#endif //_DEBUG

enum AvergingThreshold {LOW_THRESHOLD_AVG, MEDIUM_THRESHOLD_AVG,
HIGH_THRESHOLD_AVG, NO_AVERAGING};

BoostPrim::BoostPrim(int tileEdgeLength):
mTileEdgeLength(tileEdgeLength),
mVertexCount(mTileEdgeLength * mTileEdgeLength),
mEdgeCount(2 * (mTileEdgeLength * (mTileEdgeLength - 1))),
mWeights(new int[mEdgeCount]),
mEdges(new Edge[mEdgeCount]),
mHasBeenReduced(false),
mIsReductionEnabled(false)
{
    //generate random weights
    Random* rng = new Random();
    for (int counter = 0; counter != mEdgeCount; counter++)
    {
        mWeights[counter] = rng->Next(g_MaxWeight);
    }

    //construct grid graph
    int currentVertexIndex = 0;
    int currentEdgeIndex = 0;
    for (int y = 0; y != mTileEdgeLength; y++)
    {
        for (int x = 0; x != mTileEdgeLength; x++)
        {
            int yOffset = y * mTileEdgeLength;
            currentVertexIndex = yOffset + x;
            VertexEdgeWeights edgeWeights;
            if (x < mTileEdgeLength - 1)
            {
                int eastVertexIndex = currentVertexIndex + 1;
                mEdges[currentEdgeIndex++] = Edge(currentVertexIndex,
eastVertexIndex);
                edgeWeights.mEast = mWeights[currentEdgeIndex];
                if (x > 0)
                {
                    mVerticesEdgeWeights[eastVertexIndex].mWest =
mWeights[currentEdgeIndex];
                }
            }
```

```
            if (y < mTileEdgeLength - 1)
            {
                int southVertexIndex = yOffset + mTileEdgeLength + x;
                mEdges[ currentEdgeIndex++] = Edge(currentVertexIndex,
southVertexIndex);
                edgeWeights.mSouth = mWeights[ currentEdgeIndex] ;
                if (y > 0)
                {
                    mVerticesEdgeWeights[ southVertexIndex] .mNorth =
mWeights[ currentEdgeIndex] ;
                }
            }
        }

        //sanity checks
        assert(currentEdgeIndex == mEdgeCount);
        assert(currentVertexIndex == mVertexCount - 1);

        mGraph = Graph(mEdges, mEdges + mEdgeCount, mWeights,
mVertexCount);
}

BoostPrim::~BoostPrim()
{
    delete [] mEdges;
    delete [] mWeights;
}

void BoostPrim::FindMst()
{
    //test code
    //E edges[] = {
    //   Edge(0, 3), Edge(0, 1), Edge(1, 4), Edge(1, 2), Edge(2, 5),
    //   Edge(3, 6), Edge(3; 4), Edge(4, 2), Edge(4, 5), Edge(5, 8),
    //   Edge(6, 7), Edge(7, 8)};
    //int weights[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    //Graph g(edges, edges + sizeof(edges) / sizeof(Edge), weights,
9);

    //Graph g(mEdges, mEdges + mEdgeCount, mWeights, mVertexCount);
    std::vector < graph_traits < Graph >::vertex_descriptor >
p(num_vertices(mGraph));
    property_map<Graph, edge_weight_t>::type weightmap =
get(edge_weight, mGraph);

    int avgLoopCount = g_AveragingLoopCounters[ NO_AVERAGING] ;
    if (mTileEdgeLength < 64)
    {
        avgLoopCount = g_AveragingLoopCounters[ LOW_THRESHOLD_AVG] ;
    }
    else if (mTileEdgeLength < 128)
    {
        avgLoopCount = g_AveragingLoopCounters[ MEDIUM_THRESHOLD_AVG] ;
    }
    else if (mTileEdgeLength < 256)
    {
        avgLoopCount = g_AveragingLoopCounters[ HIGH_THRESHOLD_AVG] ;
    }

    clock_t start, reduction, finish;
    start = clock();
    if (mIsReductionEnabled)
    {
        Reduce();
```

```
            reduction = clock();
        }
        for (int avrCount = 0; avrCount != avgLoopCount; avrCount++)
        {
            prim_minimum_spanning_tree(mGraph, &p[ 0] );
        }
        finish = clock();

        if (mHasBeenReduced)
        {
            cout << "Reduction time    : " << ((double)(reduction - start)
    / CLOCKS_PER_SEC) << endl;
            cout << "With reduction    : ";
        }
        else
        {
            cout << "Without reduction: ";
        }
        cout << mTileEdgeLength << "," << (((double)(finish - start) /
    CLOCKS_PER_SEC) / avgLoopCount) * 1000 << endl;

        bool printDiagnostics = false;
        if (printDiagnostics)
        {
            for (int counter = 0; counter != mEdgeCount; counter++)
            {
                cout << "mWeights[ " << counter << "]  = " <<
    mWeights[ counter] << endl;
            }

            for (size_t i = 0; i != p.size(); ++i)
            {
                if (p[ i]  != i)
                {
                    cout << "parent[ " << i << "]  = " << p[ i] << endl;
                }
                else
                {
                    cout << "parent[ " << i << "]  = no parent" << endl;
                }
            }
        }
    }

    void BoostPrim::Reduce()
    {
        mHasBeenReduced = true;
        Reduction_Phase1();

        //phase 2 is not required for worst-case analysis
        //Reduction_Phase2();
    }

    void BoostPrim::Reduction_Phase1()
    {
        for (int y = 0; y != mTileEdgeLength; y += 1)
        {
            for (int x = 0; x != mTileEdgeLength; x += 1)
            {
                Step1(x, y);
                Step2(x, y);
            }
        }
    }
```

```
void BoostPrim::Reduction_Phase2()
{
    for (int y = 0; y != mTileEdgeLength; y += 1)
    {
        for (int x = 0; x != mTileEdgeLength; x += 1)
        {
            Step3(x, y);
            Step4(x, y);
        }
    }
}
```

```
void BoostPrim::Step1(int x, int y)
{
    //rule-in min edge of G vertex
    int minVertexWeight = g_MaxWeight + 1; //gurantees that the first
available edge would have a lesser value
    EdgeDirection direction = UNKNOWN; //invalid value useful for
sanity-check later on
    VertexEdgeWeights weights = mVerticesEdgeWeights[ VertexIndex(x,
y)];
    if (DoesHaveNorthNeighbour(x, y))
    {
        if (weights.mEast < minVertexWeight)
        {
            minVertexWeight = weights.mNorth;
            direction = NORTH;
        }
    }
    if (DoesHaveEastNeighbour(x, y))
    {
        if (weights.mEast < minVertexWeight)
        {
            minVertexWeight = weights.mEast;
            direction = EAST;
        }
    }
    if (DoesHaveSouthNeighbour(x, y))
    {
        if (weights.mSouth < minVertexWeight)
        {
            minVertexWeight = weights.mSouth;
            direction = SOUTH;
        }
    }
    if (DoesHaveWestNeighbour(x, y))
    {
        if (weights.mWest < minVertexWeight)
        {
            minVertexWeight = weights.mWest;
            direction = WEST;
        }
        assert (direction != UNKNOWN);
        RuleIn(x, y, direction);
    }

    void BoostPrim::Step2(int x, int y)
    {
        //rule-out max edge of G* vertex
        if (!DoesHaveSuothEastVertex(x, y))
        {
            //a G* vertex does not exist at these coordinates
            return;
        }
```

```
   // A --- X
   // |  C  |
   // y --- B
   // edges of virtual G* vertex, C, can be accessed within the
context of
   // G edges of vertices A and B
   int vertexA = VertexIndex(x, y);
   int vertexB = SuothEastVertexIndex(x, y);

   int maxVertexWeight = mVerticesEdgeWeights[ vertexA] .mEast;
   EdgeDirection direction = EAST;
   if (mVerticesEdgeWeights[ vertexA] .mSouth > maxVertexWeight)
   {
      maxVertexWeight = mVerticesEdgeWeights[ vertexA] .mSouth;
      direction = SOUTH;
   }
   if (mVerticesEdgeWeights[ vertexB] .mNorth > maxVertexWeight)
   {
      maxVertexWeight = mVerticesEdgeWeights[ vertexA] .mNorth;
      direction = NORTH;
   }
   if (mVerticesEdgeWeights[ vertexB] .mWest > maxVertexWeight)
   {
      maxVertexWeight = mVerticesEdgeWeights[ vertexA] .mWest;
      direction = WEST;
   }
   if (direction == EAST || direction == SOUTH)
   {
      //rule out a VertexA edge
      RuleOut(x, y, direction);
   }
   else
   {
      assert (direction == NORTH || direction == WEST);
      //rule out a VertexB edge
      RuleOut(x + 1, y + 1, direction);
   }
}

void BoostPrim::Step3(int x, int y)
{
   //rule-in min edge of G vertex if none of its edges is already
ruled-in
   //not required for worst-case analysis
}

void BoostPrim::Step4(int x, int y)
{
   //rule-out max edge of G* vertex if none of its edges is already
ruled-out
   //not required for worst-case analysis
}

void BoostPrim::RuleIn(int x, int y, EdgeDirection direction)
{
   //contract ruled-in edge
   int VertexA = VertexIndex(x, y);
   int VertexB = VertexIndex(x, y, direction);

   graph_traits<Graph>::out_edge_iterator edgeItr, edgeItr_end;
   for (tie (edgeItr, edgeItr_end) = out_edges(VertexB, mGraph);
edgeItr != edgeItr_end; edgeItr++)
   {
      if (target(*edgeItr, mGraph) == VertexA) //is this the edge to
```

```
be ruled in?
        {
            //remove the contracted edge
            remove_edge(VertexB, VertexA, mGraph);
            //get_target(*edgeItr, mGraph) = VertexB;
        }
        else
        {
            //all remaining edges of VertexB become edges of VertexA
            get_source(*edgeItr, mGraph) = VertexA;
        }
    }
}
void BoostPrim::RuleOut(int x, int y, EdgeDirection direction)
{
    //delete ruled-out edge
    remove_edge(VertexIndex(x, y), VertexIndex(x, y, direction),
mGraph);
}

int BoostPrim::VertexIndex(const int& x, const int& y, const
EdgeDirection& direction) const
{
    switch (direction)
    {
        case NORTH:
        {
            assert (DoesHaveNorthNeighbour(x, y));
            return ((y - 1) * mTileEdgeLength) + x;
        }
        case EAST:
        {
            assert (DoesHaveEastNeighbour(x, y));
            return (y * mTileEdgeLength) + x + 1;
        }
        case SOUTH:
        {
            assert (DoesHaveSouthNeighbour(x, y));
            return ((y + 1) * mTileEdgeLength) + x;
        }
        case WEST:
        {
            assert (DoesHaveWestNeighbour(x, y));
            return (y * mTileEdgeLength) + x - 1;
        }
        default:
        {
            assert (false); //must be one of the above
        }
    }
    return (y * mTileEdgeLength) + x;
}
void BoostPrim::VertexCoordinate(int& x, int& y, const
EdgeDirection& direction) const
{
    switch (direction)
    {
        case NORTH:
        {
            assert (DoesHaveNorthNeighbour(x, y));
            y--;
            break;
        }
        case EAST:
        {
```

```
            assert (DoesHaveEastNeighbour(x, y));
            x++;
            break;
        }
        case SOUTH:
        {
            assert (DoesHaveSouthNeighbour(x, y));
            y++;
            break;
        }
        case WEST:
        {
            assert (DoesHaveWestNeighbour(x, y));
            x--;
            break;
        }
        default:
        {
            assert (false); //must be one of the above
        }
    }
}
```