Image Oversegmentation Solved in Parallel

Mihai BUDIU – Politehnica University of Bucharest

June 1996

Contents

Chapter 1

Abstract

This paper focuses its attention on a basic problem originated in image processing — image segmentation. It considers a narrow subtask of the segmentation, namely over-segmentation reduction.

In order to solve it we first build a formal framework (using elementary graph theory) which helps to define the goals and offers us the necessary tools for attaining them. It reveals some strong connections between a classical (but still open) problem in the graph algorithms field, namely the one of computing a minimum spanning tree of a graph, and the problem of over-segmentation reduction.

We try to benefit from the results known for the minimum spanning tree, taking into account some special properties of the graphs which represent images (e.g. planarity).

After studying two very efficient classical implementations of the minimum spanning tree algorithms, extracting precious information about their complexity and behaviour in some circumstances (e.g. sorted input data), we turn our attention to parallelizing these algorithms.

We offer two tentative parallel implementations, one for each of the algorithms. We use a rather weak model of computation, compared to the ones usually employed in the literature, but which has two great merits: it models a wide variety of real machines, and can scale to use both very many as very little computing power, depending on their availability.

Our main concern is for evaluating the performances of the two parallel algorithms with respect to the resources they need. The theoretical results show that they both exploit the parallelism of the problem, although one of them does it better. Adding processors decreases the processing time of both, at some loss of efficiency (i.e. the processing time does not decrease as fast as the number of processing units increases).

Chapter 2

The Original Problem

2.1 Image Segmentation

One of the operations of image processing consists of splitting a discrete image (i.e. a collection of *pixels*) in some disjoint patches. Each of the patches is supposed to represent a unique object in the scene underlying the image. The intent is to make a first step towards the recognition of the contents of the image. This type of processing is called image *segmentation*.

A variety of methods can be used for this purpose, each based on some a-priori knowledge of the properties of the represented objects, combined with some general purpose techniques. The outcome of the segmentation is an image which has been split into contiguous groups of pixels forming the *segments*. The pixels belonging to a region are assigned a same label. The segmentation process can be viewed as a pixel labeling procedure.

Figure 2.1: Example of an over-segmented image

The result of the segmentation phase is sometimes an *over-segmented* picture, i.e. one within which false boundaries have been detected, as the figure 2.1 shows. Such images are subject to a new processing stage whose purpose is to merge regions which belong together. The general behaviour of such a stage can be described as follows: identify regions which are always distinct, and make each of the remaining regions link to such a one.

This is the problem we try solve: reduce the over-segmentation of an image. We tried to propose a solution designed in such a way as to make a parallel execution feasible (i.e. several parts of the image to be processed independently at the same time).

We devised some requirements for the prospective solution, which in our opinion make it easily applicable and general-purpose:

1. The solution should work with any number of available processing units; the algorithm should not depend on the size of the input problem. As a limit case it should work also on a typical *sequential machine*.
2. The processor model used has to be the most general possible: each processing unit has its own private storage capacity (i.e. no *shared memory* – a device widely used for theoretical evaluations of parallel

algorithms, but with a limited applicability in real environments). The *communication* between the processing units takes place by exchanging messages.

3. A participating processing unit is a classical deterministic machine with random-access storage.
4. The algorithms should be correct and proved so.
5. The time-complexity of the algorithms should be estimated.

In order to achieve these goals we have to define the problem precisely. The natural framework is graph theory.

2.2 Basic Graph Definitions and Notations

- By $|X|$ we denote the cardinal of the set X (the number of elements). We will work only with finite sets.
- A *directed graph* (or *digraph*) is a pair of sets $G = (V, E)$.
  - V is the vertex set. Denote $|V| = n$. We will call the vertices also *nodes*.
  - E is a binary relation over V, i.e. $E \subset V \times V$. E is the set of the *edges* of the graph. $|E| = m$. We will denote the edges either by e_i, with $i \in \{1 \dots m\}$, or by (u, v), the (ordered) set of their head vertices.
  - If the relation E is symmetrical, i.e. $E = E^T$, we can say that the graph is *undirected.* We will basically work with undirected graphs, but we will sometimes regard each undirected edge like a pair of directed edges in both directions.
  - For the edge $e = (u, v) \in V \times V$ the vertices u and v are called *heads* of the edge. We say that e is *incident* to u and v, and that u and v are *adjacent.*
  - In an undirected graph the *degree* of a vertex is the number of edges incident to it.
  - Sometimes one can attach weights to the edges. These are real valued functions defined on E. Let the weight of edge $e = (u, v)$ be denoted by $w(e)$ or $w((u, v))$. For our purposes the weights will always be positive.
  - If we allow several edges between a pair of nodes the graph is called *multi-graph*, else is called a *simple graph.*
  - For a simple graph we have $m \leq n(n-1)$. The equality is for a *complete graph*, having an edge between any two vertices.
- Some other useful notions: a *path* between nodes u and v is a sequence of nodes $(u = v_{k_0}, v_{k_1}, \dots v_{k_r} = v)$ such that $(v_{k_i}, v_{k_{i+1}}) \in E$ for all $i \in \{0 \dots r-1\}$. r is the length of the path in edges. Denote by $P(u, v)$ the set of all paths between u and v.
  - A *cycle* is a path whose first and last vertexes coincide.
  - A *simple path* does not pass twice through an edge or vertex.
  - A *connected* undirected graph has a path between any pair of nodes. A graph which is not connected is *disconnected.*
  - A *planar* graph is a graph which can be drawn on a plane without the edges intersecting.
- A *subgraph* $G' = (V', E')$ of $G = (V, E)$ is a graph such that $V' \subset V, E' \subset E$ and $e \in E'$ only if $e \in E$. We will denote this fact by $G' \prec G$.
- A *tree* is a connected graph having no cycles. A *forest* is a union of vertex disjoint trees.
- For a graph G a *spanning tree* T is a tree $T \prec G$ having the same vertex set as G. A *spanning forest* of G is a forest having the same vertex set as G. Any connected graph has a spanning tree. All spanning trees have $n - 1$ edges.

- We define a *minimum spanning tree* of a graph G with weighted edges as being a spanning tree of G whose total weight (sum of all edge weights) is minimum. A graph may have several minimum spanning trees, having all the same total weight. Denote the (nonempty!) set of minimum spanning trees of a connected graph G by $mst(G)$.
- A *cut* of the graph is a partition of the vertices into two sets $V = V' \cup V''; V' \cap V'' = \phi$. For a given cut some edges are *cross-edges*, i.e. $e = (v', v''); \quad v' \in V', v'' \in V''$.
- Two graphs G and G' are *isomorphic* if there is a one to one correspondence between their vertices which maps one to one the corresponding edges too: $f : V \rightarrow V'; \quad f$ is one-to-one, such that $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.
- Two graphs G and G' are *homeomorphic* if there is a one to one correspondence between the edges of one graph and the paths of the other one. To be more precise: they can be made isomorphic just by adding or deleting vertices of degree 2.
- We define on a graph $G = (V, E)$ and one of its edges $e \in E$ the operation *edge condensation*. This consists of taking the nodes adjacent to e and of merging them into a single one. The edges of the resulting node are the union of the edges of the two component nodes, and e is dropped. Let $G \circ e$ denote the condensation of the graph G by the edge e.

2.3 The Graph Associated to an Image

Associate a segmented image with an undirected graph in this way:

- Each *region* corresponds to a *vertex* in the graph.
- Each two regions that have a common border imply an *edge* incident to their corresponding nodes.
- The *weight* of an edge reflects the *distance* between the regions. The higher the distance, the lower the likeliness that these two regions belong to a same component. The way this weight is computed from the actual image is not our concern.
- Coalescing regions in the image means establishing *equivalence relations* over the vertices. Two vertices are equivalent if their regions should become one.
- Some vertices of the graph are *marked*. These vertices are certain to belong to different equivalence classes; they can be chosen to be representatives of their classes, as no two marked vertices will belong to a same class. Let the set of marked vertices be M; denote the elements of M by m_i.

2.3.1 Remarks

- For our purposes it is often useful to have all *edge weights* distinct. We can easily extend the weight order to accomplish this requirement if we further order two edges having the same weight after their indices in the set E, or after the indices of their head vertices. Since we are dealing only with *simple* graphs there are no multiple edges between any pair of vertices.
- An image graph is always *planar*. Planar graphs have important properties which can help optimizing our algorithms. We will come back to them later.
- An image graph is always *connected*.

We will give proofs for the last two assertions in a later section.

2.4 Region Merging

The problem of region merging becomes that of finding for each graph vertex a marked vertex to whom it should be assimilated. This means finding a partition of the set V such that in any resulting class there is exactly one marked vertex m_i. Obviously, the result will have $|M|$ sets. Any partition defines an *equivalence* relation over the set V. We will say that $v_i \equiv v_j$ if and only if they belong to the same set in the partition.

We can solve this problem by defining a distance between vertices; then each vertex can be assimilated to the nearest marked vertex. In other words

$$v_i \equiv m_k \quad \Leftrightarrow \quad d(v_i, m_k) = \min_{m_j \in M} \{d(v_i, m_j)\}. \tag{2.1}$$

The *distance* between two adjacent vertices should be simply the weight of the edge linking them. However, there are several possibilities for defining the distance between non-adjacent vertices. For this purpose we will extend the notion of weight to paths. Having defined the weight of a path we will be able to define the distance between two vertices as

$$d(u, v) = \min_{p \in P(u,v)} w(p)$$

(we used again the function w to denote path weight). Define $\min_{x \in \phi} w(x) = \infty$.

Two alternatives for defining the path weight come to our thought:

1. Path weight is the sum of the weights of the component edges:

$$w(P) = \sum_{e \in P} w(e). \tag{2.2}$$

2. A path is as heavy as the heaviest edge on it:

$$w(P) = \max_{e \in P} w(e). \tag{2.3}$$

It is easy to check that both weights for paths provide a *distance* between vertices, i.e. the function $d : V \times V \to \mathbf{R}$ respects the distance axioms ($d(v, v) = 0; d(u, v) = d(v, u); d(u, v) \leq d(u, x) + d(x, v) \forall u, v, x \in V$).

To distinguish between the two distances induced by these weights, we will denote the second distance function by $\mathcal{D}(u, v)$. So let us state it again:

$$\mathcal{D}(u, v) = \min_{p \in P(u,v)} \max_{e \in p} w(e). \tag{2.4}$$

Simple examples reveal that the two definitions are not equivalent. In the figure 2.2 the thick edges have weight 3, and the thin ones 2. The top and bottom-most vertices are marked.

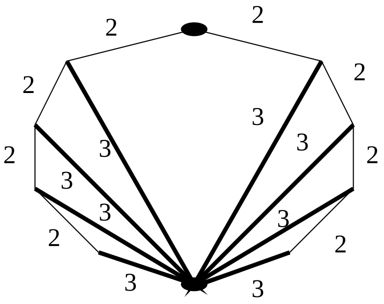

Figure 2.2: Different classes by different metrics

Using the second metric (from equation 2.3) all the other vertices depend upon the top-most one, but with the metric of equation 2.2 most of them depend upon the lowest-one.

For the purposes of image processing the semantics of the second choice proves to be more suitable, so we will continue to use only that one. It has some interesting properties, which we will study in the next chapter.

Chapter 3

Graphs and Images

3.1 Forests and Image Segments

We begin by a general lemma:

Lemma 1 *Let G be a connected graph which is not a tree, T a spanning tree of G, and $e \in E, e \notin T$. Then:*

- *$T \cup \{e\}$ has a cycle C;*
- *$\forall e' \in C$ we have $T \cup \{e\} - \{e'\}$ is a spanning tree of G.*

Proof:

- Consider the heads of e. T is a spanning tree, so both vertices are in T. Because T is connected there is a path between these two nodes with edges in T. It follows that there are two edge disjoint paths between them in $T \cup \{e\}$. These two paths form a cycle C.
- Denote $T' = T \cup \{e\} - \{e'\}$. T' has the same number of edges as T. We just have to show it is connected to prove that it is a spanning tree. Take two nodes in G and consider the path L linking them in T.
    - if $e \notin L$ then $L \in T'$ so the nodes are connected.
    - if $e \in L$ then there is a path in T' which instead of e passes through all the other edges of the cycle C.

The results follow. ▮

There is a strong connection between the distance $\mathcal{D}$ defined by equation 2.3 and the *minimum spanning trees* of a graph, as proven by the following couple of results:

Lemma 2 *Let G be a connected graph having only one marked m node and $T \in mst(G)$. Denote the path linking a node $v \in V$ to m by $P(v)$. Then $w(P(v)) = \mathcal{D}(v, m)$.*

Proof: In other words the paths in the minimum spanning tree are all of minimum weight (with the definition of equation 2.3). We can prove this result by contradiction.

Assume a node v exists, such that $\mathcal{D}(v, m) < w(P(v))$. Fixing v simplify the notation $P = P(v)$. Take the path P' from v to m having the weight $\mathcal{D}(v, m)$. Let the heaviest edge of P be h; then $w(h) = w(P)$ by the definition of the path weight. $P \neq P'$, as $h \notin P'$. The figure 3.1 depicts this situation.

Chose the first edge $l \in P' - T$ on P' when going from v to m. Add l to T. Denote $l = (a, b)$ such that $a \in P \cap P'$ and $b \in P' - P$. Because $a \in P$ we have $P = (v, \ldots, a \ldots, m)$. $h \in P$, so either

- h between v and a. But this is impossible, as the path $v \rightarrow a$ is also in P' (by the way we chose a) and $h \notin P'$.
- h between a and m.

We also have a path $(a, b, \ldots, m) \subset P'$ which does not include h. Thus there is a cycle in $T \cup \{l\}$ containing both l and h, because there are two paths from a to m, one through l and one through h. Because $l \in P'$ and $w(P') < w(P)$ we have $w(l) < w(h)$.

Now consider $T' = T - \{h\} \cup \{l\}$. This is again a spanning tree by lemma 1. However T' has a lower total weight than T, so T cannot be a minimum spanning tree of G.

This is a contradiction, so a node v satisfying the condition $\mathcal{D}(v, m) < w(P(v))$ cannot exist. Because $\mathcal{D}(v, n) \leq w(p)$ by definition for any path $p \in P(v, m)$, we have the desired result. ∎

And here is a generalization of the previous result:

Lemma 3 *Let $G = (V, E)$ be a connected graph having positive edge weights. $M \subset V$ is a set of* marked *vertices. Add to V a vertex o, and to E the edges $E' = \{(m_i, o), i \in \{1 \ldots |M|\}$ all of weight 0. Denote $T \in mst(V \cup \{o\}, E \cup E')$ by $T = (V, MS)$. The graph $(V, MS - E')$ obtained by removing the added edges is a forest. If in this forest two vertices are in a same tree then they are equivalent by the definition 2.1.*

Proof:

Because the weights are positive T will include all edges with weight 0. If it didn't we could build a lower cost tree by adding that edge and breaking the formed cycle (which doesn't contain another 0-weight edge).

Any non-marked vertex will be in exactly one tree of the forest $(V, MS - E')$ (if it is in two, there is a cycle in T, if it is in none G is not connected). No two marked vertices will be in a tree of the forest, because adding back the 0 edge we would get a cycle.

Denote by T_v the tree including v, and its unique marked 'root' m_v.

We have to show that $\mathcal{D}(v, m_v) \leq \mathcal{D}(v, m) \quad \forall m \in M - \{m_v\}$. Assume that this does not hold, i.e. $\mathcal{D}(v, m) < \mathcal{D}(v, m_v)$ for some m. Consider the path in G linking v with m_v with weight $\mathcal{D}(v, m_v)$, the path linking v with m and the edge (m, m_v). This is a cycle. All its maximum weight edges are on the path from v to m_v. With an argument like the one in the previous lemma (2), we can prove that there is a lighter spanning tree of $G \cup E'$ including the first edge of the lighter path which is not in the heavier path, and without one of the heavy edges. This is a contradiction, as $T \in mst(G \cup E')$. ∎

This result allows us to proceed solving the over-segmentation problem by computing minimum spanning trees!

3.2 Planar Graphs

Every image is a planar connected graph. (Recall that a planar graph can be drawn on a plane such that none of its edges intersect.) We will prove this and some other properties in this section, which happen to have important contributions to optimize the solutions of the problem we're studying now.

If a planar graph has been laid on a plane in the manner described above, it divides the plane into closed regions named *faces*. Let us use the notation f for the number of faces of a planar layout of a graph. This number does not depend on the layout we chose (a planar graph may be drawn in many different ways). In fact we have:

Lemma 4 *For any connected planar graph we have $f = m - n + 2$ [Euler formula].*

Proof: Consider a spanning tree of the graph (it exists because the graph is connected). It has $n - 1$ edges and only one face. For it we have $f = 1 = m - n + 2 = n - 1 - n + 2$, so the result holds. Each edge we add splits one face into two, so it increases the number of faces by one. The number of nodes remains the same, and the number of edges obviously increases by one too. The result follows. ∎

Lemma 5 *For every simple planar graph $m = O(n)$.*

Proof:

We can prove an even stronger result, which is: $m \leq 3n - 6$ for any simple connected planar graph. The proof is immediate based on the Euler formula and goes like follows:

Every face of the graph has at least three neighbouring edges (because the graph is simple), so counting the edges bordering all the faces we have at least $3f$ edges. In this way we count each edge twice. The following relation must hold: $3f \leq 2m$. Plugging into the Euler formula we get:

$$3(m-n+2) \leq 2m \Leftrightarrow m-3n+6 \leq 0 \Leftrightarrow m \leq 3n-6.$$

For a disconnected simple planar graph this relation holds for each of the components. The result follows immediately for simple planar graphs. ∎

We will need also the following result, which is somehow difficult to phrase.

Lemma 6 *If G is a planar graph whose n vertices are uniformly distributed in a rectangle in the plane, any straight line intersecting two opposite edges of the rectangle crosses $O(\sqrt{n})$ edges.*

Proof: This lemma is intuitively true, but we are not able to provide a proof right now. ∎

3.3 From Pixels to Regions

The limit case of this approach is to consider each pixel of the image to be an independent *segment* and to be adjacent to its four touching neighbours. We will call the graph having each pixel as node and edges to all four neighbours the *full image graph.* In this case the very algorithm we present for reducing the over-segmentation can be used for the segmentation task itself!

We have even more: the regions of a segmented image, no matter in which way are obtained from individual pixels, are in fact *edge condensations* of the original full image graph. Using this fact and the observation that the full image graph is obviously planar and connected, we get:

Theorem: *Any image graph is connected and planar.*

The proof is a simple consequence of the following lemmas:

Lemma 7 *An edge-condensation of a connected graph is connected.*

Proof: Assume that between the nodes u and v there is a path P in the graph G.

- If $e \notin P$ then P is a path also in $G \circ e$, because all the edges and nodes of P exist in $G \circ e$ too.
- If $e \in P$ than consider $P \circ e$ – the path obtained by short-cutting in P all the apparitions of e; then $(P \circ e) \prec (G \circ e)$, and $P \circ e$ is a path between u and v.

It follows that there is a path between u and v in the condensed graph wherever there was one in the original one. As a consequence, if the original graph was connected, so is the condensed one. ∎

Also:

Lemma 8 *Any edge-condensation of a planar graph is planar.*

Proof: By a theorem due to *Kuratowski*, a graph is planar if and only if it does *not* contain a subgraph *homeomorphic* to one of the Kuratowski graphs. (For our purposes it does not matter which are these graphs; they are the complete graph with 5 vertices, and the complete bipartite graph with 3,3 vertices.) Assume that $G \circ e$ contains a subgraph $H \prec (G \circ e)$ *homeomorphic* to one of the Kuratowski graphs. Then G will itself contain such a subgraph, because:

- If H does not contain the node created by the merging of the heads of e then $H \prec G$.
- If H contains that node, the subgraph of G obtained by expanding H with e is homeomorphic to H, by the definition of homeomorphism. Homeomorphism is transitive.

The result follows by contradiction. ∎

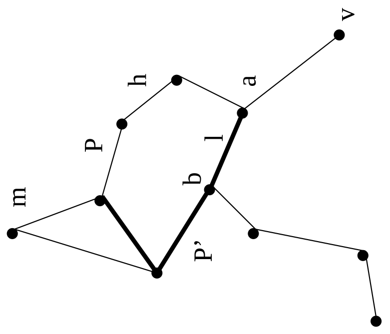

Figure 3.1: Tree paths have minimum weight

Chapter 4

Minimum Spanning Trees

As the chapter 3 points out, we can solve the problem of reducing the over-segmentation by solving a minimum spanning tree problem. In this chapter we focus our attention from an algorithmic point of view towards finding a minimum spanning tree of a graph.

For our algorithm description we will use a language very similar to Pascal.

4.1 The Red and Blue Rules

All the algorithms we shall study build the spanning tree gradually using two rules, known in the literature as the *red rule* and the *blue rule*. Here they are:

Blue rule: Select a cut of the graph that has no blue cross-edges. Among the cross-edges select a minimum-one and color it blue.

Red rule: select a maximum cost uncoloured edge belonging to a simple cycle with no other red edge. Colour it red.

These rules work because every application maintains the following invariant:

Colour Invariant: there exists a minimum spanning tree containing all the blue edges and no red edge.

If the above invariant is true we can easily build a minimum spanning tree by repeatedly applying the rules. When no edge is uncoloured the blue edges form a minimum spanning tree.

Lemma 9 *Applying the* red *and* blue *rules to a connected graph maintains the* colour invariant *and colours all the edges of the graph.*

Proof: We make the proof in two steps: first that the rules keep the invariant and next that they do colour all the edges.

1. We will prove by induction on the number of colouring steps the fact that the invariant is respected. At the beginning of the colouring all the edges are uncoloured, so the colour invariant is obviously true, because any connected graph has at least one minimum spanning tree, which includes the empty set. Assume that the invatiant was still true after $k-1$ edges were coloured. Let T be the minimum spanning tree which satisfies the invariant after $k-1$ steps, and e the edge coloured at step k.
    - If e is coloured blue we distinguish two cases:
        - if $e \in T$ we have nothing to prove — the invariant is respected;
        - if $e \notin T$, consider the cut V', V'' to which the blue rule was applied. In T there is a unique path joining the heads of e, and this path crosses the cut at least once (a spanning tree crosses *any* cut). Take an edge of T which is a cross-edge of V', V'', and name it e'. Now, e has been selected by the blue rule, so $w(e) \leq w(e')$. The graph $T \cup \{e\} - \{e'\}$ is again a spanning tree of G, because of lemma 1. Its weight is $w(T) + w(e) - w(e') \leq w(T)$, so it must be again a minimum spanning tree.

We have thus proved that there exists a minimum spanning tree *after* selecting e.

- If e is red, we deal again with two cases:
  - $e \notin T$ — we are done;
  - $e \in T$. $T - \{e\}$ is a disconnected graph; it defines a cut. The cycle from which e was selected has another edge crossing that cut, say e'. Because of the red rule selection $w(e) \geq w(e')$, and e' is not yet coloured. But $T \cup \{e'\} - \{e\}$ is again a spanning tree, with a lower weight, so it must be a minimum spanning tree. Moreover, it satisfies the invariant.

Now the second part of the lemma, the 'completness':

2. Assume that we have applied a number of times the rules and we are left with a graph having an uncoloured edge e, but no rule can be applied.
   - If there is a cut-set (V', V'') such that e crosses it, and no other blue edge is a cross edge for (V', V'') we can apply the blue rule to this cut-set and e colouring one more edge.
   - If there is no such cut-set, it means that there is a path linking the heads of e formed only of blue edges. Then we apply the red rule to the cycle containing e and this path.

   This is a contradiction.

With this all the cases are covered, so the lemma is proved. ∎

4.2 Kruskal Algorithm

Kruskal algorithm consists of repeated applications of the above rules. It makes use of the operations *makeset*, *union* and *find* used for manipulating disjoint sets and for set membership testing. These procedures are described in the appendix A. We use them to keep track of the *blue trees* that develop because of the applications of the blue rule. (A blue tree is a tree with all the edges coloured blue).

```
function KruskalMST((V, E) : graph): set of edge;

var
  MST: set of edge;
  v,u: vertex;
  e  : edge;

begin
  E := sort(E) on weight;
  MST := [];
  for v in V do makeset(v);
  for e in E do begin
    (v, u) := e;
    if find(u) <> find(v) then begin      {edge is blue}
        union(v, u);                      {colour it so}
        MST := MST + [e];
    end
   {else edge is red – discarded}
  end;
  return MST
end
```

4.2.1 Correctness of Kruskal Algorithm

The *correctness* follows immediately from the observation that all selected edges are blue and all the other edges are truly red, so the color invariant is respected.

4.2.2 Complexity of Kruskal Algorithm

Lemma 10 *The cost of the Kruskal Algorithm for a graph with m edges and n nodes is $O(m \log n)$.*

Proof:

- There are n *makeset* operations.
- The sorting of the edges takes by the best methods $m \log m$.
- The inner cycle is executed m times. All operations are $O(1)$ in the cycle except the two *find* operations. As the appendix A shows, the cost of m *find* operations on a set of n objects is $\Theta(m\alpha(m,n))$, where the function $\alpha(m,n)$ is smaller asymptotically than $m \log n$.

Because for a graph $\log m = O(\log n)$ it remains that the complexity is dominated by the sorting of the edges. ∎

4.2.3 Remarks

- We use the notation $f(n) = O(g(n))$ when $n \to \infty$ for $f, g : \mathbf{N} \to \mathbf{R}_+$ to mean that there are positive constants C, n_0, not depending on n, for which $f(n) \leq Cg(n); \forall n > n_o$.
- We use the notation $f(n) = \Theta(g(n))$ when $n \to \infty$ for $f, g : \mathbf{N} \to \mathbf{R}_+$ to mean that there are positive constants C_1, C_2, n_0, not depending on n, for which $C_1 g(n) \leq f(n) \leq C_2 g(n); \forall n > n_o$.
- If the edges are pre-sorted or can be sorted very fast (e.g. are small integers) the algorithm running time is dominated by the time of the execution of the main cycle, being thus $O(m\alpha(m,n))$.

4.3 Edge Condensation

The previous algorithm carries repeated applications of the *edge condensation* operation. Some of its properties are studied in Chapter 3. This operation is met over and over again in the minimum spanning trees algorithms, sometimes in disguise. The efficiency with which we will be able to implement it will impact on the overall complexity of the algorithm.

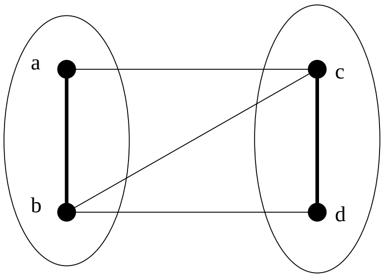

Figure 4.1: Multiple edges between collapsed nodes

Because this operation can use nodes which are already the result of previous applications, the graph will be no more a *simple* graph, as it will develop multiple edges between pairs of nodes; the figure 4.1 shows such a case. The leftmost two nodes **a** and **b** have been merged (blue edges are shown by thick lines), and so have **c** and **d**. There are three edges between the two ‘composite’ nodes.

Why is this operation so important in spanning tree algorithms? Well, first of all it is not performed on *any* pair of nodes, but only on nodes which become linked by an edge of the minimum spanning tree (thus a blue edge). This means:

- that the merged nodes are in cut-sets with no blue edge across (by the blue rule). One set of the cut-set contains just one of the two nodes.

- that they will *never* be again in different cut-sets separated only by uncoloured edges (because between them there is a blue path). This is why no other edge incident with both these two nodes will be ever selected by the blue rule. By the colouring lemma 9 we know that every edge will be either red or blue. It follows that all the remaining edges connecting these two nodes are (will be) red and can be coloured so!

As an example, the figure 4.2 shows by dashed lines the red edges and by thick lines the blue edges. There are no more uncoloured edges in that figure.

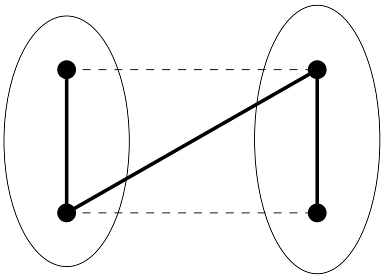

Figure 4.2: Only one blue edge between composite nodes

When merging two nodes we can colour red all the edges that link them and which are not coloured.

More than that: when we merge two nodes multiple edges develop between this newly created one and all the nodes to which the original components were linked both. Referring back to figure 4.1, when node **a** has been linked to the node **b**, both being linked to the node **c** meant that a double edge developed between this new composite node and **c**.

All these edges are linking different cut-sets with no blue edge between. However, only one of these edges will be blue; all the rest are deemed to become red.

As a consequence of these observations **we can regard each application of the blue rule as an edge condensation!**

4.4 Boruvka Algorithm

Lemma 11 *The minimum weight edge incident to a node will always be part of a minimum spanning tree.*

Proof: An edge is coloured red only if it is the heaviest edge in a cycle. Any cycle has to pass through any node twice, thus it will contain two edges incident to that node.

Assume that the minimum weight edge incident to a node is not in a minimum spanning tree. Add the edge to the tree; so a cycle will develop. This cycle must include the two edges incident to the node: the minimal one and the one in the tree. Remove the tree edge, breaking the cycle, like in lemma 1. We get so a tree with smaller weight, which must be a minimum spanning. ∎

Using this *locality* property, we can search for the edges forming a minimum spanning tree in a zone of the graph without concern for what is happening in other parts. We should always select the minimum edges incident to a node. The only restriction is not to have two edges with the same weight in the graph, because this could lead to the formation of cycles, when each node chooses its own, as the example in figure 4.3 shows.

4.4.1 One Implementation

One solution, called the *round robin* algorithm processes nodes as uniformly as possible. We make use of a function *incident(v)*, which is supposed to give the list of edges incident to the node v. Here is an implementation:

```
1  procedure BoruvkaMST((V, E): graph);
2      {computes a minimum spanning tree MST}
3  var
4      u, v, x: vertex;
5      queue  : set of vertex;
6
7  begin
8    {first some initializations}
9    queue := [];
10   for v in V do begin
11       makeset(v);
12       queue := queue + [v];
13       incident(v) := preprocess(incident(v))
14   end;
15
16   {the robin goes round:}
17   while length(queue) > 1 do begin
18       v:= head(queue);
19       (x, u) := MinBlueEdge(incident(v));
20       MST := MST + [(x, u)];
21       queue := queue − [v, u];
22       v := union(v, u);
23       incident(v) := combine(incident(v), incident(u));
24       queue := queue + [v]
25     end
26 end;
```

The algorithm can be described like follows: *select a blue tree from the front of a queue, find the minimum edge incident to it, condense the graph on this edge (i.e. colour it blue) and put the tree at the back of the queue.* Some operations can be implemented obviously fast enough: all the operations on *queue* can be done in $O(1)$ (constant) time by maintaining it as a doubly linked list; the *union* and *find* are described in the appendix A. The efficiency depends crucially upon the implementation of the other operations, notably set operations and:

```
function preprocess(set of edge): set of edge;
        {makes some preprocessing on the set of edges so as to
        simplify the operations of the following functions}

function MinBlueEdge(set of edge): edge;
        {returns a minimum weight edge which is not red
         from the indicated set}

function combine(set of edge, set of edge) : set of edge;
        {merge the two edge-sets}
```

There are many variants of implementation with dramatical impacts on the running time. They differ only with respect to the data structure used for maintaining the list of edges incident to a node (tree) so as to extract fast the minimum edge and to merge two edge-sets quick.

4.4.2 Refinements of the Implementation

As observed in section 4.3 multiple edges develop between collapsed nodes, most of them red. The *union* operation in line 22 is necessary only because the function *MinBlueEdge* has to distinguish red edges (having both ends in a same tree) from the uncoloured ones, and it will do so by checking if they are in different trees with *find*.

One alternative is to do some *clean-up* from time to time, removing the red edges. They will be anyhow eliminated, but only in later phases of the algorithm, when the nodes incident to them get mixed. Until that point they participate in all the combinations of the nodes, which carry them along, contributing much to the complexity.

The simplest way to implement the above functions is simply to keep for each node the list of its edges in unsorted order. The functions become:

```
function preprocess(E: set of edge): set of edge;
            { do nothing ! }
begin
    return E
end;

function MinBlueEdge(E: set of edge): edge;
            {search the minimum weight non-red edge in an unsorted set}
var
   m : edge;
   u, v : vertex;
begin
    m := null;                          {assume w(null) = infinite}
    for (u,v) in E do
         if w((u,v)) < w(m) and          {lower weight}
              find(u) <> find(v) then    {not red}
                   m:=(u,v);
    return m
end;

function combine(e1, e2 : set of edge) : set of edge;
            {concatenate the edge sets}
begin
   return e1 + e2
end;
```

If the set of edges incident to a node is kept as a circular list, we get the following times for each operation:

- a preprocessing step for each node takes nothing;
- the *MinBlueEdge(incident(v))* takes $O(m_v)+$time for m_v *find*s, where m_v is the number of edges incident to v;
- the *combine* function takes $O(1)$ at each call.

This is a very simple implementation, yet very efficient. It surpasses the other obvious implementation, in which the edges are kept sorted. In that case preprocessing would take $O(m_v \log m_v)$ for each vertex v, a call *MinBlueEdge* would be $O(1)$, but *combine(u, v)* would rise up to $O(m_u + m_v)$.

Observe that each invocation of *MinBlueEdge* is accompanied by one of *combine*, so we wish that their total cost to be lowest. So it is better to pay for minima than for combinations, as the latter work with larger sets.

We still have to deal with the *multiple edges* developing between nodes. We can modify the algorithm to make successive *sweeps* over the nodes in the graph. After a complete sweep follows a *cleanup*. The clean-up phase removes all red edges in the graph. Because all the edges left after a clean-up will be uncoloured, we can get rid of the *union* and *find* calls!

Mixing all together we can rewrite the algorithm like follows:

```
1  procedure BoruvkaMST((V, E): graph);
2     {computes a minimum spanning tree in MST}
3  var
4    queue, aux: set of node;
5    u, v, x     : node;
6
7  begin
8    queue := [];
9    for v in V do begin
10       queue := queue + [v];
11       dependents(v) := null;        {nodes in this blue tree}
12   end;
13   while length(queue) > 1 do begin
14     {begin sweep}
15     aux := [];
16     while length(queue) > 0 do begin
17        u:= head(queue);
18        queue := queue − [u];
19        (x, v) := minimum(incident(u));   {x <> u possibly!}
20        MST := MST + [(x, v)];
21        if v in aux then
22           aux := aux − [v]
23        else
24           queue := queue − [v];
25        dependents(u) := dependents(u) + [v] + dependents(v);
26        incident(u) := incident(u) + incident(v);
27        aux := aux + [u]
28      end;
29      queue := aux;
30      cleanup
31      {end sweep}
32    end
33 end;
```

4.4.3 Complexity of This Approach

Let us make a preliminary estimate of the complexity $C(n, m)$ of the algorithm. We start by estimating the way the *queue* decreases in the innermost cycle during a sweep. At each pass one node is removed from the queue and mixed with another one, either from *queue*, or from *aux*. In the first case two nodes from *queue* will contribute one node to the *aux*. In the other case three or more nodes will do (because any node on *aux* contains at least two nodes from *queue*). After *queue* is depleted it is renewed from *aux*, in line 29. Conclusion: in the worst, *queue* halves its length during one sweep. More precisely, denoting by $queue(i)$ the *queue* at the beginning of the i-th sweep: $|queue(i+1)| \leq \lceil |queue(i)|/2 \rceil$.

We also have $|queue(0)| = n$. The sweep happens, as you can see from the line number 12, until at phase number k: $|queue(k)| \leq 1$. We can immediately get a value for k, the number of sweeps:

$$1 = |queue(k)| \leq \left\lceil \frac{|queue(k-1)|}{2} \right\rceil \leq \left\lceil \frac{\lceil |queue(k-2)|/2 \rceil}{2} \right\rceil = \left\lceil \frac{|queue(k-2)|}{4} \right\rceil \leq \dots \leq \left\lceil \frac{|queue(0)|}{2^k} \right\rceil$$

using the properties of the $\lceil \cdot \rceil$ function. From this we get $1 \leq \lceil n/2^k \rceil$ or $k = \lceil \log_2 n \rceil$.

Now we should try to estimate the complexity of the operations done through each sweep. The only costly operations (more than $O(1)$) are on the lines 19, and 30.

- The function *minimum*: On line 19 we have a search for a *minimum* in an unordered set, which has a cost equal to the size of the set, because all the edges in the set have to be examined. We can't estimate it, but we can do better: we can *amortize* the cost over all the operations done in a single sweep. Observe that no node is found during a sweep twice in *queue*. It means that each edge of the graph can be scanned at most twice, once for each of its heads, during a single sweep. We have that the cost of all the calls to *minimum* in sweep $\#i$ is $O(2m_i)$. (Denote by m_i the number of the edges in the graph during pass i).

- The concatenation of the incident edges (in line 26) of two nodes can be done in constant time if we keep the edges as linked lists. This operation is $O(1)$.
- We keep track for each tree v of the component nodes with a list, *dependents(v)*. To this list we add in constant time the nodes of another tree when mixing, in line 25.
- The only important cost left is the one for *cleanup* (line 30), which we can't estimate, unless we suggest an algorithm for performing it. We will give below a solution which performs the clean-up on a graph with m edges in $O(m)$.

Using these values let us write the complexity of the overall algorithm: the sum of the costs of all sweeps:

$$C(m,n) = \sum_{k=0}^{\lceil \log_2 n \rceil} 2O(m_k). \tag{4.1}$$

We can immediately bound $m_k \leq m$ to get:

$$C(m,n) = O(m \log_2 n) \tag{4.2}$$

which, by the way, is just the cost of the algorithm with no clean-up. That's a very interesting result: a complexity equal with the one of the Kruskal algorithm.

4.4.4 The Clean-Up

We owe an implementation for the clean-up, which should run in time proportional to the number of edges.

Despite the apparent complexity the solution is rather plain: label the nodes with the tree they are in, and next eliminate

- all edges with same label at both heads;
- all but the minimum weight edge from all edges incident to the two same labels.

Because of the labeling the edge ellimination can be done in a single pass inspecting all the edges.

The initializations take $O(n)$, as each node is considered only once. The large **for** loop over the elements of *E1* does indeed take exactly $O(m)$, as it considers each edge, and all operations in the loop are $O(1)$.

The only critical point is the sorting of E on the two keys, the labels of the heads of the edges. But the labels are small numbers, in the interval $1 \dots n$, so we can use a fast sorting method, like *radix sort*, which is $O(m)$ again.

```
procedure cleanup;

var
  v,u : vertex;
  e, previous, min : edge;
  E1: set of edge;

begin
  for v in queue do begin                        {for all blue trees}
     for u in dependents(v) do label(u) := v;  {mark node in each tree}
     incident(v) := null                        {rebuild incident edges list}
  end;
  E1 := sort(E) on (labels of incident nodes); {quick sorting}

  E   := [];                  {prepare to rebuild the edge set}
  previous := null;
  min := null;
  for e in E1 do begin
     (u, v) := e;
     if label(u) = label(v) then                {red edge; skip}
     else begin
        {scan for minimum edge between these two labels}
        if headlabels(previous) <> headlabels(e) then begin {new pair of nodes}
           E := E + [min];
           (u, v) := min;
           incident(label(u)) := incident(label(u)) + min;
           min := e;
           previous := e
        end
        else if w(e) < w(min) then              {new minimum edge}
           min := e
     end
  end
end;
```

4.4.5 The Complexity Revisited — Planar Graphs

As the chapter 3 shows, the planar graphs have some important properties which allow us to improve the time bounds of the algorithm in such a case.

The key observation is the fact that all the graphs entering one sweep are still planar (by lemma 8), connected (by lemma 7) and simple (because of the clean-up).

As the lemma 4 shows, a planar graph with n_i nodes has less that $3n_i - 6$ edges.

We also observed that the size of *queue* halves at each sweep. But *queue* holds exactly the one node for each blue tree, which is a node of the graph considered in the course of a sweep. So we have $n_i = |queue(i)| \leq n/2^i; \forall i \leq \log_2 n$. Let us plug these estimates into the equation 4.1:

$$\begin{aligned}
C_{planar}(m,n) &= \sum_{k=0}^{\lceil \log_2 n \rceil} O(m_k) \leq \sum_{k=0}^{\lceil \log_2 n \rceil} 3Cn_k \\
&\leq \sum_{k=0}^{\lceil \log_2 n \rceil} 3Cn/2^k \leq 3Cn \sum_{k \geq 0} 1/2^k \\
&= 6Cn = O(n).
\end{aligned}$$

The complexity is linear in n if the graph is planar!

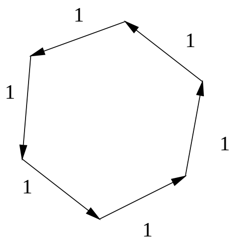

Figure 4.3: Boruvka Algorithm Failing

Chapter 5

Parallelizing the Algorithms

This chapter shows some tentative parallelizations of the derived minimum spanning tree algorithms useful in the case of image processing.

We will be using the minimum spanning tree algorithms presented in the previous chapter, but we will modify them slightly to suit our main goal, that of finding an equivalence relation between the regions in an image. For this:

- We will use the technique described in section 3.1 to build the equivalence classes: we will assume that all marked nodes are linked to a fake node with 0 weight edges.
- We will be not interested in the actual edges which form the forest but only in the equivalence between the nodes.

5.1 Splitting an Image On a Processor Array

We suppose that the pixels in an image are arranged in a rectangle of constant width and height (i.e. which has the same number of pixels along each vertical and horizontal section). We will split the image in the easiest way like follows: divide the rectangle into smaller rectangles with the same orientation, and give each rectangle to one processor. Ideally there would be an equal number of splits along both the vertical and the horizontal dimension.

Our image processing algorithms will be easier to express (and will make more efficient use of the resources) if the number of splits along one side is a power of two. We will assume this holds; a generalization is straightforward.

In the 'real world' the image may have been split in earlier stages of the processing (before the segmentation phase), so the cost of distributing it to the processor array will not be taken into consideration when we analyze our algorithms. (The distribution may have been made even in the stage of acquisition.)

Processors need to share some information with their neighbour, as there will be edges from regions in different processors. One method is to give a processor and each of its neighbours some pixels in common, as in figure 5.1.

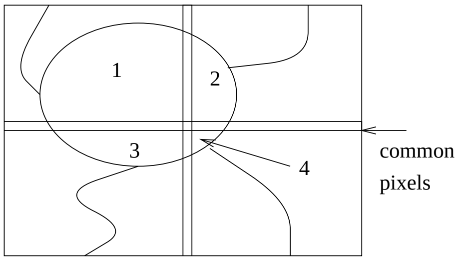

Figure 5.1: Image distribution to 4 processors

We also assume that the process of segmentation has been carried on a per processor-basis. Referring again to the figure 5.1 the ellipse region, which would normally span all the four processors, is actually recognized as four different regions (the ones labeled with numbers in the figure). No region spans several processors. It remains a task to the following processing steps, which are precisely intended to reduce over-segmentation, to merge the four indicated regions into a single one. Figure 5.2 shows the graph which would be created for the image in figure 5.1.

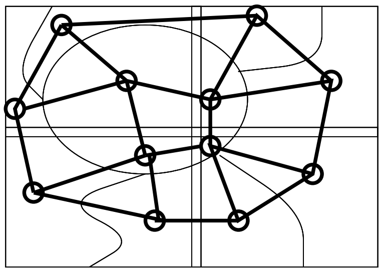

Figure 5.2: The graph built from the previous image

How do you decompose a graph into subgraphs whose union is the initial graph? If the graph is connected (and we know that images are) there is no way to *partition* it (in disjoint graphs). Thus we have to *replicate* information on the processors. The simplest way to deal with this is to duplicate some *edges* of the graph, and to represent their heads in both subgraphs.

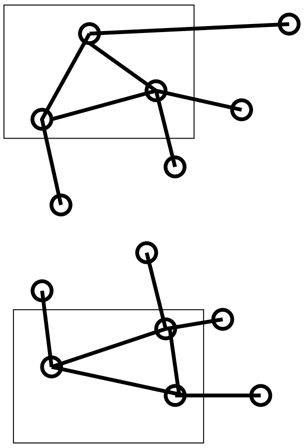

Figure 5.3: The graphs of two processors — from previous image

As the image 5.3 shows, in this process the edges linking neighbouring regions in distinct processors get duplicated. (So do their head nodes.)

The only remaining problem is to derive the *weights* of the duplicated edges. This is actually a problem of the meaning of the image. If the weight of an edge spanning two processors can be determined solely on the values of the pixels lying on the frontier (a realistic assumption under some circumstances) each processor will compute it from the common strip. Otherwise some preliminary stages of information exchange would be necessary between neighbouring processors. We will not concern ourselves with this matter, assuming it is part of the input data.

5.2 Merging the Partial Results

How will the image processing proceed from now on? Each processor will work as much as possible on its own part of the graph. When everybody extracted as much as it could from the local data the results are merged somehow between neighbouring processors and the computation proceeds.

This is like the *conquer* phase from a divide and conquer algorithm. (The division was already performed on the image before our algorithms started.)

The functioning will be SIMD-like with a large grain. That is, all processors will perform the same processing simultaneously; after all are ready each pair of processors merges the results; in each pair of processors one goes to sleep and the other one continues to process the combined data.

```
{external view of the processing}
while processors > 1 do begin
    each processor(partial mst);
    group processors in pairs;
    each pairs mixes results;
    processors := processors / 2;
    one in each pair goes to sleep
end;
```

Or, subjectively, we can view the processing of each individual like this:

```
{code of each processor}
while (I am active) do begin
    compute partial mst;
    if (I will remain active) then
          get(data from my neighbour)
    else begin
          send(data to my neighbour);
          go to sleep
    end
end;
```

Fortunately there's no need for *barriers* or some other kind of global synchronization mechanism, because all the processors do not have necessarily to perform in lock-step. There is implicit synchronization in the message-passing mechanism: one will go to work as soon as it has the data from its neighbour.

There are several methods of combining the partial results. We will assume the following simple pattern of communication, depicted in figure 5.4, which halves after each step the number of processors, and keeps the processor array almost square.

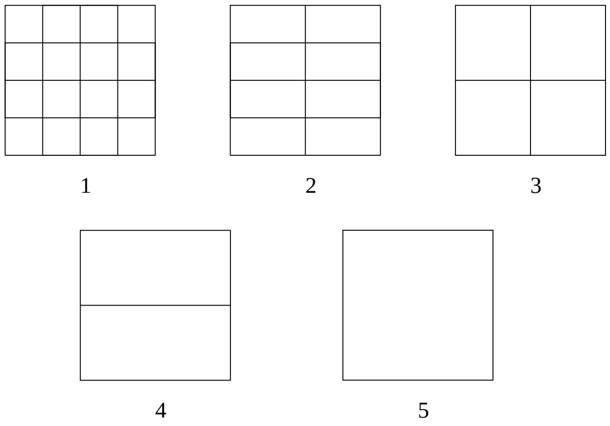

Figure 5.4: Communication phases for 4x4 processors

To be more precise:

```
{code of processor with coordinates (x, y) in grid}
function communicate(step: integer) : boolean;
  {communicates data; returns true if more work to do}
var active: boolean;

begin
  active := true;    {default}
  if even(step) then begin {horizontal communication}
    if (odd(x)) then begin
      send(processor(x+1, y));
      active := false        {I am out of job from now on}
    else
      receive(processor(x−1, y));
    x := x div 2
  end
  else {if odd(step)}
    ...  {same processing on y coordinate}
  end;
  return active
end;

{main operation}

step := 0;
active := true;
preparation;                        {takes a SU time}

while (active) do begin
  compute partial mst;              {takes a PR(step) time}
  active := communicate(step);  {takes a COM(step) time}
  step := step + 1
end;
```

Using this pattern we can readily estimate the number of processing *steps.*

During step k there are $P(k) = P(0)/2^k$ active processors because their number halves at each step. The computation will finish in $k_{max} = \log_2 P(0)$ steps. The analysis is a bit more complex if these numbers are not integer.

During step k, $P(0) \cdot (1 - 1/2^k)$ processors are idle.

5.2.1 Analysis Method

To effectively analyze the running times we will use another apparently very strong assumption, but which is essential for our estimates. We suppose that *in each step of the processing the nodes and edges of the graph are uniformly distributed over all the existing processors.*

If the graph is not uniform we can still use our results, but substituting for n a higher value. Namely we consider the processor which holds the highest number of nodes of the graph (at each moment all processors compute on areas of equal size) and extrapolate this node density to all other. However the worst-case results obtained in this way are overly pessimistic and of no practical interest.

Because all the (active) processors compute in parallel during each step, we really have to add the times of the processing of *one* of them, no matter which, because of the uniformity assumption.

To compute the time complexity $C(m, n, P)$ of such an implementation we will have to sum up several components:

- The *set-up time SU*. The operations needed for *preparation*, which bring the input data in a form suitable for the latter uniform processing.

- The *processing time* $PR(k)$ of each of the steps. This should be summed for all the steps, from $k = 0$ to $k = \log_2 P$.
- The *communication time* $COM(k)$. This is proportional to the amount of data one processor sends to its active neighbour. Sum it for all k from 0 to $\log_2 P - 1$. The -1 is because the last step of the computation if not followed by any communication.
- We may also consider *communication start-up time*, which is the overhead incurred for any sent message. But in our case each processing step concludes with one message being sent, irrespective of the nature of the processing, so this time is really constant through all the algorithms, and equal to $\log_2 P$ times the overhead for one message.

Summarizing we get:

$$C(m,n,P) = SU(m,n,P) + \sum_{k=0}^{\log_2 P} PR(m,n,P,k) + \sum_{k=0}^{\log_2 P-1} COM(m,n,P,k). \tag{5.1}$$

We used the following notations:

- k is the step number.
- $P = P(0)$ is the total number of participating processors. In all the problems we study $P \in \{1 \ldots n\}$. $P(k)$ is the number of active processors in step k.
- n and m characterize the size of the initial graph. In reality SU, $PR(k)$ and $COM(k)$ are all functions of m, n and P, but we simplified the notation.
- Because of the lemma 6 and the way we distribute the graph to the processors (duplicating some edges and nodes), the total number of nodes processed in a step exceeds the total number of nodes in the graph.
  - at odd steps processors work on square areas. If one area has inside n nodes, we assume it has $\sqrt{n}$ edges crossing one side of the square if there is a neighbour on that side. The other end of these edges has to be included in the processed nodes; there are at most $4\sqrt{n}$ more nodes to be considered;
  - at even steps the processors work on rectangular areas, formed from two adjoining squares. If the number of nodes in a square is n (and in the rectangle $2n$), there will be less than $6\sqrt{n}$ more nodes to be considered.

  The values are lower for marginal processors. To simplify the discussion, majorate the coefficient of the square root with $a \leq 4$. We then have, uniformly:

  Each processor works during step k on at most:

  $$n(k) = n/P(k) + a\sqrt{n/P(k)} \tag{5.2}$$

  nodes at a time. We can substitute $P(k)$ to get

  $$n(k) = n2^k/P + a\sqrt{2^k n/P}.$$

The asymptotic running time of the resulting algorithm (as $n \to \infty$) will be certainly no better than the time of the best sequential algorithm (else the sequential algorithm can be optimized).

The problem is that times are usually obtained as *asymptotic* values, not precise ones, because we are really counting *operations* and not *time* (worse than that, we are counting only *dominating* operations). Asymptotic evaluations disregard constant factors, like $1/P$, which are for us the most interesting.

We will try to compute the *speed-up* of the parallel algorithm, defined as the ration between the time of a sequential implementation and that of a parallel implementation. Because our algorithms are working in the case $P = 1$ too, we can simply use

$$S(m,n,P)=\frac{C(m,n,1)}{C(m,n,P)}. \tag{5.3}$$

We will try to analyze the behaviour of the algorithm in two limit cases: when P, the number of processors is very small compared to the size of the graph, and when P is on the order of n.

P **small** — P will have a constant value, practically independent of n; $P \ll n$. In this case, probably SU will be dominant, because the size of the smallest problem will be big. On the other hand there will be rather few computation steps: $\log_2 P$ is small.

P **big** — We will use for P a value depending on n: n/f, where f is a small constant factor (possibly 1); $P=\Theta(n)$. In this case SU should become unimportant, but the sums for COM and PR will come to have $\log_2(n/f)$ terms, and will bring a higher contribution to the overall complexity.

5.3 A First Method — Kruskal in Parallel

In the pseudo-code detailed previously the only step that needs customization depending on the applied method is the one we named *compute partial mst*. With the help of the following lemma we can readily conceive a possible solution:

Lemma 12 *Let* $G_k, k \in \{1 \ldots r\}$ *be* r *(non-disjoint) connected subgraphs of* G*:* $G_k \prec G$*, such that* $\bigcup_{k=1}^{r} G_k = G$*. If* $T_k \in mst(G_k)\, \forall k$*, we have* $\exists T \in mst(G)$ *such that* $T \subseteq \bigcup_{k=1}^{r} T_k$*.*

Proof: First observe that because $\bigcup_k G_k \supseteq G$ we have that $\bigcup_k T_k$ spans all the vertices of G. Take an arbitrary $T \in mst(G)$ and consider and edge $e \in T - (\bigcup_k T_k)$.

- If no such e exists we have done, as T is the tree we were searching for.
- Otherwise consider the heads of e. Because every edge in G is in some G_i, $e \in G_k$ for some k, so its both ends lie in some T_k. Then $e \cup T_k$ has a cycle. e is a maximal edge in this cycle, because of the red rule. However, $e \in T$, so there must be another edge e' with the same weight in this cycle. Then $(T - \{e\} \cup \{e'\}) \in mst(G)$ and we have built a new minimum spanning tree without e.

In this way we can find a $T \subseteq \bigcup_k T_k$. ∎

We can apply the result obtained in the following way: each processor computes a minimum spanning tree of the part of the graph it possesses. Merging partial results is a simple union of graphs, on which subsequent computations can be carried on.

Any sequential algorithm may do for the computation of the minimum spanning tree. We can use the best ones available.

One good candidate would be Kruskal, which is easy to implement and rather efficient, because we are in the situation that after the first step the received edges are already sorted and can be merged 'on the fly'.

We can clarify the procedures unspecified in the pseudo-code:

```
{code for one processor}
procedure preparation;
begin
  sort(edges) on weight;
end;

procedure compute partial mst;
begin
  graph := KruskalMST(graph);
end;
```

```
procedure send(p : processor);
begin
    transmit(p, edges(graph));  {send sorted edge list}
end;

procedure receive(p : processor);

var received edges: list of edge;

begin
    received edges := get(p);
    edges := merge(edges, received edges);  {merge two sorted lists}
end;
```

The graph is kept as the list of edges, which is preliminary sorted. Afterwards each processor when receiving merges the sorted edge lists it possesses and it receives.

5.3.1 Analysis of the Approach

Observe that a transmitted graph has always less than $n-1$ edges, as it is a forest. Also, in our special case, the graphs being planar and *simple*, $m = O(n)$.

Recall the complexity of the sequential algorithm, derived in section 4.2.2: $C(m, n, 1) = O(m \log n)$.

Let us estimate each of the three component times for the parallel version.

SU, the set up time: the set-up means each processor computing a minimum spanning tree of its local graph; this is $O(m \log m)$ using Kruskal on a graph with m edges. Assuming that the edges are distributed uniformly across processors we get using equation 5.2:

$$SU = O((m/P + a\sqrt{m/P}) \log(m/P + a\sqrt{m/P})).$$

$PR(k)$, the processing time of step $k > 0$: the time of the Kruskal algorithm for a graph with sorted edges, with less than $2n(k-1)$ nodes (the ones received from the previous step computation), and $2n(k-1)$ edges (the edges of the trees computed at the previous step by itself and its neighbour). Or, substituting

$$PR(k) = O\left(2n(k-1)\alpha(2n(k-1), 2n(k-1))\right).$$

$COM(k)$, the communication time: just counting the number of transmitted edges at step k, which is $n(k)$:

$$COM(k) = \Theta(n(k)).$$

Complexity for $P \ll n$

When summing over k the $PR(k)$ and $COM(k)$ do not change their asymptotic behaviour *if* the number of terms in the sum is constant with respect to n. In other words $\sum PR(k) = O(n\alpha(n, n))$ and $\sum COM(k) = O(n)$, from their maximum term, $n(\log_2 P) = n$. The term dominating asymptotically $C(n, m, P)$ is SU, which is $O\left((m/P) \log(m/P + a\sqrt{m/P})\right)$.

This means that asymptotically (i.e. for m and $n \to \infty$ and P constant) we are spending as much time as a sequential algorithm solving a problem almost P times smaller. To be more precise the ratio of the problems is given by:

$$\begin{aligned} R &= \frac{m \log m}{\frac{m}{P} \log\left(\frac{m}{P} + a\sqrt{\frac{m}{P}}\right)} = P \frac{\log m}{\log\left[\frac{m}{P}\left(1 + \sqrt{\frac{P}{m}}\right)\right]} \\ &= \frac{P}{1 - \frac{\log P}{\log m} + \frac{\log\left(1 + a\sqrt{\frac{P}{m}}\right)}{\log m}} \approx \frac{P}{1 + \frac{a\sqrt{P/m} - \log P}{\log m}} \approx P. \end{aligned}$$

Why does this happen? Because the first phase (the set-up) of the algorithm is the most complex asymptotically, and can be effectively carried out in parallel. The rest of the computations contribute with only a lower order factor to the overall complexity of the algorithm.

The key is the fact that the edges come sorted in latter stages, and thus the minimum spanning tree algorithm carried at subsequent steps looses a logarithmic factor in complexity.

However, the impact of the lower-order terms cannot be underestimated in real applications.

Complexity for $P = \Theta(n)$

We replace in the above formulas $P = n/f$. SU becomes constant and ignorable.

This time the sums have $\log_2(n/f)$ terms, so they are no more asymptotically unimportant. Let us try to estimate COM:

$$\begin{aligned} COM &= \sum_{k=0}^{\log_2 n/f-1} \Theta\left(f2^k + a\sqrt{f2^k}\right) = \Theta\left(f\frac{2^{\log_2 n/f}-1}{2-1} + a\sqrt{f}\frac{\sqrt{2}^{\log_2 n/f}-1}{\sqrt{2}-1}\right) \\ &= \Theta\left(f\left(\frac{n}{f}-1\right) + a\sqrt{f}\left(\sqrt{\frac{n}{f}}-1\right)\frac{1}{\sqrt{2}-1}\right) = \Theta(n). \end{aligned}$$

and PR; instead of summing from 1 terms in $n(k-1)$ sum from 0:

$$\begin{aligned} PR &= \sum_{k=0}^{\log_2 n/f-1} O(2n(k)\alpha(2n(k), 2n(k))) \\ &= \sum_{k=0}^{\log_2 n/f-1} O(2(f2^k + a\sqrt{f2^k})\alpha(2f2^k + 2a\sqrt{f2^k}, 2f2^k + 2a\sqrt{f2^k}) \\ &= O(n\alpha(n + a\sqrt{n}, n + a\sqrt{n})). \end{aligned}$$

The (asymptotic) efficiency increases, but by an order less than the number of available processors. From $O(n \log n)$ it goes to $O\left((n\alpha(n + a\sqrt{n}, n + a\sqrt{n})\right)$ for $O(n)$ processors.

$$S(m, n, n) = \frac{C(m, n, 1)}{C(m, n, P)} = \frac{O(\log n)}{O\left(\alpha(n + a\sqrt{n}, n + a\sqrt{n})\right)}. \tag{5.4}$$

This holds for planar graphs and for high P.

The speed-up is less than logarithmic.

5.4 Parallelized Boruvka

In the previous case we could not avoid transmitting all the edges, as any of them could be a candidate for removal in next stages of processing.

In this section we explore the parallelization of the exposed variant of Boruvka. This time we will make edge condensations irrevocably (recall that the parallel Kruskal variant submitted again condensed edges to latter processing steps). Each processor will condense edges which are certain to belong to the final global minimum spanning tree. These edges do not have to be transmitted to further levels of processing. The number of vertices taken into consideration by later stages of the processing will be lower and lower, so we can expect a decrease of the communication and processing costs.

We have only to take care of the possibility of the application of blue rules when we do *not* know all the edges incident to a node, as is the case of the vertices outside the region of a processor.

We base our presentation on the Boruvka algorithm presented in section 4.4. One step of the modified algorithm goes like this:

```
procedure ParallelBoruvka;

var u, v, z: vertex;

begin
  passive:= passive+ neighbour passive;
  active := [bogus];            { mark sweeps with this node }

  for v in frozen do begin
      (u, z) := MinBlueEdge(incident(v));
      if (z in neighbour frozen) or (z in active) then begin
          if z in active then active := active - [z]
          else neighbour frozen := neighbour frozen - [z];
          frozen := frozen - [v];
          active := active + [u, z];
      end;
      {else z is not an internal node}
  end;
  frozen := frozen + neighbour frozen;
            { here the real processing begins }
  while active <> [] do begin
      v := head(active);
      active := active - [v];
      if v = bogus then begin        { end of sweep }
          cleanup;
          active := active + [v]
      end
      else begin
          if incident(v) = null then
              closed := closed + [v];
          else begin
              (u, z) := MinBlueEdge(incident(v));
              dependents(z) := dependents(z) + [v] + dependents(v);
              incident(z) := incident(v) + incident(z);
              if z in active then begin
                          active := active - [z,v];
                          active := active + [z]        {link at the end}
              end
              else if z in frozen or
                          external(z) then
                          frozen := frozen + [v]
                          {linked to an external node: freeze}
              else if z in passive then
                          passive := passive + [v]
                          {linked to a marked component: make passive}
          end
      end
   end
end;
```

Here is the data processed by the algorithm:

- input data:
  - *frozen* — list of nodes frozen during the previous step;
  - *passive* — list of nodes made passive in previous steps;
  - *neighbour frozen* — frozen list of nodes received from neighbour;

  - *neighbour passive* — received passive list;
  - *closed* — list of nodes which have been solved (no more neighbours);
- output data:
  - *frozen* — list of nodes frozen during this pass;
  - *passive* — list of nodes coloured during this pass;
  - *closed* — list of nodes which are done.

We use four node queues to represent it:

active are processed in this step;

closed have no other neighbour; this is a collapsed component of the image which has been fully identified;

passive are marked nodes or connected to a marked one.

frozen not to be processed because they have the minimum edge towards a node not in this processor.

The algorithm description is simple: at the first step all nodes are *frozen*. Next all nodes which can be processed are unfrozen and put on the *active* list. A node is unfrozen if its minimum weight incident edge is internal.

The algorithm next proceeds condensing the graph on the minimum edges, except that it has to be careful about nodes having no more neighbours (because the result is a set of trees, not connected) and about edges crossing processor boundaries. The latter cause their incident nodes to freeze until both ends will be in the same processor.

The marked nodes are always frozen-like because they are linked to each other by 0-weight edges, which can be considered as cross-processor edges. This is why they are put on a special list, *passive*.

The nodes which are collapsed with a marked node become passive too, because semantically they have now got a cross-edge: the one with zero weight, linking all marked nodes.

Only the reduced graphs are passed from one processor to its neighbour.

We also have to pay special attention to the fact that a processor does not transmit nodes with their own edges. Figure 5.5 explains this.

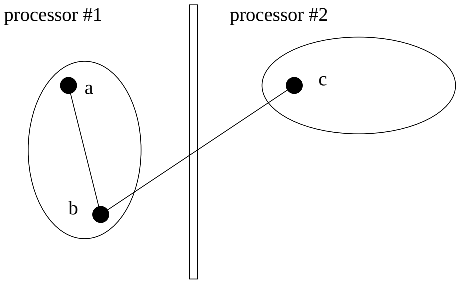

Figure 5.5: Recognizing external nodes

The processor #1 has collapsed nodes **a** and **b**. It has chosen **a** as representative for the resulted node. When communicating with processor #2, the node **a** will be sent, which has inherited during the collapsing the edge **b**–**c**. The other processor knew only about node **b**, to which its own node **c** has a link. How can it determine that the received node **a** contains the old **b**? Well, it can do that, if the node **a** has kept in the description of its incident edge the original heads, **b** and **c**.

5.4.1 The Reverse Propagation

This algorithm is rather effective, because it does not pass any information about a node which has been incorporated into another one during a collapsing. In this way much less information is passed from one step to the following.

The *closed* nodes need not be passed at all.

However, as soon as a processor finds a *closed* or *passive* node (in the last step all non-marked nodes will become so) it should inform the original owner of the node about its label, in a *reverse propagation* stage. This would be necessary in order to label the nodes which were shrunken to create this one, and which are known only by the processor which did the shrinking.

Refering again to figure 5.5, presume that processor #2 determined the received node **a** is linked to a marked node **c** (processor #1 knows nothing about **c** being a marked node). **a** is 'contaminated' with this markage: i.e. the region it represents has to be labeled like **c**. Processor #1 has to be informed about the markage of **a**, because he alone can mark node **b**.

This stage of the algorithm can proceed after the first one has ended, in symmetric steps: each processor tells to the processors from which it got received nodes the marker to which they were connected. We can do even better, by propagating a node backwards as soon as it becomes *closed* or *passive*.

The implementation is rather straightforward, but needs some book-keeping: which node has been received from whom. A high-level description would be:

```
{step k of reverse stage}
begin
  $L$ := receive list of nodes with their zone markers;
  for $v$ in $L$ do
     for $u$ in dependents($v$) do marker($u$) := marker($v$);
  send(list of nodes received in step $k-1$, with their markers)
end;
```

Complexity Analysis of Reverse Propagation

As we will see the cost of this part of the algorithm is smaller than the cost of the first part, so it does not influence the overall behaviour. For each reverse step k we will have to add the time of:

- reverse communication — proportional to the number of received nodes at the same step in the 'forward' stage, so less than the forward communication time, which used to send edges too;
- node labeling, which is again proportional to twice the number of received nodes, because all the received nodes and locally owned nodes are touched once, and their numbers are approximately equal (due to uniformity of distribution).

The cost of a reverse step is dominated by the cost of the corresponding forward step.

5.4.2 Analysis of the Parallelized Boruvka Algorithm

We will have again to estimate the three types of component times. The sequential algorithm time is linear, by the analysis we made a bit earlier, in section 4.4.3.

For Small P

- SU is given by the application of the algorithm to a graph P times smaller. The complexity is even smaller than for a full minimum spanning tree computation, because the processing will fail to shrink the entire graph due to external edges. The terms in $\sqrt{n/P}$ from 5.2 do not appear, as the external nodes do not participate in the computation. $SU = O(n/P)$.
- $COM(k)$ is given by the amount of data processors communicate. This includes the marked nodes, the *frozen* nodes and all their incident edges after clean-up. In a processing stage all the nodes become either *closed*, *frozen* or *passive*. The closed nodes do not take part in the communication, but the passive ones propagate until the last step, because other nodes may cling to them. There are two involved costs:

  - a term for the *frozen* nodes, which is $O(\sqrt{n/P(k)})$, because only the nodes having one external edge freeze, and these are less on the order of $a\sqrt{n/P(k)}$.
  - a term for the *passive* nodes. This is easier estimated globally for all steps, because each node is communicated $\log_2 P$ times. If there are $|M|$ marked nodes the total time for *all* the phases will be at most $|M| \log_2 P$ (if all marked nodes are in one processor initially).

- $PR(k)$ encompasses two parts:
  - the computation of a minimum spanning tree (lines 20-50) on the union of two collapsed graphs. Even better, only the active nodes take part in the computation; we just have to estimate their number. But active nodes become only some of the locally frozen and received frozen ones, and they do not exceed (if uniformly distributed) $\sqrt{n/P(k)}$. The time of Boruvka is linear in the input size, so the cost of such a processing is $O(\sqrt{n/P(k)})$.
  - The building of the *active* list itself (lines 5-20) is linear in the length of the frozen lists, and contributes with a term of the same magnitude to the complexity.

- As observed earlier the reverse stage does not add significant cost to the complexity, as each of its steps is dominated by the corresponding forward step; we can assimilate its cost fully in the forward pass, both for small and for large P.

Summing up (for $P \ll n$) we have the same pleasant surprise as for the Kruskal algorithm, namely that the significant factor is the set-up time. This time when using P processors we are really solving a problem P times smaller.

For Large P

Substituting $P = n/f$ in the cost formulas and summing we get:

- $SU = O(n/P) = O(f) = O(1)$ — insignificant term.
- $COM(k) = O(\sqrt{n/P(k)}) = O(\sqrt{f2^k})$ plus the contribution of the marked nodes, which is overall (for all k) $|M| \log_2 (n/f)$.
- $PR(k) = \Theta(\sqrt{n/P(k)}) = \Theta(\sqrt{f2^k})$.

Or, estimating globally:

$$\begin{aligned} COM &= \sum_{k=0}^{\log_2 n/f-1} O(\sqrt{f2^k}) + |M| \log_2 \frac{n}{f} = O\left(\sqrt{f}\frac{\sqrt{2}^{\log_2 n/f}-1}{\sqrt{2}-1}\right) + |M| \log_2 \frac{n}{f} = O(\sqrt{n}). \\ PR &= \sum_{k=0}^{\log_2 n/f} \Theta(\sqrt{f2^k}) = \Theta(\sqrt{n}). \end{aligned}$$

Again parallelizing the algorithm we get for a large number of processors a worse asymptotic behaviour than for the case with few processors. The performance increases from $O(n)$ to $O(\sqrt{n})$ when using $O(n)$ processors.

The speed-up is however better than in the previous case (Kruskal):

$$S(n, m, n) = \frac{C(n, m, 1)}{C(n, m, P)} = \frac{O(n)}{O(\sqrt{n})}. \tag{5.5}$$

The speed-up is like the square root of the number of processors.

Chapter 6

Final Conclusions

To summarize, we can draw the conclusion that the parallel variant Boruvka algorithm is more efficient in most respects when compared with the one of Kruskal:

- it is asymptotically faster (like $O(n)$);
- it has a higher speed-up when using many processors (like $\sqrt{P}$);
- it has a good performance when using few processors (speed-up like P).

The parallelization of the Kruskal algorithm has the following features:

- asymptotical complexity $O(nlogn)$;
- a rather ineffective speed-up when using many processors (like $logP$);
- a good performance when using few processors (speed-up like P);
- much easier to implement.

For small problems, only experimental tests can show which is better. For large images, and especially for many processors, Boruvka algorithm looks better suited. However, the reverse phase is a bit of a nuisance.

The nature of the Kruskal algorithm suggests its utilization in a pipeline, where the merging of the partial results are done simultaneously with the processing of new ones.

Let us conclude this paper with the initial oversegmented image (courtesy of Alina Moga), now processed using a variant of the parallel Boruvka algorithm:

Figure 6.1: The over-segmented image processed

Appendix A

Union-Find Algorithm

In the presented algorithms we faced the problem of maintaining some disjoint sets. We had n vertices, each belonging to some set, which represents a connected component of the graph. We faced the question whether two vertices belong to a same set. This problem can be solved with a simple abstract data type, *set*, adorned with three basic operations:

```
procedure makeset(v : element);
    {Build necessary data structure for future set operations}

function union(v1, v2 : element): element;
    {Destroys both sets and makes one of the union of all their elements.
     If sets not distinct no action is taken.
     Return the canonical element of the new set. }

function find(v : element) : element;
    {returns the canonical element from set to which this element belongs}
```

To implement these methods one would represent each set element as a node in a tree. The tree root is the "canonical" element of the set. We could then identify a set with the canonical element.

It can be proven that with suitable implementations we can derive a precise and tight upper bound for the cost of m *find* operations on a collection of sets comprising n nodes, no matter in which order the (at most) $n-1$ *unions*, the n *makeset* and the m *finds* are carried out.

The complexity of these operations is $\Theta(m\alpha(m,n))$, where the function $\alpha(m,n)$ is defined to be a very slowly growing function, a functional inverse of the Ackermann function $A(x,y)$. Here is the slightly adjusted definition of $A(x,y)$ used for defining α:

$$\begin{array}{rcl} A(1,y) & = & 2^y \quad x \geq 1 \\ A(x,1) & = & A(x-1,2) \quad y \geq 1 \\ A(x,y) & = & A(x-1,A(x,y-1)) \quad x,y > 1 \end{array}$$

The definition of α is

$$\alpha(m,n) = \min\{i; A(i,\lfloor m/n \rfloor) > \log_2 n\}.$$

With this definition $\alpha(m,n) \leq 4$ for any $n \leq 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$, where the stack of 2's has a height of 16. For all practical purposes α is smaller than 4.

References

Finding minimum spanning trees — David CHERITON and Robert Endre TARJAN
SIAM Journal of Computation, vol 5, No. 4, December 1976

The algorithm of section 4.4 is taken from here, together with the analysis of its complexity for the planar case.

Data Structures and Network Algorithms — Robert E. TARJAN
SIAM Press, Philadelphia, 1983

Kruskal algorithm in section 4.2 is presented and analyzed in the 6-th chapter of this book. The red and blue rules are explained plainly, and the proof of the 9 lemma is given.

Worst-Case Analysis of Set Union Algorithms — Robert E. TARJAN and Jan van LEEUWEN
Journal of the ACM, Vol. 31, No. 2, April 1984, pp 245-281

The complexity of the disjoint set-operations used mainly for the Kruskal algorithm implementations, and described briefly in the Appendix A are deduced in this paper.

Designing and Building Parallel Programs — Ian FOSTER
http://www.mcs.anl.gov:80/dbpp/text/book.html, An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation.

Very useful techniques for the analysis and estimation of parallel implementations have been extracted; namely the division of the time into its communication, processing and set-up components, as it is presented in section 5.2.1.

Algorithmic Graph Theory — Allan GIBBONS
Cambridge University Press, Cambridge, 1989

Basic results in graph theory with their proofs, used as starting point for our Chapter 3 are here: Kuratowski theorem, Euler formula and planar graph properties.

Concrete Mathematics — Donald E. KNUTH, Oren PATASHNIK, Ronald GRAHAM
Addison-Wesley, 1989

General useful techniques in the analysis of the algorithms, like the $O(\cdot)$ notation properties are found here.

Private Communications — Alina MOGA
Tampere University of Technology, 1996

The very problem of image segmentation has been sugested by Alina MOGA, which worked simultaneously in the same direction. The parallel Kruskal algorithm from section 5.3 is from her. The actual parallel implementation (using pvm and MPI) of both parallel algorithms, testing and experimental results belong to her.