UNIVERSITY OF OSLO
Department of Informatics

PQ-trees and maximal planarization

An approach to skewness

Gørril Vollen

Cand Scient Thesis

February 1998

Trying to get an edge on this

- I went astray -

not seeing the wood for the trees

Preface

This thesis is a result of my study for the Cand. Scient. degree at the Department of Informatics, University of Oslo. The work has been carried out in the time period between September 1995 and February 1998.

For the acknowledgments, I would first and foremost like to thank my supervisor, Almira Karabeg, for introducing me to a very interesting species of trees. She always had the time to discuss with me the different aspects of trees and bushes.

I am also very grateful to professor Stein Krogdahl who had mercy on me when I desperately needed someone to discuss the final outline of the chapters and this thesis with. He provided me with invaluable help in the final stages of this thesis.

For their ability to back me up and inspire me in the early days of this study, I would like to thank Ahmed El-abbadi and Kjetil Karlsen.

Great thanks to Dag-Erling Smørgrav for his patience and interest when I got lost in the tricky parts of C++ templates.

My fellow students at "lesesal 1302" have been a good support as well, with a cake break once in a while, and lots of moral support in the final spurt of this thesis. A special thanks to Leif John, who let his Companion[1] be my Companion when I needed one.

For knowing when to back me up and when to back off, I want to thank my good friend Vibeke.

Finally, my boyfriend Roffy has been most patient, always encouraging me to keep at it until I was done, and my parents who have always been there, supporting me when I needed it.

Blindern, February 1998

Gørril Vollen

[1] M. Goossens, F. Mittelbach, and A. Samarin: *The LaTeX Companion*, Addison-Wesley, 1994.

Contents

1 Introduction **1**

2 Graphs and planarity **3**
2.1 Definitions 3
2.1.1 Properties of planarity and non-planarity 6
2.2 Degrees of non planarity 6
2.2.1 Skewness 6
2.2.2 Thickness 7
2.2.3 Crossing number 8
2.2.4 Some relationships between these problems 9

3 *PQ*-trees **11**
3.1 *PQ*-trees and Permissible Permutations 11
3.1.1 The *PQ*-tree data structure 12
3.1.2 The reduction algorithm 14
3.2 Planarity testing with *PQ*-trees 18
3.2.1 *st*-numbering 20
3.2.2 Planar 22
3.3 Implementation 24
3.3.1 S. Leipert's implementation of *PQ*-trees 26
3.3.2 Implementation of Planar 27

4 *PQ*-trees and obstructions to planarity **29**
4.1 Additions to the *PQ*-tree 30
4.1.1 Additional information 30
4.1.2 Additional data structure 32
4.1.3 Building the pruned subtree 36
4.2 Maintaining additional information in the *PQ*-tree 38
4.2.1 Template procedures 39
4.2.2 Utility procedures 48
4.3 Finding the obstruction when the *PQ*-tree fails 49

4.3.1 Cases 49
4.3.2 Procedures 51
4.3.3 Implementation of additional data structure 56
4.4 OBSTRUCTIONS and skewness number 57

5 Approaches to planarizing graphs using PQ-trees 61
5.1 A planarization algorithm 62
5.1.1 Finding edges to delete 64
5.1.2 Deleting edges 67
5.1.3 PLANARIZE 68
5.2 Algorithms for maximal planarization 69
5.2.1 Initial approach [JTS89] 69
5.2.2 Changes made by Kant 70
5.2.3 Problems discovered by Leipert 73
5.2.4 An attack on the approach of [JTS89, Kan92] 76
5.2.5 A new approach 77

6 Towards a skewness algorithm 79
6.1 Combining PLANARIZE and `Boundary` 80
6.1.1 REMOVENODES 81
6.2 Finding near pairs 83
6.2.1 A closer look at the bush form 83
6.2.2 A closer look at `Boundary` 85
6.3 Skewness algorithm 89
6.3.1 Time complexity 90
6.4 Remaining problems 91
6.4.1 In search for perfection 93

7 Conclusion 97
7.1 Further work 97

Bibliography 99

A Implementation of `mypqtree` 103
A.1 Guidelines 103
A.1.1 vbcTool 104
A.1.2 Altered files of Leipert's implementation 104
A.2 Code files main.cc and mypqtree.h 106
A.2.1 main.cc 106
A.2.2 `pqtree` 107
A.3 Code files for parameter types 140

A.3.1 `edge` 140
A.3.2 `Chain` 141
A.3.3 `Boundary` 143
A.3.4 element 165

List of Figures and Tables

2.1 Graph with multiple edges and loops. 3
2.2 Trees. 4
2.3 Three embeddings of K_4, (c) with the four faces marked. . . . 4
2.4 K_5 and $K_{3,3}$ with respective homeomorphs. 5

3.1 $\mathcal{U}$ = {A,B,C,D,E,F,G,H}, S_1 = {A,B,C}. 12
3.2 Two more restrictions from $\mathcal{S}$ have been added to the PQ-tree of fig. 3.1(b), S_2 = {E,G,H}, S_3 = {E,D}. 12
3.3 The pruned subtree of the tree in fig. 3.2 when S_4 = {B,C,G}. 13
3.4 Templates 15
3.5 Tree of fig. 3.2 after reduction of S_4 = {B,C,G}. 17
3.6 Graph G, embedded subgraph G_4, and bush form B_4. 19
3.7 A biconnected graph. Left: Drawing. Right: Adjacency list. . 21
3.8 Graph after DFS. 21
3.9 st-numbered graph on the left. Outgoing and incoming edge sets on the right, used by PLANAR. 22
3.10 The correspondence between a PQ-tree and its bush form. . . 23
3.11 The initial tree, and the result of the first two iterations of the main loop of PLANAR, $i = 2, 3$. 24
3.12 Running of the example graph through PLANAR, $i = 4, 5, 6$. . 25
3.13 Class structure of Leipert's implementation. 26

4.1 Outer mesh of a biconnected graph 30
4.2 Biconnected component and boundary of triangle. 31
4.3 Building chains in the PQ-tree. T_4, T_5, and T_6. 32
4.4 `Boundary` of Q-nodes. T_6, T_7, and T_8. 33
4.5 The reduction of T_8 and T_9. 34
4.6 Reduction of T_9. 35
4.7 Case 1 49
4.8 Case 2 50
4.9 Case 3 50
4.10 Case 4 50

4.11 A sketch of the $K_{3,3}$ subgraph for Case 1. 51
4.12 A sketch of the $K_{3,3}$ subgraph for Case 2. 52
4.13 A sketch of the $K_{3,3}$ subgraph for Case 3. 53
4.14 A sketch of the $K_{3,3}$ subgraph for Case 4 when P-node is proper. 54
4.15 A sketch of the K_5 subgraph for Case 4 when P-node is non proper. 55
4.16 The data structure of `pqtree` 56
4.17 Reducing T_{k-1}. 58
4.18 Bush forms B_{k-1}, corresponding to T_{k-1} of fig. 4.17(a). 58
4.19 T_{k-1} obtained for each of G_1 and G_2. 59

5.1 The deletion of l causes G_p to not be maximal. 70
5.2 Two intersecting near pairs $l, si(l)$ and $l', si(l')$. 71
5.3 Two non intersecting near pairs, part of a consecutive sequence. 73
5.4 Example of top down reduction of near pairs suggested in [Kan92]. 74
5.5 Only one of the potential leaves l_1 and l_2 can be reduced. . . . 75
5.6 MaxPlanarize of [JTS89] does not work when the st-numbering is not legal for G_p. 76

6.1 Pertinent subtree of T_8. 82
6.2 Pertinent subtree of T_8 after Reduce, and bush form B_9. . . . 82
6.3 Situation causing a stray edge to be deleted. 84
6.4 New inner faces. 85
6.5 Q-nodes and boundaries corresponding to bush forms in fig. 6.4. 86
6.6 A new inner face is checked for near pairs in T_{i-1}. 87
6.7 Example of embeddable edge not discovered by the PQ-tree. . 92
6.8 Pertinent Q-node corresponding to biconnected component of fig. 6.7. 92
6.9 Example graph from [Kan92] and [GT94]. 94

A.1 Example of input file 103
A.2 Example of file displayed by vbcTool 104
A.3 File represented in fig. A.2. 105

3.1 PQ-tree definitions. 13

Chapter 1

Introduction

The purpose of this thesis is to explore the possibilities that the PQ-tree data structure of Booth and Lueker [BL76] gives for designing an efficient heuristic for the skewness problem for graphs. The skewness number of a non planar graph G is the least number of edges that must be removed from G to make the resulting subgraph G' planar. Determining the size of this set is $\mathcal{NP}$-hard. The corresponding problem of finding the maximum planar subgraph determined by removing such a minimum set from G is thus $\mathcal{NP}$-hard.

The need for planar subgraphs arise for instance for automatic graph drawing, facility layout, and design of electronic circuits. Thus, heuristics for deleting as few edges as possible to obtain a planar subgraph are of great interest. A related problem, which is in $\mathcal{P}$, is to find a maximal planar subgraph G_p of G, a subgraph that will no longer be planar if any additional edge from G is added to G_p.

At first, the goal of this thesis was to find and extensively test a simple heuristic for finding the skewness number. At that time, the problem of finding a maximal planar subgraph using the PQ-tree data structure was believed to be solved [JTS89, Kan92]. However, [Lei96] shows that there are serious deficiencies with the approaches given in the earlier articles attempting to solve the problem. This result shifted the emphasis of this thesis towards solving the maximal planar subgraph problem correctly. A progress in that direction has been made and is described in Chapter 6.

When this thesis was nearly finished, a new result of [JLM96] gave an explanation as to why the PQ-tree data structure may not lend itself well to solving skewness or maximal planar subgraph problems. Our algorithm in Chapter 6 solves all the problems pointed out in [Lei96], but it does not produce a maximal planar subgraph for all instances due to the problem with the PQ-tree data structure as described in [JLM96].

The rest of this thesis is organized as follows:

Chapter 2 presents some fundamental aspects of graph theory needed to understand this thesis. Different $\mathcal{NP}$-hard graph problems, related to the skewness problem both in theory and in the way they may be heuristicly solved, are also presented.

Chapter 3 describes the PQ-tree data structure and the planarity testing algorithm of [BL76].

Chapter 4 presents some additional data structure proposed in [Kar90], that will maintain more information in the PQ-tree about the embedded subgraph. This data structure will be used for a different purpose in chapter 6.

Chapter 5 gives a survey of the work done by others on the maximal planar subgraph problem based on the PQ-tree data structure.

Chapter 6 proposes a new approach for the maximal planar subgraph problem, a result that can also be viewed as a heuristic for the skewness number. This approach is primarily based on the additional data structure of chapter 4.

Chapter 7 contains concluding remarks as well as some thoughts on how this work can be further developed.

Source code for the implementation of the data structure presented in chapter 4 is included in appendix A.

The text

To make it easier to read and understand the various algorithms, procedures and data structures presented, different text styles have been used. All procedures and algorithms mentioned in this thesis are written in SmallCaps. Class names are written in `typeWriterStyle`, while variables related to procedures or classes are written in `CAPITAL_TYPEWRITER`.

Chapter 2

Graphs and planarity

This chapter is a short introduction to the graph theory needed to give the reader the necessary background for understanding the graph theoretical aspects of this thesis. Some basic notions, definitions, and properties of graphs are given. The chapter ends with a short overview of some $\mathcal{NP}$-hard problems on graphs, among them skewness, the problem which is the main focus of the present work.

2.1 Definitions

A graph $G(V, E)$ consists of a vertex set V and an edge set E. The *order* of G is $|V| = n$ and the *size* is $|E| = m$. An edge is a pair of vertices, drawn as a line between them. Two vertices $u, v \in V$ are *adjacent* if and only if there is an edge $e = \{u, v\} \in E$. A graph has *multiple edges* if there exist edges $e_1, e_2, \ldots, e_i \in E$, $i \geq 2$, where $e_1 = e_2 = \cdots = e_i = \{u, v\}$ for some vertices u, v. An edge $\{u, u\}$ is called a *loop*. See fig. 2.1.

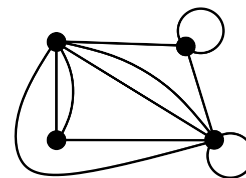


Figure 2.1: Graph with multiple edges and loops.

A graph is *finite* if both vertex- and edge-sets are finite. A graph is *simple* if there are no loops or multiple edges in the graph. Only finite, simple graphs will be considered in this thesis.

If the graph G is *directed*, the edges are ordered pairs, written (u, v) where the direction is from u to v. A directed edge is drawn as an arrow, u being the tail and v the head. (u, v) is an *outgoing* edge of u and an *incoming* edge of v. The *degree* of a vertex is the number of edges adjacent to it. For directed graphs, a vertex has *out-degree* and *in-degree* equal to the number of its outgoing and incoming edges, respectively.

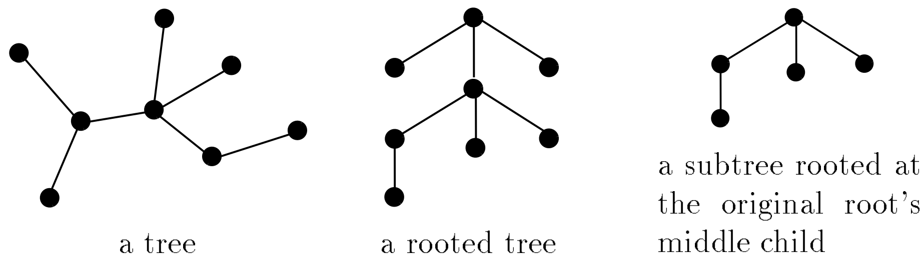


Figure 2.2: Trees.
The two first trees are the same graph, but the latter is rooted.

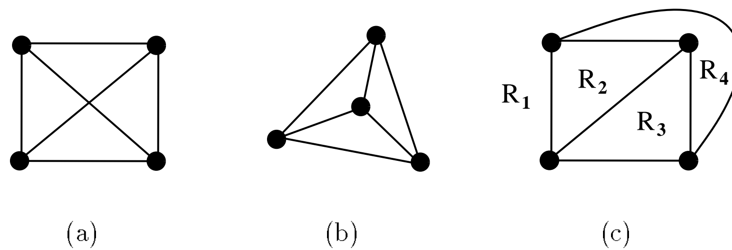


Figure 2.3: Three embeddings of K_4, (c) with the four faces marked.

A *path* from u to v, $\{u, e_1, u_1, e_2, \dots, u_{k-1}, e_k, v\}$, is a sequence of vertices and edges in which all vertices are distinct and all edges $e_i \in E$. For directed graphs, the edges must have the same direction as the path. That is, $e_i = (u_{i-1}, u_i)$, for all $i = \{1, \dots, k\}, u = u_0, v = u_k$. If $v = u$, we have a *cycle*. A graph is *connected* if there is a path between every pair of vertices. A graph is *biconnected* if there are two vertex disjoint paths between every pair of vertices. A vertex is a *cut vertex* if its removal disconnects the graph.

A connected graph is a *tree* if every vertex of degree higher than one is a cut vertex (fig. 2.2). The vertices of a tree are called *nodes*. *Rooted trees* are drawn as the middle tree in fig. 2.2, with the *root* on top. Rooted trees are considered to be directed in direction *from* the root. Thus, the root is the single node in a tree with no incoming edges. Nodes with no outgoing edges are called *leaves*, and nodes with both incoming and outgoing edges are called *internal*. The root is the *ancestor* of all nodes below it, and they are the *descendants* of the root. Leaves are their own descendants. A node, not root, and all its descendants define a *subtree*, with the node as root of the subtree, fig. 2.2. The immediate ancestor and descendants of a node are often referred to as *parent* and *children*. Nodes that are children of the same parent are *siblings*.

A drawing of a graph in the plane is called an *embedding*. Two edges cross in an embedding if they intersect at some point other than a common endpoint, as in fig. 2.3(a). A graph is *planar* if it can be embedded in the plane without edge-crossings. If no such embedding exists, the graph is non-planar. In a planar embedding of a graph, the edges describe *faces*. The four faces in a planar embedding of K_4 are marked in fig. 2.3(c). R_1, the unbounded region, is called the *outer face*, while the other three are *inner faces*.

A graph is *complete* on n vertices, denoted K_n, if there is an edge between every pair of vertices. A *bipartite* graph is a graph $G(V_1 \cup V_2, E)$, with $V_1 \cap V_2 = \varnothing$, where for every edge $\{u, v\} \in E$, $u \in V_1$ and $v \in V_2$ or vice versa. A bipartite graph is complete if every vertex in V_1 is adjacent to every vertex in V_2. If $|V_1| = n_1$, and $|V_2| = n_2$, the graph is denoted K_{n_1,n_2}. Fig. 2.4 shows K_5 and $K_{3,3}$ as examples of complete graphs.

A *subgraph* $H(V', E') \subseteq G(V, E)$ have $V' \subseteq V$ and $E' \subseteq E$. A connected subgraph with exactly $n - 1$ edges is called a *spanning subtree* of G. For a non planar graph G, if H is planar, it is called a *planar subgraph*. If no edge $e \in E - E'$ can be added to H without destroying planarity, it is a *maximal planar subgraph*. The largest of all such subgraphs of G with respect to E', is the *maximum planar subgraph* of G. A subgraph $H(V', E')$ where $V' \subset V$ and E' contains all edges $\{u, v\} \in E$ where $u, v \in V'$, is called an *induced subgraph on* V'. The maximal biconnected subgraphs of a graph is called its *biconnected components* or *blocks*. A biconnected graph has exactly one biconnected component.

A subdivision of an edge $e = \{u, v\}$ is the insertion of a new vertex w on

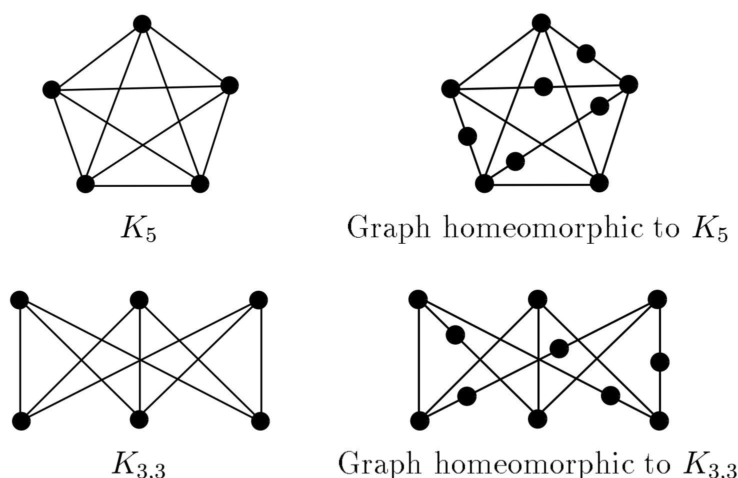


Figure 2.4: K_5 and $K_{3,3}$ with respective homeomorphs.

e, dividing e into $e_1 = \{u, w\}$ and $e_2 = \{w, v\}$. One graph is a *homeomorph* of another, if the first can be obtained from the second by a sequence of subdivisions of edges. Fig. 2.4 also show two graphs homeomorphic to K_5 and $K_{3,3}$.

2.1.1 Properties of planarity and non-planarity

As stated earlier, a planar graph, by definition, has a planar embedding, and any planar embedding shows the number of faces of the graph. Then Euler's theorem on the relationship between the number of vertices n, edges m and faces f of a connected planar graph states:

Theorem 1 $n - m + f = 2$

In a planar embedding of a maximal planar graph, every face must be a triangle. The number of edges in such a graph is thus $m = 3n - 6$, and any graph on n vertices with $m > 3n - 6$ edges is non-planar. For a planar bipartite graph, all regions have four edges surrounding it, so the maximum number of edges in a planar bipartite graph is $2(n_1 + n_2) - 4$. A proof of this theorem can be found in, for instance, [Gri94].

Another famous and important result, this time on non-planarity, was published in 1930 by Kuratowski ([Kur30]). It is known as the Kuratowski Theorem.

Theorem 2 *A graph G is non-planar if and only if there is a subgraph of G which is homeomorphic to either $K_{3,3}$ or K_5.*

This is why K_5 and $K_{3,3}$ often are called *Kuratowski graphs.* K_5 and $K_{3,3}$ are the smallest non planar graphs.

2.2 Degrees of non planarity

There are several $\mathcal{NP}$-hard problems that relate to the degree of non-planarity of a given graph. Definitions and some theoretical results are given for three such problems: skewness, thickness and crossing number. All three can be viewed as *measures of non-planarity* of a graph G.

2.2.1 Skewness

The *skewness number* or just *skewness* $\mu(G)$ of a graph $G(V, E)$ is the minimum number of edges whose removal makes G planar. Thus, if E' is a set of

such edges, $\mu(G) = |E'|$. If E' is given, the resulting subgraph $H(V, E - E')$ is a maximum planar subgraph of G. For G planar, $\mu(G) = 0$ and $H = G$. Finding the Skewness of a graph was shown $\mathcal{NP}$-hard by Liu and Geldmacher in [LG79].

Theoretical bounds

In a complete graph K_n, the number of edges is $\frac{1}{2}n(n-1)$. The maximum planar subgraph of K_n is of course a maximal planar graph on n vertices, containing $3n - 6$ edges. This graph is also triangulated, and has a unique embedding. The skewness number for the complete graphs is thus given by

$$\mu(K_n) = \frac{n(n-1)}{2} - (3n-6) = \frac{(n-3)(n-4)}{2} \quad \text{for } n \geq 3. \tag{2.1}$$

For the complete bipartite graph K_{n_1,n_2}, the number of edges is $n_1 \cdot n_2$. Since the maximum planar subgraph has $2(n_1 + n_2) - 4$ edges, the skewness number for the bipartite graph is

$$\mu(K_{n_1,n_2}) = (n_1 n_2) - (2(n_1 + n_2) - 4) = (n_1 n_2) - 2(n_1 + n_2) + 4 \quad \text{for } n_1, n_2 \geq 2 \tag{2.2}$$

2.2.2 Thickness

The thickness of a graph, $\theta(G)$, is the smallest number of planar subgraphs of G whose union is G. For G planar, $\theta(G) = 1$, since it is its own planar subgraph. For a proof of $\mathcal{NP}$-hardness, see [Man83].

Theoretical bounds

From $m = 3n - 6$ comes the lower bound for the thickness of a graph:

$$\theta(G) \geq \frac{m}{3n-6}$$

For the complete graph, this gives

$$\theta(K_n) \geq \left\lceil \frac{\frac{1}{2}n(n-1)}{3n-6} \right\rceil = \left\lfloor \frac{\frac{1}{2}n(n-1) + 3n - 7}{3n-6} \right\rfloor = \left\lfloor \frac{n+7}{6} \right\rfloor \tag{2.3}$$

It was believed that (2.3) in most cases was an equality. The case was first settled by Beineke and Harary in 1965 for all $n \neq 4 \pmod 6$. Accurate results were found for several values of $n = 4 \pmod 6$ over the next years, the last case of $n = 16$ was settled in 1972 by Mayer [May72].

This gives the equation (2.4) for the thickness of the complete graph.

$$\theta(K_n) = \left\lfloor \frac{n+7}{6} \right\rfloor, \ n \neq 9, 10 \quad \text{and} \quad \theta(K_9) = \theta(K_{10}) = 3 \tag{2.4}$$

(For a more complete historical review and further references, see e.g. [Har69] or [Tho95]).

2.2.3 Crossing number

For G non planar, if embedded in the plane, some of G's edges will cross, since no planar embedding exists. The *crossing number* $\nu(G)$ is the minimum number of such crossings in any possible embedding of G. If the drawing is required to have only straight lines, we get the *rectilinear crossing number* $\overline{\nu}(G)$. Naturally, $\overline{\nu}(G) \geq \nu(G)$. Since any planar graph can be embedded using only straight lines, $\overline{\nu}(G) = \nu(G) = 0$ for G planar.

[GJ83] states that the decision problem

Given a graph G and a natural number k, is $\nu(G) \leq k$?

is $\mathcal{NP}$-complete, which makes the optimization version of crossing number $\mathcal{NP}$-hard.

Theoretical bounds

Crossing number seems to be somewhat harder than the other two. Accurate crossing numbers are known for very few graphs.

$$\nu(K_n) \leq \frac{1}{4} \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \left\lfloor \frac{n-2}{2} \right\rfloor \left\lfloor \frac{n-3}{2} \right\rfloor \tag{2.5}$$

This upper bound on the crossing number has been shown to be an equality for $n \leq 10$ [Guy72].

For the rectilinear crossing number, (2.6) is the best known upper bound for the complete graph [Tho95]:

$$\overline{\nu}(K_n) \leq \left\lfloor \frac{7n^4 - 56n^3 + 128n^2 + 48n\lfloor \frac{n-7}{3} \rfloor + 108)}{432} \right\rfloor \tag{2.6}$$

The crossing number problem were originally given for bipartite graphs (Turán's brick-factory problem [Mut94, Tho95]). Zarankiewicz gave a proof of equality for (2.7) in 1954, but it was later found that he had only proven the upper bound (see [Guy72]).

$$\nu(K_{m,n}) \leq \left\lfloor \frac{m}{2} \right\rfloor \left\lfloor \frac{m-1}{2} \right\rfloor \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor \tag{2.7}$$

(2.7) is still believed to be an equality, but this has so far only been proven for $\min(m,n) \leq 6$ and $K_{7,q}$, where $q \leq 10$. (See [Tho95] and [Mut94] for further history and references).

2.2.4 Some relationships between these problems

The three problems discussed above all try to give a numerical value for how far a non planar graph is from planarity. They are somewhat related in the way they may be heuristicly solved, as a heuristic solution for one problem may provide a heuristic solution for the other.

This section gives a few examples of these relationships. Since all three problems are $\mathcal{NP}$-hard, no exact, polynomial algorithms are known for the general graph. An intuitive heuristic for skewness is proposed, based on a solution to or estimate of the crossing number.

Given a heuristic or accurate algorithm for the maximum planar subgraph problem, a straight forward heuristic for the thickness problem is to repeatedly apply the planar subgraph algorithm and remove this subgraph, until the graph is empty. The thickness estimate is then the number of maximum or maximal planar subgraphs extracted [Cim95]. The connection between the thickness $\theta(G)$ and crossing number $\nu(G)$ is given by the relation $\theta(G) \leq \nu(G) + 1$.

An algorithm for crossing number that also determines the edges involved in each crossing can give a heuristic result for the skewness number: Assume that each edge has received a set of (pointers to) crossing edges by the crossing number algorithm. Edges are then given a priority according to the number of crossings they are involved in. Edges with priority > 0 are put in a priority queue, the rest are left in the planar subgraph. Each edge that is removed from the front of the queue, is counted in the skewness number. When no longer part of the graph, the priority of the edges it crosses with, is decremented by 1. Edges fall out of the queue and into the planar subgraph when their priority reaches zero.

```
procedure SKEWNESS(E)
begin
    SKEW := 0; {SKEW will count the skewness number μ(G)}
    for i = 1 to |E| do
        PRIORITY(e_i) := |CROSSING_EDGES(e_i)|;
        if PRIORITY(e_i) > 0 then place e_i in PRIORITY_QUEUE;
        else put e_i back into graph;
    od;
    while |PRIORITY_QUEUE| > 0 do
        dequeue e;
        SKEW := SKEW + 1;
        for each element d in CROSSING_EDGES(e) do
            PRIORITY(d) := PRIORITY(d) - 1;
            if PRIORITY(d) = 0 then put d back into graph;
            else reposition d in PRIORITY_QUEUE according to new priority;
        od;
    od;
end;
```

The resulting planar subgraph is maximal planar (any removed edge will create a crossing if added to the planar subgraph). Whether it is maximum planar, depends on the embedding used for the crossing number. As Guy states in [Guy72],

> "Almost all questions that one can ask about crossing numbers remain unanswered."

Therefore, no more questions about crossing number will be asked, and this connection between crossing number and skewness is only used to see, one more time, the connection between problems in $\mathcal{NP}$.

Chapter 3

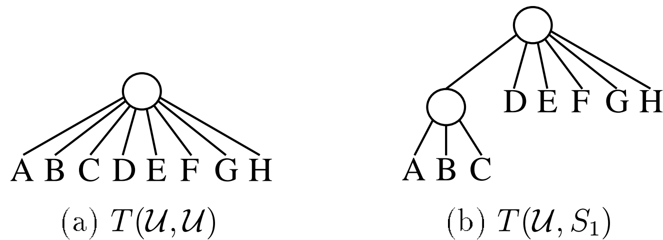
PQ-trees

The PQ-tree data structure, used for planarity testing of graphs, is the basic building block in this thesis. The PQ-tree data structure was designed by Booth and Lueker [BL76] for sorting out permissible permutations of a set, where some subsets have to be consecutive. As will be described in section 3.2, this sorting algorithm is also useful in planarity testing, which were one of the main uses for it described in [BL76]. The other two were tests for *consecutive ones property* and *interval graphs*. Today, the PQ-tree data structure is used in biology, chemistry, graph theory, graph drawing, circuit layout, matrix manipulation, and other areas where certain types of legal permutations are of interest.

We will give a short and intuitive description of the general idea of the PQ-tree data structure, the basic reduction algorithm, and the planarity testing algorithm. For a more complete description of the data structure and reduction algorithm, the reader is referred to [BL76].

3.1 PQ-trees and Permissible Permutations

PQ-trees are rooted trees, whose internal nodes are of two types, called P- and Q-nodes. The leaves of the tree correspond to elements of a set $\mathcal{U}$. The PQ-tree is designed to constrain the permutations of elements of $\mathcal{U}$, revealing the permissible permutations with respect to $\mathcal{S}$, a family of subsets of $\mathcal{U}$. Permissible permutations of the set $\mathcal{U}$ with respect to $S_i \in \mathcal{S}$, are the permutations where no two elements of S_i are separated by an element not belonging to S_i. The PQ-tree ensures this by imposing restrictions on how P- and Q-nodes can rearrange their children.



(a) $T(\mathcal{U},\mathcal{U})$ (b) $T(\mathcal{U},S_1)$

Figure 3.1: $\mathcal{U} = \{A,B,C,D,E,F,G,H\}$, $S_1 = \{A,B,C\}$.

3.1.1 The *PQ*-tree data structure

A *PQ*-tree, representing permutations of a set $\mathcal{U}$, is built step by step. In each step a new restriction, given as a subset $S \in \mathcal{U}$, is introduced, and the tree altered to also represent this restriction. Initially, there are no restrictions, and the *PQ*-tree represents all permutations of $\mathcal{U}$. This is reflected by the tree in fig. 3.1(a), in which the root, drawn as a circle, imposes no order on its children; it is a *P*-node.

When a subset is introduced, the tree must ensure that elements of the subset can permute only among themselves. To represent this, it gathers them under another *P*-node, so that no outsiders may mingle, as in fig. 3.1(b). When additional subsets are introduced (fig. 3.2), they may not be disjoint from or contained in the sets already reduced, thus placing a rather strict ordering on the elements of $\mathcal{U}$. In the tree, this is solved by introducing a new type of node, the *Q*-node, whose children must maintain a strict order. *Q*-nodes are drawn as rectangles, to illustrate that they are more stringent parents than the *P*-nodes.

The restrictions in definitions 3, 4 and 5 of table 3.1 define a *proper* *PQ*-tree, and are made to avoid costly chains and redundancy in the data structure. Reversing and permuting two elements are in essence the same.

The *frontier* of a node in the *PQ*-tree is the sequence of its descendant leaves, read from left to right. The frontier of the root of the *PQ*-tree in fig. 3.2 is ABCFDEGH. Reversing the *Q*-node will give a different frontier, namely ABCFGHED. To find all permutations the *PQ*-tree represents, all rearrangements of children of *P*-nodes and reversals of *Q*-nodes must be taken into account. It is shown in [BL76] that a *PQ*-tree

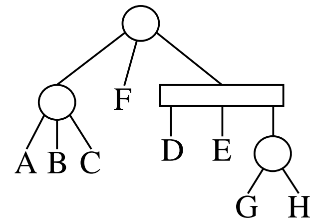


Figure 3.2: Two more restrictions from $\mathcal{S}$ have been added to the *PQ*-tree of fig. 3.1(b), $S_2 = \{E,G,H\}, S_3 = \{E,D\}$.

1	The *universal tree*, fig. 3.1, has all elements of $\mathcal{U}$ as leaves and children of the root, a single P-node.
2	Every element $a \in \mathcal{U}$ is a tree, consisting of one leaf, with itself as the root.
3	Every element of $\mathcal{U}$ appears exactly once, as a leaf, in the PQ-tree representing $\mathcal{U}$.
4	P-nodes have at least two children.
5	Q-nodes have at least tree children.
6	Two trees are identical if and only if one can be obtained from the other by zero or more *equivalence transformations.* There are two equivalence transformations: • P-nodes may permute their children arbitrarily. • Q-nodes may only be flipped over, reversing the order of their children, and always leaving the same two children *endmost*, and the rest *interior.*
7	When a PQ-tree T is restricted by S_i, the tree is said to be *reduced with respect to* S_i, and the new tree is denoted $T(\mathcal{U}, S_i)$.

Table 3.1: PQ-tree definitions.

$T(\mathcal{U}, S)$ represents exactly the permissible permutations of $\mathcal{U}$ where the elements of S appear as a consecutive subsequence. If no such permutations exists, then the reduction of T with respect to S fails and the null tree is returned.

Given a PQ-tree T and a subset S_i, a node in T is *full* if all its descendant leaves are in S_i. If none of them are in S_i, the node is *empty.* An internal node that is neither empty nor full, is *partial* with respect to S_i. Full and partial nodes are called *pertinent.* The smallest subtree of T having all elements of S_i in its frontier is the *pertinent subtree* of T with respect to S_i. The root of this subtree is the *pertinent root*, also called ROOT(T, S_i). In the case of S_4 and the tree in fig. 3.2, the whole tree is the pertinent subtree and the root ROOT(T, S_4). If all empty nodes in the pertinent subtree, and the edges pointing to them, are made invisible, one gets the *pruned subtree*, fig. 3.3. When the full leaves are gathered together by the reduction

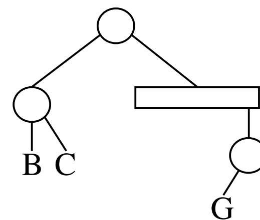


Figure 3.3: The pruned subtree of the tree in fig. 3.2 when $S_4 = \{$B,C,G$\}$.

step, they will form a consecutive sequence in every possible frontier of the tree, as ABC does in fig. 3.1(b). This is called a *pertinent sequence*.

3.1.2 The reduction algorithm

There is only one operation on PQ-trees that alters the tree, namely *reduction with respect to* S, S a subset of $\mathcal{U}$. As stated in the following theorem, proven in [BL76], that is exactly what is needed.

Theorem 3 ("The fundamental theorem of PQ-trees") *Let $S \subseteq \mathcal{U}$ be a subset of $\mathcal{U}$, and let T be a PQ-tree with exactly the elements of $\mathcal{U}$ in its frontier. Let $T(\mathcal{U}, S)$ be T reduced with respect to S. The permutations of $\mathcal{U}$ permissible by $T(\mathcal{U}, S)$ are then exactly those permutations permissible by T in which the elements of S occur consecutively.*

If this set of permutations is empty, then T is said to be *irreducible* with respect to S, and $T(\mathcal{U}, S)$ will be the null tree.

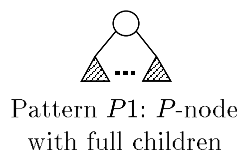
Usually, a set $\mathcal{U}$ and a set of subsets of $\mathcal{U}$, $\mathcal{S}$, are given, and the task is to produce a PQ-tree representing all the restrictions imposed by these subsets. To obtain this, we start by initializing the PQ-tree to $T(\mathcal{U},\mathcal{U})$, and for each $S_i \in \mathcal{S}$, REDUCTION(T, S_i) is called. The final result is a PQ-tree defining the permissible permutations, represented by the null tree if no such permutations exist.

REDUCTION works in two steps, called BUBBLE and REDUCE. BUBBLE builds the pruned subtree with respect to S_i, and then REDUCE groups the leaves together. This rearrangement of the PQ-tree is done by applying one or more *template matchings*.

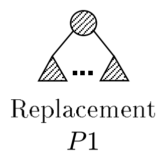
A *template* consists of a pattern, describing a PQ-tree, and a replacement to be made for the pattern if a match is found. There are nine *legal*[1] templates all given in fig. 3.4 on the facing page. For further description, see [BL76]. In addition, there is a template $L1$ for leaves, that marks the leaf `FULL` if it is in S_i, else does nothing. Partial nodes matching $P4$, $P5$, and $Q2$ are referred to as *singly partial*, while partial nodes matching $P6$ and $Q3$ are referred to as *doubly partial* nodes. These templates covers all reducible situations in the PQ-tree, and if none of them apply, the PQ-tree is irreducible.

Both BUBBLE and REDUCE works with a bottom-up strategy, always processing the children before the parent. The bottom-up strategy enables processing of only the necessary parts of the tree, but it also requires that children point to their parent. Since children frequently change parents in the reduction step, maintaining parent pointers for all nodes would be too

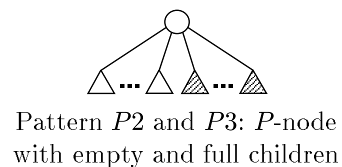
[1] $P0$ and $Q0$ of [BL76] is not counted since they are not used.



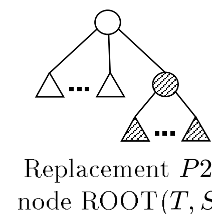
Pattern $P1$: P-node with full children



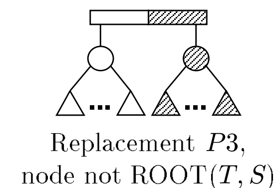
Replacement $P1$



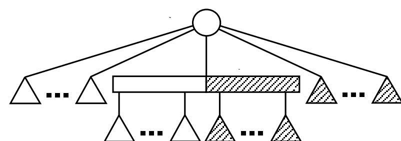
Pattern $P2$ and $P3$: P-node with empty and full children



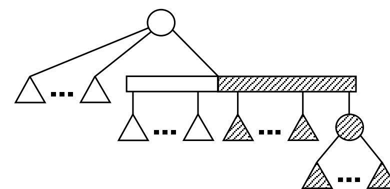
Replacement $P2$, node ROOT(T, S)



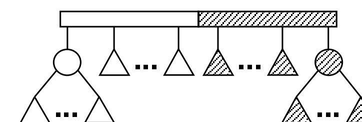
Replacement $P3$, node not ROOT(T, S)



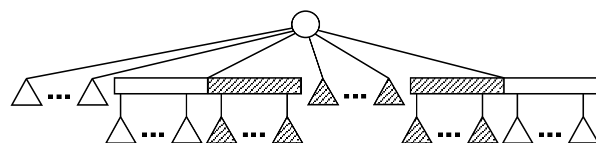
Pattern $P4$ and $P5$: P-node with one partial child



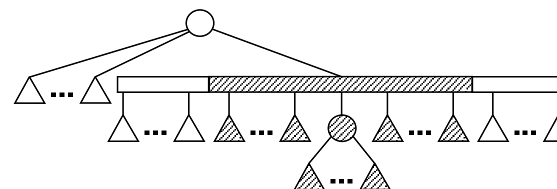
Replacement $P4$, node ROOT(T, S)



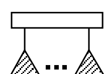
Replacement $P5$, node not ROOT(T, S)



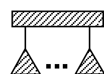
Pattern $P6$: P-node with two partial children



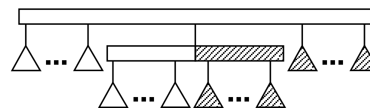
Replacement $P6$, node ROOT(T, S)



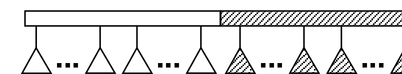
Pattern $Q1$: Q-node with full children



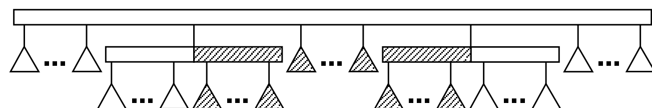
Replacement $Q1$



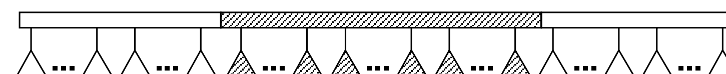
Pattern $Q2$: Singly partial Q-node



Replacement $Q2$



Pattern $Q3$: Doubly partial Q-node



Replacement $Q3$, node ROOT(T, S)

Figure 3.4: Templates

time consuming. Hence only children of P-nodes and endmost children of Q-nodes have *valid* parent pointers between reductions, while interior children of Q-nodes borrow one from their endmost siblings when they become part of a pruned subtree.

Bubble and Blocks

BUBBLE builds the pruned subtree of T with respect to S_i from the leaves upwards, connecting it by giving valid parent pointers to interior children of Q-nodes, and setting a count at each internal node of its number of pertinent children. If BUBBLE reaches the root of T before it has determined the root of the pruned subtree, a flag is set (`OFF_THE_TOP`).

When an interior child of a Q-node is encountered, and none of its immediate siblings have a valid parent pointer, it is *blocked* for the time being. If one or both of the node's siblings are also blocked, they become a *block* of blocked nodes, or a *blocked sequence*. In a later step, when a sibling of a blocked node receives a valid parent pointer from another sibling, the pointer is passed on, and the entire block is *unblocked*. A count of both the number of blocked nodes and of blocks are kept (`BLOCK_COUNT`). BUBBLE is only allowed to leave one block in the PQ-tree, the sequence under the Q-node, root of the pertinent subtree, that matches the pattern of template $Q3$. In this case, a *pseudo node* is used, alluding the doubly partial Q-node. If any other blocked sequences remain, BUBBLE returns $T(\varnothing, \varnothing)$, the null tree.

Procedure BUBBLE can be briefly sketched as follows.

```
procedure BUBBLE(T, S)
begin
    for each leaf X ∈ S do enqueue X;
    while |queue| + BLOCK_COUNT + OFF_THE_TOP > 1 do
        if |queue| = 0 then return T(∅, ∅); {No consecutive sequence is possible}
        else
            dequeue X and mark X BLOCKED;
            if X has valid parent pointer or is adjacent to unblocked sibling then
                mark X UNBLOCKED;
            if X is marked UNBLOCKED then
                if X is ROOT(T, S_i) then OFF_THE_TOP := 1;
                else
                    Y := PARENT;
                    update blocked siblings of X;
                    if Y has not been enqueued earlier then enqueue Y;
                fi;
            else update BLOCK_COUNT;
        fi;
    od;
    if BLOCK_COUNT = 1 then make PSEUDONODE ROOT(T, S_i);
```

```
    return T;
end;
```

Nodes touched by BUBBLE are put on a stack, and reset after each reduction. New nodes created by REDUCE are also put on the stack.

Reduce and Template Matchings

REDUCE carries out the template matching for every node in the pruned subtree, in order to gather the full leaves in one pertinent sequence. Care is taken so that no node is matched until all its pertinent children are matched. The last node matched is ROOT(T, S_i), every node matched before that has only part of the pertinent sequence in its frontier. This is why some templates apply only to ROOT(T, S_i), and others only to nodes *not* root of the pertinent subtree.

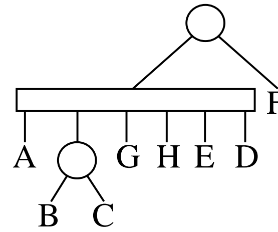


Figure 3.5: Tree of fig. 3.2 after reduction of $S_4 = \{B,C,G\}$.

Template $P3$ creates a non proper tree, since the new Q-node has just two children. This is always corrected by a later template, that either incorporate the node as part of another Q-node, or adds more children to it. If a P-node ends up with only one child after replacement, the child takes its place and the P-node is removed from the tree. Hence the PQ-tree is always proper after REDUCE.

If a node does not match any legal template pattern, then no permissible permutations of $\mathcal{U}$ with respect to $\mathcal{S}$ exists, and $T(\emptyset, \emptyset)$ is returned. Fig. 3.5 shows the PQ-tree of fig. 3.3 after reduction with respect to S_4. Trying to reduce $S_5 = \{B,E,H\}$ results in the null tree, since these elements cannot be made consecutive in any frontier of $T(\mathcal{U}, S_4)$ of fig. 3.5.

Procedure REDUCE can be briefly sketched as follows.

```
procedure REDUCE(T, S)
begin
    for each leaf X ∈ S do enqueue X;
    while |queue|>0 do
        dequeue X;
        if X is ROOT(T, S_i) then
            if some template for ROOT(T, S_i) applies to X then
                substitute the replacement for X in T;
            else return T(∅, ∅);
        else {X is not ROOT(T, S_i)}
            if some template for nodes not ROOT(T, S_i) applies to X then
                substitute the replacement for X in T;
```

```
            else return T(∅, ∅);
        fi;
        if ROOT(T, S_i) is reached then return T;
        else if every pertinent sibling of X has been matched then
            enqueue the parent of X;
    od;
end;
```

Some templates apply only to nodes ROOT(T, S_i), others only to nodes not ROOT(T, S_i), while some apply to both kinds of nodes. This divides the templates into two not disjoint sets. Templates are called in a special order by REDUCE, the lexicographic order within each set, so that the templates have implicit knowledge of the node being matched. For instance, a P-node reaching $P4$ must have at least one partial child, since otherwise it would have been matched and handled by $P1$ or $P2$.

3.2 Planarity testing with PQ-trees

Since planarity can be characterized by the existence of a planar embedding, a natural way to check for planarity, is to search for such an embedding. If no planar embedding can be found, the graph is not planar. This method underlies both main types or approaches for planarity testing algorithms, *path-addition* (also called *edge-addition*) and *vertex-addition*. Although no embedding is explicitly exhibited by either approach, all information needed to do so is present during the course of the algorithm ([HT74, BL76, CNAO85, Joh98]).

The first linear planarity testing algorithm was based on path addition and is due to Hopcroft and Tarjan [HT74]. The first planarity testing algorithm based on the vertex addition approach that reached the linear time bound, is due to Booth and Lueker [BL76], and is built on the algorithm given in [LEC67]. The idea of Lempel, Even and Cederbaum was that when trying to find a planar embedding of a graph, an ordering, or permutation, of the edges around each vertex has to be established, ensuring that no edges will cross. Planarity is verified for the induced subgraph on vertices 1 through k, and then checked for the incoming edges of vertex $k + 1$.

For this algorithm, the graph is assumed to be *st-numbered*. The concept of st-numbering is due to Lempel, Even and Cederbaum. They proved in [LEC67] that a graph can be given an st-numbering if and only if it is biconnected.

Definition 3.1 ([ET76]) *Given any edge (s, t) in a biconnected graph G with n vertices, the vertices of G can be numbered from 1 to n so that vertex*

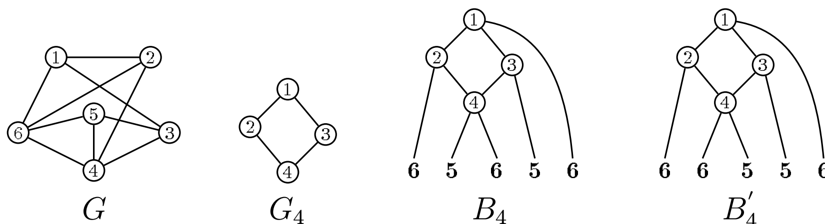


Figure 3.6: Graph G, embedded subgraph G_4, and bush form B_4. B_4' have all virtual vertices labeled 5 consecutive.

s receives number 1 and vertex t number n, and any vertex except s and t is adjacent both to a lower-numbered and to a higher-numbered vertex. Such a numbering is called an st-**numbering** *of G.*

The next section explains this concept further, and gives an example of how a graph is given an st-numbering, using the linear algorithm of [ET76].

The cornerstone of the approach of [LEC67] is the *bush form.*

Definition 3.2 *The induced subgraph on vertices 1 through i of G is G_i, $1 \leq i \leq n$. The* **bush form** *B_i is defined as follows: G_i is the embedded vertices and edges of B_i. In addition, B_i consists of all edges (j, k) of G, where $1 \leq j \leq i$, and $i < k \leq n$. These edges are called* **virtual edges**, *and their head vertices are called* **virtual vertices**. *Virtual edges and vertices are not yet embedded. See fig. 3.6.*

Virtual vertices have only one entering edge each, so there may be several virtual vertices with the same label. Bush forms are drawn with all virtual vertices on the outer face, lined up below G_i. Since a bush form grows step by step downwards from the first embedded vertex, vertex 1 will always be on top. Vertex $i + 1$ is added to B_i by making all virtual vertices labeled $i + 1$ consecutive. This is done by rearranging outgoing edges of cut vertices and flipping biconnected components.

Lempel, Even, and Cederbaum showed in [LEC67] that an st-numbered graph G is planar if and only if for every B_i, $2 \leq i \leq n - 2$, there exists a planar drawing of B_i such that all virtual vertices labeled $i + 1$ appear as a consecutive sequence. This results in the following lemma, shown by [Eve79].

Lemma 1 *Let $G = (V, E)$ be a planar, biconnected graph with an st-numbering, and let $1 \leq i \leq n$. If G is embedded such that both s and t lie on the outer face, then all vertices and edges of $G - G_i$ are drawn in the outer face of the induced subgraph G_i of G.*

Lempel, Even, and Cederbaum used formulas to keep track of legal permutations of the virtual vertices of the bush form, and did not achieve a linear time bound. With the linear st-numbering algorithm of [ET76], Booth and Lueker [BL76] achieved a linear bound on the vertex-addition algorithm using their novel data structure PQ-trees. How PQ-trees and bush forms correspond is explained in section 3.2.2.

Planarity is not a graph property dependent upon whether the graph is connected or biconnected, directed or undirected, simple or with multiple edges. A disconnected graph is planar if and only if its connected components are planar, and a connected graph is planar if and only if its biconnected components are planar. Multiple edges can be embedded in parallel. Thus any graph can be tested for planarity, by testing the underlying simple, undirected graph, one biconnected component at a time. The biconnected components can be found by a slight modification to the depth first search procedure of the st-numbering algorithm [Eve79].

We will here give a short description of the basic elements of the st-numbering algorithm. Graphs considered will be finite, simple, undirected, and biconnected.

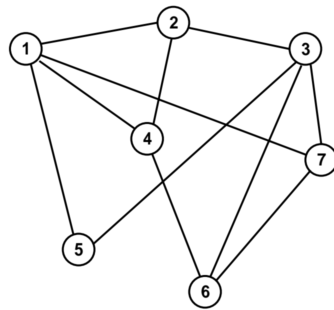
3.2.1 st-numbering

The purpose of the st-numbering is to order the vertices by numbering, in such a way that when checking the graph for planarity, the algorithm will not "run dry". That is, when finishing one vertex, there will always be outgoing edges to higher-numbered vertices, and all vertices, except the first one, will always have incoming edges from lower-numbered vertices. Because of this, the output graph of the st-numbering algorithm will in essence be treated as a directed graph, with edges directed from lower-numbered to higher-numbered vertices. This gives a network with one source (node s), one sink (node t), and no cycles.

Given a biconnected graph, this is achieved by selecting an edge (s, t) and giving vertex s number 1 and vertex t number n. Numbering of the remaining vertices is done such that each vertex is adjacent both to a lower numbered and to a higher numbered vertex. If edges then are imagined to have direction from lower to higher numbered vertices, we get the network described above.

In [ET76], Even and Tarjan gave a linear (that is, $O(n + m)$) algorithm for giving an st-numbering to a biconnected graph. Here we give a brief introduction to the algorithm with an example, the reader is referred to [ET76] or [Eve79] for further details.

The algorithm takes a biconnected graph G as input, and produces as



Vertex no:	Adjacent to vertices:
1	2 4 5 7
2	1 3 4
3	2 5 6 7
4	1 2 6
5	1 3
6	3 4 7
7	1 3 6

Figure 3.7: A biconnected graph. Left: Drawing. Right: Adjacency list.

output an st-numbering for G. A biconnected graph is given in fig. 3.7. The adjacency lists in the right column is the form of both input and output of the algorithm, with an additional line first, containing n, the number of vertices.

First, a depth first search (DFS) is carried out, in order to find a spanning subtree of G and to give the vertices a temporary, preorder numbering according to the DFS (fig. 3.8). The depth first search start in vertex t, with edge (t, s). Edges in the spanning subtree are called *tree edges* (with an imagined direction *from* the root), the rest are called *back edges* (with direction *towards* the root).

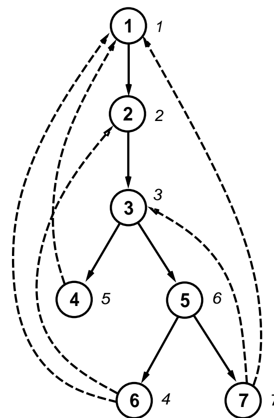


Figure 3.8: Graph after DFS, vertices with preorder numbering, original numbers in italics.

The second part of the algorithm is the PATHFINDER procedure. It investigates every path between two given vertices, marking the edges and vertices *old* as they become part of a path. When all paths have been investigated, an empty path is returned. The initial path is $\{t, (t, s), s\}$.

The final part is STNUMBER, that uses PATHFINDER to give the vertices st-numbers. The vertices of the path returned are placed on a stack, the last one on top. The edge between the two topmost vertices are then extended to a path by PATHFINDER. When an empty path is received, the top vertex on the stack is given the next st-number. The vertex to receive number 1 is s. t will stay the bottom element of the stack throughout, and receive number n. Fig. 3.9 gives the resulting st-numbering of the graph of figs. 3.7 and 3.8.

All graphs considered for the rest of this thesis are assumed to be st-

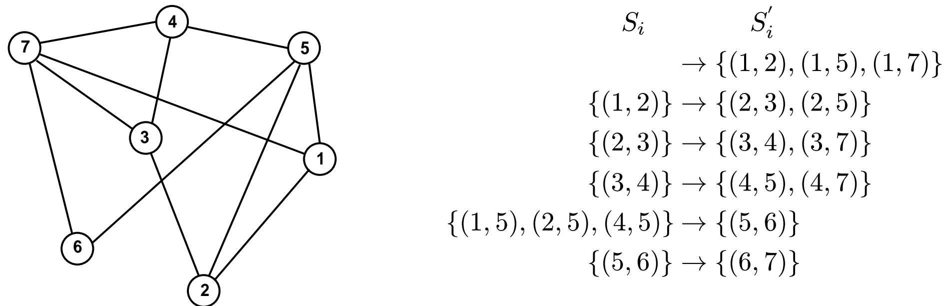


Figure 3.9: *st*-numbered graph on the left. Outgoing and incoming edge sets on the right, used by PLANAR.

numbered, and all references to *vertex number* refer to the *st*-number of each vertex.

3.2.2 PLANAR

When operating on the vertices and edges of a graph in order to check planarity, the "universal" set $\mathcal{U}$ changes for each vertex. In the beginning, $\mathcal{U}_1$ consists of the outgoing edges of vertex 1 in G, S_1'. Then for each vertex i, $i = \{2, \ldots, n-1\}$, the tree is reduced with respect to S_i, the set of incoming edges of i. In the ith step, after the reduction of T_{i-1}, the new vertex i is added to the tree, by exchanging the consecutive sequence of elements from S_i with the set of outgoing edges from i, S_i', thus creating T_i. $\mathcal{U}_i$ is updated accordingly.

The *PQ*-trees $T_1, T_2, \ldots, T_{n-1}$ built by PLANAR represent exactly the bush forms $B_1, B_2, \ldots, B_{n-1}$ of [LEC67]. Leaves are virtual edges, *P*-nodes are cut vertices, and *Q*-nodes are biconnected components. This is exemplified in fig. 3.10. A bush form B_i with all vertices labeled $i+1$ as a consecutive sequence exists if and only if the *PQ*-tree T_i can be reduced such that all its pertinent leaves appear as a consecutive sequence in the frontier of all permutations of T_i [BL76]. T_{n-1} does not need to be reduced, all its leaves represent incoming edges to vertex n.

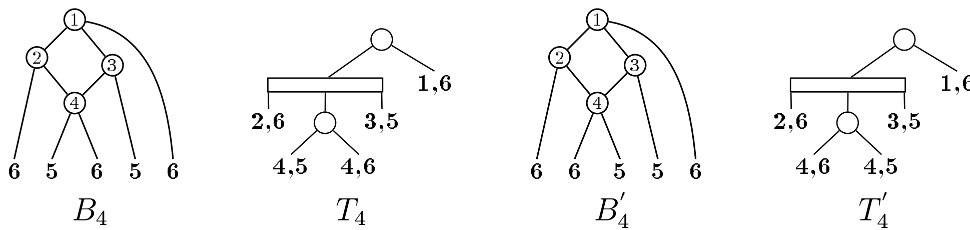


Figure 3.10: The correspondence between a PQ-tree and its bush form. B_4 and B_4' are from fig. 3.6. T_4 and T_4' are equivalent PQ-trees.

```
boolean procedure PLANAR(V, E)
begin
    U_1 := the set of edges whose lower-numbered vertex is 1;
    T_1 := T(U_1, U_1);
    for i := 2 to n − 1 do
        {Reduction step}
        S_i := the set of edges whose higher numbered vertex is i;
        T_{i−1} := BUBBLE(T_{i−1}, S_i);
        T_{i−1} := REDUCE(T_{i−1}, S_i);
        if T_{i−1} = T(∅, ∅) then return FALSE;
        else
            {Vertex addition step}
            S'_i := the set of edges whose lower-numbered vertex is i;
            if ROOT(T_{i−1}, S_i) is a partial Q-node then
                replace the full children of ROOT(T_{i−1}, S_i) and their descendants
                    by T(S'_i, S'_i);
            else replace ROOT(T_{i−1}, S_i) and its descendants by T(S'_i, S'_i);
            U_i := U_i − S_i ∪ S'_i;
        fi;
    od;
    return TRUE;
end;
```

If $\mathrm{ROOT}(T_{i-1}, S_i)$ after reduction is a Q-node with one empty child, the vertex addition step will leave it with only two children. To keep the tree proper, the non proper Q-node is replaced by a P-node.

The table of fig. 3.9 on the preceding page shows the sets S_i and S_i' of the incoming and outgoing edges of vertex i for our example graph. Here $\mathcal{U}_1 = S_1' = \{(1,2),(1,5),(1,7)\}$ and $S_2 = \{(1,2)\}$. After the first reduction, S_2 is removed from $\mathcal{U}_1$, and $S_2' = \{(2,3),(2,5)\}$, the outgoing edges of vertex 2, is added, so that in the second step $\mathcal{U}_2 = \{(1,5),(1,7),(2,3),(2,5)\}$.

The trees are numbered accordingly: $T_1 = T(\mathcal{U}_1, \mathcal{U}_1)$, and then the subscript changes with $\mathcal{U}$'s subscript (fig. 3.11), so that T_{i-1} is reduced in step i, creating T_i.

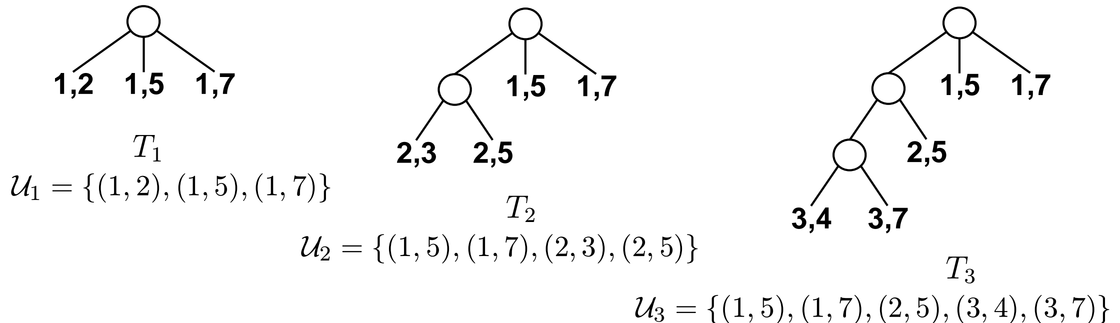


Figure 3.11: The initial tree, and the result of the first two iterations of the main loop of PLANAR, $i = 2, 3$.

The remaining *PQ*-trees and some corresponding bush forms from running the example graph of figs. 3.9 and 3.11 through PLANAR, are shown in fig. 3.12.

Correctness and complexity

This planarity testing algorithm is merely an efficient implementation of the planarity testing algorithm presented in [LEC67]. A proof of the correctness can be found there. Otherwise the algorithm is quite intuitive, and the reader should be able to convince himself that the algorithm works.

The *st*-numbering algorithm of [ET76] is $\mathcal{O}(n+m)$. The total number of nodes handled by all calls to BUBBLE is $\mathcal{O}(n+m)$. The amount of work done for each node is bounded by a constant. The number of *P*-nodes in a *PQ*-tree T_i is at most i, the number of *Q*-nodes is also at most i, and the number of pertinent leaves is $|S_{i+1}|$. Summing up these numbers for all T_i's gives $\mathcal{O}(n+m)$. The time bound for REDUCE is also $\mathcal{O}(n+m)$, following the same arguments as for BUBBLE. Initialization and the vertex addition step takes $\mathcal{O}(n+m)$ time as well, rendering a total time complexity of $\mathcal{O}(n+m)$. Since PLANAR only needs to be called for graphs with $m \leq 3n-6$ (see section 2.1.1), the algorithm is $\mathcal{O}(n)$, linear in the number vertices of G.

3.3 Implementation

PQ-trees are a tool, a basis for further work in this thesis. Therefore, an existing implementation were sought. The choice fell on an implementation provided by Sebastian Leipert ([Lei97]), now also a LEDA-Extension Package (`http://www.mpi-sb.mpg.de/LEDA/`). The main reasons for choosing

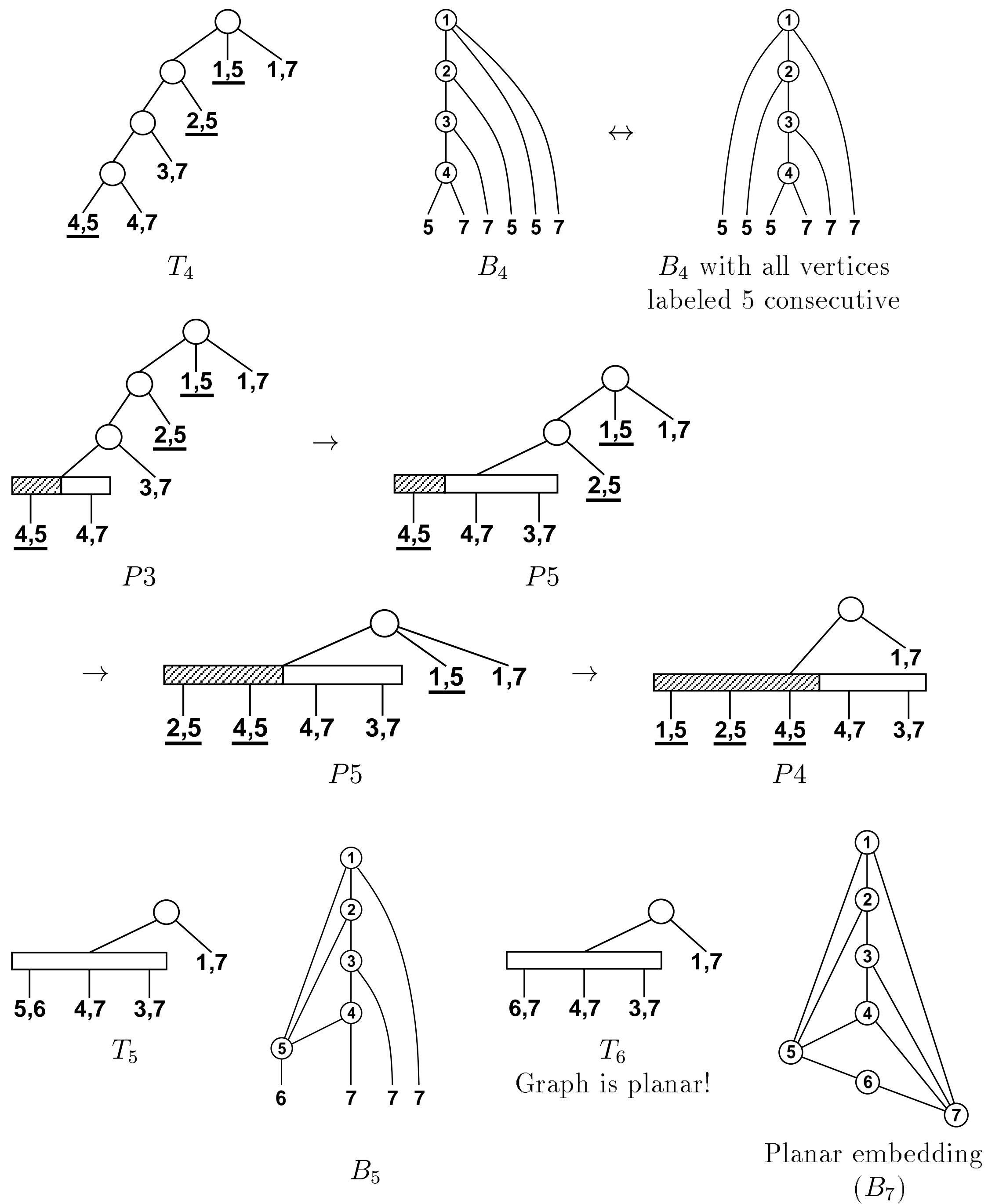


Figure 3.12: Running of the example graph through PLANAR, $i = 4, 5, 6$. Elements of S_4, matched by $L1$, are marked with a ___ underneath, pertinent internal nodes are marked by shading. The two middle rows show the result of all non trivial matchings done by REDUCE. The template name is stated for each.

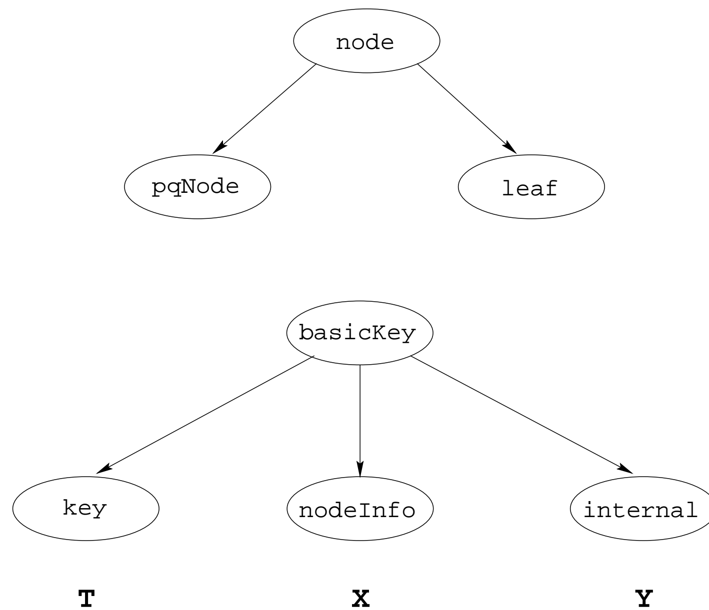


Figure 3.13: Class structure of Leipert's implementation.
Top figure show the class structure of the nodes in the PQ-tree, the bottom is the information keepers. Each information keeper has a pointer of the specified argument type.

Leipert's implementation, was the idea behind it: an implementation that provides the features necessary to extend it to other purposes, and the extensive documentation provided with it ([Lei97]).

Leipert has implemented the general PQ-tree data structure of [BL76] as a template class in C++. Readers not familiar with C++ and templates are referred to [Str97]. The specification as a template class enables the user to specify any type of elements of $\mathcal{U}$ to be handled by the PQ-tree.

3.3.1 S. Leipert's implementation of PQ-trees

The PQ-tree template has three template arguments, `T`, `X`, and `Y`. All classes of the PQ-tree implementation are parameterized, with the types `T`, `X`, and `Y` as their parameters.

The class structure for the classes in the PQ-tree is shown in fig. 3.13. Leipert uses the two subclasses of `node`, `pqNode` and `leaf`, to build the PQ-tree. All internal nodes are of one type, `pqNode`. The only difference between P- and Q-nodes is the values of certain fields and how the child list is accessed. This means that children are linked in a doubly linked list both

for P- and Q-nodes, but for endmost children of Q-nodes one sibling pointer is nil.

`pqNode` and `leaf` are only concerned with their place in the tree, they leave all information handling to the subclasses of `basicKey`. Type `T` is essential for the PQ-tree template to work. It is the type of the elements of $\mathcal{U}$. Every `leaf` object in the PQ-tree must be associated with an element of $\mathcal{U}$. The maintenance of this pointer is fully taken care of by the implementation, it is part of the general structure of PQ-trees. Thus, class `leaf` has a pointer of type `key` and class `key` has a pointer of type `T`. The implementation ensures that each `key` object is assigned to exactly one `leaf` object, and that every element of $\mathcal{U}$ is assigned to exactly one `key`.

Additional information, specific either to P- and Q-nodes or of general interest to all nodes in the PQ-tree, can also be included. Class `node` has a pointer of type `nodeInfo`, and `nodeInfo` has a pointer of type `X`. Thus `nodeInfo` is meant to hold information that may concern all nodes, while information specific to internal nodes should be made type `Y`. Class `pqNode` has a pointer to `internal`, that has a pointer of type `Y`. To include information of type `X` and `Y` in the tree, the user must make a derived class of `PQTree` to maintain these pointers.

3.3.2 Implementation of Planar

For the implementation of Planar, we made a derived subclass `pqtree` of the template class `PQTree`. This work was done jointly with J.-E. Bye Johansen, who also provided an implementation of the st-numbering algorithm of section 3.2.1. A description of this implementation can be found in [Joh98].

A procedure `Planar`, as well as a procedure `readStNumbering` for reading st-numbered graphs from file, was added to the class `pqtree`. The edge sets S_i and S_i', necessary for procedure `Planar`, are built.

Since the possibility to handle self-made types was present, and mapping the edges to integers makes output from the PQ-tree cumbersome to read, a new type, class `edge` was designed. Each `edge` object carries the number of its head and tail vertex, making it easier to give readable output from the program. As long as a mapping number is one of the attributes of class `edge`, included for handling edges outside of `Planar`, it is quite possible to implement Planar by using integer as template type `T` instead of edge (see [Joh98]). In this version, however, edge was used.

All code files are found in appendix A.

Chapter 4

PQ-trees and obstructions to planarity

In chapter 3, a planarity testing algorithm using the PQ-tree data structure was presented. Both this algorithm, the vertex addition algorithm of [LEC67] and [BL76], and the path addition algorithm of [HT74] have been used for other purposes than planarity testing ([CHT93, Cim95, JTS89, Kan92, CNAO85, Kar90]). This chapter will concentrate on one of these articles, [Kar90], where a linear algorithm for explicitly identifying obstructions to planarity is given. This article formed the basis for our search for a new approach to a skewness heuristic.

Given a non planar graph G, then a subgraph G' of G, where G' is homeomorphic to $K_{3,3}$ or K_5, is an "obstruction to planarity". G can be made closer to planarity by removing such an obstruction. To remove an obstruction, deleting one edge from the subgraph is enough. In [Kar90], Karabeg uses PLANAR, the planarity test of section 3.2.2, with some additional data structure added to the PQ-tree, to detect obstructions to planarity when PLANAR fails.

A challenging part of designing a skewness heuristic using PQ-trees, is how to add to the existing data structure so that more information can be handled by PQ-trees, while preserving efficiency. The data structure of [Kar90] proved to provide information useful for determining a minimal set of edges to remove from a planar graph. Accordingly, sections 4.1 and 4.2 are devoted to this new data structure that enables obstructions to planarity to be found by the PQ-tree. Section 4.3 presents how this additional data structure was used in [Kar90], while section 4.4 discusses to what extent the contents of this chapter can be used for a skewness heuristic.

4.1 Additions to the PQ-tree

During PLANAR, the planarity testing algorithm described in the previous chapter, a series of PQ-trees $T_1, T_2, \ldots, T_{n-1}$ are constructed, corresponding to the bush forms defined on page 19. A PQ-tree T_i only carries information about $\mathcal{U}_i$, the virtual edges of the bush form, and how they are connected to the embedded subgraph G_i. The structure of this subgraph is to some extent represented by the internal nodes in the tree. P-nodes represent cut vertices on the outer face of the bush form, as long as their children can be rearranged. Biconnected components are represented by Q-nodes, as long as they have more than two vertices with outgoing edges. Thus, only structures that can cause virtual vertices to change place are represented, and no information about the vertices and edges that these structures consist of, is available.

4.1.1 Additional information

When a Kuratowski subgraph is detected, some of it has to have been embedded already, so in order to exhibit its vertices and edges, additional information about structures in the embedded subgraph needs to be present. An essential part of this information is the *vertex number*, an integer representing the st-number of each vertex, and head and tail vertex of each edge. Vertices of degree two build *chains* that must be kept track of. For biconnected components, the *outer mesh* is necessary.

Definition 4.1 *An* **outer mesh** *is the cycle made up of the edges and vertices of the unbounded region in a planar embedding of a biconnected graph.*

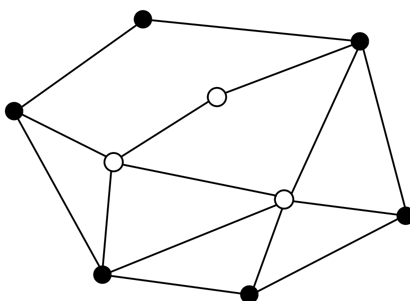
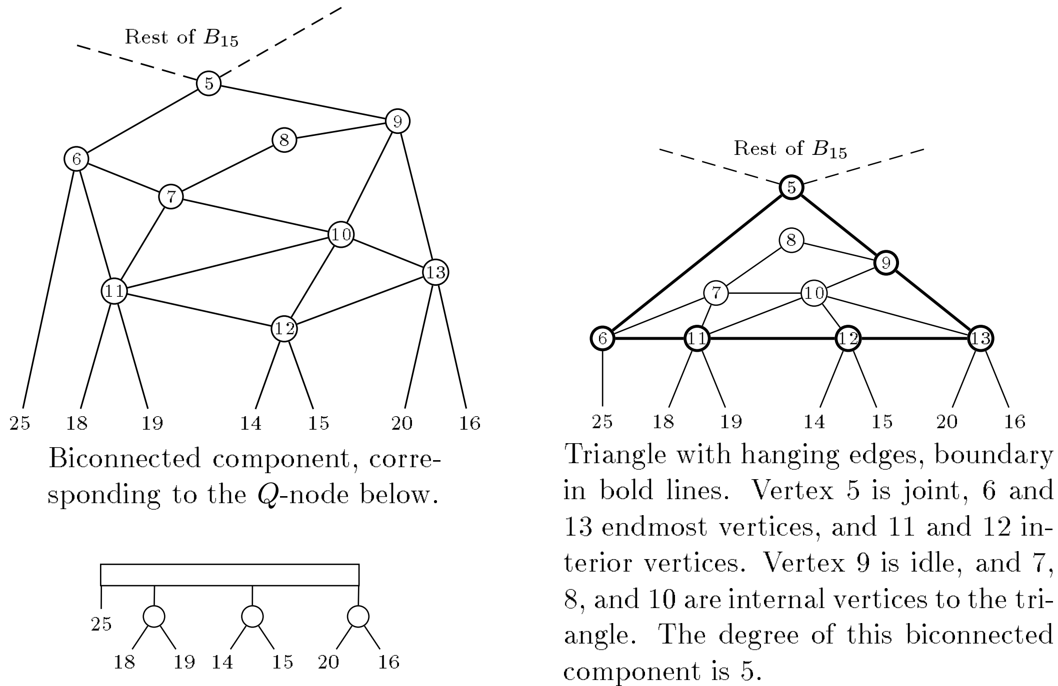


Figure 4.1: Outer mesh of a biconnected graph
The filled vertices determine the outer mesh of the graph. The open vertices are *internal*.

Definition 4.2 *Outgoing edges of vertices on the outer mesh of a biconnected component, not embedded in the same component, are called* **hanging edges***.*

Definition 4.3 *A biconnected component in a bush form, attached to the rest of the subgraph at a single vertex, is called a* **triangle***. The vertex of attachment is called the* **joint***. The outer mesh of a triangle is called the* **boundary** *of the triangle. The first two vertices with hanging edges encountered when walking along the boundary in both directions, are* **endmost** *vertices. Other vertices with hanging edges, between endmost vertices, are* **interior** *vertices. Vertices on the boundary without hanging edges are* **idle***.*



Biconnected component, corresponding to the Q-node below.

Triangle with hanging edges, boundary in bold lines. Vertex 5 is joint, 6 and 13 endmost vertices, and 11 and 12 interior vertices. Vertex 9 is idle, and 7, 8, and 10 are internal vertices to the triangle. The degree of this biconnected component is 5.

Figure 4.2: Biconnected component and boundary of triangle.

Definition 4.4 *The* **degree of a biconnected component** *is the number of vertices with edges entering or leaving the component, in other words, the number of vertices with hanging edges plus one for the joint.*

Fig. 4.2 shows examples of these concepts.

4.1.2 Additional data structure

All explicit information present in a PQ-tree is in the leaves, holding the elements of $\mathcal{U}$. When working with graphs, this information is in the virtual edges of the corresponding bush form. The embedded subgraph of the bush form is represented by P- and Q-nodes, corresponding to biconnected components and cut vertices. Accordingly, information about structures in the subgraph must be held by internal nodes of the PQ-tree. The additional data structure in the expanded PQ-tree is described in the following. The same example graph G is used in figs. 4.3–4.6.

Vertex number

The embedded subgraph consists of vertices and edges. As mentioned earlier, the number of each vertex is essential information. The set $S_i^{'}$ is the outgoing edges of vertex i, introduced to the tree T_i in iteration i. When $|S_i^{'}| > 1$, a new P-node will be parent of the new leaves and receive i as its `vertexNumber`. All P-nodes are introduced to the PQ-tree this way.If $|S_i^{'}| = 1$, the tail vertex of the single edge will be i, and so the number is present in the PQ-tree when needed. Numbers inside P-nodes in the PQ-trees of fig. 4.3 are `vertexNumbers`.

Chain

Chains of vertices of degree two are not visible in the PQ-tree, and biconnected components of degree two meet the same fate (fig. 4.3). `Chains` are built when vertices of degree two are embedded, that is, when $|S_i| = |S_i^{'}| = 1$ (T_4 and T_6 of fig. 4.3), or when all hanging edges from a biconnected component are tied up in the new vertex, leaving a component of degree two (T_5 of

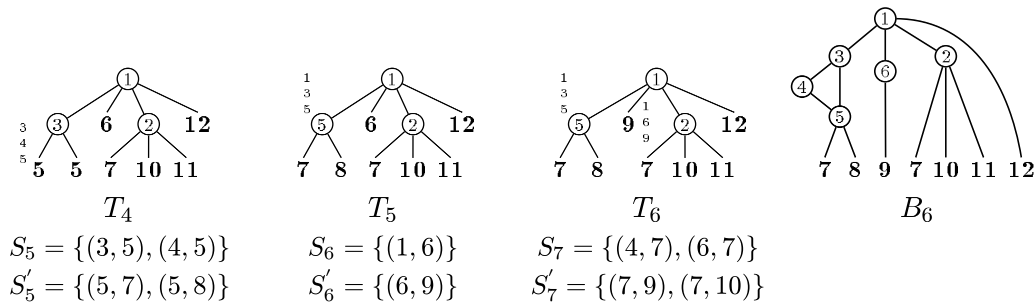


Figure 4.3: Building chains in the PQ-tree. T_4, T_5, and T_6.
`Chains` are represented as upside down stacks of vertex numbers.

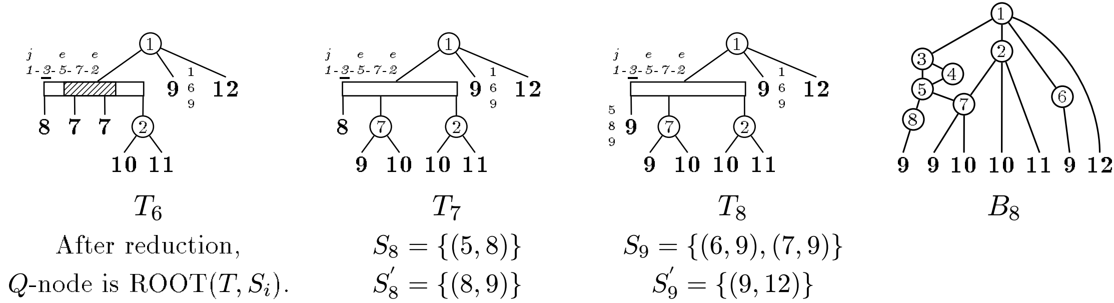


Figure 4.4: `Boundary` of Q-nodes. T_6, T_7, and T_8.
A biconnected component with boundary has been constructed. A chain is part of the boundary, creating an idle element (*3*). Boundaries are represented as a list of vertex numbers, joint and endmost elements marked.

fig. 4.3). The vertices connecting the chain to the rest of the subgraph is also part of the `Chain`.

In the PQ-tree, this is the case when $P1$ or $Q1$, and sometimes $P2$, matches ROOT(T, S_i), as in T_4 of fig. 4.3. Biconnected components of degree two are represented by a simple path from the joint along the boundary to the single vertex with hanging edges. This component may later become part of the outer mesh of another biconnected component, but it will still be free to flip so that the side represented by the `Chain` can be on the boundary (B_8 of fig 4.4).

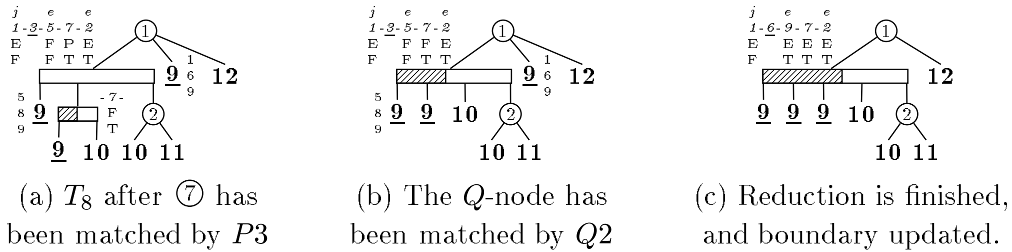
Since parents in the PQ-tree have several children, while children have only one parent, a `Chain` is held by the child. This ensures that children of Q-nodes always have the number of the vertex on the boundary to which they are connected, and are able to update their parents' boundary when the extended PQ-tree is traversed bottom up.

Boundary

Q-nodes represent triangles, biconnected components in the embedded subgraph, and thus they have a `Boundary`.

A Q-node left with only two children after the vertex addition step, is replaced by a P-node. Such a P-node, that represents a biconnected component of degree three in the bush form, is a *non proper* P-node, and inherits the Q-node's `Boundary` (T_{10} of fig. 4.6). Accordingly, only proper P-nodes will have `vertexNumber`, while non proper P-nodes and Q-nodes have `Boundary`.

All Q-nodes start out as a non proper Q-node, a Q-node with only two children, created by template $P3$. (See fig. 3.4 on page 15.) This new Q-



(a) T_8 after ⑦ has been matched by $P3$

(b) The Q-node has been matched by $Q2$

(c) Reduction is finished, and boundary updated.

Figure 4.5: The reduction of T_8 and T_9.
The third row of the boundary is the `STATUS` of each element, E = `EMPTY`, F = `FULL`, P = `PARTIAL`. The second row is their `CHILD` field, telling if the corresponding node has empty children. F = `FALSE`, T = `TRUE`.

node will grow into a proper Q-node before the last step of the reduction. A biconnected component with an outer mesh is created when the new vertex is added, and the boundary comes into existence. (Figs. 4.4 and 4.5.) In the PQ-tree however, in order to have the required information about the new `Boundary` at this point, the `Boundary` of a Q-node is built bit by bit by each template that handles it. When $\text{ROOT}(T, S_i)$ is reached, the boundary can be completed, since all necessary information is available (fig. 4.5 and T_6 of fig. 4.4).

`Boundary` of a node X, denoted `Boundary(`X`)`, consists of elements in a circular, doubly linked list, one element for each vertex on the outer mesh of the triangle. Each element has a `NUMBER` equal to the number of the vertex it represents. Track is kept by `Boundary` of the joint and two endmost elements, representing the corners of the triangle. To be able to distinguish between the two endmost elements, and find interior elements, the joint is defined as the “beginning and end” of the circular list, its left pointer always leading to the left endmost first (`END1`), and the right pointer to the right endmost first (`END2`). This doubly linked list is represented as a simple list in figures, with the joint as the leftmost element, see for instance figs. 4.4 and 4.5.

Each element has a `STATUS` that equals the status of the node representing the vertex in the PQ-tree. Idle elements will have status `EMPTY`. In addition the element has a field `CHILD`, with the value `TRUE` as long as the corresponding node has empty children, and value `FALSE` if that node has only full children or if the corresponding vertex is idle already. These fields are shown for non idle elements in the reduction of T_8 in fig. 4.5. `JOINT` and endmost elements, as well as idle elements, are marked for all boundaries in figs. 4.4, 4.5, and 4.6. Elements are also capable of holding a `Chain`. For instance, the `Chain` 5 8 9 in (a) and (b) of fig. 4.5 has actually been passed on to element

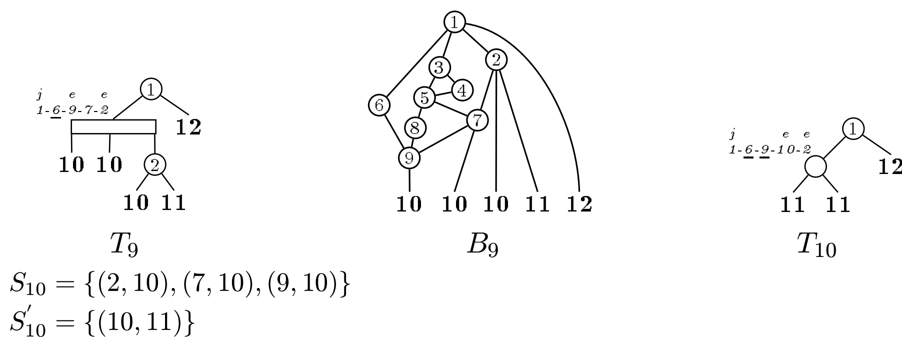


Figure 4.6: Reduction of T_9.
The actual reduction is not shown. The `Boundary` of the Q-node, non proper after vertex addition, is passed on to the non proper P-node of T_{10}.

5 by template $L1$. This fact is not important, and for simplicity, this will not be reflected in the figures. On the other hand, when such `Chains` obviously will be internal to a biconnected component when the new vertex is added, as with 5 8 9 in (c) of fig. 4.5, they will be removed, since the corresponding element will be thrown away from `Boundary`.

A `Boundary` that is part of a pruned subtree has a pertinent sequence. The first pertinent element encountered walking left to right along the boundary from `JOINT`, is denoted `BEGFULL`, and the last one `ENDFULL`. These can be endmost or interior. The last element encountered before `BEGFULL` is denoted `ENDEMPTY`, and `BEGEMPTY` is the first element encountered after `ENDFULL`. These can be any element in `Boundary`, including idle elements and `JOINT`.

Elements on the boundary are updated when their children are matched. In fig. 4.5(a), all children of the Q-node have been matched, and the pertinent sequence consists of elements 5 and 7, `BEGFULL` and `ENDFULL`, respectively. This makes element 3 `ENDEMPTY` and 2 `BEGEMPTY`. The partial child has been matched by $P3$, creating a non proper Q-node with a single element in its boundary. This element represents the same vertex as the P-node did, and has status `FULL` and child field `TRUE`, since the Q-node has a full child. The corresponding element on the parent's boundary is set to `PARTIAL` when parent is told of partial child.

When the Q-node is matched by $Q2$, its partial child is "swallowed". In `Boundary(`Q`)`, the partial element is replaced by the `Boundary` of the partial child, in this case one element. This is visible in the changed status of element 7.

ROOT(T_8, S_9), node 1, is matched by $P6$. $P6$ leaves the partial child as root of the pertinent subtree. Since this is the last step of the reduction,

the vertex addition is carried out in `Boundary(`Q`)` at this point. The `Chain` of the full child of node 1 is inserted between element 1 and the new element, creating a new idle element. On the other side, the new element is connected to `ENDFULL`. Those elements that are removed from the boundary, represent vertices that become internal to the biconnected component when the new vertex is added, so that the boundary of the Q-node in T_9 reflects the boundary of the biconnected component in B_9 (fig. 4.6).

Elements touched by the reduction are reset when nodes in the pruned subtree are reset, after the vertex addition step of PLANAR. In (c) of fig. 4.5, `Boundary(`Q`)` is shown after this resetting.

The procedures handling the construction of boundaries during reductions are described in section 4.2. They are called by the corresponding template procedures in the PQ-tree, who also provide the necessary information about e.g. partial children.

Complexity of the additional data structure

From the above examples, it should be clear that, if implemented correctly, maintaining the `Boundary` requires no more time than handling the PQ-tree. Each template called by REDUCE will take a constant amount of time extra to deal with the `Boundary`. Building and maintaining a chain does also take constant amount of time. The total number of elements in all boundaries, not counting the joints, are $\mathcal{O}(n)$. The number of joints equals the number of biconnected components of degree at least three in the bush form. This number is at most $\mathcal{O}(\frac{1}{3}n)$, since one component consists of at least three vertices.

The pointers to `BEGFULL`, `ENDFULL`, `ENDEMPTY`, and `BEGEMPTY` must be set once for all boundaries in the pruned subtree. At most $\mathcal{O}(n)$ elements will be traversed in total in each reduction. Pointers `JOINT, END1` and `END2` are maintained by each boundary, and take only constant time to be set.

Thus the conclusion should be clear, this additional data structure does not increase the time complexity of the PQ-tree. A more thorough proof can be found in [Kar90].

4.1.3 Building the pruned subtree

PLANAR of [BL76] is only concerned with deciding whether a graph is planar or not. REDUCE stops if a node cannot be matched, BUBBLE breaks off and returns the empty tree if it discovers that the pruned subtree cannot be built.

When working with non planar graphs, it is essential that BUBBLE always completes the building of the pruned subtree. Therefore, it must be modified.

New version of Bubble

This new version, denoted BubblExt, will be a slightly modified version of Bubble. For completeness, the parts of Bubble that are unchanged are described as well. Recall also the short outline of Bubble given on page 16.

All nodes in the PQ-tree have a mark, initially set to `UNMARKED`. This field is used by Bubble, and reset after each reduction, along with all fields used by Bubble and Reduce, such as the counter of descendant pertinent leaves. Bubble has two counters, `BLOCKED_NODES` and `BLOCK_COUNT`.

The first time a node is seen by Bubble, it is put in a queue and marked `QUEUED`. When dequeued, the node is marked `BLOCKED`. If Bubble succeeds in finding a valid parent pointer, the node is marked `UNBLOCKED`, and its parent is put in the queue if it is still `UNMARKED`. Nodes that remain blocked, are counted by `BLOCKED_NODES`. Sequences of blocked nodes, blocks, are counted in `BLOCK_COUNT`. When a sibling of a blocked node receives a valid parent pointer in a later step, it is passed on to the whole block, and the numbers `BLOCKED_NODES` and `BLOCK_COUNT` are counted down accordingly.

In BubblExt, all nodes ever counted in `BLOCKED_NODES` are put on a stack. If Bubble is not able to unblock all blocks before the queue empties, it returns the empty tree. BubblExt will instead find valid parent pointers for all blocked nodes. The stack is emptied, node by node. Nodes marked `UNBLOCKED` are just thrown away. For each node X marked blocked, a procedure FindParent is called. FindParent traverses the siblings of X until a valid parent pointer is found. This pointer is given to X, and then passed on to its blocked siblings in the same way described above for Bubble. The parent of X is also told of its pertinent children and, if still `UNMARKED`, put in the queue. `BLOCKED_NODES` and `BLOCK_COUNT` are also updated. When `BLOCKED_NODES` reaches zero, the stack is emptied, since no more blocked nodes remains.

If the queue is no longer empty, Bubble continues to build the pruned subtree, as described above. If the queue empties with one block left, without the root being reached, Bubble places that block as children of a *pseudo node*, and makes that node ROOT(T, S_i). BubblExt has to find the real pertinent root, so FindParent is called in this situation as well.

FindParent is only called when Bubble has failed to find a valid parent pointer for some blocked nodes. FindParent will only go as far as necessary to find a valid parent pointer, but it might have to traverse the sibling chain all the way to the end if no unblocked node is encountered. Since a Q-node may have several blocks of blocked children, FindParent may pass other blocks in its search for a valid parent pointer for X. When FindParent is called for these blocks later on, it might end up traversing the same nodes

traversed for X. To avoid this extra work, without jeopardizing the updating of the parent, FINDPARENT introduces a new mark. All nodes marked `BLOCKED` traversed by FINDPARENT are put on a local stack. All such nodes will receive the valid parent pointer from FINDPARENT, and their mark is set to `BLOCKED_WITH_PARENT`. When BUBBLEXT pops nodes with this mark off its stack, it does exactly the same as for nodes marked `BLOCKED`, but it needs not call FINDPARENT.

Complexity of BUBBLEXT

From the above discussion, it should be clear that FINDPARENT is called at most $|S_{i+1}|$ times by BUBBLEXT(T_i, S_{i+1}). The total number of children of Q-nodes in T_i is at most i, thus at most $\mathcal{O}(i)$ nodes are traversed by all calls to FINDPARENT by BUBBLEXT(T_i, S_{i+1}). Summing this up for all calls to BUBBLEXT made by the reduction step, gives time $\mathcal{O}(n+m)$.

BUBBLE as presented in section 3.1.2, takes time $\mathcal{O}(n+m)$ as well (see page 24). BUBBLEXT handles exactly the same nodes as BUBBLE, performing exactly the same work, as long as T_i is reducible with respect to S_{i+1}. The only exception is that the pseudo node is not used if the real root of the pertinent subtree cannot be found. As argued above, these calls amount to at most $\mathcal{O}(n+m)$.

The extra work done by BUBBLEXT when the tree is irreducible, is due to the fact that the graph is non planar. Since the number of edges of a non planar graph is $\mathcal{O}(n^2)$, the complexity of BUBBLEXT is also $\mathcal{O}(n^2)$.

4.2 Maintaining additional information in the PQ-tree

The new data structure of the previous section has to be incorporated into the PQ-tree data structure so that the intended information can be maintained. `Chain` and `Boundary` maintain additional information in the PQ-tree about structures in the embedded subgraph of the corresponding bush form. New structures build when a new vertex is added to the bush form. The PQ-tree prepares for the new vertex in the reduction step, making sure that all incoming edges to the vertex can be made consecutive. The new vertex is then added in the vertex addition step. `Boundary` and `Chain` must therefore be an integrated part of the reduction and vertex addition steps of the PQ-tree algorithm, in order to maintain the correct information about the structures of the bush form.

All handling of boundaries is done within REDUCE. Each time a node, not ROOT(T, S_i), has been matched, the parent's `Boundary` is updated. Each element thus updated, is placed on a stack, so that it can be reset or deleted after the vertex addition step, in the same way the nodes of the pruned subtree are reset.

As noted in section 3.3, the implementation has only one type of internal nodes, allowing only one type of information reserved for internal nodes. Therefore all P- and Q-nodes will have `Boundary` objects, but for proper P-nodes only `vertexNumber` will be valid, and for non proper P-nodes and Q-nodes, all `Boundary` pointers, such as `JOINT`, will be valid. This also allows templates to update a parent's `Boundary` without being concerned if that parent actually has a boundary or not. And proper P-nodes can hold `Chains` from their full children, allowing correct boundaries to be constructed in $P2$, $P4$, and $P5$.

REDUCE handles constructions of new chains when biconnected components of degree two are built. The vertex addition step handles chains of vertices of degree two, that build when $|S_i| = |S_i'| = 1$. If ROOT(T, S_i) has a chain, it is passed on to the new vertex.

There is one situation in the PQ-tree that calls for a `Chain` object not representing a chain in the embedded subgraph. If ROOT(T, S_i) is full and child of a Q-node or non proper P-node, then no new element representing the new node has been added to the boundary of this parent. Hence, there will be no element to update when the new node becomes pertinent in a later reduction. If there already exists a `Chain` that will be passed on from ROOT(T, S_i) to the new node, this will not be a problem. The first element of the `Chain` will be the number of the correct element to update. Otherwise, the new node will receive a `Chain` with only two numbers, the number of the old and new vertex. The first number is then the number of the corresponding element on parent's `Boundary`, the element that should be updated when the new node later becomes pertinent.

4.2.1 Template procedures

In PLANAR, REDUCE handles all template matchings of the pertinent nodes of the PQ-tree, and ensures that the pertinent sequence is constructed, if possible. The nine templates were given in fig. 3.4 on page 15. Now REDUCE will have the responsibility to maintain `Chain` and `Boundary` as well. To ensure that this is done properly, each template of the PQ-tree calls a corresponding procedure in the `Boundary` of the node being matched. Any `Chains` held by partial children, and other necessary information, is provided by the caller.

A template matching a node descendant of $\text{ROOT}(T, S_i)$, has responsibility to inform the parent of this node of the status of its pertinent child. When this is done, the `Boundary` of the parent node is also informed about the correct status, what element that corresponds to this child, if any, and whether or not this child has a chain. If it does, a pointer to the chain is sent along.

In Leipert's implementation, when new Q-nodes are created by $P3$, the full child is always made rightmost and the empty child leftmost. This rule is also followed in `Boundary`, and is the reason why special cases arise only for `END1` *or* `END2`, and not both, when `Boundary` is under construction. As noted in chapter 3, children of a node are implemented as a doubly linked list without direction. The same structure is used for the doubly linked list of elements in `Boundary`. That is why the element on the other side is needed when traversing the list.

$P4$ and $P5$ are interchanged in the following presentation, since $P4$ and $P6$ match $\text{ROOT}(T, S_i)$ and will both apply vertex addition in the `Boundary`. Utility procedures are given in section 4.2.2

L1

$L1$ is not really a template, it only marks the leaf `FULL`.

If the leaf is not $\text{ROOT}(T, S_i)$, $L1$ tells its parent about its status. The `Boundary` of its parent is also updated.

P1

$P1$ matches a full node as well. Regardless of whether it is $\text{ROOT}(T, S_i)$ or not, it is responsible for a whole little subgraph, so it must prepare a `Chain` in case this subgraph ends up as (part of) a biconnected component. This work is done in `Boundary`.

In `Boundary(`X`)`:

```
procedure ProperP1(X, i)
begin
    if X has chain then
        if all children of X have chain then add one of them to Chain(X);
        else add i to Chain(X);
    else {X has no chain}
        if all children of X have chain then make one of them Chain(X);
        else make new chain of vertexNumber(X) and i;
    fi;
end;
```

For non proper P1, see Q1.

P2 (ROOT(T, S_i))

$P2$ gathers all its full children under a new P-node, and makes that node ROOT(T, S_i). $P2$ will always be a proper P-node, since it must have at least three children. In the bush form, a biconnected component of degree two will be created. If all the full children have a chain, one must be kept as chain of the full node, and given to the new node in the vertex addition step, in case this component later becomes part of the boundary of another biconnected component.

In `Boundary(X)`:

```
procedure P2(X)
begin
    if all full children of X have chain then
        make one of them Chain(X);
end;
```

P3 (not ROOT(T, S_i))

$P3$ creates a new Q-node with two children, one empty and one full. Its parent is told about its partial child, and parent's boundary about partial element. The P-node matching $P3$ can be both proper and non proper. If it is non proper, it already has a boundary, and is treated as $Q2$, not ROOT(T, S_i) and with no partial child. The only thing that will be done in this case is to mark the pertinent sequence.

The boundary-object of a proper P-node is empty, except for the vertex number. Since the corresponding component is still a cut vertex, that is correct so far. But the new Q-node is the building block of a new component and a proper Q-node, and should carry the necessary information to be a part of that boundary. Hence an element that will represent the same vertex in the embedding, and the two children in the PQ-tree, is constructed and given the number of that vertex. If the full child has a chain, it is kept in this element.

In `Boundary(X)`:

```
procedure ProperP3(X)
begin
    make new element El with vertexNumber(X), STATUS(FULL), CHILD(TRUE);
    if all full children of X have chain then give one of them to El;
    END1 := El ; {Left endmost is only corner of triangle}
end;
```

For non proper P3, see Q2.

P5 (not ROOT(T, S_i))

$P5$ attaches its full children, if any, to the full end of the partial child and, if left with enough empty children, itself to the empty end. Since the partial child switches place with the P-node, any chain held by the P-node is transferred to the partial node. partial node gets the old boundary.

The vertex a proper $P5$ represents, will be placed on the "empty" side of the boundary of the triangle, regardless of whether it has empty children, full children, or both. But the `CHILD` field will be TRUE only if the node has empty children. Thus, the boundary created by $P5$ will be far from what has been defined earlier.

If the P-node has empty children, the joint's children are made part of the triangle's boundary. This is solved by letting the empty endmost point to the same element as the joint, since this element also represents an endmost corner. The situation is corrected when the parent node is matched, by giving the partial boundary a new joint, or by including it in an already existing boundary.

On the other hand, if the P-node has only full children besides the partial child, the joint is still idle. But if the partial child was matched by $P3$, the boundary consists of only two elements, one idle and one full. For later templates to find full and empty end of boundary, both `JOINT` and `END1` must point of the idle element. This situation will be corrected when the new vertex is added in ROOT(T, S_i), if not sooner.

Non proper $P5$ is treated as $Q2$, not ROOT(T, S_i) and with one partial child, except it's joint will be empty end if the P-node had empty children. In this case, the boundary of the P-node is expanded to include the boundary of the partial child, thus the P-node's boundary must then be given to the partial child.

In `BOUNDARY(partialChild(`X`))`:

```
procedure PartialP5(vertexNumber(X), HASEMPTY)
begin
    if this boundary has only one element then
        make new element JOINT with vertexNumber(X),
            STATUS(EMPTY), CHILD(HASEMPTY);
        END2 := END1;
        END1 := JOINT;
        connect the two elements;
        if there is a chain between partial node and X then
            insert it between END1 and END2;
    else    {Boundary has at least two elements,}
            {JOINT, END1 and END2 all valid pointers}
        if END1 has status EMPTY then
            if JOINT has NUMBER different from vertexNumber(X) then
```

```
                make new element JOINT with vertexNumber(X),
                    STATUS(EMPTY), CHILD(HASEMPTY);
                if there is a chain between partial node and X then
                    insert it between new and old JOINT;
            else let JOINT get CHILD(HASEMPTY);
            if HASEMPTY is TRUE then make END1 point to JOINT;
        else if END2 has status EMPTY then
            if JOINT has NUMBER different from vertexNumber(X) then
                make new element JOINT with vertexNumber(X),
                    STATUS(EMPTY), CHILD(HASEMPTY);
                if there is a chain between partial node and X then
                    insert it between new and old JOINT;
            else let JOINT get CHILD(HASEMPTY);
            if HASEMPTY is TRUE then make END2 point to JOINT;
        fi;
        mark pertinent sequence;
    fi;
end;
```

And for non proper $P5$, in `Boundary(X)`:

```
procedure NonProperP5(X)
begin
    call NonRootQ2(X);
    if HASEMPTY is TRUE then
        let JOINT get CHILD(TRUE);
        if END1 has status EMPTY then make END1 point to JOINT;
        else make END2 point to JOINT;
    fi;
end;
```

For NonRootQ2, see $Q2$.

P4 (ROOT(T, S_i))

$P4$ attaches its full children to the full end of its partial child, and makes the Q-node new ROOT(T, S_i). If the P-node has no empty children, it is deleted from the tree, and the partial child inherits its parent's chain, if there was one.

Since the reduction finishes at this point, the boundary of the partial node can be "cleaned up", or connected. An element representing the new vertex just about to be added, is created, and connected to each end of the pertinent sequence in the boundary, throwing away those elements of the pertinent sequence that will become internal vertices in the vertex addition step of the bush form.

If the P-node is non proper, it will be treated as $Q2$, ROOT(T, S_i) and with one partial child. The P-node disappears from the tree, leaving its boundary to the partial child.

In `Boundary(`X`)`:

```
procedure PROPERP4(X, i)
begin
    if all full children of X have chain then place one of them in CH_PTR;
    call PARTIALP4(vertexNumber(X), i, CH_PTR) in partialChild(X);
end;

procedure PARTIALP4(vertexNumber(X), i, CH_PTR)
begin
    if this boundary has only END1 then
        make new element JOINT with vertexNumber(X),
            STATUS(EMPTY), CHILD(FALSE);
        make new END2 with i, STATUS(EMPTY), CHILD(TRUE);
        connect the three elements;
        if there is a chain between partial node and X then
            insert it between JOINT and END1;
        if END1 holds a chain then insert it between END1 and END2;
        if CH_PTR holds a chain then insert it between JOINT and END2;
    else    {Boundary has at least two elements,}
            {JOINT, END1 and END2 all valid pointers}
        if END1 has status EMPTY then
            if JOINT has NUMBER different from vertexNumber(X) then
                make new element JOINT with vertexNumber(X),
                    STATUS(EMPTY), CHILD(FALSE);
                connect new JOINT to old JOINT and its right neighbor;
                if there is a chain between partial node and X then
                    insert it between new and old JOINT;
                if END1 is idle then END1 := END2;
                {Old END2 is only active element on boundary.}
            fi;
            make new element END2 with i, STATUS(EMPTY), CHILD(TRUE);
            connect new END2 to JOINT and BEGFULL;
            remove any disconnected pertinent and idle elements;
            {They will be internal to the triangle}
            if BEGFULL has a chain then insert it between BEGFULL and new END2;
            if CH_PTR holds a chain then insert it between JOINT and new END2;
        else if END2 has status EMPTY then
            if JOINT has NUMBER different from vertexNumber(X) then
                make new element JOINT with vertexNumber(X),
                    STATUS(EMPTY), CHILD(FALSE);
                connect new JOINT to old JOINT and its left neighbor;
                if there is a chain between partial node and X then
                    insert it between new and old JOINT;
            fi;
            make new element END1 with i, STATUS(EMPTY), CHILD(TRUE);
```

```
            connect new END1 to JOINT and ENDFULL;
            remove any disconnected pertinent and idle elements;
            {They will be internal to the triangle}
            if ENDFULL has a chain then insert it between ENDFULL and new END1;
            if CH_PTR holds a chain then insert it between JOINT and new END1;
        fi;
    fi;
end;
```

P6 (ROOT(T, S_i))

$P6$ attaches its full children to the full end of one of its partial children. Then it connects the two partials into one Q-node, and makes that node new ROOT(T, S_i). If the P-node is not left with any empty children, it is deleted from the tree, and the partial node inherits its parent's chain, if any.

The reduction is finished at this point, so the boundary of the partial node can be "cleaned up", or connected. Accordingly, a new element representing the new vertex just about to be added, is created, and connected to the interior end of the pertinent sequence of each partial boundary, throwing away those elements of the pertinent sequence that will become internal vertices in the vertex addition step of the bush form. Full children of the P-node are not given any attention, since they are doomed to be internal.

If the P-node is non proper, it will be treated as $Q2$, ROOT(T, S_i) and with one partial child. The P-node disappears from the tree, leaving its boundary to the partial child.

In BOUNDARY(partialChild1(X)):

```
procedure ProperP6(vertexNumber(X),i,partialChild2(X))
begin
    if this boundary has only END1 then
        make new element JOINT with vertexNumber(X),
            STATUS(EMPTY), CHILD(FALSE);
        connect JOINT and END1;
        if there is a chain between this partial node and X then
            insert it between JOINT and END1;
        connect boundary of partialChild2(X) to this boundary,
            and insert new element i;
    else    {Boundary has at least two elements,}
            {JOINT, END1 and END2 all valid pointers}
        if END1 has status EMPTY then
            if JOINT has NUMBER different from vertexNumber(X) then
                make new element JOINT with vertexNumber(X),
                    STATUS(EMPTY), CHILD(FALSE);
                connect new JOINT to old JOINT and its right neighbor;
                 Old JOINT is on same side as END1
                if there is a chain between this partial node and X then
```

```
                insert it between new and old JOINT;
            if END1 is idle then make END1 point to END2;
            {Old END2 is only active element on boundary.}
        fi;
        connect boundary of partialChild2(X) to this boundary,
            and insert new element i;
    else if END2 has status EMPTY then
        if JOINT has NUMBER different from vertexNumber(X) then
            make new element JOINT with vertexNumber(X),
                STATUS(EMPTY), CHILD(FALSE);
            connect new JOINT to old JOINT and its left neighbor;
            if there is a chain between this partial node and X then
                insert it between new and old JOINT;
        fi;
        connect boundary of partialChild2(X) to this boundary,
            and insert new element i;
    fi;
  fi;
end;
```

For non proper $P6$, see $Q3$.

Q1

$Q1$ is much like $P1$, except it has a boundary. This boundary is treated the same, regardless of whether the full Q-node is $\mathrm{ROOT}(T, S_i)$ or not. The same chain must be built anyhow.

In `Boundary(X)`:

```
procedure Q1(X, i)
begin
  if Chain(X) is empty then
    make new Chain(X);
    insert NUMBER of JOINT as first element of Chain(X);
  fi;
  add one side of the boundary to Chain(X);
  make sure i is last entry in Chain(X);
  fi;
end;
```

Q2

$Q2$ is a singly partial Q-node. If it has a partial child, its children are made children of the Q-node being matched. $Q2$ matches Q-nodes both $\mathrm{ROOT}(T, S_i)$ and not. If the node is $\mathrm{ROOT}(T, S_i)$, the boundary must be cleaned up, while if not, the boundary should be built. Hence, there must be two procedures, one for each possibility.

In `Boundary`(X):

```
procedure NonRootQ2(X)
begin
    mark pertinent sequence;
    if X has partial child then
        connect boundary of partialChild2(X) to this boundary;
        mark pertinent sequence;
    fi;
end;

procedure RootQ2(X, i)
begin
    mark pertinent sequence;
    if X has partial child then
        connect boundary of partialChild(X) to this boundary,
            and insert new element i;
    else {X has no partial child}
        if END1 is FULL then
            make new END1 with i, STATUS(EMPTY), CHILD(TRUE)
                and insert it between BEGFULL and ENDFULL;
        else if END2 is FULL then
            make new END2 with i, STATUS(EMPTY), CHILD(TRUE),
                and insert it between BEGFULL and ENDFULL;
        fi;
        if BEGFULL has chain then insert it between BEGFULL and i;
        if ENDFULL has chain then insert it between ENDFULL and i;
    fi;
end;
```

Q3 (ROOT(T, S_i))

$Q3$ can only be ROOT(T, S_i). It is a doubly partial Q-node. If it has partial children, their children are included as children of the doubly partial Q-node. $Q3$ can have one, two, or no partial children. In either case, the boundary is "cleaned up", or connected, introducing a new element i. This new element will be interior, since both endmost children are empty.

In `Boundary`(X):

```
procedure RootQ3(X)
begin
    mark pertinent sequence;
    if X has no partial children then
        make new element with i, STATUS(EMPTY), CHILD(TRUE)
            and insert it between BEGFULL and ENDFULL;
        if BEGFULL has chain then insert it between BEGFULL and i;
        if ENDFULL has chain then insert it between ENDFULL and i;
    else if X has one partial child then
```

```
        connect boundary of partialChild(X) to this boundary,
            and insert new element i;
    else if X has two partial children then
        connect boundary of partialChild1(X) to this boundary;
        connect boundary of partialChild(X) to this boundary,
            and insert new element i;
    fi;
end;
```

4.2.2 Utility procedures

insert chain Receives a chain, and two elements it should be connected to. Makes new elements representing the vertices of the chain. If top and bottom elements match the two received elements, this chain of new elements is connected to them.

mark pertinent sequence Starts out the left pointer of `JOINT`, and traverses the list until it reaches an element with status `FULL` or `PARTIAL`. This element is marked `BEGFULL`, and previous element are marked `ENDEMPTY`. The same is done with the right pointer setting `ENDFULL` and `BEGEMPTY` respectively.

In regard of time complexity, this procedure should only be called once for NONROOTQ2, namely before `Boundary` of the partial child is inserted. If done right, all pointers to the pertinent sequence can be updated correctly during this insertion. In this first version of the implementation, this was not done, and unfortunately, there was not enough time to make a second version.

connect boundaries Most nodes in the pruned subtree disappear during reduction. The same fate strikes their boundaries. `Boundary(`X`)` must connect with `Boundary(`Y`)` when node X with partial child Y is reduced. These procedures are called from `Boundary(`X`)`, and executed in `Boundary(`Y`)`. `Boundary(`X`)` sends along the elements it want `Boundary(`Y`)` to connect to, and then `Boundary(`Y`)` connects its full end to the full element, and its empty end to the empty element.

- without inserting i: X is not $\text{ROOT}(T, S_i)$. The connection elements will be the two elements to either side of the partial element on `Boundary(`X`)`.
- with insertion of i: X is $\text{ROOT}(T, S_i)$, so `Boundary(`X`)` should be cleaned up. The correct elements are sent along to `Boundary(`Y`)`, and those parts of the boundary that will be interior are thrown

away. `Boundary(Y)` makes a new element i its full end before connecting, and throw away surplus elements of its pertinent sequence.

In ProperP6, one of the partial children are made ROOT(T, S_i), and the other one is connected. For Q3, with two partial children, one boundary is connected without i, and the other inserts i.

4.3 Finding the obstruction when the PQ-tree fails

The number of Kuratowski subgraphs in a graph can be much larger than the skewness number, since two different subgraphs need only differ by a single path. Therefore, when the PQ-tree fails, several Kuratowski subgraphs may be detected, and removing a single edge may in fact remove several $K_{3,3}$ or K_5.

To restore planarity in a Kuratowski homeomorph, it is enough to disconnect it by removing one edge, an entire path does not have to be removed. The cases presented next, ended up as not being a part of the skewness algorithm presented in chapter 6, for reasons discussed in section 4.4.

4.3.1 Cases

The reduction algorithm of Booth & Lueker fails when the elements of S cannot be made a consecutive sequence. This is detected either by Reduce, if no matching template for the present node is found, or by Bubble, if a connected pruned subtree is impossible to achieve.

The situations that may occur in a node of the PQ-tree are limited. The legal ones from [BL76] are shown in fig. 3.4 on page 15, and A. Karabeg shows in [Kar90] that there are only four illegal ones.

Case 1 (fig. 4.7) is the situations where the patterns of $Q1$, $Q2$ and $Q3$ will not match the Q-node because empty or partial children occur inside the pertinent sequence.

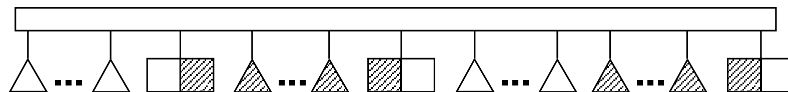


Figure 4.7: Case 1

Case 2 (fig. 4.8) The Q-node matches the pattern of $Q3$, but is *not* root of the pertinent subtree.

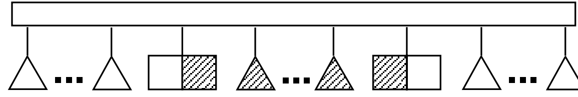


Figure 4.8: Case 2

Case 3 (fig. 4.9) A P-node, ROOT(T, S_i), with three or more partial children.

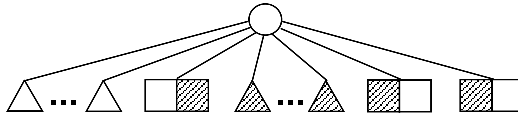


Figure 4.9: Case 3

Case 4 (fig. 4.10) P-node, not ROOT(T, S_i), with two or more partial children.

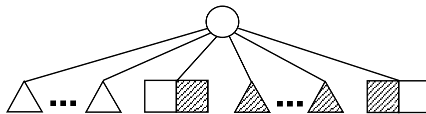


Figure 4.10: Case 4

Every situation that can occur in the pertinent PQ-tree is covered by these four Cases and the nine templates of fig. 3.4 on page 15.

Theorem 4 ([Kar90]) *The reduction process terminates before the last node is reduced if and only if the last processed node in T_i can be classified as one of Cases 1, 2, 3 or 4.*

Proof is given in [Kar90].

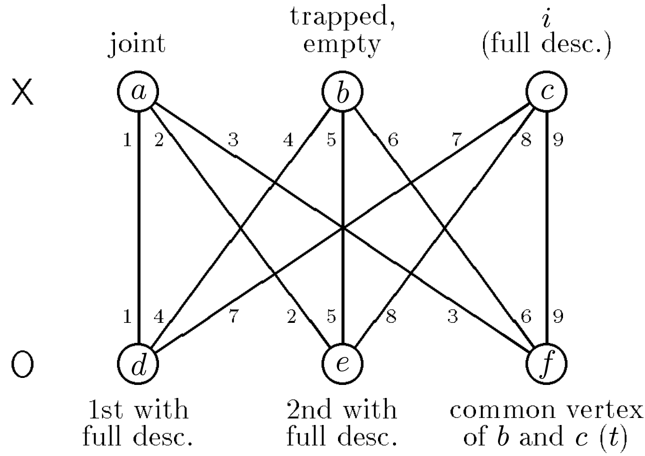


Figure 4.11: A sketch of the $K_{3,3}$ subgraph for Case 1.
Each of the edges 1 through 9 may actually be a path.

4.3.2 Procedures

In [Kar90], procedures to find the vertex sets of each of the four Cases are given, but an edge set procedure is presented for Case 1 only. Here we will give procedures for all edge sets as well. From the outset it was intended to use these procedures as a foundation for a skewness approach, together with the additional data structure of this chapter. As will be seen in section 4.4, these Cases are not directly used, but studying them provided quite a bit of insight into what happens in the PQ-tree when the reduction fails. For that reason, they are included.

Case 1

procedure VertexSet1
begin
  mark `JOINT` of `BOUNDARY(`Q`)` by X;
  find interior element with empty descendant such that on each side of this element
    on the boundary, there are elements with full descendants;
  mark this element by X;
  mark two elements with full descendants, one on each side of the
    previous element, by O;
  mark a full descendant itself by X;
  find paths from the last two X's to t;
  if paths are disjoint **then mark** t by O;
  else mark the first common vertex on the paths by O;
end;

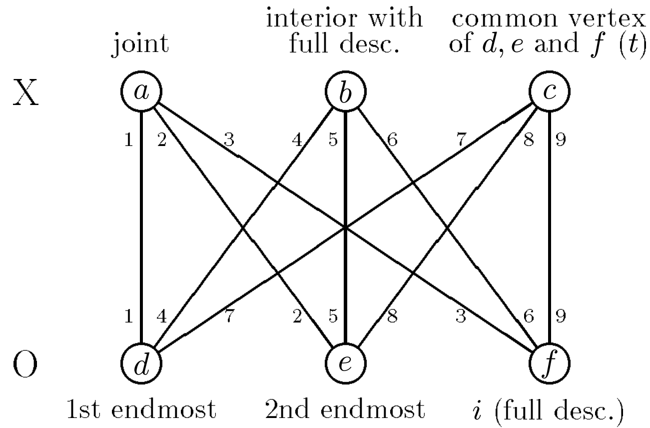


Figure 4.12: A sketch of the $K_{3,3}$ subgraph for Case 2.
Each of the edges 1 through 9 may actually be a path.

procedure EDGESET1
begin
  find the four paths on `BOUNDARY`(Q);
  find paths from vertices that have a full descendant, to that descendant;
  {Two paths are found in VERTEXSET1*}*
  find path from the last vertex marked by O to `JOINT`, containing edge $\{s, t\}$;
end;

Case 2

procedure VERTEXSET2
begin
  mark `JOINT` of `BOUNDARY`(Q) by X;
  mark two endmost elements by O;
  mark interior element with full descendant by X;
  mark that descendant by O;
  find paths from the three vertices marked O to t;
  if all three paths are disjoint **then mark** t by X;
  else if the path from the full child and one of the other two paths intersect **then**
    mark the first common vertex by X;
  else
    {All three paths join}
    mark the first common vertex (of two or all three paths) by X;
  fi;
end;

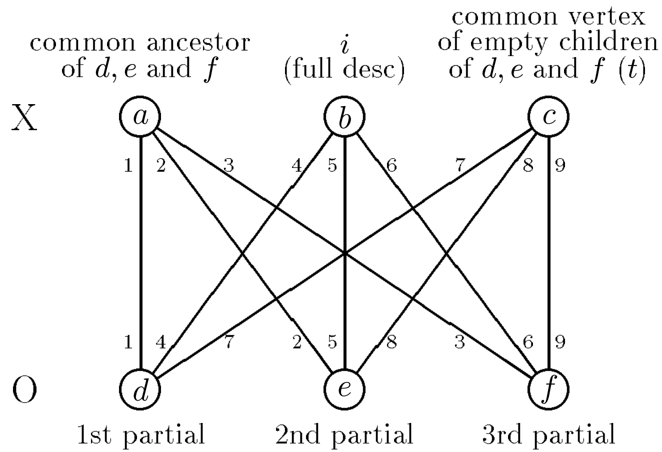


Figure 4.13: A sketch of the $K_{3,3}$ subgraph for Case 3.
Each of the edges 1 through 9 may actually be a path.

procedure EdgeSet2
begin
  find the four paths on `BOUNDARY(`Q`)`;
  find path from vertex with full descendant, to that descendant;
  {Three paths are found in VertexSet2*}*
  find path from `JOINT` to a full descendant outside the Q-node;
end;

Case 3

procedure VertexSet3
begin
  mark three partial nodes by O;
  mark their common ancestor (the P-node) by X;
  mark any full descendant by X;
  find paths from one of the empty children of the three partial nodes to t;
  mark the smallest numbered common vertex for any two of these paths by X;
end;

procedure EdgeSet3
begin
  find paths from P-node to partial children;
  find paths from partial children to full descendant;
  {Three paths are found in VertexSet3*}*
end;

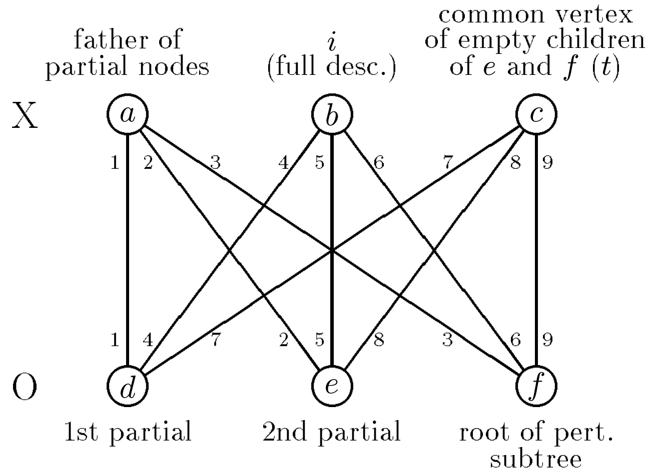


Figure 4.14: A sketch of the $K_{3,3}$ subgraph for Case 4 when P-node is proper. Each of the edges 1 through 9 may actually be a path.

Case 4

procedure VERTEXSET4
begin
  mark `JOINT` of two partial nodes by O;
  if P-node is proper **then**
    mark it by X;
    mark ROOT(T, S) by O;
    mark any full descendant by X;
    find paths from one of the empty children of the two partial nodes to t;
    mark the smallest numbered common vertex of these paths by X;
  else
    {P-node not proper}
    mark `JOINT` of Q-node and one full descendant by O;
    find paths from one of the empty children of the two partial nodes to t;
    mark the smallest numbered common vertex of these paths by O;
    {They form a vertex set of K5}
  fi;
end;

procedure EDGESET4
begin
  if a K_5 were found **then**
    find the three paths on `BOUNDARY`(Q);
    find paths from partial nodes and Q-node (via ROOT(T, S_i)) to full descendant;
    {Two paths are found in VertexSet4}
    find path from t to `JOINT` of Q-node, containing edge $\{s, t\}$;

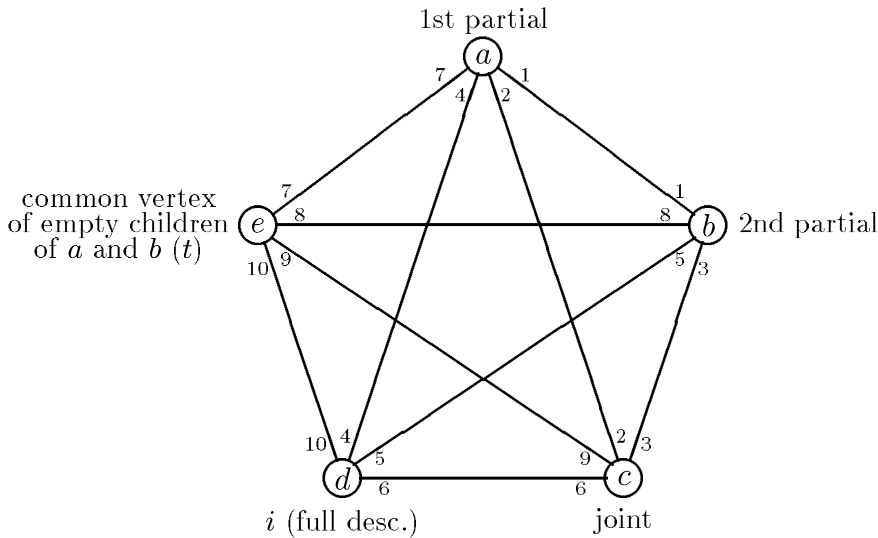


Figure 4.15: A sketch of the K_5 subgraph for Case 4 when P-node is non proper. Each of the edges 1 through 10 may actually be a path.

```
        find path from full descendant to t;
    else
        {A K3,3 were found}
        find paths from P-node to two partial nodes;
        find path from ROOT(T, S) to P-node;
        find paths from ROOT(T, S) and partial nodes to full descendant;
        {Two paths were found in VertexSet4}
        find path from the last vertex marked by X to ROOT(T, S),
            containing edge {s, t};
    fi;
end;
```

Finding obstructions in linear time.

The algorithm presented in [Kar90] assumes that templates corresponding to Cases 1 through 4 are added to those given by Booth and Lueker in [BL76]. This can easily be done. The procedure for explicitly exhibiting a subgraph homeomorphic to a Kuratowski graph when the PQ-tree fails can then be sketched like this:

```
procedure OBSTRUCTIONS(G)
begin
    if Case(i) was detected by PLANAR(G) then
        call the vertex and edge set procedures of Case i;
end;
```

The vertex and edge set procedures presented above are clearly linear in the number of vertices of the graph. The paths that goes beyond the vertices and edges of the bush form (all paths leading to t) can be found from the sorted adjacency lists that represents the graph. Common vertices of these paths exist if consecutive occurrences of vertices occur.

The additional data structure maintained for this algorithm were shown linear in section 4.1.2, and thus, given that the actual amount of time spent by Planar before it fails for non planar graphs is always $\mathcal{O}(n)$, Obstructions is $\mathcal{O}(n)$.

4.3.3 Implementation of additional data structure

As Leipert's implementation is a template class with three parameters, opening for additional information in the nodes of the PQ-tree, it was perfect for implementing the additional data structure of this chapter. The focus during the development of the procedures of section 4.2 was on the specifications and conditions of [Kar90]. Only the data structure has been implemented, not the Obstructions procedures and additional templates of section 4.3.

The derived subclass `pqtree` of section 3.3.2 was extended to implement the additional data structure of this chapter. `Chains` are held by children, and all nodes in the PQ-tree, except $\mathrm{ROOT}(T, S_i)$, is child of another node. Accordingly, class `Chain` was constructed and used as parameter `X`. See figs. 3.13 and 4.16. `Boundary` is meant to contain information specific to

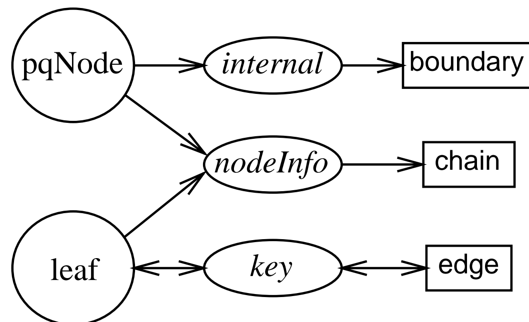


Figure 4.16: The data structure of `pqtree`
Circles are the objects of the PQ-tree. Ovals are their respective information handlers, and the squares the information holders specified at initialization, as parameters to the `pqtree` object. The double arrows denote pointers that are being maintained by Leipert's implementation, single arrows the main access direction maintained by `pqtree`.

Q-nodes and non proper P-nodes, while proper P-nodes are intended to have `vertexNumber` only. Since there is no distinguishing between the information held by P- and Q-nodes in Leipert's implementation, `Boundary` and `vertexNumber` was combined, as mentioned in section 4.2. Class `Boundary` is thus used for all P- and Q-nodes. Proper and non proper P-nodes are distinguished between by checking whether or not the `JOINT` pointer is active.

The main procedures necessary to maintain `Chain` and `Boundary` were outlined in section 4.2. All template procedures of Leipert's implementation were overridden in class `pqtree`, so that the procedures of `Boundary` can be called. Procedure `Bubble` was also overridden, and changed to match BUBBLEXT of section 4.1.3. Code files and some further descriptions are given in appendix A.

4.4 OBSTRUCTIONS and skewness number

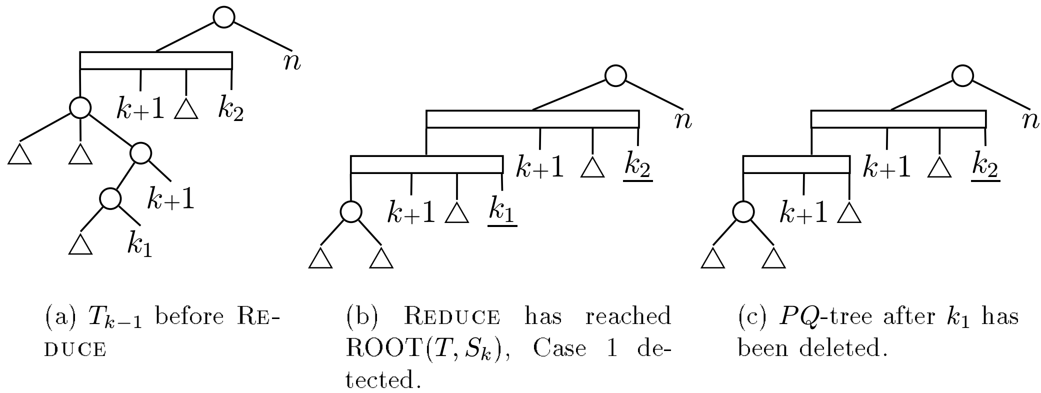
Cases 1 through 4 of algorithm OBSTRUCTIONS exhibit a Kuratowski subgraph when the PQ-tree fails. The procedures of section 4.3.2 provide us with all vertices and edges of the subgraph. Therefore, the possibility of using Cases and additional data structure of this chapter as an approach to a heuristic for the skewness number, was investigated.

The pen and paper method was used. By running small graphs through PLANAR, keeping track of boundaries and chains, rules for letting the Cases decide the edges to delete, were sought. It soon became apparent that the `Boundary` could provide useful information about when a deleted edge could be reintroduced to the embedded graph without destroying planarity. By marking the elements that represented vertices with deleted, outgoing edges, it was easy to see when all obstacles causing the deletion had been removed, thus enabling a planar embedding.

However, finding consistent rules for the Cases proved to be harder. It was therefore decided to implement the additional data structure as if it would be used for exhibiting homeomorphs of $K_{3,3}$ and K_5 when one was detected. The plan was to test different sets of rules on the computer instead of by hand. But before the Cases were programmed, a serious problem with this approach came to the surface.

Problems with this approach

Consider an irreducible PQ-tree T_i. Let one of the reducible nodes in the pruned subtree of T_i be a partial P-node with no partial child. This node will be made into a Q-node with two children by template $P3$, a non proper Q-



(a) T_{k-1} before REDUCE

(b) REDUCE has reached ROOT(T, S_k), Case 1 detected.

(c) PQ-tree after k_1 has been deleted.

Figure 4.17: Reducing T_{k-1}.

node by the definitions of table 3.1. Since T_i is now a non proper PQ-tree, it no longer corresponds to the bush form B_i. In B_i, the cut vertex represented by the P-node will stay a cut vertex until the new vertex is embedded. For PLANAR and planar graphs, this is no problem, as a PQ-tree T_{i+1} will match the bush form B_{i+1} when the vertex is added, maintaining the bush form as an invariant of the PQ-tree.

Since this PQ-tree is irreducible, the pertinent children of the new Q-node may not end up as part of the maximal pertinent sequence in T_i. Thus a Q-node representing cut vertices in the bush form B_{i+1} may be part of the PQ-tree after REDUCE has finished. The bush form can thus no longer be used as an invariant for the remaining PQ-trees. As the next example will show, this may lead to more edges than necessary being deleted from the graph, rendering a planar subgraph that is not maximal.

Fig. 4.17 pictures this situation. (a) shows the PQ-tree T_{k-1} before REDUCE is called. The tree has two pertinent leaves, k_1 and k_2. The corresponding bush form B_{k-1} is shown in fig. 4.18. The two leaves and cor-

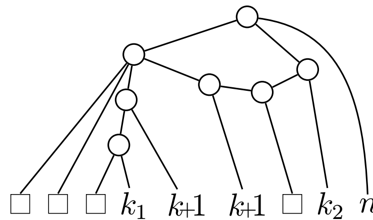


Figure 4.18: Bush forms B_{k-1}, corresponding to T_{k-1} of fig. 4.17(a).

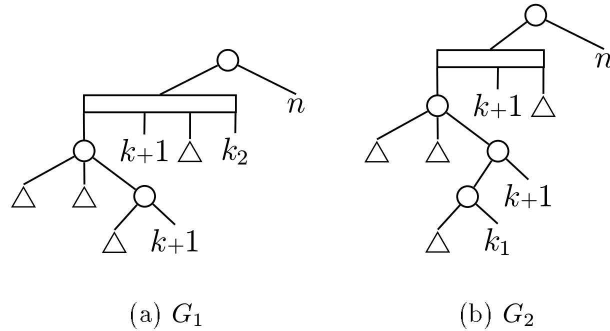


Figure 4.19: T_{k-1} obtained for each of G_1 and G_2.

responding virtual vertices can clearly not be made consecutive in neither the PQ-tree nor the bush form. Accordingly, REDUCE(T_{k-1}) detects Case 1 when ROOT(T_{k-1}, S_k) is reached, fig. 4.17(b). One of k_1 and k_2 will have to be deleted to make the PQ-tree reducible and the subgraph planar. If k_1 is deleted from T_{k-1}, the PQ-tree will have two Q-nodes, of which only one represents a biconnected component in the bush form. The situation remains the same if k_1 is kept and k_2 deleted.

In fig. 4.17(c), k_1 has been deleted. In the vertex addition step, a new P-node will replace k_2. Without loss of generality, the new vertex is assumed not to have outgoing edges leading to vertex $k+1$. Thus, when T_k is reduced, only the two leaves labeled $k+1$ in T_{k-1} will be pertinent in T_k. As seen in fig. 4.17(c), these leaves will not be able to from a consecutive sequence in the PQ-tree. Case 2 will be detected, and another leaf deleted. A look at the bush form B_{k-1} in fig. 4.18 shows that it is possible to embed both incoming edges to vertex $k+1$ in the subgraph. Deleting k_2 instead will lead to the exact same situation.

The problem is that the presence of such Q-nodes may cause several PQ-trees to be irreducible, and thus even more edges to be deleted from the graph. To show how the PQ-tree will behave if the situation in T_{k-1} is removed, consider two subgraphs G_1 and G_2, obtained from G by removing one edge from each, edges k_1 and k_2 respectively. The respective PQ-trees T_{k-1} obtained for each graph is given in fig. 4.19. Clearly, the next tree T_k can be reduced without problems for both graphs, and no edges will be deleted from these trees. Again, it can be assumed w.l.o.g., that G_1 and G_2 are planar. Thus, the skewness number of G is 1. In addition, both G_1 and G_2 are maximal (and maximum) planar subgraphs of G.

Using the Cases of algorithm OBSTRUCTIONS will give a connected and planar subgraph of any non planar input graph. But for some graphs, the

obtained subgraph will obviously not be maximal, and the given estimate for the skewness number will be equally far from optimal.

This is caused by PQ-trees that no longer reflect the structures of the corresponding bush form, and thereby loose some knowledge of the embeddable subgraph obtained so far. As seen in the above example, this may lead to the deletion of clearly embeddable edges.

A better result is wanted, and seems to be obtainable if a way to decide when, and which, leaves should be deleted from a PQ-tree *before* REDUCE is called, and the structures of the PQ-tree changed.

The search for such a method led to revisiting a number of articles, resulting in the survey presented in the next chapter.

Chapter 5

Approaches to planarizing graphs using PQ-trees

Many heuristics for planarizing graphs have been published. Most of them also claim to find a maximal planar subgraph. This chapter surveys some of these works, with emphasis on approaches based on PQ-trees.

Many applications require a graph to be planar or separated into planar subgraphs. The removal of edges in order to obtain a planar subgraph of a given non planar graph is called planarization. Ideally, one would want an algorithm that produces a maximum planar subgraph within reasonable time. However, that problem is $\mathcal{NP}$-hard, and in practice a heuristic must be used.

A planar subgraph where no additional edges from the original graph can be added without destroying planarity, is maximal planar. Determining a maximal planar subgraph is in $\mathcal{P}$. Thus any heuristic for maximum planar subgraph should at least come up with a maximal planar subgraph. The straight forward way of finding a maximal planar subgraph is to start with one edge, and then, for every edge that is added to the subgraph, test if planarity is preserved. This gives an $\mathcal{O}(nm)$ algorithm, given that planarity is tested in time $\mathcal{O}(n)$.

More efficient algorithms have been presented in the literature. Cai, Han, and Tarjan [CHT93] uses the path addition algorithm of [HT74] to find a maximal planar subgraph, achieving an $\mathcal{O}(m \log n)$ time bound. The algorithm of Di Battista and Tamassia [DT89] checks in $\mathcal{O}(\log n)$ time whether or not an edge can be added to G without destroying planarity, rendering an $\mathcal{O}(m \log n)$ algorithm as well. An algorithm claiming to give results closer to optimum than what [DT89], [Kan92] (see below), and earlier versions of the algorithm of [CHT93] give, has been presented by Goldschmidt and Takvorian [GT94]. They use a very interesting approach, but the worst case time

complexity of this algorithm is at least $\mathcal{O}(nm)$.

The first approach using PQ-trees to planarize a non planar graph, is due to Ozawa and Takahashi [OT81]. They claimed that their $\mathcal{O}(nm)$ algorithm finds a maximal planar subgraph. Jayakumar, Thulasiraman, and Swamy later showed that this is not the case. Any maximal planar subgraph of a biconnected graph is connected, and the algorithm of [OT81] fails to always include all vertices in the planar subgraph. In [JTS89] they modified the approach of [OT81] into an $\mathcal{O}(n^2)$ algorithm PLANARIZE, rendering a connected planar subgraph G_p that is not necessarily maximal. They also present a second phase, based on PQ-trees as well. If the subgraph G_p found by PLANARIZE is biconnected, MAXPLANARIZE augments it into a maximal planar subgraph G_p' of G.

G. Kant showed in [Kan92] that the second phase of [JTS89] is not correct, and gave an improved version of MAXPLANARIZE that augments G_p into a maximal planar subgraph of G, independent of whether G_p is biconnected or not. This improved version has been shown incorrect by S. Leipert in [Lei96]. He points out three problems, but suggests corrections only to two of them.

In addition to this, Jünger, Leipert, and Mutzel [JLM96] recently pointed out serious deficiencies in the approaches of [JTS89] and [Kan92], one directly related to the nature of the PQ-tree. The reader should by now be convinced that searching for a heuristic for the maximal planar subgraph problem (or the skewness problem) based on the PQ-tree data structure is not a simple task!

Some of the results presented in this chapter will be used in the next chapter as a basis for a new approach for the maximal planar subgraph problem. This new approach will avoid most of the fundamental problems of the approaches discussed in this chapter. Parts of PLANARIZE [JTS89], that will be used directly in chapter 6, are thoroughly described in section 5.1, while results from the articles concerning MAXPLANARIZE is briefly discussed in section 5.2.

5.1 A planarization algorithm

This section will describe the basic principles and procedures of the planarization algorithm PLANARIZE of [JTS89]. Rules used by procedures COMPUTE and DELETENODES are given in extenso, since they will be used for a new approach for the skewness number presented in chapter 6.

As seen in the previous chapter, during a reduction, some pertinent leaves may cause a node not to match any of the legal templates of [BL76], thus signaling that the graph is not planar. The goal in PLANARIZE is to solve

these situations before each reduction, by deleting just enough leaves to make each tree reducible. As few edges as possible are deleted, in such a way that the remaining graph G_p is a connected planar subgraph of G. G_p is only guaranteed to be maximal planar if G is complete ([JTS89, Kan92]).

Based on the frontier of a node, its descendant leaves read from left to right, [JTS89] classify the nodes in a new way. This classification is later used to decide if any of its descendants need to be deleted. The corresponding notions used in [BL76] and chapter 3 are given in parenthesis for each type. Remember that the frontier of a leaf is the leaf itself.

Type W: A node is said to be Type W, if its frontier consists of only empty leaves. (Empty node)

Type B: A node is said to be Type B, if its frontier consists of only pertinent leaves. (Full node)

Type H: A node X is said to be Type H if the subtree rooted at X can be arranged such that all the descendant pertinent leaves of X appear consecutively at either the left end or at the right end of the frontier. (Singly partial node)

Type A: A node X is said to be Type A if the subtree rooted at X can be arranged such that all the descendant pertinent leaves of X appear consecutively in the middle of the frontier with at least one empty leaf appearing at each end of the frontier. (Doubly partial node)

PLANAR of section 3.2.2 constructs a sequence of PQ-trees $T_2, T_3, \ldots, T_{n-1}$ provided that each T_i, $1 \leq i \leq n-2$, is reducible with respect to S_{i+1}. Recall the definitions given in section 3.1.1. The principle of the planarity testing algorithm of [BL76] and [LEC67] can thus be restated as the following theorem, the cornerstone of the planarization algorithm of [JTS89].

Theorem 5 ([JTS89]) *A graph G is planar if and only if the pertinent roots of all subtrees in $T_2, T_3, \ldots, T_{n-1}$ of G are Type B, H, or A.*

A PQ-tree whose pertinent root is Type B, H, or A is reducible, otherwise it is irreducible. An irreducible PQ-tree is made reducible by appropriately deleting some of its pertinent leaves.

A full node can be made Type B or, by deleting all its pertinent children, Type W. A partial node can be made Type W, Type H, or Type A, but not Type B, as PLANARIZE do not allow deletion of empty leaves. The reason why is explained next, as well as how the type of each node is determined.

5.1.1 Finding edges to delete

Since the aim of the algorithm is to find a maximal planar subgraph, and it usually pays to be greedy in heuristics, as few edges as possible is sought to be deleted in each step. Of course, if T_i is reducible, no edges need to be deleted.

The reason the algorithm of [OT81] does not necessarily produce a spanning subgraph, is that they allow empty leaves to be deleted. If a deleted empty leaf $(j, k), j < k$, represented the only incoming edge to vertex k in T_{k-1}, then there will be no pertinent leaves to reduce in the PQ-tree, and nowhere to add the outgoing edges of k. Thus k, and maybe some of its descendants, will not be a part of the planar subgraph embedded by the PQ-tree. Accordingly, only pertinent leaves will be deleted by PLANARIZE to ensure that G_p is connected.

If a PQ-tree T_i is irreducible, the strategy of the algorithm is to delete some of its pertinent leaves to make it reducible. As concluded in section 4.4 of the previous chapter, these leaves should be found and removed before the tree is reduced. In other words, all nodes in the pertinent subtree should be one of Type W, B, H, or A before calling REDUCE. The w-, h-, and a-number of a node X in the pertinent subtree of T_i is the minimum number of pertinent descendant leaves of X that should be deleted from T_i in order to make X Type W, H, and A, respectively. The tuple of numbers thus associated with a node is hence denoted by $[w,h,a]$. Since only pertinent leaves can be deleted, the b-number is not needed. The $[w,h,a]$-number of a node X is computed from the $[w,h,a]$-numbers of its children, so the computation is done in a bottom up process.

The actual type of the nodes is decided after COMPUTE(T_i) reaches ROOT(T_i, S_{i+1}). If the minimum of h and a for ROOT(T_i, S_{i+1}) is zero, then T_i is reducible. Otherwise the $[w,h,a]$-numbers are used by the procedure DELETENODES of section 5.1.2 to decide the correct type for ROOT(T_i, S_{i+1}) and all its descendant nodes, and delete the appropriate nodes.

Computation of $[w,h,a]$-numbers

The pruned subtree is assumed to be completely built and connected by a modified version of the BUBBLE procedure of [BL76], e.g. the BUBBLEXT of section 4.1.3.

Procedure COMPUTE computes bottom up the $[w,h,a]$-number for each node X in the pruned subtree. It also finds the partial children of X that should be made Type H or A if X is made Type H or A. These children are denoted h-child1(X), h-child2(X), and a-child(X).

The $[w,h,a]$-number of a node is based on the $[w,h,a]$-numbers of its pertinent children. The computation formulas from [JTS89] are included here for the sake of completeness. $P(X)$ denotes the set of pertinent children of X and $Par(X)$ denotes its set of partial children.

I) *X is a pertinent leaf.*

$w = 1$, $h = 0$, and $a = 0$.

II) *X is a pertinent internal node.*

$w = \sum_{i \in P(X)} w_i$

(i) *X is a full node, or partial P-node with no partial children.*

$h = 0$ and $a = 0$.

(ii) *X is a partial P-node with partial children.*

- Type H:
  Make all pertinent children of X Type B, one partial child Type H, and all other partial children Type W. The partial child chosen to be Type H is made h-child1(X).

$$h = \sum_{i \in Par(X)} w_i \; - \max_{i \in Par(X)} \{(w_i - h_i)\}$$

- Type A:
  1. Make one partial child of X Type A and all other pertinent children Type W. The partial child chosen to be Type A is made a-child(X).

$$\alpha_1 = \sum_{i \in Par(X)} w_i \; - \max_{i \in Par(X)} \{(w_i - a_i)\}$$

  2. Make two partial children Type H, all full children Type B and all other partial children Type W. max1 and max2 determine the two partial children to be made Type H, h-child1(X) and h-child2(X).

$$\alpha_2 = \sum_{i \in Par(X)} w_i - \underset{i \in Par(X)}{\max 1} \{(w_i - h_i)\} - \underset{i \in Par(X)}{\max 2} \{(w_i - h_i)\}$$

  This gives $a = \min\{\alpha_1, \alpha_2\}$. If the value of a is different from α_1, a-child(X) is made empty.

(iii) *X is a partial Q-node.*

- Type H:
  X can only be made Type H if one of its endmost children is pertinent. Assume one endmost is pertinent. Then an array $P_1(X)$ is filled with a consecutive sequence of pertinent children of X, starting with the endmost pertinent child. Only the last node is allowed to be partial. If the other endmost is pertinent also, an array $P_2(X)$ is filled in the same way. The last node in the maximal array is made h-child1(X).

$$h = \sum_{i \in P(X)} w_i - \max\left\{ \sum_{i \in P_1(X)} (w_i - h_i), \sum_{i \in P_2(X)} (w_i - h_i) \right\}$$

- Type A:
  1. Make one pertinent child Type A and all other pertinent children Type W. The child chosen to be Type A will be made a-child(X).

$$\beta_1 = \sum_{i \in P(X)} w_i - \max_{i \in P(X)} \{(w_i - a_i)\}$$

  2. Let $P_A(X)$ be a maximal consecutive sequence of children of X such that the two endmost are either full or partial, and all interior nodes are full. X can then be made Type A if all full nodes in $P_A(X)$ are made Type B, any partial nodes in $P_A(X)$ Type H, and all pertinent children outside $P_A(X)$ Type W. There may be more than one $P_A(X)$. An endmost node in the selected $P_A(X)$ is made h-child2(X).

$$\beta_2 = \sum_{i \in P(X)} w_i - \max_{P_A(X)} \left\{ \sum_{i \in P_A(X)} (w_i - h_i) \right\}$$

  This gives $a = \min\{\beta_1, \beta_2\}$. If β_2 is minimum, a-child(X) is made empty.

These numbers are computed bottom up for each T_i by COMPUTE(T_i). The tree is reducible if the minimum of h and a for ROOT(T, S_i) is zero. If not, some pertinent leaves must be deleted to make it reducible.

COMPUTE(T_i) traverses at most all nodes in the pruned subtree, plus all children of all partial Q-nodes. The number of children of all Q-nodes in a PQ-tree T_i is at most i, the number of vertices embedded so far. The number

of P-nodes in a PQ-tree T_i is also at most i, and the number of pertinent leaves is $|S_{i+1}|$, the in-degree of vertex $i+1$. Thus the amount of work done by COMPUTE for one T_i is $\mathcal{O}(n + \text{in-degree}(i+1))$. The total for all T_i's are thus $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$, so COMPUTE has complexity $\mathcal{O}(n^2)$ [JTS89].

5.1.2 Deleting edges

The type of a node determines uniquely the types of its children. So when T_i is irreducible, the type of ROOT(T, S_i) is decided according to the minimum of h and a, and then DELETENODES(T_i) is called. DELETENODES traverses the pruned subtree top down, and for each node X the type of all its children is determined and the pertinent leaves made Type W deleted.

A node X is made Type B if $h = a = 0$. The subtree rooted at a node made Type B is not touched by the procedure; all its pertinent leaves are to stay in the tree. When X is made some type other than B, all pertinent children of X will be processed, and the correct type is set for each one. If X is a leaf of Type W, it is deleted from the PQ-tree, and the corresponding edge is removed from the graph.

Assume X is an internal node whose type is set. Then there are three possibilities:

1. *X is Type W.*

   All children of X are made Type W. All descendant pertinent leaves will be deleted, and the corresponding edges removed from the graph.

2. *X is Type H.*
   - (i) *X is a P-node.*

     The partial child h-child1(X) is made Type H, all other partial children Type W, and all full children Type B.
   - (ii) *X is a Q-node.*

     It is determined if h-child1(X) is part of $P_1(X)$ or $P_2(X)$, all full nodes in the chosen array are made Type B, h-child1(X) is made Type H if it is partial, and all other pertinent children of X are made Type W.

3. *X is Type A.*
   - (i) *X is a P-node.*

     If a-child(X) is not empty, that child is made Type A, and all other pertinent children are made Type W. On the other hand, if a-child(X) is empty, h-child1(X) and h-child2(X) are made Type H, all other partial children Type W, and all full children Type B.

(ii) *X is a Q-node.*
If a-child(X) is not empty, that child is made Type A and all other pertinent children are made Type W. If a-child(X) is empty, all full nodes in $P_A(X)$ are made Type B. If endmost nodes in $P_A(X)$ are partial, they are made Type H, and all other pertinent children of X are made Type W.

These rules determine the actions of procedure DELETENODES. When necessary, the number of descendant leaves for each node is updated. The leaves deleted from T_{i-1} is removed from S_i, the set corresponding to incoming edges of vertex i, and added to the set E_i'. The leaves in all sets E_i', $3 \leq i \leq n-1$ correspond to all edges in $G - G_p$.

DELETENODES(T_i) traverses most of the pruned subtree. Full nodes are not touched, neither are nodes outside the pruned subtree. Deletion and updating of nodes can be done in constant time for each node. Thus, following the same arguments used for COMPUTE, DELETENODES is an $\mathcal{O}(n^2)$ algorithm.

5.1.3 PLANARIZE

Adding these new procedures to the main loop of procedure PLANAR (section 3.2.2 on page 22) gives this outline of PLANARIZE:

```
procedure PLANARIZE(G)
begin
    construct initial tree T_1;
    for i := 2 to n − 1 do
        BUBBLEXT(T_{i−1}, S_i);
        COMPUTE(T_{i−1});
        if min{h, a} ≠ 0 for ROOT(T_{i−1}, S_i) then
            {Deletion step}
            make ROOT(T_{i−1}, S_i) Type H or A corresponding to minimum of h and a;
            DELETENODES(T_{i−1});
        fi;
        REDUCE(T_{i−1}, S_i);
        {Vertex addition step}
        make T_i by replacing full nodes with new P-node X with all outgoing edges
            of vertex i as children of X;
    od;
end;
```

PLANARIZE finds a planar, connected subgraph G_p of G, and by doing so, also constructs a sequence $T_2, T_3, \ldots, T_{n-1}$ of reducible PQ-trees.

As shown in [JTS89], G_p is not necessarily a maximal planar subgraph. Assume a pertinent leaf l is deleted by DELETENODES in T_i because some

empty leaves are between l and the maximal pertinent sequence in all permutations of T_i. The corresponding edge e is deleted from G_p in order to preserve planarity. If the empty leaves are later deleted as well, then edge e can be included in G_p without destroying planarity, and consequently, G_p is not maximal planar.

Given that PLANAR of [BL76] is $\mathcal{O}(m+n)$, and COMPUTE and DELETE-NODES both are $\mathcal{O}(n^2)$, this procedure runs in time $\mathcal{O}(n^2)$ for non planar graphs [JTS89].

5.2 Algorithms for maximal planarization

In order to find a maximal planar subgraph of G, Jayakumar et al. suggested a second phase MAXPLANARIZE, also based on PQ-trees [JTS89]. This second phase tries to find all leaves in $G - G_p$ that can be added to G_p without destroying planarity, thus creating a maximal planar subgraph $G_p^{'}$. The three versions of MAXPLANARIZE that, to our knowledge, have been published so far, are briefly described here.

5.2.1 Initial approach [JTS89]

The main goal of Jayakumar et al. was to detect leaves that were unnecessarily deleted from G by the first phase, and augment G_p into a maximal planar subgraph $G_p^{'}$, containing G_p as a subgraph. Their second phase demands that G_p be biconnected for $G_p^{'}$ to be maximal planar.

MAXPLANARIZE sets out to construct the same sequence $T_2^{'}, T_3^{'}, \ldots, T_{n-1}^{'}$ of PQ-trees for G as PLANARIZE did, but this time no such leaves as l, mentioned at the end of the previous section, are deleted from G. Leaves corresponding to edges of $G - G_p$ are ignored until they become pertinent. Nodes with only ignored children are also ignored. In a PQ-tree T_i, leaves representing incoming edges to vertex $i+1$ in G_p are called *preferred leaves* and make up the *preferred sequence*. None of these leaves will be deleted by MAXPLANARIZE. Leaves in $E_{i+1}^{'}$ are called *new pertinent leaves*, and these leaves can be added to the preferred sequence if ignored nodes are the only nodes between two preferred leaves. Such ignored nodes will then be deleted from the PQ-tree. Edges corresponding to the new pertinent leaves added to the preferred sequence are added to the planar subgraph. Hence, for each $T_i^{'}$ as many as possible of the leaves in $E_{i+1}^{'}$ are kept and reduced, in order to create a maximal planar subgraph $G_p^{'}$ of G, containing G_p as a subgraph.

If G_p is not biconnected, $G_p^{'}$ may not be maximal [JTS89]. Let vertex k have degree 1 in G_p. Its only adjacent edge will be an incoming edge. If the

corresponding leaf is the only node between a pertinent leaf l and its maximal pertinent sequence in a PQ-tree T'_{i-1}, $i < k$, then l will be deleted from T'_{i-1} to make it reducible, and the corresponding edge e will be removed from G'_p (fig. 5.1). But in the vertex addition step of T_{k-1}, no new vertex is added, meaning that l could have been made adjacent to its pertinent sequence if S_k had been reduced before S_i. Thus adding e to G'_p will not destroy planarity, and G'_p is not maximal planar.

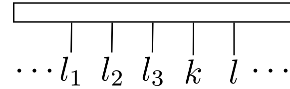


Figure 5.1: The deletion of l causes G_p to not be maximal. $l_1, l_2, l_3, l \in S_i, i < k$

5.2.2 Changes made by Kant

G. Kant showed in [Kan92] that MaxPlanarize of [JTS89] does not always produce a maximal planar subgraph even when G_p is biconnected. One problem is the fact that ignored-empty leaves have to be deleted in order to augment the preferred sequence with new pertinent leaves. In general there is a choice between different sets of ignored leaves. Arbitrarily choosing one set to keep may lead to deleting an ignored leaf that would have been adjacent to its preferred sequence in a later step.

Another problem stems from the fact that adding a new pertinent leaf to the maximal pertinent sequence of T'_i may lead to empty nodes being bound to new places, causing the following PQ-trees $T'_j, i < j < n$, of the second phase not to be equivalent any more to the PQ-trees T_j of the first phase. Accordingly, it may not be possible in some T'_j to form a consecutive sequence of the preferred leaves of the maximal pertinent sequence in T_j. This forces some preferred leaves to be deleted in order to make T'_j reducible, and G_p is no longer a subgraph of G'_p.

Kant suggests a new version of MaxPlanarize in order to solve these problems. He introduces *sequence indicators*[1] to keep track of the place of the pertinent sequence in the PQ-tree. Thereby he is able to delay the question of whether an edge can be added to G'_p without destroying planarity, until enough information is available. Leaves from $G - G_p$ are treated as normal, empty leaves until it is time for them to become pertinent in some T'_i. Instead of being made pertinent, leaves in E'_{i+1} are made *potential*. The maximal pertinent sequence of G_p is then reduced, and in the vertex addition step, if $|E'_{i+1}| > 0$, a sequence indicator $< i+1 >$ is added to T'_i adjacent to the new node. Both potential leaves and sequence indicators are ignored for the rest of their lifetime in the PQ-tree.

[1]Based on the *direction indicator* of [CNAO85].

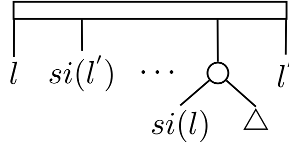


Figure 5.2: Two intersecting near pairs $l, si(l)$ and $l', si(l')$. The Q-node and all its descendants are ignored nodes.

Taking care not to bind empty nodes to new places, a sequence of PQ-trees $T_2', T_3', \ldots, T_{n-1}'$ is constructed by MaxPlanarize, equivalent to the sequence constructed by Planarize. Since all non pertinent leaves of each T_i will be present as ordinary leaves in T_i', G_p need not be biconnected.

When, in a later T_i', a potential leaf l can be made adjacent to its sequence indicator $si(l)$ in every permutation of T_i', by only deleting ignored nodes, the corresponding edge can be added to G_p without destroying planarity.

Definition 5.1 *A potential leaf l is* **near** *its sequence indicator $si(l)$, if the PQ-tree T_i, $1 \leq i < n$ can be reduced such that they are adjacent siblings, by deleting only ignored nodes and not binding empty nodes to new places. If a potential leaf l is near its sequence indicator $si(l)$, then l and $si(l)$ are called a* **near pair**.

This does not mean that every near pair can be reduced. Other potential leaves may form near pairs with $si(l)$ as well, causing l to be deleted when reducing for near pairs. Another situation is when two near pairs, $l, si(l)$ and $l', si(l')$, *intersect*, that is, in every permutation of the PQ-tree, either l or $si(l)$ are between l' and $si(l')$. See fig. 5.2. When two near pairs intersect, only one of them can be reduced [Kan92].

Leaves that form near pairs and are successfully reduced, are removed from the tree and the corresponding edges are added to G_p'. Sequence indicators stay in the tree as long as they have potential leaves and are not deleted in a reduction.

How near pairs are found and reduced is described next.

Finding and reducing near pairs

Since there, by definition, are no empty leaves in a pertinent sequence, Kant tests for near pairs only within pertinent sequences.

Each time Reduce matches a pertinent node X, the frontier of X is searched for near pairs l and $si(l)$. To reduce a found near pair, two arrays PL_X and SI_X are introduced for every internal node X in the pruned subtree.

Those children of X that are ancestors of potential leaves corresponding to incoming edges of vertex i, are placed in $PL_X[i]$. The child of X that is ancestor of sequence indicator $< i >$ is placed in $SI_X[i]$.

Since REDUCE works bottom up, all near pairs in the frontier of every pertinent child of X have been found and reduced already. Also, all potential leaves or sequence indicators that are between pertinent leaves in the frontier of a child Z of X in all permissible permutations, have been removed from the tree. Thus, PL_Z and SI_Z are up to date for every pertinent child Z of X. It can be shown that all near pairs will be found by using the PL_X and SI_X arrays [Kan92].

Reduction of near pairs is done top down. When a near pair is found, if X is a P-node, the two children of X that are ancestors of l and $si(l)$ are placed as children of a new Q-node, to ensure that the reduction of the near pair starts with a Q-node.

The procedure reducing near pairs is here called REDUCENEAR. Other near pairs, not intersecting with the one that is being reduced, are reduced first. Other potential leaves l_i, also forming a near pair with $si(l)$, are added to the planar subgraph.

Definition 5.2 *An ignored P-node is said to be of* **Type U***, if all children except one child, yet unknown, must be removed with their descendants from the PQ-tree in a later step. An ignored Q-node is said to be of Type U, if it has one special marked child Y, and all children of the Q-node between Y and one of the endmost children, yet unknown, must be removed with their descendants from the PQ-tree in a later step.*

A Type U node is used to delay the question of which set to keep when exactly one arbitrary set of ignored nodes can be kept in the tree. The reduction process is briefly described. After a near pair $l, si(l)$ corresponding to an incoming edge of vertex i is found, the following situation occurs: The first common ancestor of l and $si(l)$ is a Q-node X. Let node Y_1 be child of X and ancestor of l, let Y_k, $k \geq 2$, be another child of X and ancestor of $si(l)$, and let $Y_2, \ldots, Y_{k-1}$ be the sequence of ignored nodes between them. REDUCENEAR first removes the children $Y_2, \ldots, Y_{k-1}$ and their descendants from the tree. Any deleted leaf l_i corresponding to an incoming edge of vertex i form a near pair with $si(l)$, and is added to the graph. Then REDUCENEAR goes along the path from Y_1 to l applying a top down reduction. Any leaves l_i that form a near pair with $si(l)$, are added to the graph. The same is done with the path from Y_k to $si(l)$. If there are other near pairs in the frontier of Y_1 and Y_k, they are reduced correspondingly. In the end, all ignored nodes between l and $si(l)$ are removed from the PQ-tree, and arrays PL and SI

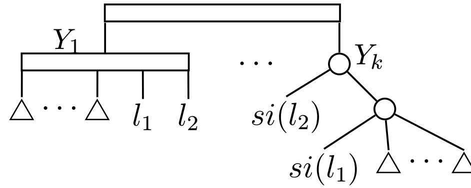


Figure 5.3: Two non intersecting near pairs, part of a consecutive sequence. Reducting $l_1, si(l_1)$ first leads to the deletion of l_2, so that $l_2, si(l_2)$ cannot be reduced after the reduction of $l_1, si(l_1)$.

are updated. Any leaf thus deleted corresponding to an incoming edge of i is added to G_p.

Proof that these changes are within the time bound of $\mathcal{O}(n^2)$ is presented in [Kan92].

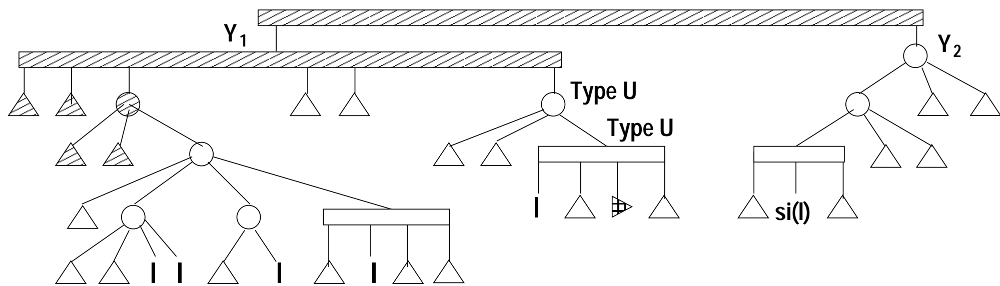
5.2.3 Problems discovered by Leipert

S. Leipert [Lei96] found that REDUCENEAR does not reduce near pairs correctly, due to the following problems.

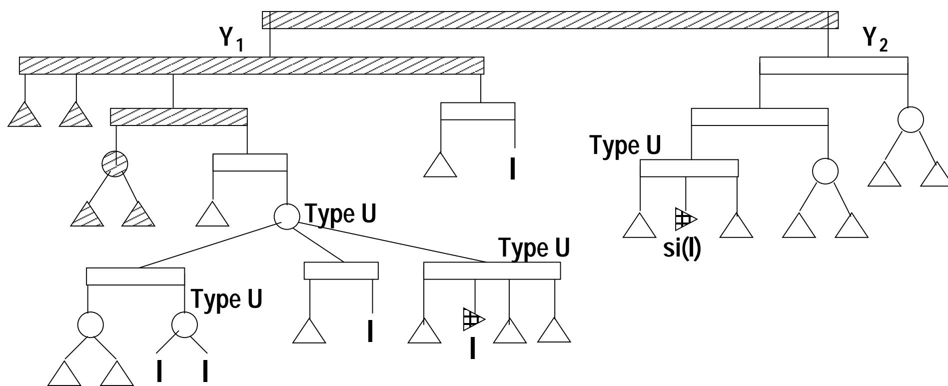
1. If one of $Y_2, \ldots, Y_{k-1}$ is a Type U node, adding all edges corresponding to leaves l_i to the graph will create non planarity.

2. The top down reduction does not restrict the permissible permutations in such a way that l and $si(l)$ form a consecutive sequence in all permissible permutations of the PQ-tree. Fig. 5.4 give an example of this, taken from [Lei96]. Hence the top down reduction performed by REDUCENEAR of [Kan92] may lead to wrong conclusion about the presence of near pairs.

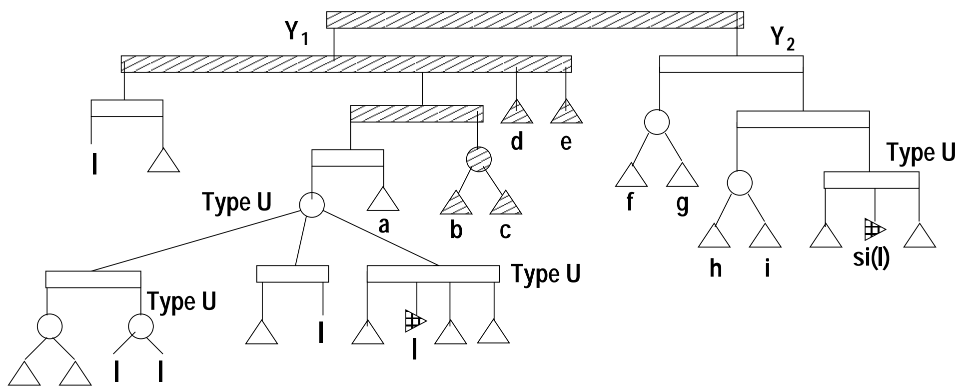
3. There may exist near pairs between l and $si(l)$ in the frontier of their least common parent, near pairs that will not be detected because one part of the pair is deleted before the other one is found. An example from [Lei96] is given in fig. 5.3. Obviously, both near pairs can be reduced, but reducing $l_1, si(l_1)$ first causes l_2 to be deleted, making it impossible to reduce $l_2, si(l_2)$ afterwards. The edge corresponding to l_2 will therefore not be added to G_p, and G_p will not be maximal planar.



(a) Before the reduction.



(b) After the reduction.



(c) A permissible permutation of the tree in (b), where nodes a, b, c, d, e, f, g, h, and i are between the sequence indicator $si(l)$ and all potential leaves l.

Figure 5.4: Example of top down reduction of near pairs suggested in [Kan92]. All potential leaves denoted as l form a near pair with the sequence indicator $si(l)$. Other ignored nodes are drawn white. Full nodes are shaded. ⊞▷ marks the "middle" of a Type U Q-node. Example is taken from [Lei96].

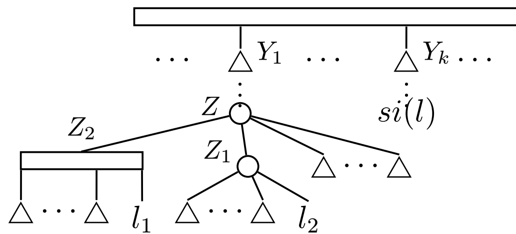


Figure 5.5: Only one of the potential leaves l_1 and l_2 can be reduced.

Suggested solutions

The first problem demands that a maximal subset of the potential leaves forming near pairs with $si(l)$ is found, which guarantees that all leaves of the subset can be reduced together. This can be solved by following the steps of ReduceNear as before, reducing the near pairs in the frontier of $Y_1, Y_2, \ldots, Y_k$. But when a Type U node is encountered, only one set of the children is kept. Furthermore, when an ordinary P- or Q-node, with children that have both ignored and pertinent children, is encountered, a choice resembling the choice made for Type U nodes will have to be made, so that no reduced leaf can cause non planarity in the graph. See fig. 5.5.

For the second problem, the reduction of a near pair $l, si(l)$ has to make l and $si(l)$ a consecutive sequence in all permissible permutations. This is exactly what the templates of [BL76] do. But since the reduction for near pairs involves not two, but four types of nodes, new templates must be introduced. The four types are as follows: Pertinent nodes, ignored nodes with part of the near pair as a descendant (*ignored-pertinent* nodes), ignored nodes with no such descendants (*ignored-empty* nodes), and full nodes with ignored-pertinent leaves as descendants (*pertinent-partial* nodes). In addition, an ignored-pertinent node may be *ignored-partial* if it has both ignored-empty and ignored-pertinent children. New templates must be found for all new situations that may occur, some of them will also involve Type U nodes [Lei96].

The third problem turns out to be most difficult. An ordering of near pairs, if more than one is present, has to be found, such that the reduction of one near pair does not hinder the reduction of the others. The fact that near pairs may intersect, aggravates the possibilities of computing a suitable order of reductions, and no solution to this problem is presented by Leipert.

According to [JLM96], the suggested solutions to the first two problems are far beyond any reasonable implementation.

5.2.4 An attack on the approach of [JTS89, Kan92]

In [JLM96] two major problems with the approach of [JTS89] are detected, both based on the use of the st-numbering of G. In [JTS89], a biconnected G_p was a condition for MaxPlanarize to work correctly. But Jayakumar et al. did not compute a new st-numbering for G_p, in order to be able to construct the same sequence of PQ-trees as in the first phase. The st-numbering of G is used instead, a numbering that is not necessarily a legal st-numbering for G_p. An st-numbering for G does not imply an st-numbering for a biconnected subgraph of G. All outgoing edges of a vertex in G may have been deleted by Planarize, even if G_p is biconnected. The same argument Jayakumar et al. used to show that G_p had to be biconnected for their algorithm MaxPlanarize to find a maximal planar subgraph (section 5.1.3, fig. 5.1) is used in [JLM96] to show that G_p must have a legal st-numbering.

In fig. 5.6, part of a reduction step of MaxPlanarize for a biconnected G_p is shown. When the incoming vertices of vertex l, $l < k$, are reduced, leaves k_1 and k_2 are between l and its maximal pertinent sequence in all permutations of the PQ-tree, so l is deleted. Assume vertex k has no outgoing edges in G_p. Thus, the new node added to T_k will be an ignored node. If this node and all its children, representing the outgoing edges of vertex k in G, are later deleted from the tree, then edge e_l can be added to G_p' without destroying planarity. Accordingly, G_p' is not maximal planar.

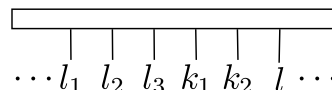


Figure 5.6: MaxPlanarize of [JTS89] does not work when the st-numbering is not legal for G_p.

Kant seems to avoid the problem of a legal st-numbering for G_p since G is the input graph to his version of MaxPlanarize, and leaves such as l are kept in the graph as potential leaves. But as already pointed out, his version of MaxPlanarize has other flaws.

The second problem presented by [JLM96] is not so easily solved. Lemma 1 (page 19) gives the invariant that allowed [LEC67] to test a graph for planarity by constructing a sequence of bush forms $B_i, 1 \leq i \leq n$. The lemma states that in a given embedding of a planar, st-numbered graph G, all incoming edges to vertex $i+1$ are embedded in the outer face of the induced subgraph G_i. This invariant is not obeyed by MaxPlanarize. To find a maximal planar subgraph, some edges may have to be embedded in an inner face of some G_i. MaxPlanarize will not detect this. It only embeds edges that can be embedded in the outer face of G_k. This problem is partly based in the question of a legal st-numbering for the obtained subgraph. None of the corrections suggested by [Kan92] or [Lei96] solves this problem. An

example of this situation is given in section 6.4, where one way of getting around it is sketched. The three problems with the reduction of near pairs in [Kan92] pointed out by [Lei96], are repeated in [JLM96], but no further solutions are presented.

5.2.5 A new approach

The aim of the next chapter is to work out a new approach, based on the additional data structure of chapter 4, which tries to avoid the problems of the approach of [JTS89, Kan92]. The presented heuristic will produce a maximal planar subgraph in a single run through the PQ-tree, except for situations matching the second problem of [JLM96] stated above.

Unfortunately, the present form of the algorithm presented in chapter 6 seems to have time complexity $\mathcal{O}(nm)$, hence no better than the straight forward approach. It is believed that further development can improve on that to give an $\mathcal{O}(n^2)$ time bound.

Chapter 6

Towards a skewness algorithm

The previous chapter showed that no algorithm based on PQ-trees, is known that produces a maximal planar subgraph for all biconnected graphs G. Such an algorithm would be a heuristic for the skewness number, so the focus of this chapter is to find such an algorithm, based on the data structure of chapters 3 and 4.

The skewness number of a graph $G(V, E)$ is the size of the minimum set E' of edges that must be removed from G to make the resulting subgraph $G'(V, E - E')$ planar. Then the remaining set of edges, $E - E'$, induces a maximum planar subgraph of G.

Chapter 4 introduced additional data structure to the PQ-tree that provides extra information about structures in the outer face of the embedded subgraph of the bush form (definition 3.2). This extra information was used in [Kar90] to exhibit a non planar subgraph homeomorphic to $K_{3,3}$ or K_5 when PLANAR [BL76], described in section 3.2.2, finds the graph to be non planar. The conclusion of that chapter, however, was that the additional templates proposed by [Kar90] are not immediately suitable for constructing a planar subgraph that is maximal.

Chapter 5 reviewed what has been done in the field of maximal planar subgraph algorithms based on PQ-trees. As the last few years have shown ([Lei96, JLM96]), the approaches made by [JTS89] and [Kan92] do not provide a correct algorithm, computing a maximal and planar subgraph for all graphs. In addition, all the corrections and modifications proposed have lead to an algorithm that is no longer implementable with reasonable efforts [JLM96].

Therefore, the aim of this chapter will be to come up with a new approach using PQ-trees, that can give a better, and hopefully more intuitive, algorithm for finding a maximal planar subgraph. As we shall see, the resulting proposal will produce a maximal planar subgraph in all but a few cases

(discussed in section 6.4). Thus, it can also act as a fairly good skewness heuristic based on *PQ*-trees. This aspect is further elaborated in the last section.

6.1 Combining PLANARIZE and `Boundary`

As mentioned in chapter 5, procedures COMPUTE and DELETENODES of section 5.1 will be used to find the leaves causing a *PQ*-tree to be irreducible. The findings of [Kan92] and [Lei96] presented in section 5.2, concerning near pairs and how they should be determined to ensure a maximal and planar subgraph, are taken advantage of. Thus, some terminology of chapter 5, as well as of chapter 3 and section 4.1, will be used in this chapter.

MAXPLANARIZE of section 5.2 is based on two passes through the *PQ*-tree. In the first phase, PLANARIZE, edges found to prevent reducibility are carelessly thrown away. Then, in the second phase, the same edges are kept as ignored leaves of the *PQ*-tree, long enough to find those that did not prevent planarity after all.

If `Boundary` is used to maintain information about edges that prevent planarity, only one pass through the *PQ*-tree is needed. This requires that no `Boundary` containing such information is lost. Deleting a leaf with only one sibling from the *PQ*-tree will cause their parent to be deleted. The sibling is either a leaf or an internal node with a `Boundary` of its own, thus making it difficult for the *PQ*-tree to keep the `Boundary` containing information about the deleted leaf. This can be solved by not deleting these leaves, and instead mark them as *ignored* right away. This enables both the current and the following *PQ*-trees to disregard these leaves. Ignored leaves will thus not prevent a *PQ*-tree from being reducible.

We will follow the terminology of [Kan92], and call ignored leaves in the *PQ*-tree *potential leaves*. A virtual edge corresponding to a potential leaf has both its tail and head vertex embedded in the subgraph, but is not embedded itself. Since it may still be added to the subgraph, the edge is not included in the set E', estimating the skewness of the graph, either. Thus virtual edges in the bush form, corresponding to potential leaves in the *PQ*-tree, will be called *stray edges*. Their virtual vertices are called *stray vertices*.

The embedded head vertex of a stray edge is referred to as its *home vertex*. The boundary element corresponding to a home vertex is called a *home element*. Again, to ensure that a `Boundary` object will maintain this information in the *PQ*-tree, a special node, called a *home node*, will be placed as a child of the new node added in the vertex addition step, if the corresponding vertex is a home vertex.

The concept of home node is the same as the sequence indicator of [Kan92], in so far as they both mark the place of the pertinent sequence and new node in the tree. The reason for introducing new terminology concerning this node, is the difference in the function they serve in the PQ-tree. The home node and corresponding potential leaf take no part in determining near pairs in SKEW. SKEW rely on `Boundary` to detect near pairs. The ignored nodes are merely kept for `Boundary` to be able to maintain the proper information. Since the `Boundary` represents parts of the outer face of the bush form, near pairs will be defined in terms of the bush form in this chapter. Definition 6.1 follows definition 5.1 of the previous chapter.

Definition 6.1 *A stray edge is* **near** *its home vertex if the edge can be embedded in the outer face of the bush form by only crossing other stray edges. Such a* **near pair** *is said to be* **embedded** *if the stray edge is added to the planar subgraph.*

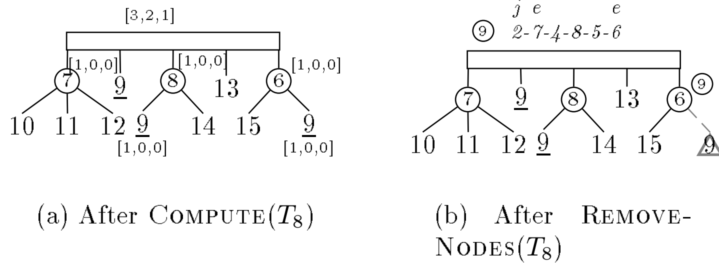
As explained in section 5.2.2, not all near pairs can be embedded. `Boundary` will have responsibility to detect near pairs and embed as many as possible.

Before the procedure finding these near pairs in the `Boundary` is presented, the procedure that marks leaves as potential and stores information in `Boundary` is described.

6.1.1 REMOVENODES

DELETENODES(T_i) of section 5.1.2 deletes all pertinent leaves made Type W, according to the [w,h,a]-numbers computed by COMPUTE(T_i) of section 5.1.1. All internal nodes thus left without children are also deleted, and the PQ-tree T_i is updated accordingly. As argued above, SKEW needs these nodes to be kept in the PQ-tree. A modified version of DELETENODES, REMOVENODES, will perform this task. In addition to updating the nodes of the PQ-tree and pruned subtree about the correct number of descendant leaves, REMOVENODES(T_i) will also update the `Boundary` of each node about potential children and descendants. In addition, a set P_{i+1} will be built, containing all pertinent leaves made potential by REMOVENODES(T_i). Nodes with only potential leaves as descendants will be ignored by the PQ-tree, and are thus marked as ignored.

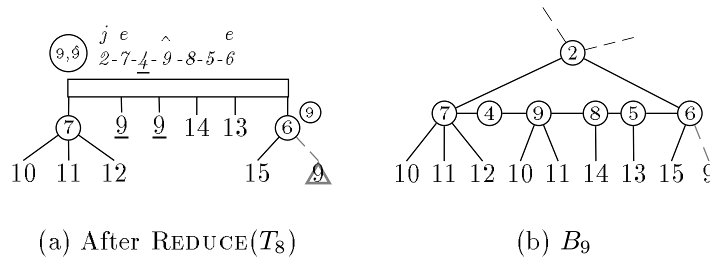
For each leaf made potential, the `Boundary` of the parent receives a pointer to this child. If parent is a Q-node or non proper P-node, the pointer is given to the corresponding element on the `Boundary`, representing the child. This element will place the pointer in a *potential set*. If the parent is a proper P-node, the pointer is also placed in a potential set, held directly by



(a) After COMPUTE(T_8)

(b) After REMOVENODES(T_8)

Figure 6.1: Pertinent subtree of T_8.



(a) After REDUCE(T_8)

(b) B_9

Figure 6.2: Pertinent subtree of T_8 after REDUCE, and bush form B_9.

the `Boundary`. The potential leaf receives a pointer to this potential set in return. The set and the leaf will follow each other for as long as they stay in the PQ-tree. No maintenance is necessary until the leaves are deleted.

Parent's `Boundary` is updated in the same way when an internal node is marked ignored. If the ignored node is a Q-node or non proper P-node, the potential sets of its boundary elements are placed in a list, that are again placed in a potential set. This set corresponds to the potential set of proper P-nodes, and it is these sets that their parent's `Boundary` receives a pointer to. Accordingly, the potential sets will reflect the structures of the PQ-tree. They may contain pointers to ignored leaves or to other sets, representing ignored children of P- or Q-nodes. A figure illustrating this structure can be found on page 87.

REMOVENODES traverses the tree top down, from ROOT(T, S_i) to the pertinent leaves to be made potential, carrying with it information about how many descendant leaves of each node will end up as potential. This information is left in the `Boundary` of every node, as a total count and a count for each i. REMOVENODES also knows when it reaches the node that will be root of the new pertinent subtree. Thus, any ancestor of this node

will be marked as ancestor of the new home node. The new root itself is not marked, since this will be done by REDUCE.

Figs. 6.1 and 6.2 give an example of how REMOVENODES works. The pertinent root in fig. 6.1(a) is not reducible, since the minimum of h and a of its $[w,h,a]$-number is not 0. a is the smaller, so the root is made Type A and REMOVENODES(T_8) called. The element added in the `Boundary` of ROOT(T_8, S_9) in fig. 6.1(b) is the count of potential descendants; there is one, corresponding to an incoming edges of vertex 9. In fig. 6.2(a), the new element, added by template $Q3$, is marked as a home element (the circumflex symbolizes a roof). This mark is actually placed in a potential set. Fig. 6.2(b) is the corresponding part of the bush form after vertex 9 has been added.

6.2 Finding near pairs

The information left by REMOVENODES gives every node complete knowledge of potential leaves and home nodes among its descendants. The question is thus how to detect all near pairs that form, and embed all edges that do not create crossings in the planar subgraph. The approach presented here will maintain the bush form as an invariant, never creating a PQ-tree that does not match the corresponding bush form. This is necessary for the boundary to maintain information and decide when near pairs are formed and can be embedded.

6.2.1 A closer look at the bush form

The bush form is defined to be a planar drawing, with all virtual vertices drawn in the outer face. Thus edges must be deleted from the bush form when they create crossings.

When a home vertex j is made internal to a biconnected component, it cannot form a near pair with any of its stray edges anymore. Accordingly, all stray vertices labeled j are deleted from the bush form together with the corresponding edges. If any stray vertices are caught inside the sequence of virtual vertices labeled i in B_{i-1}, like the stray vertex labeled k in fig. 6.3, they are deleted to avoid edge crossings. And when a near pair is embedded, all stray edges thereby crossed are also deleted.

This gives all situations that cause stray edges to be deleted from the bush form, and thereby from the planar subgraph. Disregarding the embedding of near pairs, only the embedding of new vertices is left. As long as the maximal pertinent sequence of B_{i-1} consists of at least two virtual vertices labeled i, embedding vertex i to form the next bush form B_i will create *new*

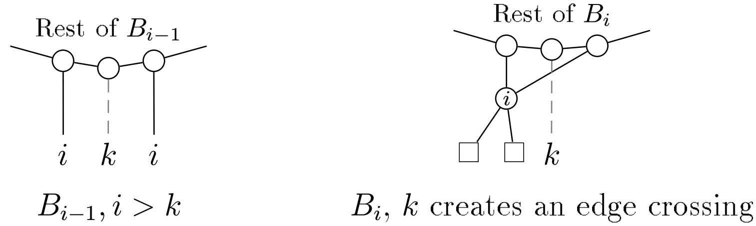


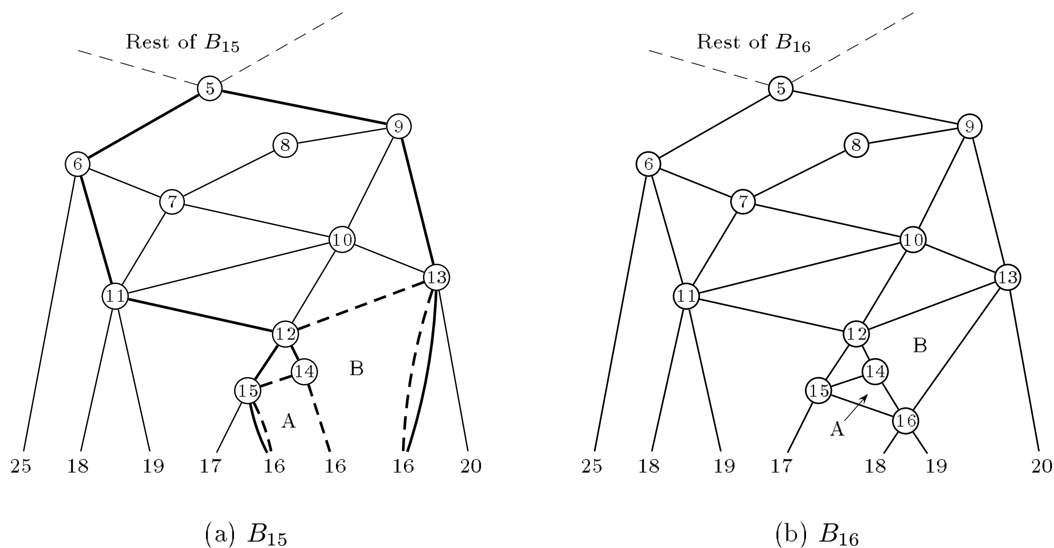
Figure 6.3: Situation causing a stray edge to be deleted.
The stray edge, drawn as a dashed, gray line, causes an edge crossing after vertex i is embedded. The stray edge corresponds to a potential leaf in the PQ-tree T_{i-1}, interior to the pertinent sequence in all permutations of T_{i-1}. The potential leaf will be deleted form T_{i-1} in the vertex addition step. The squares are virtual vertices whose labels are indifferent to the example.

inner faces in the embedded subgraph. Fig. 6.4 shows an example of such new inner faces.

All near pairs can be found by checking only new inner faces. In the PQ-tree, this corresponds to only checking for near pairs within the pertinent sequence of the `Boundary`. To make this task a little simpler, one exception is made. If RemoveNodes detects that a near pair is present in the subtree rooted at a node made ignored, it will be embedded right away, instead of waiting until the root of this subtree becomes part of a pertinent sequence.

As stated above, whenever a home vertex is made internal to a biconnected component, the corresponding stray edges are removed from the bush form. In the PQ-tree, this happens when home nodes are interior to a pertinent sequence, and are deleted in the vertex addition step. When the home node labeled i is deleted, the set P_i, built by RemoveNodes(T_{i-1}), is emptied. Each leaf that were in P_i is removed from the PQ-tree, and the corresponding edges added to E', the set estimating the skewness of the graph. The information in the potential sets holding pointers to the deleted leaves can be updated thanks to the reverse pointer set by RemoveNodes.

The last bush form created when embedding a graph, is B_n, a planar embedding of the graph itself. When testing for planarity, the PQ-trees are only constructed as far as T_{n-1}; it is reducible by default. To be able to detect near pairs forming in the new inner faces created by embedding vertex n, and in the outer face of the planar subgraph, all boundaries of T_{n-1} must be checked for near pairs as well.



(a) B_{15}

(b) B_{16}

Figure 6.4: New inner faces.
A and B mark the new inner faces of the biconnected component created by embedding vertex 16. In (a), bold lines mark the new outer mesh, stapled lines the new inner faces.

6.2.2 A closer look at `Boundary`

RemoveNodes provides `Boundary` with information about potential leaves and home nodes. To locate near pairs, the boundary must search the new inner faces created. Some small changes to the procedures of section 4.2.1 will add to the pertinent sequence enough elements to represents the vertices bounding each new inner face.

Fig. 6.5 shows the extended boundary of the new biconnected component created in fig. 6.4 with all vertices of the outer mesh represented. Since idle elements represent vertices that are indifferent to the existence of near pairs, they do not have to be represented in the `Boundary` any more. Thus, no chains will need to be inserted either. A natural way to obtain the double occurrence of elements, as element 12 of fig. 6.5(a), is to disconnect the doubly linked list of boundary elements for partial boundaries during a reduction. The two endmost elements will both represent the joint. These two elements will also be the empty and full end of the partial boundary, and automatically give the double occurrences needed to represent new inner faces. This should of course not be done for the `Boundary` of the node

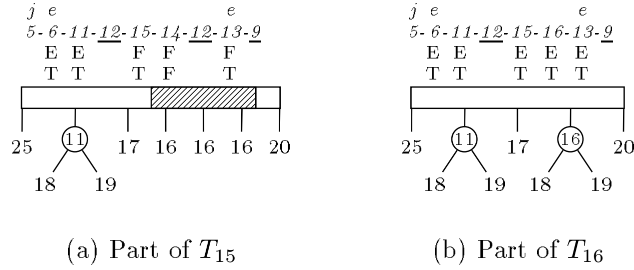


Figure 6.5: Q-nodes and boundaries corresponding to bush forms in fig. 6.4. In (a), all nodes on the new outer mesh is present, causing two occurrences of element 12. (b) is the Q-node after vertex addition. Multiple occurrences of element 12 were removed when the new element was added.

that is ROOT(T, S_i), as this `Boundary`, after vertex addition, should again represent only the outer mesh.

Embedding of near pairs

New inner faces are checked for near pairs as they are created by the template matchings in REDUCE. A procedure SEARCHNEAR walks the path defining a new inner face, from one pertinent element to the next. The pertinent child of both pertinent elements have already been checked for near pairs, and all potential leaves and home vertices interior to the new inner faces have been deleted. The potential sets of the pertinent elements are therefore up to date, containing no elements that are prevented from forming near pairs.

Any potential set held by elements encountered by SEARCHNEAR on this walk, is put on a stack. A list is maintained of how many potential leaves are found, and what vertex they are incoming to. This count has been set by REMOVENODES. When a home element $\hat{x}$ is found, the list is checked to see if a near pair can be found. If the entry for x in the list is not zero, one or more near pairs can be formed. Potential sets are popped off the stack until one containing x is found. If elements without any occurrences of x is popped before all x's are found, all potential leaves in these sets are deleted. They correspond to stray edges being crossed by the embedding of near pairs. The corresponding edges are deleted forever from the subgraph, and added to E', the set of edges estimating the skewness number.

Fig. 6.6(a) gives an example of a new inner face in a `Boundary` containing near pairs. u and z are perinent elements whose pertinent descendants define a new inner face. SEARCHNEAR starts up in u. The potential set held by

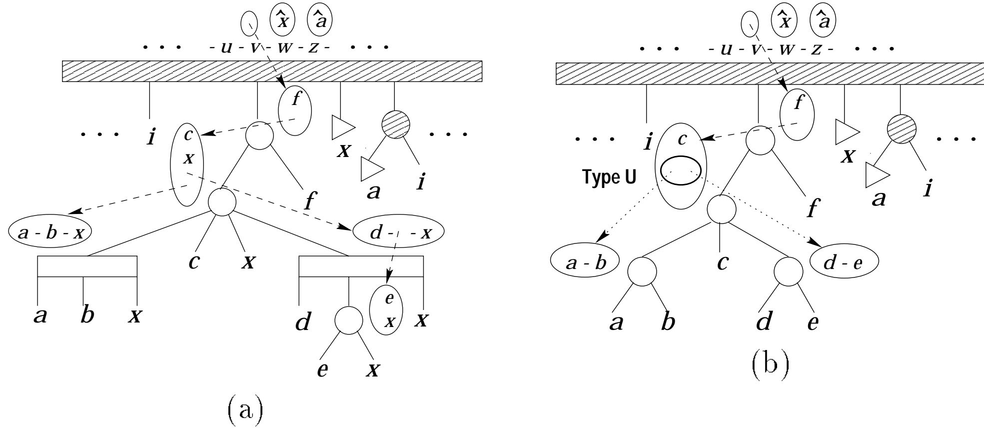


Figure 6.6: A new inner face is checked for near pairs in T_{i-1}. The boundary is not complete, only interesting part is shown. The two leaves labeled i are pertinent, defining a new inner face. All other descendants of the Q-node included in the figure are ignored. Home vertices are represented by $\triangleright$. Ovals represent potential sets. Bold lines mark Type U sets.

element v is put on the stack. The next element encountered is w, holding home element $\hat{x}$. Obviously, some leaves preventing the potential leaves labeled x from being consequtive with the maximal pertinent sequence, have been deleted. The probable cause is that they became potential themselves, and were deleted when their home nodes were deleted. The corresponding element on the `Boundary` is therefore idle, and not interesting anymore. The list kept by SEARCHNEAR counts four occurrences of potential leaves that can form a near pair with $\hat{x}$. The top element of the stack, containing only g, is popped. The corresponding potential leaf is removed from the PQ-tree, and the edge added to E'. The next potential set on the stack contains all four occurrences of x. A procedure EMBED is called. EMBED will make sure that all embeddable near pairs are found, and that the potential sets are updated correctly, so that only those potential leaves corresponding to edges that will create crossings in the embedded subgraph, are deleted.

All potential leaves that belong to the same home node and are descendants of the same ignored node, will form a near pair if one of them does. The challenge is to find the potential leaves corresponding to stray edges that do not create crossings when these near pairs are embedded. Such leaves need not be deleted. In fig. 6.6(a), when all potential leaves labeled x forming near pairs with $\hat{x}$ have been embedded, only potential leaves from one of the

sets $a - b$ and $d - e$ can be embedded without creating a crossing. Which set that should be kept cannot be decided at this point.

The idea behind the Type U node (definition 5.2) introduced by [Kan92] is used to solve this problem. Here a special set is defined, called a *Type U set* (U stands for unknown).

Definition 6.2 *A Type U set has at least two elements*[1]*. These elements are potential sets, containing one or more elements themselves. Only elements from one of these potential sets can be embedded, and when one is chosen, all other potential sets in the Type U set is deleted.*

A Type U set is used when a choice between two or more potential sets must be taken, without the appropriate information present. A Type U set is, among other matters, used when interior children of an ignored Q-node is embedded. One side of the Q-node will end up on the outer face, but which one cannot be decided yet. The two sequences obtained by walking from the endmost children towards the potential children forming near pairs, will make up the list for each of the two elements of the Type U set. When a Type U set is encountered by EMBED, only one of the sets it contains can be used. The others are discarded, and the contained potential leaves with them.

After EMBED has embedded all near pairs formed with home node $\hat{x}$, the sets held by the ignored nodes look like fig. 6.6(b). The last element checked by SEARCHNEAR is z. A new home node, a, is encountered. The potential set of element v is still on the stack, and the count still shows one potential leaf marked a. When the set containing a is found and removed, the other potential set of the Type U set is deleted.

Since z is pertinent, SEARCHNEAR stops searching for near pairs, it has reached the end of the path defining the new inner face. All potential sets still held by the potential elements between u and z are deleted. They will all be internal to the new inner face determined by the pertinent descendants of u and z. Since a near pair has formed with home node a inside the new inner face, vertex a, as well as vertex x, will be internal to the biconnected component after the vertex addition step. All leaves in P_a and P_x are deleted form the PQ-tree, and the corresponding edges added to E'. The corresponding potential sets are updated, and if left empty, deleted along wiht the corresponding idle element.

When SEARCHNEAR has checked the new inner faces on both sides of an interior pertinent element holding a potential set, the potential set is deleted.

[1] The word 'element' of definition 6.2 refers strictly to the set elements. It is not to be confused with the boundary elements of the `Boundary`.

It cannot form any more near pairs. If, on the other hand, the element is endmost in the pertinent sequence, what is left of the potential set is kept. The contents may still form near pairs in later reductions.

To ensure that all near pairs are embedded correctly, EMBED can also be called by REMOVENODES. If a node made ignored by REMOVENODES is least common parent of a potential leaf and its home node, they form a near pair that should be embedded right away. In such cases, EMBED is called to embed this near pair, delete the necessary potential leaves, and update the potential sets. This way, no potential sets representing an ignored subtree contain near pairs when encountered by SEARCHNEAR.

EMBED

EMBED descends through the potential set and all its subsets, subtracting all leaves that form a near pair with the given home element, creating new Type U sets when necessary, deleting the appropriate potential leaves, and updating the counts of potential descendant leaves. The last task can easily be done if EMBED makes a list of what is deleted. After EMBED is finished, SEARCHNEAR can update the count according to this list.

6.3 Skewness algorithm

Since most of the new procedures determining the planar subgraph is called by REDUCE and the template procedures of the `Boundary`, there is not much difference between the outline of SKEW and the outline of PLANARIZE of [JTS89] given on page 68. The main difference lies in the extra work done by REMOVENODES, REDUCE, and `Boundary` to obtain a maximal planar subgraph in one run through the PQ-tree.

algorithm SKEW(G)
begin
construct initial tree T_1;
for $i = 2$ **to** n **do**
BUBBLEXT(T_{i-1}, S_i);
COMPUTE(T_{i-1});
if $\min\{h, a\} \neq 0$ for ROOT(T_{i-1}, S_i) **then**
{Deletion step}
make ROOT(T_{i-1}, S_i) Type H or A corresponding to minimum of h and a;
REMOVENODES(T_{i-1});
fi;
REDUCE(T_{i-1}, S_i);
{Vertex addition step}
make T_i by replacing full nodes with new P-node X with all outgoing edges

of vertex i as children of X;
od;
end;

After BUBBLEXT has built the complete pruned subtree, COMPUTE of [JTS89] computes the [w,h,a]-numbers, ignoring the presence of ignored nodes in the PQ-tree. If the PQ-tree is not reducible, the type of the root of the pertinent subtree is descided, and REMOVENODES is called. If $h = a > 0$, no specific rule is given in [JTS89] of what type to make ROOT(T, S_i). Our testing indicates that Type A is to prefer in such cases. It seems that the `Boundary` will provide better information this way.

The necessary leaves are made potential by REMOVENODES, the potential sets updated, and the set P_i built. If a near pair is detected, EMBED is called. During REDUCE, all new inner faces are checked for near pairs. The template procedures called in the `Boundary` of each node takes care of this.

There has not been enough time to develop every detail of SKEW. When this is done, and the result tested, better solutions than those proposed for the information keeping and determining of near pairs, may be found. But those proposed here will work, and in deed find a maximal planar subgraph for most graphs.

6.3.1 Time complexity

None of the procedures called directly from SKEW takes more than $\mathcal{O}(n^2)$ time. This has been stated earlier for BUBBLEXT (section 4.1.3), COMPUTE (section 5.1.1) and REDUCE (section 3.1.2). The additional updating of nodes performed by REMOVENODES takes constant time for each node. Thus, REMOVENODES is $\mathcal{O}(n^2)$, just as DELETENODES (section 5.1.2).

SEARCHNEAR and EMBED are the only ones that can disturbe the complexity of $\mathcal{O}(n^2)$. All searching for, and embedding of near pairs are performed by these procedures. The number of faces of an embedded planar graph is $\mathcal{O}(n)$. So SEARCHNEAR can be called at most $\mathcal{O}(n)$ times. The total number of existing near pairs has an upper bound of $\mathcal{O}(m)$. Hence, EMBED can be called at most $\mathcal{O}(m)$ times. Without further analysis, this gives an upper time bound of $\mathcal{O}(nm)$, the same complexity as the straight forward algorithm for maximal planar subgraph problem. This analysis is not complete.

The sketched approach of SEARCHNEAR and EMBED are intuitive first ideas on how to detect and embed near pairs in the `Boundary`. Accordingly, the preferred overall timebound of $\mathcal{O}(n^2)$ may still be obtainable, and it is my conjecture, given the resemblance with the $\mathcal{O}(n^2)$ data structure and algorithm of [Kan92], that this is possible. (See also section 7.1).

6.4 Remaining problems

Since SKEW is a one-phase algorithm, the problems of [JTS89, Kan92] based in the two phases of MAXPLANARIZE are avoided. The Type U sets ensures that no edge is faulty included in the planar subgraph. All near pairs are found and embedded by SEARCHNEAR and EMBED.

But the second problem of [JLM96] still remains. An example is given in figs. 6.7 and 6.8. A bush form B_{k-1} is given in fig. 6.7, where only three of the four virtual vertices labeled k can be made consecutive. The forth is thus not embedded when the new vertex is added. The corresponding PQ-tree T_{k-1} is given in fig. 6.8. Assume that when all vertices of G has been embedded in the planar subgraph, there is a path from at least one of the vertices causing k_4 not to be embedded in B_{k-1}, to vertex n. Thus, SKEW cannot have embedded the corresponding edge in the planar subgraph, since this edge and the corresponding home vertex have not been part of the same new inner face, or at any point formed a near pair. Assume further that no outgoing edges of vertex k is embedded in the planar subgraph. This means that the deleted edge, incoming to vertex k, can be made adjacent to vertex k when the structures of the biconnected component are as in fig. 6.7. Vertex k is made part of the same old inner face as the tail vertex of the deleted edge, by flipping part of the biconnected component. This enables the deleted edge to be embedded in this face. Fig. 6.7(b) shows how this would look in bush form B_k.

It seems that this situation cannot be detected by adding a reasonable amount of extra information to the PQ-tree. But the situation is easy to detect after SKEW has finished. A vertex in G that has no outgoing edges in G_p, as well as deleted incoming edges, may resemble the situation of k in fig. 6.7. Some of these incoming edges may thus be embeddable in G_p without destroying planarity. To see if that is the case, they can be added to G_p, creating G_p'. G_p' is given a new st-numbering, and its biconnected blocks can then be tested for planarity, or run through SKEW again. How much a second run through SKEW will improve the result, has not been analysed. But it seems that the situation of fig. 6.7 is so rare that the number of edges added to G_p' will be small. Accordingly, a maximal planar subgraph can be efficiently obtained from G_p by testing if planarity is preserved for one edge at a time. Improvement is possible if some heuristic can be found that estimates the optimal order in which to test these edges.

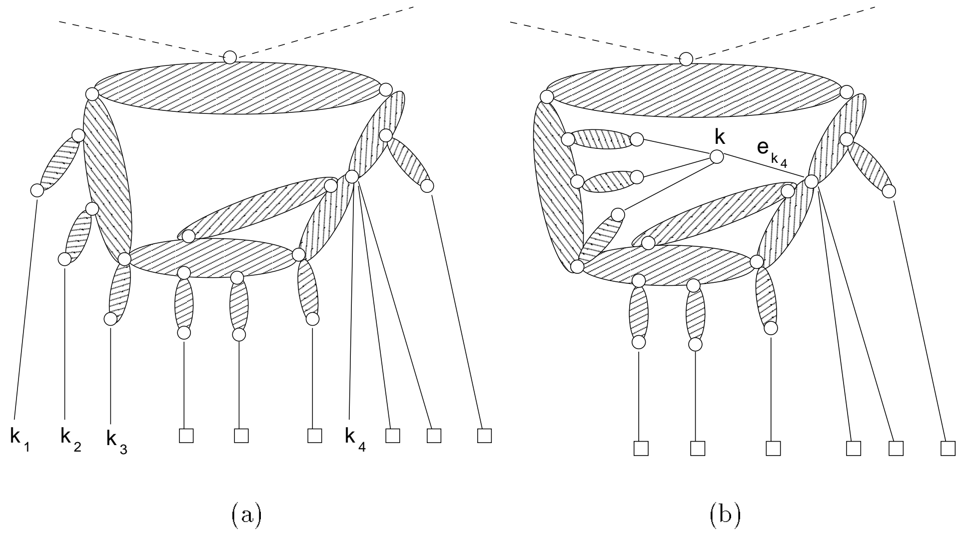


Figure 6.7: Example of embeddable edge not discovered by the PQ-tree. Shaded areas are embedded biconnected components.

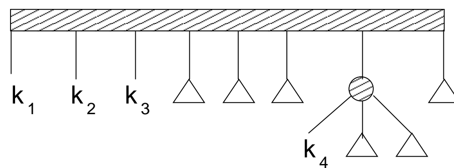


Figure 6.8: Pertinent Q-node corresponding to biconnected component of fig. 6.7.

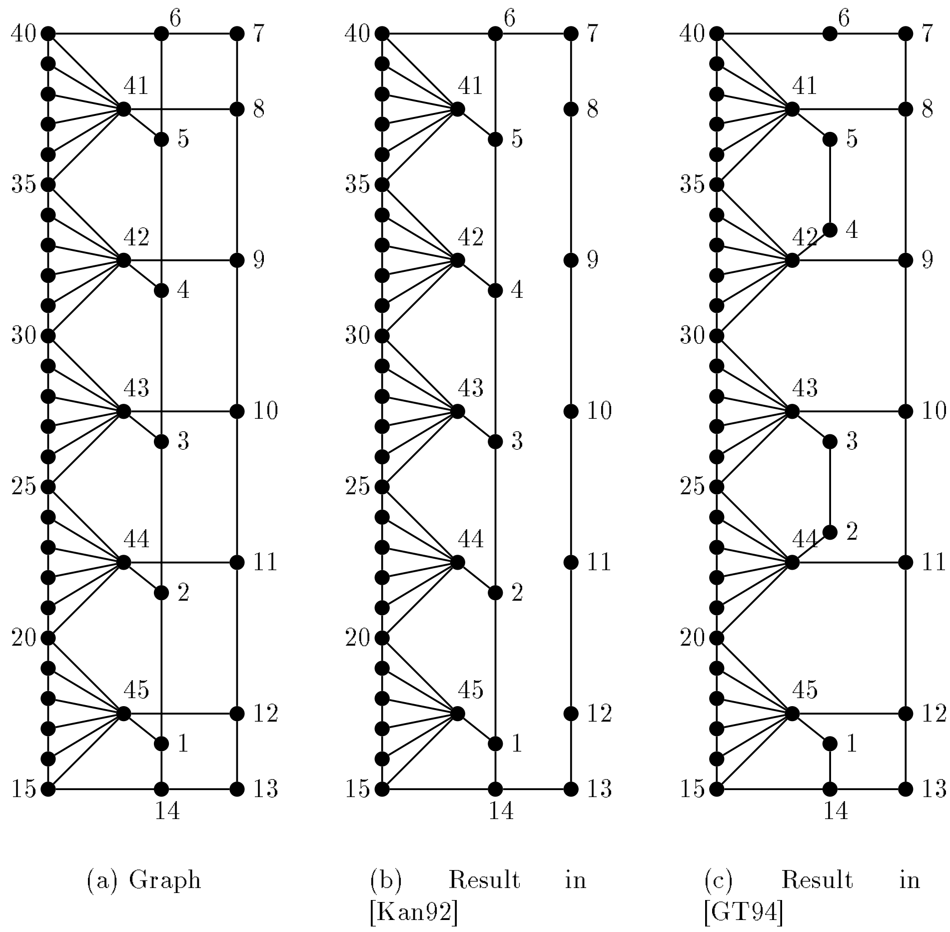
6.4.1 In search for perfection

The planar subgraph determined by SKEW is not necessarily very close to the optimum, maximal or not. Some additional approaches are suggested, that will improve on the size of the obtained subgraph, if not on the simplicity of SKEW. Deleting a pertinent leaf in the maximal pertinent sequence might lead to additional near pairs being embedded. This actually corresponds to deleting empty leaves from an irreducible PQ-tree. As pointed out by Jayakumar et al. [JTS89], this cannot be done uncritically. If all pertinent leaves are deleted, there is nowhere to add the new node, and the subgraph may end up disconnected. But when several new inner faces form, they can be joined. Always keeping the two endmost pertinent nodes ensures that the new node can be added, and that the structures in the outer face of the bush form remain unaltered. This is important for maintaining the bush form as an invariant to the PQ-tree. An efficient way to decide if an edge should be deleted in favor of a set of embeddable near pairs, will thus have to be found.

The tests performed with OBSTRUCTIONS and Cases of chapter 4, showed that edge $(1, n)$ often blocks several other edges from being embedded. This implies that testing several different st-numberings of a graph may pay off. Since the edge (s, t) can be an argument to the st-numbering algorithm, rules determining smart choices for this edge may be developed.

There is another aspect of the PQ-tree where the choice of st-numbering can strongly influence the result. Once an edge has been embedded in the subgraph, it cannot be removed by the PQ-tree. If an edge, essential in the minimum sets of edges that can be removed, is embedded before the graph is discovered to be non planar, then a minimum set cannot be discovered, and the estimate will not be optimum. A good example is an example graph from [Kan92], used there to demonstrate the superiority of his version of MAXPLANARIZE over the algorithm of [DT89] and an earlier version of the algorithm of [CHT93]. The graph is given in fig. 6.9(a).

Fig. 6.9(b) gives the result of MAXPLANARIZE. Five edges have been deleted from the graph, $(8, 41), (9, 42), (10, 43), (11, 44)$, and $(12, 45)$. In [GT94], only three edges are deleted from this graph, determining the skewness number to be 3. There are several sets of size three that will make the resulting graph planar. In fig. 6.9(c), edges $(1, 2), (3, 4)$, and $(5, 6)$ have been removed. Another set is $(1, 14), (2, 3)$, and $(4, 5)$. We have reached the conclusion that at least one of edges $(1, 2), (2, 3)$, and $(3, 4)$ is contained in all minimum sets for this graph. Since the PQ-tree cannot discover non planarity before at least four vertices have been embedded, these sets cannot be determined by the present st-numbering. Accordingly, the estimate for the skewness number, and the obtained planar subgraph, may be increased



(a) Graph (b) Result in [Kan92] (c) Result in [GT94]

Figure 6.9: Example graph from [Kan92] and [GT94]. (c) is a maximum planar subgraph of the graph in (a)

by running the graph through the PQ-tree more than once, with different st-numberings.

Chapter 7

Conclusion

If I were to write this thesis all over again, both the focus and the result would probably have been different. The new results of [JLM96] came to my attention too late to have a larger impact on this thesis. As it is, the focus of this thesis has been on exploring the possibilities that the PQ-tree data structure gives, in order to design a heuristic for solving the maximal planar subgraph problem and, ultimately, the skewness problem. My own research as well as that of [JTS89, Kan92, Lei96, JLM96] has shown that the PQ tree data structure may not be the structure of choice when attempting to solve those problems.

A survey of previously published articles attempting to solve the maximal planar subgraph problem is given. My augmentation of the PQ-tree data structure is described and partially implemented. This part of the thesis ended up being much smaller than originally planed, in favor of a more theoretical emphasis. Finally, a new approach for a heuristic to the maximal planar subgraph and skewness number heuristic is given.

7.1 Further work

The algorithm proposed in chapter 6 has not been implemented and tested. It is my belief that, if time for a more thorough analysis of how to organize the information and search for near pairs had been available, a version of SKEW with better complexity could be obtained. Implementing the algorithm and performing tests can determine how well this approach can compete with other algorithms such as [CHT93, GT94], both in the time complexity and in the number of deleted edges.

The OBSTRUCTIONS algorithm of chapter 4 can be extended into a planarizing algorithm. With extensive testing, it may very well prove to be

a good and simple heuristic for the skewness number for some families of graphs.

Section 6.4 presented some thoughts on how the chosen st-numbering may inflict on the results of MAXPLANARIZE and SKEW. Further analysis of this may give interesting results that can provide new approaches to the maximal planar subgraph and skewness problems, based on the PQ-tree data structure.

Bibliography

[BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using *PQ*-tree algorithms. *J. Comput. System Sci*, 13(3):335–379, 1976.

[CHT93] J. Cai, X. Han, and R. E. Tarjan. An $\mathcal{O}(m \log n)$-time algorithm for the maximal planar subgraph problem. *SIAM J. Comput.*, 22(6):1142–1162, 1993.

[Cim95] R. Cimikowski. On heuristics for determining the thickness of a graph. *Information Sciences*, 85(1-3):87–98, 1995.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using *PQ*-trees. *J. Comput. System Sci.*, 30(1):54–76, 1985.

[DT89] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th Annual Symposium an Foundations of Computer Science*, pages 436–441. IEEE Comput. Soc. Press, 1989.

[ET76] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2(3):339–344, September 1976.

[ET77] S. Even and R. E. Tarjan. Corrigendum: Computing an *st*-numbering. *Theoretical Computer Science*, 4(1):123, February 1977.

[Eve79] S. Even. *Graph Algorithms*. Pitman, 1979.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman & Co, New York, 1979.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is $\mathcal{NP}$-complete. *SIAM J. Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[Gri94] R. P. Grimaldi. *Discrete and Combinatorial Mathematics, An Applied Introduction*. Addison-Wesley Publishing Co., 3 edition, 1994.

[GT94] O. Goldschmidt and A. Takvorian. An efficient graph planarization two-phase heuristic. *Networks*, 24:69–73, 1994.

[Guy72] R. K. Guy. Crossing number of graphs. In Y. Alavi, D. R. Lick, and A. T. White, editors, *Proc. Graph Theory and Applications: Western Michigan University, May, 1972*, volume 303 of *Lecture Notes in Mathematics*, pages 111–124. Springer Verlag, 1972.

[Har69] F. Harary. *Graph Theory*. Addison-Wesley Publishing Company, 1969.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. of the Association for Computing Machinery*, 21(4):549–568, 1974.

[JLM96] M. Jünger, S. Leipert, and P. Mutzel. On computing a maximal planar subgraph using *PQ*-trees. Technical report, Informatik, Universität zu Köln, 1996. `ftp://ftp.zpr.uni-koeln.de/pub/paper/zpr96-227.ps.gz`.

[Joh98] J.-E. Bye Johansen. Master Thesis. To appear. Implements the embedding algorithm of [CNAO85], May 1998.

[JTS89] R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy. $\mathcal{O}(n^2)$ algorithms for graph planarization. *IEEE Trans. on Computer-Aided Design*, 8(3):257–267, 1989.

[Kan92] G. Kant. An $\mathcal{O}(n^2)$ maximal planarization algorithm based on *PQ*-trees. Technical Report RUU-CS-92-03, Dept. of Comp. Science, Utrecht University, 1992. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1992/1992-03.ps.gz`.

[Kar90] A. Karabeg. Classification and detection of obstructions to planarity. *Linear and Multilinear Algebra*, 26(1-2):15–38, 1990.

[Kur30] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs: International symposium: Rome, Italy, July 1966*, pages 215–232. Gordon and Breach, New York, 1967.

[Lei96] S. Leipert. The problem of computing a maximal planar subgraph using PQ-trees is still not solved. In *Special Proceedings for Students, European Consortium of Mathematics in Industry, Kaiserslautern, Germany, 1994, EMCI'94*, aug 1996.

[Lei97] S. Leipert. PQ-trees, an implementation as template class in C++. Technical report, Informatik, Universität zu Köln, 1997. `http://www.mpi-sb.mpg.de/LEDA/www/leps/pq_tree.html`.

[LG79] P. C. Liu and R. C. Geldmacher. On the deletion of nonplanar edges of a graph. In *Proc. 10th S-E Conf. Combinatorics, Graph Theory, and Computing: Boca Raton, Florida*, pages 727–738, 1979.

[Man83] A. Mansfield. Determinig the thickness of graphs is $\mathcal{NP}$-hard. *Math. Proc. of Cambridge. Philos. Soc.*, 93:9–23, 1983.

[May72] J. Mayer. Décomposition de K_{16} en trois graphes planaires. *J. Combin. Theory B*, 13:71, 1972.

[Mut94] P. Mutzel. *The Maximum Planar Subgraph Problem.* PhD thesis, Mathematisch-Naturwissenschaftlichen Fakultät der Universität zu Köln, 1994.

[OT81] T. Ozawa and H. Takahashi. A graph-planarization algorithm and its application to random graphs. In N. Saito and T. Nishizeki, editors, *Proc. Graph Theory and Algorithms: Sendai, Japan, October 24-25, 1980*, volume 108 of *Lecture Notes in Computer Science*, pages 95–107, New York, 1981. Springer Verlag.

[Str97] Bjarne Stroustrup. *The C++ programming language.* Addison Wesley, 1997.

[Tho95] C. Thomassen. Embeddings and minors. In R. L. Graham, M. Grötschel, and L. Lovàsz, editors, *Handbook of Combinatorics*, volume 1, chapter 5. Elsevier Science B. V., Amsterdam, Netherlands, 1995.

Appendix A

Implementation of `mypqtree`

The newest version of Leipert's implementation of PQ-tree [Lei97] is available at `http://www.mpi-sb.mpg.de/LEDA/`.

Files are stored at `/home/skidbladnir/e/gorrilv/hovedfag/OPPGAVE/Prog/`.

A.1 Guidelines

Input files should look like the file in fig. A.1

The derived subclass of `PQTree` is called `pqtree`. `pqtree` is implemented only to work with parameter types `edge`, `Chain`, and `Boundary`. Thus, it is no longer a template in itself.

```
12
 2  3  4  5 12  0
 1  3  7  0
 1  2  8  0
 1  9  0
 1  6  0
 5  7 12  0
 2  6 10 11  0
 3  9  0
 4  8 11  0
 7 11  0
 7  9 10 12  0
 1  6 11  0
```

Figure A.1: Example of input file

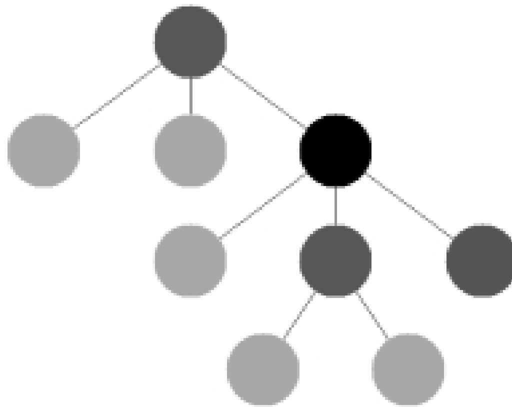


Figure A.2: Example of file displayed by vbcTool

A.1.1 vbcTool

Leipert has provided a visual debugging tool, vbcTool, that visualizes the *PQ*-tree. Print procedures have been added to classes `edge`, `chain`, and `boundary` in order to present this additional information as well in these drawings. vbcTool can be downloaded from
`http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html`.
Fig. A.2 shows a typical tree, one of the trees obtained from running the graph in fig. A.1. Dark grey nodes are *P*-nodes, colored red. Black node is *Q*-node. Leaves are colored blue. When full, they are colored green. Fig. A.3 shows the file generated by Bubble, and displayed by vbcTool in fig. A.2.

A.1.2 Altered files of Leipert's implementation

All *.cc files provided with the template classes of Leipert's implementation, were altered, since the present implementation is not a template. `PQTree.cc` is given as an example. For all files, `edge.h`, `chain.h`, and `boundary.h` must be included, in addition to those specified by Leipert's implementation.

```
/*********************************************************************

            Filename    :    PQTree.cc
            Version     :    1.0    1997
            Author      :    Sebastian  Leipert
            Language    :    C++

*********************************************************************/

#include "PQTree.h"
#include "key.h"
```

```
#TYPE: COMPLETE TREE
#TIME: NOT
#BOUNDS: NONE
#INFORMATION: STANDARD
#NODE_NUMBER: NONE
n 1 \iROOT:  P-NODE, ID#: 0, Status:  EMPTY , Vertex 1 fChs: 0 \i
c 1 4
e 1 2
n 2 \i LEAF ID#: 12, Status:  EMPTY Chain: 1-4-9 (4,9)\i
c 2 8
e 1 3
n 3 \i Q-NODE ID#: 18, Status:  EMPTY , Vertex 6 Bound:
1(j)(E)(F) - 5(E)(F) - 6(l)(E)(T) - 7(E)(T) - 2(E)(F) - 3(r)(E)(T) fChs: 0 \i
c 3 6
e 1 4
n 4 \i LEAF ID#: 5, Status:  EMPTY  (1,12)\i
c 4 8
e 3 5
n 5 \i LEAF ID#: 16, Status:  EMPTY  (6,12)\i
c 5 8
e 3 6
n 6 \i LEAF ID#: 10, Status:  EMPTY  (3,8)\i
c 6 9
e 3 7
n 7 \i P-NODE ID#: 19, Status:  EMPTY , Vertex 7 fChs: 0 \i
c 7 4
e 7 8
n 8 \i LEAF ID#: 20, Status:  EMPTY  (7,10)\i
c 8 8
e 7 9
n 9 \i LEAF ID#: 21, Status:  EMPTY  (7,11)\i
c 9 8
```

Figure A.3: File represented in fig. A.2.

```
#include "edge.h"
#include "chain.h"
#include "boundary.h"

template class PQTree<edge,chain,boundary>;
```

A.2 Code files main.cc and mypqtree.h

A.2.1 main.cc

```
/*******************************************************************
                                        File: main.cc

                                        Author: Gørril Vollen
                                        Last update: 20/02/1998
*******************************************************************/

#include <stdio.h>
//#define DEBUG     /* remove // if debugging */
//#define PRINT     /* remove // if using vbcTool */

#include <iostream.h>
#include <fstream.h>
#include "mypqtree.h"
#include "edge.h"
#include "chain.h"
#include "boundary.h"

/*******************************************************************
   TallTilTekst
   Transforms an integer into it's equivalent character string.
   Only used for debugging purposes.
   Arguments: number - the integer to be transformed
                 width - number of characters in the string to be returned
   Returns: a pointer to the character string
*******************************************************************/
char * TallTilTekst(int number, int width)
{
      ostrstream string;
      char r[width];

      string << number << ends;
      strcpy(r, string.str());
      strstreambuf * buf_ptr = string.rdbuf();
      buf_ptr->freeze(0);
      return r;
};

int main(int argc, char *argv[])
{
      char *readFile = argv[1];
      pqtree<edge, chain, boundary>* mytree = new pqtree<edge, chain, boundary>();
      if (argc == 2)
            mytree->readStNumbering(readFile);
      if(!mytree->Planar()) cout << "Not ";
      cout << "planar!" << endl;
}
```

A.2.2 pqtree

All procedrues called in `mypqtree.h`, not part of any files presented here, are part of `PQTree` or other parts of Leipert's implementation. Documentation of these procedures are found in [Lei97].

```
/*******************************************************************
                                    File: pqtree.h

                                    Author: Gørril Vollen
                                    Last update: 18/02/1998
*******************************************************************/
#ifndef MYPQTREE_H
#define MYPQTREE_H
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>
#include <map.h>

#include "PQTree.h"
#include "key.h"
#include "internal.h"
#include "nodeInfo.h"

#define MAXNODES 100


/*******************************************************************
class pqtree
Definition of class pqtree implemented as a subclass of the template
class PQTree. PQTree is part of an implementation of the PQ-tree data
structure (Booth & Lueker) by S. Leipert. This subclass extends PQTree
with the planarity testing algorithm in the procedrue planar, as well
as handling of the additional data structure of boundary and chain
(Karabeg).
Bubble is also extended to always build a pruned subtree, regardless of
whether or not the graph is planar.
*******************************************************************/

template<class T, class X, class Y>
class pqtree : public PQTree<T, X, Y>
{

     typedef map<int, edge, less<int> > maps;
     maps incoming[MAXNODES]; /* set for storing incoming edges
                                 for nodes, i.e. S[i] sets */
     maps outgoing[MAXNODES]; /* set for storing outgoing edges
                                 for nodes, i.e. S'[i] sets */

     int N, M;
     int iterationNumber;

     int NumberOfNodes;
     int NumberOfEdges;

     stack<element*> *pertinentElements; /* elements whose attributes are
                                            changed during a reduction is
                                            placed on this stack, to be
                                            cleaned up by cleanUpElem */
public:
     void readStNumbering(char *filename);
     int Planar();
```

```
    void destroyNodeAndChildren(node<T,X,Y>* node_ptr);
    void cleanUpElem();
    node<T,X,Y>* findParent(node<T,X,Y>* child);

    int Initialize(int numberOfElements, key<T,X,Y>** ArrayOfElements);

#ifdef DEBUG
    int     Bubble(int numberOfConsecutiveElements,
                   key<T,X,Y>** ArrayOfElements,
                   int redNumber);
#else
    int     Bubble(int numberOfConsecutiveElements,
                   key<T,X,Y>** ArrayOfElements);
#endif

#ifdef PRINT
    int     Reduce(int numberOfConsecutiveElements,
                   key<T,X,Y>** ArrayOfElements,
                   int iterationNumber,int printYes); /* If printYes is set to TRUE,
                              a file will be made after every template matching */
#else
    int     Reduce(int numberOfConsecutiveElements,
                   key<T,X,Y>** ArrayOfElements);
#endif

    int     template_L1(node<T,X,Y> *node_ptr,
                        int isRoot);

    int     template_P1(node<T,X,Y> *node_ptr,
                        int isRoot);

    int     template_P2(node<T,X,Y> **node_ptr);

    int     template_P3(node<T,X,Y> *node_ptr);

    int     template_P4(node<T,X,Y> **node_ptr);

    int     template_P5(node<T,X,Y> *node_ptr);

    int     template_P6(node<T,X,Y> **node_ptr);

    int     template_Q1(node<T,X,Y> *node_ptr,
                        int isRoot);

    int     template_Q2(node<T,X,Y> *node_ptr,
                        int isRoot);

    int     template_Q3(node<T,X,Y> *node_ptr);

};

/********************************************************************
pqtree::readStNumbering
readStNumbering reads from a given file an st-numbered graph,
and produces the data for the incoming and outgoing sets of edges
as well as setting the NumberOfNodes and NumberOfEdges variables
********************************************************************/

template<class T, class X, class Y>
void pqtree<T,X,Y>::readStNumbering(char *filename)
{
    int edg;
```

```
    int j;
    int edgecount = 0;
    maps sh;
    edge *ep;
    ifstream inClientFile(filename, ios::in);

    inClientFile >> NumberOfNodes;

    for(j = 1; j <= NumberOfNodes; j++)
    {
        inClientFile >> edg;

        while(edg != 0)
        {
            if(edg > j)
            {
                edgecount++;
                ep = new edge(edgecount, j, edg);
                incoming[edg-1][edgecount] = *ep;
                outgoing[j-1][edgecount] = *ep;
            }
            inClientFile >> edg;
        }
    }
    NumberOfEdges = edgecount;
}

/*******************************************************************
pqtree::cleanUpElem
cleanUpElem resets the appropriate attributes of the elements on stack
pertinentElements after each step of the reduction. Elements marked as
TO_BE_DELETED are deleted.
*******************************************************************/

template<class T, class X, class Y>
void pqtree<T,X,Y>::cleanUpElem()
{
    element *elem;

    while (!pertinentElements->stackEmpty())
    {
        elem = pertinentElements->pop();
        switch(elem->status())
        {
        case EMPTY:
            break;
        case TO_BE_DELETED:
            delete elem;
            break;
        case PARTIAL:
            elem->reset(EMPTY, TRUE, NULL, TRUE);
            break;
        case PERTINENT:
            elem->reset(EMPTY, TRUE, NULL, FALSE);
            break;
        case FULL:
            if (elem->child() == TRUE)
                elem->reset(EMPTY, TRUE, NULL, TRUE);
            else /* child is FALSE */
                elem->reset(EMPTY, FALSE, NULL, FALSE);
            break;
        default:
```

```
                cerr << "boundary::cleanUpElem: unknown status: "
                        << elem->status() << endl;
            }
        }
} //cleanUpElem


/*******************************************************************
pqtree::Planar
Planar checks a given st-numbered graph for the planarity property.
The edges incident to each vertex of the graph has been partitioned
into two sets: incoming and outgoing. Based on the algorithm of
Booth & Lueker, Planar utilizes the PQ-tree data structure to
successibely add vertices of the graph while checking if the graph
remains planar. A number of sets, U,S, and Sm (s'), are used and these
are implemented as arrays of pointers of type key. In addition, all
new internal nodes constructed are given a boundary object. In the
vertex addition step, a chain is maintained if necessary.

Variables: Edges - pointer to array of key. Used to hold all edges in
                   the graph.
           U - pointer to array fo key. Used to initialze pqtree
           S - pointer to array of key. Holds the set of incoming edges
                 of the current vertex to be added.
           Sm -  pointer to array of key. Holds the set of outgoing
                   edges of the current vertex to be added.
           + a number of iterator variables, pointers to different types
              of nodes (needed for accessing node data) and counters.
Returns: TRUE (defined as 1) if graph is planar
*******************************************************************/

template<class T, class X, class Y>
int pqtree<T,X,Y>::Planar()
{
    int                    i;          /* iterator variable */
    int                    success;
    int                    U_Size           = outgoing[0].size();
    int                    S_Size, Sm_Size;
    maps::iterator      it;
    key<T,X,Y>          **Edges; /* holds pointers to all edges */
    key<T,X,Y>          **U;        /* U = S'[1] */
    key<T,X,Y>          **S;
    key<T,X,Y>          **Sm;       /* S'[] */
    pqNode<T,X,Y>       *newPnode          = NULL;
    pqNode<T,X,Y>       *newNode           = NULL;
    leaf<T,X,Y>         *newLeaf           = NULL;
    node<T,X,Y>         *firstNode         = NULL;
    node<T,X,Y>         *currentNode       = NULL;
    node<T,X,Y>         *pertinentRoot  = NULL;
    node<T,X,Y>         *emptyNode         = NULL;
    node<T,X,Y>         *replacement       = NULL;
    chain               *newChain          = NULL;
    boundary            *newBoundary       = NULL;
    nodeInfo<T,X,Y>  *newInfo           = NULL;
    internal<T,X,Y>  *newInternal       = NULL;

    Edges = new key<T,X,Y>*[NumberOfEdges];
    for (i = 1; i <= NumberOfEdges; i++)
        Edges[i-1] = NULL;

    /* U = the set of edges whose lower-numbered vertex is 1 i.e.
        outgoing[0] */
```

```
	U = new key<T,X,Y>*[U_Size];
	it = outgoing[0].begin();

	for(i = 1; i <= U_Size; i++)
	{
		U[i-1] = new key<T,X,Y>((*it).second);
		Edges[(*it).first-1] = U[i-1];
		it++;
	}

	/* Make T(U,U) */

	Initialize(U_Size, U);

	/* Main loop of planar algorithm */

	for(iterationNumber=2; iterationNumber < NumberOfNodes; iterationNumber++)
	{

		/* S = the set of edges whose higher-numbered vertex is
		   iterationNumber, i.e. incoming[iterationNumber-1] */

		S_Size = incoming[iterationNumber-1].size();
		S = new key<T,X,Y>*[S_Size];

		it = incoming[iterationNumber-1].begin();

		for(i = 1; i <= S_Size; i++)
		{
			S[i-1] = Edges[(*it).first-1];
			it++;
		}

		/* BUBBLE(T,S) */

#ifdef DEBUG
		success = Bubble(S_Size, S, iterationNumber);
#else
		success = Bubble(S_Size, S);
#endif
		if(!success) return success;

		/* REDUCE(T,S) */

#ifdef PRINT
		success = Reduce(S_Size, S, iterationNumber, TRUE);
#else
		success = Reduce(S_Size, S);
#endif
		if(!success) return success;
#ifdef DEBUG
		else cout << "reduce success" << endl;
#endif

		/* S' = the set of edges whose lower-numbered vertex is
		   iterationNumber, i.e. outgoing[iterationNumber-1] */

		Sm_Size = outgoing[iterationNumber-1].size();
		Sm = new key<T,X,Y>*[Sm_Size];

		it = outgoing[iterationNumber-1].begin();
```

```
#ifdef DEBUG
        printOutCurrentTree("PQ_R_Print",TallTilTekst(iterationNumber,3));
        cout << "S'[" << iterationNumber << "] = ";
#endif
        for(i = 1; i <= Sm_Size; i++)
        {
            Sm[i-1] = new key<T,X,Y>((*it).second);
            Edges[(*it).first-1] = Sm[i-1];
            it++;
        }

        /* Make T(S',S') */
        /* If pertinent root is replaced, its chain must be passed on */
        if (_pertinentRoot->status() == FULL)
            newInfo = _pertinentRoot->getNodeInfo();
        if (Sm_Size > 1)
        {
            newBoundary = new boundary(iterationNumber);
            newInternal = new internal<T,X,Y>(*newBoundary);

            /* Make P node and attach leaves, if chain exists, keep it */
            if (S_Size==1) /* pertinentRoot is LEAF */
            {
                /* If parent has boundary, short chains need to be kept too. */
                if ((newInfo == NULL) &&
                    ((_pertinentRoot->_parentType == Q_NODE) ||
                     (_pertinentRoot->_parent->getInternal()->
                      userStructInternal()->joint() != NULL)))
                {
                    newChain = new chain(
                        _pertinentRoot->getKey()->userStructKey()->from(),
                        iterationNumber);
                    newInfo = new nodeInfo<T,X,Y>(*newChain);
                }
            }
            if (newInfo != NULL)
                newNode = new pqNode<T,X,Y>(_identificationNumber++,
                                            P_NODE, EMPTY, newInternal,
                                            newInfo);
            else
                newNode = new pqNode<T,X,Y>(_identificationNumber++,
                                            P_NODE, EMPTY, newInternal);
            addNewLeavesToTree(newNode, Sm, Sm_Size);
            replacement = newNode;
            newInfo = NULL;
            newChain = NULL;
        }
        else     /* T(S',S') consists of only 1 leaf, |S'|==1 */
        {
            if (S_Size == 1)
            {
                /* A chain is being built */

                if (newInfo != NULL)
                    newChain = newInfo->userStructInfo();
                if (newChain == NULL)
                {
                    newChain = new chain(
                        _pertinentRoot->getKey()->userStructKey()->from(),
                        iterationNumber,Sm[0]->userStructKey()->to());
                }
                else     /* newChain != NULL */
```

```
                newChain->insert(Sm[0]->userStructKey()->to());
        }
        else if (newInfo != NULL) /* old chain must be continued */
            newInfo->userStructInfo()->
                insert(Sm[0]->userStructKey()->to());

        if (newInfo == NULL && newChain != NULL)
        {
            newInfo = new nodeInfo<T,X,Y>(*newChain);
            newChain->setInfo(newInfo);
        }
        if (newInfo != NULL)
            newLeaf = new leaf<T,X,Y>(_identificationNumber++,
                                      EMPTY, Sm[0], newInfo);
        else
            newLeaf = new leaf<T,X,Y>(_identificationNumber++,
                                      EMPTY, Sm[0]);
        replacement = newLeaf;
        newChain = NULL;
        newInfo = NULL;
}
_pertinentNodes->push(replacement);

pertinentRoot = _pertinentRoot;

switch(pertinentRoot->type())
{

case Q_NODE:
        /* replace the full children of ROOT(T,S) and their
            descendants by T(S',S') */

        if (pertinentRoot->_identificationNumber == -1)
        {

            /* ROOT(T,S) is a pseudo node. All children of
                ROOT(T,S) must be replaced with T(S',S') */

            firstNode = pertinentRoot->fullChildren->pop();
            while(!pertinentRoot->fullChildren->stackEmpty())
            {
                currentNode = pertinentRoot->fullChildren->pop();
                removeChildFromSiblings(currentNode);
                destroyNodeAndChildren(currentNode);
            }
            exchangeNodes(firstNode, replacement);
            destroyNodeAndChildren(firstNode);
        }
        else if (pertinentRoot->status() == FULL)
        {

            /* All children of the Q-node are full so delete it and
                replace with T(S',S') */
            exchangeNodes(pertinentRoot, replacement);
            destroyNodeAndChildren(pertinentRoot);
        }
        else
        {
            if ((pertinentRoot->childCount() -
                    pertinentRoot->fullChildren->count()) == 1)
            {
```

```
        /* ROOT(T,S) is  a Q-node with only 1 empty child.
           The Q-node should be replaced with a P-node */
        /* The empty child of Q_Node must be located on
           either endmost slot*/

        if (pertinentRoot->_leftEndmost->status() == EMPTY)
             emptyNode = pertinentRoot->_leftEndmost;
        else
             emptyNode = pertinentRoot->_rightEndmost;

        /* remove ROOT(T,S), make new P-node and attach
           empty child of ROOT(T,S) and newNode */
        newInternal = pertinentRoot->getInternal();
        newInfo = pertinentRoot->getNodeInfo();
        if (newInfo != NULL)
             newPnode = new pqNode<T,X,Y>(_identificationNumber++,
                                             P_NODE, EMPTY,
                                             newInternal, newInfo);
        else
             newPnode = new pqNode<T,X,Y>(_identificationNumber++,
                                             P_NODE, EMPTY,
                                             newInternal);
        _pertinentNodes->push(newPnode);

        while(!pertinentRoot->fullChildren->stackEmpty())
        {
             currentNode = pertinentRoot->fullChildren->pop();
             removeChildFromSiblings(currentNode);
             destroyNodeAndChildren(currentNode);
        }

        addNodeToNewParent(newPnode, emptyNode);
        addNodeToNewParent(newPnode, replacement, emptyNode, NULL);
        exchangeNodes(pertinentRoot, newPnode);
        destroyNode(pertinentRoot);
}
else
{

        /* The Q-node has at least 2 empty children so only the full
           children of ROOT(T,S) should be destroyed and replaced */

        if (pertinentRoot->_leftEndmost->status() != FULL &&
             pertinentRoot->_rightEndmost->status() != FULL)
        {
             /* both endmost are empty */

             firstNode = pertinentRoot->fullChildren->pop();

             while(!pertinentRoot->fullChildren->stackEmpty())
             {
                  currentNode = pertinentRoot->fullChildren->pop();
                  removeChildFromSiblings(currentNode);
                  pertinentRoot->_childCount--;
                  destroyNodeAndChildren(currentNode);
             }
             exchangeNodes(firstNode, replacement);
             destroyNodeAndChildren(firstNode);
        }
        else
        {
             /* one of the endmost is full */
```

```
                              if (pertinentRoot->_leftEndmost->status() == FULL)
                                   firstNode = pertinentRoot->_leftEndmost;
                              else
                                   firstNode = pertinentRoot->_rightEndmost;
                              if (firstNode->status() != FULL)
                                   cout << "ERROR!" << endl;

                              exchangeNodes(firstNode, replacement);
                              while(!pertinentRoot->fullChildren->stackEmpty())
                              {
                                   currentNode = pertinentRoot->fullChildren->pop();
                                   if (currentNode != firstNode)
                                   {
                                        removeChildFromSiblings(currentNode);
                                        pertinentRoot->_childCount--;
                                        destroyNodeAndChildren(currentNode);
                                   }
                              }
                              destroyNodeAndChildren(firstNode);
                         }
                    }
               }
               break;

          case LEAF:
               /* easy case: replace ROOT(T,S) with T(S',S') */

               exchangeNodes(pertinentRoot, replacement);
               destroyNode(pertinentRoot);
               break;

          case P_NODE:
               /* ROOT(T,S) is a P node so replace it and it's full children with
                    T(S',S'). ROOT(T,S) will only have full children. */

               if (pertinentRoot->status() == FULL)
               {
                    exchangeNodes(pertinentRoot, replacement);
                    destroyNodeAndChildren(pertinentRoot);
               }
               else
                    cout << "ERROR: not full P_NODE!";
               break;

          default:
               cout << "something's wrong" << endl;
               return 0;
               break;
          }

          cleanUpElem();
          emptyAllPertinentNodes();

          delete S;
          delete Sm;
     }
     return 1;
}

/**********************************************************************
pqtree::destroyNodeAndChildren
```

Marks the subtree rooted at a given node TO_BE_DELETED. This subtree is then deleted, freeing memeory occupied by node objects, when the pqtree is cleaned up between each iteration of the planar algorithm, by calling emptyAllPertinentNodes. The procedure is based on the front procedure from PQTree.h.
Arguments: node_ptr – pointer to the root of the subtree to be deleted.
***/

```
template<class T, class X, class Y>
void pqtree<T,X,Y>::destroyNodeAndChildren(node<T,X,Y>* node_ptr)
{
      node<T,X,Y>* checkNode = NULL;
      node<T,X,Y>* firstSon = NULL;
      node<T,X,Y>* nextSon = NULL;
      node<T,X,Y>* oldSib = NULL;
      node<T,X,Y>* holdSib = NULL;
      node<T,X,Y>* lastSon = NULL;

      queue<node<T,X,Y>*> *helpqueue = new queue<node<T,X,Y>*>;
      helpqueue->enQueue(node_ptr);

      while(!helpqueue->queueEmpty())
      {
            checkNode = helpqueue->deQueue();
            checkNode->status(TO_BE_DELETED);

            if(checkNode->type() == P_NODE)
            {
                  if(checkNode->_referenceChild != NULL)
                  {
                        firstSon = checkNode->_referenceChild;
                        helpqueue->enQueue(firstSon);

                        if(firstSon->_sibRight != NULL)
                              nextSon = firstSon->_sibRight;
                        while(nextSon != firstSon)
                        {
                              helpqueue->enQueue(nextSon);
                              nextSon = nextSon->_sibRight;
                        }
                  }
            }
            else if (checkNode->type() == Q_NODE)
            {
                  oldSib = NULL;
                  holdSib = NULL;

                  firstSon = checkNode->_leftEndmost;
                  helpqueue->enQueue(firstSon);

                  lastSon = checkNode->_rightEndmost;
                  helpqueue->enQueue(lastSon);

                  nextSon = lastSon->getNextSib(oldSib);
                  oldSib = lastSon;
                  while (nextSon != firstSon)
                  {
                        helpqueue->enQueue(nextSon);
                        holdSib = nextSon->getNextSib(oldSib);
                        oldSib = nextSon;
                        nextSon = holdSib;
                  }
```

```
        }
    }
    delete helpqueue;
}

/*********************************************************************
pqtree::Initialize
Derived from PQTree::Initialize. Builds the universal tree, a P-node with
all elements of U as children. The P-node is given a boudnary object with
vertex number 1.
Arguments: ArrayOfElements - the elements of U
           numberOfElements - the size of U
Returns: True if successfull initialization
*********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::Initialize(int numberOfElements,key<T,X,Y>** ArrayOfElements)
{
   pqNode<T,X,Y>     *_newNode     = NULL;
   pqNode<T,X,Y>     *_newNode2    = NULL;
   boundary *bp = new boundary(1);
   internal<T,X,Y> *ip = new internal<T,X,Y>(*bp);

   _pertinentNodes = new stack<node<T,X,Y>*>;
   pertinentElements = new stack<element*>;

   if ( numberOfElements > 0)
   {
      _newNode = new pqNode<T,X,Y>(_identificationNumber++,P_NODE,EMPTY,ip);
      _root = _newNode;
      _root->_sibLeft = _root;
      _root->_sibRight = _root;

      _newNode2 = new pqNode<T,X,Y>(-1,Q_NODE,PARTIAL);
      _pseudoRoot = _newNode2;
      return addNewLeavesToTree(_newNode,ArrayOfElements, numberOfElements);
   }
   else return FALSE;
}

/*********************************************************************
pqtree::findParent
A utility procedure for Bubble. Traverses all siblings of a node until
a valid parent pointer is found.
Arguments: child - the node needing a parent pointer
Returns: the parent pointer found

Note! findParent is not fully implemented as described in thesis.
Will always find a parent pointer for every child of a Q-node sent as
a parameter. New mark BLOCKED_WITH_PARENT is not included.
*********************************************************************/

template<class T,class X,class Y>
node<T,X,Y>* pqtree<T,X,Y>::findParent(node<T,X,Y>* child)
{
  node<T,X,Y>* nextSib;
  node<T,X,Y>* newSib;
  node<T,X,Y>* oldSib;

  nextSib = child->getNextSib(NULL);
  oldSib = child;
```

```
   while (nextSib->mark() != UNBLOCKED && !nextSib->endmostChild())
   {
      newSib = nextSib->getNextSib(oldSib);
      oldSib = nextSib;
      nextSib = newSib;
   }
   return nextSib->parent();
}

/*********************************************************************
pqtree::Bubble
Bubble has been extended to implement BubblExt of thesis. Will always
build a complete pruned subtree, without using a pseudonode.
Arguments: ArrayOfElements - the current S, defining the pertinent leaves
           number - the size of S
           redNumber - used if DEBUG is defined in printing out
           information to a degub-file, readable with the vbcTool provided
           by S. Leipert
Returns: always TRUE, as the pruned subtree is always completed.

Note: Not fully implemented as described in thesis. New mark
      BLOCKED_WITH_PARENT is not included. Thus findParent is called for
      every blocked node on stack blocks.
Note: If the part of Bubble that is repeated by BubbleExt is made
      into a procedure of its own, handleBlockedNodes, the loop of
      pqtree::Bubble will be a lot shorter and easier to manage.
      This has not been done here since the procedure would have to be
      external to pqtree::Bubble.
*********************************************************************/

#ifdef DEBUG
template<class T,class X,class Y>
int pqtree<T,X,Y>::Bubble(int number,
                          key<T,X,Y>** ArrayOfElements,
                          int redNumber)
#else
template<class T,class X,class Y>
int pqtree<T,X,Y>::Bubble(int number,
                          key<T,X,Y>** ArrayOfElements)
#endif
{
    queue<node<T,X,Y>*> *_processNodes = new queue<node<T,X,Y>*>;

#ifdef DEBUG
    stack<node<T,X,Y>*> *_processLeaves = new stack<node<T,X,Y>*>;
#endif

    stack<node<T,X,Y>*> *blocks = new stack<node<T,X,Y>*>;
                        /* Stack of all nodes that are marked blocked */

    int         _blockCount      = 0;
    int         _blockedNodes    = 0;
    int         _offTheTop       = 0;

    int         _blockedSiblings = 0;
    int         i                = 0;

    node<T,X,Y>*  _checkLeaf  = NULL;
    node<T,X,Y>*  _checkNode  = NULL;
    node<T,X,Y>*  _checkSib   = NULL;
    node<T,X,Y>*  _holdSib    = NULL;
    node<T,X,Y>*  _oldSib     = NULL;
```

```
    node<T,X,Y>*      _parent           = NULL;
    node<T,X,Y>*      _lastBlocked  = NULL;

    for (i = 0; i <= (number-1); i++)
    {
        _checkLeaf = ArrayOfElements[i]->nodePointer();
        _checkLeaf->mark(QUEUED);
        _processNodes->enQueue(_checkLeaf);
        _pertinentNodes->push(_checkLeaf);
#ifdef DEBUG
        _checkLeaf->status(FULL);
        _processLeaves->push(_checkLeaf);
#endif
    }

#ifdef DEBUG
    char nb[INFOSIZE];
    sprintf(nb,"%d",redNumber);
    printOutCurrentTree("PQ_B_Print",nb);
    while (!_processLeaves->stackEmpty())
    {
        _checkLeaf = _processLeaves->pop();
        _checkLeaf->status(EMPTY);
    }
#endif

    while ((_processNodes->queueSize() + _blockCount + _offTheTop) > 1)
    {
        if (_processNodes->queueSize() == 0) /* HandleBlockedNodes */
        {   /* The tree is not reducible */

            while (!blocks->stackEmpty() && _blockCount > 0)
            {
                _checkNode = blocks->pop();
                if (_checkNode->mark() == BLOCKED)
                {
                    _parent = findParent(_checkNode);
                    /* Since the root is never blocked, a parent pointer
                       will always be found */
                    _checkNode->parent(_parent);
                    _checkNode->mark(UNBLOCKED);

                    /* Pass parent pointer on to all blocked siblings */
                    if (clientSibLeft(_checkNode) != NULL)
                    {
                        _checkSib = clientSibLeft(_checkNode);
                        _oldSib = _checkNode;
                        _holdSib = NULL;
                        while (_checkSib->mark() == BLOCKED)
                        {
                            _checkSib->mark(UNBLOCKED);
                            _checkSib->_parent = _parent;
                            _blockedNodes--;
                            _parent->_pertChildCount++;

                            _holdSib = clientNextSib(_checkSib,_oldSib);
                            _oldSib = _checkSib;
                            _checkSib = _holdSib;
                            if (_checkSib == NULL)
                            {
                                cout << "ERROR: PQTree->Bubble: Blocked node "
                                     << "as endmost child of a Q_NODE."
```

```
                                        << endl;
                                _checkSib = _oldSib;
                        }
                }
        }

        if (clientSibRight(_checkNode) != NULL)
        {
                _checkSib = clientSibRight(_checkNode);
                _oldSib = _checkNode;
                _holdSib = NULL;
                while (_checkSib->mark() == BLOCKED)
                {
                        _checkSib->mark(UNBLOCKED);
                        _checkSib->_parent = _parent;
                        _blockedNodes--;
                        _parent->_pertChildCount++;

                        _holdSib = clientNextSib(_checkSib,_oldSib);
                        _oldSib = _checkSib;
                        _checkSib = _holdSib;
                        if (_checkSib == NULL)
                        {
                                cout << "ERROR: PQTree->Bubble: Blocked node as "
                                        << " endmost child of a Q_NODE." << endl;
                                _checkSib = _oldSib;
                        }
                }
        }
        _parent->_pertChildCount++;
        if (_parent->mark() == UNMARKED)
        {
                _processNodes->enQueue(_parent);
                _pertinentNodes->push(_parent);
                _parent->mark(QUEUED);
        }
        _blockCount = _blockCount - _blockedSiblings;
        _blockedSiblings = 0;

    }
  }
  _blockCount = 0;
  _blockedNodes = 0;
}
/* If HandleBlockedNodes leave the queue empty,
   Bubble is finished here */

if (!_processNodes->queueEmpty())
{
  _checkNode = _processNodes->deQueue();
  _checkNode->mark(BLOCKED);
  _blockedSiblings = 0;

  if ((_checkNode->_parentType != P_NODE) && (_checkNode != _root))
      // _checkNode is son of a Q_NODE.
      // Check if it is blocked.
  {
      if (clientSibLeft(_checkNode) == NULL)
          // _checkNode is endmost child of
          // a Q_NODE. It has a valid pointer
          // to its parent.
      {
```

```
            _checkNode->mark(UNBLOCKED);
            if (clientSibRight(_checkNode)->mark() == BLOCKED)
                _blockedSiblings++;
        }
        else if (clientSibRight(_checkNode) == NULL)
            // _checkNode is endmost child of
            // a Q_NODE. It has a valid pointer
            // to its parent.
        {
            _checkNode->mark(UNBLOCKED);
            if (clientSibLeft(_checkNode)->mark() == BLOCKED)
                _blockedSiblings++;
        }
        else
            // _checkNode is not endmost child of
            // a Q_NODE. It has not a valid pointer
            // to its parent.
        {
            if (clientSibLeft(_checkNode)->mark() == UNBLOCKED)
                // _checkNode is adjacent to an
                // unblocked node. Take its parent.
            {
                _checkNode->mark(UNBLOCKED);
                _checkNode->_parent = clientSibLeft(_checkNode)->_parent;
            }
            else if (clientSibLeft(_checkNode)->mark() == BLOCKED)
                _blockedSiblings++;

            if (clientSibRight(_checkNode)->mark() == UNBLOCKED)
                // _checkNode is adjacent to an
                // unblocked node. Take its parent.
            {
                _checkNode->mark(UNBLOCKED);
                _checkNode->_parent = clientSibRight(_checkNode)->_parent;
            }
            else if (clientSibRight(_checkNode)->mark() == BLOCKED)
                _blockedSiblings++;
        }
    }
    else
        // _checkNode is son of a P_NODE
        // and children of P_NODEs
        // cannot be blocked.
        _checkNode->mark(UNBLOCKED);

    if (_checkNode->mark() == UNBLOCKED)
    {
        _parent = _checkNode->_parent;
        if (_blockedSiblings > 0)
        {
            if (clientSibLeft(_checkNode) != NULL)
            {
                _checkSib = clientSibLeft(_checkNode);
                _oldSib = _checkNode;
                _holdSib = NULL;
                while (_checkSib->mark() == BLOCKED)
                {
                    _checkSib->mark(UNBLOCKED);
                    _checkSib->_parent = _parent;
                    _blockedNodes--;
                    _parent->_pertChildCount++;
```

```
                    _holdSib = clientNextSib(_checkSib,_oldSib);
                    _oldSib = _checkSib;
                    _checkSib = _holdSib;
                    if (_checkSib == NULL)
                    {
                         cout << "ERROR: PQTree->Bubble: Blocked node as "
                              << " endmost child of a Q_NODE." << endl;
                         _checkSib = _oldSib;
                    }
               }
          }

          if (clientSibRight(_checkNode) != NULL)
          {
               _checkSib = clientSibRight(_checkNode);
               _oldSib = _checkNode;
               _holdSib = NULL;
               while (_checkSib->mark() == BLOCKED)
               {
                    _checkSib->mark(UNBLOCKED);
                    _checkSib->_parent = _parent;
                    _blockedNodes--;
                    _parent->_pertChildCount++;

                    _holdSib = clientNextSib(_checkSib,_oldSib);
                    _oldSib = _checkSib;
                    _checkSib = _holdSib;
                    if (_checkSib == NULL)
                    {
                         cout << "ERROR: PQTree->Bubble: Blocked node as "
                              << " endmost child of a Q_NODE." << endl;
                         _checkSib = _oldSib;
                    }
               }
          }
     }

     if (_parent == NULL)
     {
          // _checkNode is root of the tree.
          _offTheTop = 1;
     }
     else
          // _checkNode is not the root.
     {
          _parent->_pertChildCount++;

          if (_parent->mark() == UNMARKED)
          {
               _processNodes->enQueue(_parent);
               _pertinentNodes->push(_parent);
               _parent->mark(QUEUED);
          }
     }
     _blockCount = _blockCount - _blockedSiblings;
     _blockedSiblings = 0;
}
else /* _checkNode is blocked */
{
     _blockCount = _blockCount + 1 - _blockedSiblings;
     _blockedNodes++;
     _lastBlocked = _checkNode;
```

```
                    blocks->push(_checkNode);
                }
            }
        }

        if (_blockCount == 1)
        {
            /* The ordinary Q3 situation, but the use of _pseudoNode is avoided */
            _parent = findParent(_lastBlocked);
            _lastBlocked->_parent = _parent;
            _parent->_pertChildCount = 1;
            if (clientSibLeft(_lastBlocked) != NULL)
            {
                _checkSib = clientSibLeft(_lastBlocked);
                _oldSib = _lastBlocked;
                _holdSib = NULL;
                while (_checkSib->mark() == BLOCKED)
                {
                    _checkSib->mark(UNBLOCKED);
                    _checkSib->_parent = _parent;
                    _parent->_pertChildCount++;
                    _holdSib = clientNextSib(_checkSib,_oldSib);
                    _oldSib = _checkSib;
                    _checkSib = _holdSib;
                    if (_checkSib == NULL)
                    {
                        cerr << "ERROR: PQTree->Bubble: Blocked node as "
                             << " endmost child of a Q_NODE." << endl;
                        _checkSib = _oldSib;
                    }
                }
            }

            if (clientSibRight(_lastBlocked) != NULL)
            {
                _checkSib = clientSibRight(_lastBlocked);
                _oldSib = _lastBlocked;
                _holdSib = NULL;
                while (_checkSib->mark() == BLOCKED)
                {
                    _checkSib->mark(UNBLOCKED);
                    _checkSib->_parent = _parent;
                    _parent->_pertChildCount++;
                    _holdSib = clientNextSib(_checkSib,_oldSib);
                    _oldSib = _checkSib;
                    _checkSib = _holdSib;
                    if (_checkSib == NULL)
                    {
                        cerr << "ERROR: PQTree->Bubble: Blocked node as "
                             << " endmost child of a Q_NODE." << endl;
                        _checkSib = _oldSib;
                    }
                }
            }
        }

        delete _processNodes;

        return TRUE;
    }

    /*********************************************************************
```

```
pqtree::Reduce
The only difference between pqtree::Reduce and PQTree::Reduce, is the
numbering of additional debugging files made. For further explantion,
see [Lei97].

Note: If PRINT is defined, and prinitYes is set to TRUE by the call to
Reduce, a file is made for each iteration of the while loop.
*****************************************************************/

#ifdef PRINT
template<class T,class X,class Y>
int pqtree<T,X,Y>::Reduce(int number,key<T,X,Y> **ArrayOfElements,
                          int iterationNumber,int printYes)
#else
template<class T,class X,class Y>
int pqtree<T,X,Y>::Reduce(int number,
                          key<T,X,Y> **ArrayOfElements)
#endif

{


    int                    i                = 0;
    node<T,X,Y>*           _checkLeaf       = NULL;
    node<T,X,Y>*           _checkNode       = NULL;
    int                    _pertLeafCount   = 0;
    queue<node<T,X,Y>*>*   _processNodes    = new queue<node<T,X,Y>*>;

#ifdef PRINT
    int                    _tens;
    int                    _count = 1;
    char                   _number[8];
#endif


    for (i = 0; i <= (number-1); i++)
    {
        _checkLeaf = ArrayOfElements[i]->nodePointer();
        _checkLeaf->status(FULL);
        _checkLeaf->_pertLeafCount = 1;
        _processNodes->enQueue(_checkLeaf);
        _pertLeafCount++;
    }



    _checkNode = _processNodes->front();
    while ((_checkNode != NULL) && (_processNodes->queueSize() > 0))
    {
        _checkNode = _processNodes->deQueue();
        if (_checkNode->_pertLeafCount < _pertLeafCount)
        {

            _checkNode->_parent->_pertLeafCount =
                _checkNode->_parent->_pertLeafCount + _checkNode->_pertLeafCount;

            _checkNode->_parent->_pertChildCount--;
            if (!_checkNode->_parent->_pertChildCount)
                _processNodes->enQueue(_checkNode->_parent);

            if (!template_L1(_checkNode,FALSE))
            if (!template_P1(_checkNode,FALSE))
```

```
                if (!template_P3(_checkNode))
                if (!template_P5(_checkNode))
                if (!template_Q1(_checkNode,FALSE))
                if (!template_Q2(_checkNode,FALSE))
                    _checkNode= NULL;

#ifdef PRINT
                if (printYes)
                {
                    _tens = iterationNumber*10;
                    if (_tens > 99) _tens = _tens*10;
                    _tens = _tens + _count++;
                    sprintf(_number,"%d",_tens);
                    printOutCurrentTree("RED_",_number);
                }
#endif
            }
            else
            {
                if (!template_L1(_checkNode,TRUE))
                if (!template_P1(_checkNode,TRUE))
                if (!template_P2(&_checkNode))
                if (!template_P4(&_checkNode))
                if (!template_P6(&_checkNode))
                if (!template_Q1(_checkNode,TRUE))
                if (!template_Q2(_checkNode,TRUE))
                if (!template_Q3(_checkNode))
                    _checkNode = NULL;

#ifdef PRINT
                if (printYes)
                {
                    _tens = iterationNumber*10;
                    _tens = _tens + _count++;
                    sprintf(_number,"%d",_tens);
                    printOutCurrentTree("RED_",_number);
                }
#endif
            }
        }

        _pertinentRoot = _checkNode;

        delete _processNodes;
        if (_pertinentRoot == NULL)
            return FALSE;
        else
            return TRUE;
}


/***********************************************************************
Template procedrues.
Only changes from PQTree is the calls to procedures in Boundary.
The appropriate information is obtained, both to decide what procedure
to call, and to supply the needed parameters.
a description of what is done is found in thesis (4.2)
***********************************************************************/

/***********************************************************************
                                    template_L1
***********************************************************************/
```

```
template<class T,class X,class Y>
int pqtree<T,X,Y>::template_L1(node<T,X,Y> *node_ptr,
                                   int isRoot)
{
      int numb = 0;
      chain* ch_ptr = NULL;
      element* elem = NULL;

      if ((node_ptr->type() == LEAF) && (node_ptr->status() == FULL))
      {
            if (!isRoot)
            {
                  node_ptr->_parent->fullChildren->push(node_ptr);

                  /* UPDATE PARENT BOUNDARY */
                  if (node_ptr->getNodeInfo() != NULL)
                  {
                        ch_ptr = node_ptr->getNodeInfo()->userStructInfo();
                        numb = ch_ptr->first();
                        node_ptr->setNodeInfo(NULL);
                        /* Chain will be held by parent's boundary */
                  }
                  else numb = node_ptr->getKey()->userStructKey()->from();
                  if (node_ptr->_parent->identificationNumber() != -1)
                        elem = node_ptr->_parent->getInternal()->userStructInternal()->
                              updateBound(numb, FULL, ch_ptr);
                  if (elem != NULL)
                        pertinentElements->push(elem);
            }
#ifdef PRINT
            cout << "Template L1 successfull for node "
                  << node_ptr->identificationNumber() << endl;
#endif
            return TRUE;
      }
      else
            return FALSE;
}

/*********************************************************************
                                   template_P1
*********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P1(node<T,X,Y> *node_ptr,
                                   int isRoot)
{
      if (node_ptr->type() != P_NODE ||
            node_ptr->fullChildren->count() != node_ptr->_childCount)
            return FALSE;
      else
      {
            node<T,X,Y>          *parent      = NULL;
            nodeInfo<T,X,Y>  *info_ptr  = NULL;
            chain                *ch_ptr      = NULL;
            boundary             *bound       = NULL;
            element              *elem        = NULL;
            int                  numb         = 0;
            int                  from         = 0;

            node_ptr->status(FULL);
```

```
            /* CREATE CHAIN */
            if (node_ptr->getNodeInfo() != NULL) /* Chain exists */
                  ch_ptr = node_ptr->getNodeInfo()->userStructInfo();
            bound = node_ptr->getInternal()->userStructInternal();

            if (bound->joint() == NULL) /* proper P-node */
            {
                  parent = node_ptr->_parent;
                  if (parent != NULL && parent->identificationNumber() != -1 &&
                        parent->getInternal()->userStructInternal()->joint() == NULL)
                        from = parent->getInternal()->userStructInternal()->vertexNumber();
                  bound->properP1(iterationNumber, from,
                                          node_ptr->fullChildren->count(), &ch_ptr);
                  numb = bound->vertexNumber();
            }
            else                                   /* non proper P-node */
            {
                  bound->Q1(iterationNumber, &ch_ptr);
                  numb = bound->joint()->number();
            }

            if (!isRoot)
            {
                  if (ch_ptr != NULL)
                  {
                        numb = ch_ptr->first();
                        node_ptr->_parent->fullChildren->push(node_ptr);
                        node_ptr->setNodeInfo(NULL);
                        /* Chain will be held by parent's boundary */
                  }

                  /* UPDATE PARENT BOUNDARY */
                  if (node_ptr->_parent->identificationNumber() != -1)
                        elem = node_ptr->_parent->getInternal()->userStructInternal()->
                              updateBound(numb, FULL, ch_ptr);
                  if (elem != NULL)
                        pertinentElements->push(elem);
            }
            else if (node_ptr->getNodeInfo() == NULL && ch_ptr != NULL)
            {
                  /* A new chain has been built, give it to the P-node, ROOT(T,S), */
                  /* to pass on to the new vertex */
                  info_ptr= new nodeInfo<T,X,Y>(*ch_ptr);
                  ch_ptr->setInfo(info_ptr);
                  node_ptr->setNodeInfo(info_ptr);
            }
#ifdef PRINT
            cout << "Template P1 succesfull for node "
                   << node_ptr->identificationNumber() << endl;
#endif
            return TRUE;
      }
}

/**********************************************************************
                                    template_P2
**********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P2(node<T,X,Y> **node_ptr)
```

```
{
	node<T,X,Y>		*_newNode  =  NULL;
	nodeInfo<T,X,Y> *_newInfo  =  NULL;
	chain			*ch_ptr    =  NULL;

	if  ((*node_ptr)->type() != P_NODE ||
		(*node_ptr)->partialChildren->count() > 0)
		return FALSE;

	else
	{
		(*node_ptr)->_childCount = (*node_ptr)->_childCount -
			(*node_ptr)->fullChildren->count() + 1;
		// Gather all full children of node_ptr
		// as children of the new P-node.
		// Delete them from node_ptr.

		_newNode = createNodeAndCopyFullChildren((*node_ptr)->fullChildren);
		// Correct parent-pointer and
		// sibling-pointers of the new P-node.

		_newNode->_parent = (*node_ptr);
		_newNode->_sibRight = (*node_ptr)->_referenceChild->_sibRight;
		_newNode->_sibLeft = _newNode->_sibRight->_sibLeft;
		_newNode->_sibLeft->_sibRight = _newNode;
		_newNode->_sibRight->_sibLeft = _newNode;
		_newNode->_parentType = P_NODE;

		/* CREATE CHAIN */
		(*node_ptr)->getInternal()->userStructInternal()->
			P2(_newNode->_childCount, &ch_ptr);
		if (ch_ptr != NULL)
		{
			_newInfo = new nodeInfo<T,X,Y>(*ch_ptr);
			ch_ptr->setInfo(_newInfo);
			_newNode->setNodeInfo(_newInfo);
		}
		// The new P-node now is the root of
		// the pertinent subtree.
		(*node_ptr) = _newNode;

#ifdef PRINT
		cout << "Template P2 succesfull."
			<< (*node_ptr)->identificationNumber() << endl;
#endif

		return TRUE;
	}
}

/***********************************************************************
				template_P3
***********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P3(node<T,X,Y> *node_ptr)
{
	pqNode<T,X,Y>	*_newNode	= NULL;
	node<T,X,Y>	*_newQnode	= NULL;
	node<T,X,Y>	*_newPnode	= NULL;
	node<T,X,Y>	*_emptyNode	= NULL;
	internal<T,X,Y> *_newInternal = NULL;
```

```
boundary        *_bound       = NULL;
boundary        *_bound_2     = NULL;
element         *elem         = NULL;
int             number        = 0;
int             fullCount     = 0;

if (node_ptr->type() != P_NODE || node_ptr->partialChildren->count() > 0)
    return FALSE;
else
{
    _newNode = new pqNode<T,X,Y>(_identificationNumber++,Q_NODE,PARTIAL);
    _newQnode = _newNode;
    _pertinentNodes->push(_newQnode);

    exchangeNodes(node_ptr,_newQnode);
    node_ptr->_parent = _newQnode;
    node_ptr->_parentType = Q_NODE;

    _newQnode->_leftEndmost = (node_ptr);
    _newQnode->_childCount = 1;


    fullCount = node_ptr->fullChildren->count();

    if (fullCount > 0)
    {
        node_ptr->_childCount = node_ptr->_childCount - fullCount;

        _newPnode = createNodeAndCopyFullChildren(node_ptr->fullChildren);
        _newPnode->_parentType = Q_NODE;

        // Update _newQnode.
        _newQnode->_childCount++;
        _newQnode->fullChildren->push(_newPnode);
        // Update sibling pointers.
        node_ptr->_sibRight = _newPnode;
        _newPnode->_sibLeft = node_ptr;
        _newQnode->_rightEndmost = _newPnode;
        _newPnode->_parent = _newQnode;
    }

    // Check if node_ptr contains only one son.
    // If so, node_ptr will be deleted from the tree.
    _emptyNode = node_ptr->_referenceChild;
    checkIfOnlyChild(_emptyNode,node_ptr);
    // Update partial_children stack of
    // the parent of the new Q-node.
    _newQnode->_parent->partialChildren->push(_newQnode);

    _bound = node_ptr->getInternal()->userStructInternal();

    if (_bound->joint() == NULL) /* Proper P-node */
    {
        elem = _bound->properP3(fullCount);
        pertinentElements->push(elem);

        number = _bound->vertexNumber();
        /* If node_ptr is kept in the tree, it must get a new boundary,
           and loose its chain if it had one*/
        if (node_ptr->status() != TO_BE_DELETED)
        {
            _bound_2 = new boundary(_bound->vertexNumber());
```

```
                _newInternal = new internal<T,X,Y>(*_bound_2);
                node_ptr->setInternal(_newInternal);
            }
        }
        else
        {   /* Non-proper P-node. node_ptr is deleted from the tree */
            _bound->nonRootQ2(NULL, NULL);          /* Calls setFull() */
            number = _bound->joint()->number();
        }

        /* Give new Q-node boundary and chain of old P-node */

        _newQnode->setInternal(node_ptr->getInternal());
        if (node_ptr->getNodeInfo() != NULL)
        {
            _newQnode->setNodeInfo(node_ptr->getNodeInfo());
            number = node_ptr->getNodeInfo()->userStructInfo()->first();
            /* number holds number of element to update if parent has boundary */
            node_ptr->setNodeInfo(NULL); /* If P-node is kept, it doesn't */
                                         /*    have a chain anymore */
        }

        /* UPDATE PARENT BOUNDARY */
        if (node_ptr->_parent->identificationNumber() != -1)
            elem =_newQnode->parent()->getInternal()->userStructInternal()->
                updateBound(number, PARTIAL, NULL);
        if (elem != NULL)
            pertinentElements->push(elem);

#ifdef PRINT
        cout << "Template P3 succesfull."
             << node_ptr->identificationNumber() << endl;
#endif

        return TRUE;
    }
}

/*********************************************************************
                         template_P4
*********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P4(node<T,X,Y> **node_ptr)
{
    node<T,X,Y>      *_partialChild = NULL;    // pointer to the _partialChild
    boundary         *partialBound  = NULL;
    boundary         *nodeBound     = NULL;
    chain            *partialChain  = NULL;
    int              count;

    if ((*node_ptr)->type() != P_NODE     ||
        (*node_ptr)->partialChildren->count() != 1)
        return FALSE;

    else
    {
        count = (*node_ptr)->fullChildren->count();
        _partialChild = (*node_ptr)->partialChildren->pop();

        copyFullChildrenToPartial(*node_ptr,_partialChild);
```

```
        partialBound = _partialChild->getInternal()->userStructInternal();
        nodeBound = (*node_ptr)->getInternal()->userStructInternal();
        if (_partialChild->getNodeInfo() != NULL)
        {
            partialChain = _partialChild->getNodeInfo()->userStructInfo();
            _partialChild->setNodeInfo(NULL); /* The chain will be thrown away */
        }
        if (nodeBound->joint() == NULL) /* P-node proper */
            /* Both boundaries are updated correctly */
            nodeBound->properP4(partialBound, count, iterationNumber,
                                partialChain);
        else /* P-node not proper, it has one partial and one full child! */
        {    /* Its boundary resembles template Q2. */
            nodeBound->rootQ2(partialBound, partialChain, iterationNumber);
            /* node_ptr will disappear from the tree, its boundary must be
               transferred to _partialChild */
            _partialChild->setInternal((*node_ptr)->getInternal());
            delete partialBound; /* HERE? Something rootQ2 should handle?? */
        }
        // If node_ptr does not have any
        // empty children, then it has to
        // be deleted and the partial node
        // is occupying its place in the tree.
        checkIfOnlyChild(_partialChild,*node_ptr);

        /* If node_ptr is deleted from the tree, its chain must be passed on */
        if ((*node_ptr)->status() == TO_BE_DELETED &&
            (*node_ptr)->getNodeInfo() != NULL)
            _partialChild->setNodeInfo((*node_ptr)->getNodeInfo());

        // The partial child now is
        // root of the pertinent subtree.
        *node_ptr = _partialChild;

#ifdef PRINT
        cout << "Template P4 succesfull."
             << (*node_ptr)->identificationNumber() << endl;
#endif
        return TRUE;
    }
}

/*********************************************************************
                            template_P5
*********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P5(node<T,X,Y> *node_ptr)
{

    node<T,X,Y>         *_partialChild      = NULL;
    node<T,X,Y>         *_checkNode         = NULL;
    node<T,X,Y>         *_emptyNode         = NULL;
    internal<T,X,Y>     *_newInternal       = NULL;
    boundary            *nodeBound          = NULL;
    boundary            *partialBound       = NULL;
    boundary            *_bound_2           = NULL;
    chain               *partialChain       = NULL;
    element             *elem               = NULL;
    int                 _emptyChildCount    = 0;
    int                 hasEmpty            = FALSE;
    int                 number              = 0;
```

```
if ((node_ptr->type() != P_NODE) || (node_ptr->partialChildren->count() != 1))
      return FALSE;
else
{
      _emptyChildCount = node_ptr->_childCount -
            node_ptr->fullChildren->count() - 1;
      _partialChild = node_ptr->partialChildren->pop();
      node_ptr->_parent->partialChildren->push(_partialChild);

      removeChildFromSiblings(_partialChild);
      exchangeNodes(node_ptr,_partialChild);

      copyFullChildrenToPartial(node_ptr,_partialChild);

      if (_emptyChildCount > 0)
      {
            hasEmpty = TRUE;
            if (_emptyChildCount == 1)
            {
                  _emptyNode = node_ptr->_referenceChild;
                  removeChildFromSiblings(_emptyNode);
            }
            else
            {
                  _emptyNode = node_ptr;
                  _emptyNode->_childCount = _emptyChildCount;
            }
            if (clientLeftEndmost(_partialChild)->status() == EMPTY)
            {
                  _checkNode = _partialChild->_leftEndmost;
                  _partialChild->_leftEndmost = _emptyNode;
            }
            else if (clientRightEndmost(_partialChild)->status() == EMPTY)
            {
                  _checkNode = _partialChild->_rightEndmost;
                  _partialChild->_rightEndmost = _emptyNode;
            }
            else
                  printf("template_P5: ERROR Endmost_child not found\n");

            linkChildrenOfQnode(_checkNode,_emptyNode);
            _emptyNode->_parent = _partialChild;
            _emptyNode->_parentType = Q_NODE;
            _partialChild->_childCount++;
      }

      // If node_ptr did not have any empty
      // children it has to be deleted.
      if (_emptyChildCount <= 1)
            destroyNode(node_ptr);

      nodeBound = node_ptr->getInternal()->userStructInternal();
      partialBound = _partialChild->getInternal()->userStructInternal();
      if (_partialChild->getNodeInfo() != NULL)
      {
            partialChain = _partialChild->getNodeInfo()->userStructInfo();
            _partialChild->setNodeInfo(NULL); /* The chain will be thrown away */
      }
      if (nodeBound->joint() == NULL) /* Proper P-node */
      {     /* boundary of partial child is updated. */
            nodeBound->properP5(partialBound,hasEmpty,partialChain);
```

```
            number = nodeBound->vertexNumber();
            /* If node_ptr is kept in the tree, it must get a new boundary,
                and loose its chain if it had one*/
            if (node_ptr->status() != TO_BE_DELETED)
            {
                  _bound_2 = new boundary(number);
                  _newInternal = new internal<T,X,Y>(*_bound_2);
                  node_ptr->setInternal(_newInternal);
            }
      }
      else /* Non-proper P-node */
      {
            nodeBound->nonProperP5(partialBound,hasEmpty,partialChain);
            /* node_ptr is deleted, partial node inherits boundary. */
            _partialChild->setInternal(node_ptr->getInternal());
            delete partialBound;
            if (node_ptr->status() == TO_BE_DELETED)

            number = nodeBound->joint()->number();
      }

      if (node_ptr->getNodeInfo() != NULL)
      {
            number = node_ptr->getNodeInfo()->userStructInfo()->first();
            /* This chain now belongs to the partial node */
            _partialChild->setNodeInfo(node_ptr->getNodeInfo());
            node_ptr->setNodeInfo(NULL);
            /* If the P-node is kept, it no longer has a chain */
      }
      if (node_ptr->_parent->identificationNumber() != -1)
            elem = _partialChild->parent()->getInternal()->
                  userStructInternal()->updateBound(number, PARTIAL, NULL);

      if (elem != NULL)
            pertinentElements->push(elem);

#ifdef PRINT
      cout << "Template P5 succesfull."
             << node_ptr->identificationNumber() << endl;
#endif

      return TRUE;
   }
}

/**********************************************************************
                                    template_P6
**********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_P6(node<T,X,Y> **node_ptr)

{
   node<T,X,Y>   *_partial_1          = NULL;
   node<T,X,Y>   *_partial_2          = NULL;
   node<T,X,Y>   *_fullEnd_1          = NULL;
   node<T,X,Y>   *_fullEnd_2          = NULL;
   node<T,X,Y>   *_emptyEnd_2         = NULL;
   node<T,X,Y>   *_realEmptyEnd_2     = NULL;
   boundary      *nodeBound           = NULL;
   boundary      *bound1              = NULL;
   boundary      *bound2              = NULL;
```

```
chain          *chain1          = NULL;
chain          *chain2          = NULL;
int            proper           = TRUE;
if ((*node_ptr)->type() != P_NODE || (*node_ptr)->partialChildren->count() != 2)
    return FALSE;
else
{
    _partial_1 = (*node_ptr)->partialChildren->pop();
    _partial_2 = (*node_ptr)->partialChildren->pop();

    nodeBound = (*node_ptr)->getInternal()->userStructInternal();
    if (nodeBound->joint() != NULL) proper = FALSE;
    bound1 = _partial_1->getInternal()->userStructInternal();
    bound2 = _partial_2->getInternal()->userStructInternal();
    if (_partial_1->getNodeInfo() != NULL)
    {
        chain1 = _partial_1->getNodeInfo()->userStructInfo();
        _partial_1->setNodeInfo(NULL); /* The chain will be thrown away */
    }
    if (_partial_2->getNodeInfo() != NULL)
    {
        chain2 = _partial_2->getNodeInfo()->userStructInfo();
        _partial_1->setNodeInfo(NULL); /* The chain will be thrown away */
    }

    removeChildFromSiblings(_partial_2);
    (*node_ptr)->_childCount--;
    copyFullChildrenToPartial(*node_ptr,_partial_1);

    if (clientLeftEndmost(_partial_1)->status() == FULL)
        _fullEnd_1 = _partial_1->_leftEndmost;
    else if (clientRightEndmost(_partial_1)->status() == FULL)
        _fullEnd_1 = _partial_1->_rightEndmost;
    else
        cout << "ERROR: template_5: "
             << "partial child with no FULL endmost child detected."
             << endl;

    if (clientLeftEndmost(_partial_2)->status() == FULL)
        _fullEnd_2 = _partial_2->_leftEndmost;
    else if(clientLeftEndmost(_partial_2)->status() == EMPTY)
    {
        _emptyEnd_2 = _partial_2->_leftEndmost;
        _realEmptyEnd_2 = clientLeftEndmost(_partial_2);
    }
    else
        cout << "ERROR: template_5: partial child with "
             << "no FULL or EMPTY endmost child detected" << endl;

    if (clientRightEndmost(_partial_2)->status() == FULL)
        _fullEnd_2 = _partial_2->_rightEndmost;
    else if(clientRightEndmost(_partial_2)->status() == EMPTY)
    {
        _emptyEnd_2 = _partial_2->_rightEndmost;
        _realEmptyEnd_2 = clientRightEndmost(_partial_2);
    }
    else
        cout << "ERROR: template_5: partial child with "
             << "no FULL or EMPTY endmost child detected" << endl;

    if (_fullEnd_2 == _emptyEnd_2)
        cout << "ERROR: template_5: partial child with "
```

```
                << "same type of endmost child detected" << endl;

        while (!_partial_2->fullChildren->stackEmpty())
            _partial_1->fullChildren->push(_partial_2->fullChildren->pop());

        linkChildrenOfQnode(_fullEnd_1,_fullEnd_2);
        if (_partial_1->_leftEndmost == _fullEnd_1)
            _partial_1->_leftEndmost = _emptyEnd_2;
        else
            _partial_1->_rightEndmost = _emptyEnd_2;

        _emptyEnd_2->_parent = _partial_1;
        _emptyEnd_2->_parentType = Q_NODE;

        _realEmptyEnd_2->_parent = _partial_1;
        _realEmptyEnd_2->_parentType = Q_NODE;

        _partial_1->_childCount = _partial_1->_childCount +
            _partial_2->_childCount;

        if (proper)
            nodeBound->properP6(bound1, bound2, chain1, chain2, iterationNumber);
        else
        {
            nodeBound->rootQ3(bound1, bound2, chain1, chain2, iterationNumber, 2);
            /* _partial_1 will take node_ptr's place in the tree,
               and should have its boundary */
            _partial_1->setInternal((*node_ptr)->getInternal());
        }

        destroyNode(_partial_2);
        // If node_ptr does not have any
        // empty children, then it has to
        // be deleted and the partial node
        // is occupying its place in the tree.
        checkIfOnlyChild(_partial_1, *node_ptr);

        /* If node_ptr dissappears from the tree, its chain, if it has any,
           must be given to the partial node */
        if (((*node_ptr)->status() == TO_BE_DELETED) &&
            ((*node_ptr)->getNodeInfo() != NULL))
            _partial_1->setNodeInfo((*node_ptr)->getNodeInfo());

        // _partial_1 is now root of the
        // pertinent subtree.
        *node_ptr = _partial_1;

#ifdef PRINT
        cout << "Template P6 succesfull."
             << (*node_ptr)->identificationNumber() << endl;
#endif
        return TRUE;
    }
}

/**********************************************************************
                                template_Q1
**********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_Q1(node<T,X,Y> *node_ptr,
                               int isRoot)
```

```
{
      node<T,X,Y>*       _seqStart  =  NULL;
      node<T,X,Y>*       _seqEnd  =  NULL;
      nodeInfo<T,X,Y>  *info_ptr  =  NULL;
      chain               *ch_ptr      =  NULL;
      element            *elem          =  NULL;
      int                    numb          =  0;

      if  (node_ptr->type()  ==  Q_NODE  &&  node_ptr  !=  _pseudoRoot  &&
            clientLeftEndmost(node_ptr)->status()  ==  FULL  &&
            clientRightEndmost(node_ptr)->status()  ==  FULL)
      {
            if  (checkChain(node_ptr,clientLeftEndmost(node_ptr),&_seqStart,&_seqEnd))
            {
                  node_ptr->status(FULL);
                  if  (node_ptr->getNodeInfo()  !=  NULL)  /* Chain exists */
                        ch_ptr  =  node_ptr->getNodeInfo()->userStructInfo();
                  node_ptr->getInternal()->userStructInternal()->
                        Q1(iterationNumber,  &ch_ptr);

                  if  (!isRoot)
                  {
                        node_ptr->_parent->fullChildren->push(node_ptr);
                        /* UPDATE PARENT BOUNDARY */
                        if  (ch_ptr  !=  NULL)
                        {
                              numb  =  ch_ptr->first();
                              node_ptr->setNodeInfo(NULL);
                              /* chain will be held by parent's boundary */
                        }
                        else  numb  =  node_ptr->getInternal()->userStructInternal()->
                                          joint()->number();
                        if  (node_ptr->_parent->identificationNumber()  !=  -1)
                              elem  =  node_ptr->_parent->getInternal()->
                                    userStructInternal()->updateBound(numb,  FULL,  ch_ptr);
                        if  (elem  !=  NULL)
                              pertinentElements->push(elem);
                  }
                  else  if  (node_ptr->getNodeInfo()  ==  NULL  &&  ch_ptr  !=  NULL)
                  {
                        /* A new chain has been built, give it to the Q-node to pass */
                        /* on to the new vertex */
                        info_ptr=  new  nodeInfo<T,X,Y>(*ch_ptr);
                        ch_ptr->setInfo(info_ptr);
                        node_ptr->setNodeInfo(info_ptr);
                  }
#ifdef  PRINT
            cout  <<  "Template Q1 succesfull for node "
                      <<  node_ptr->identificationNumber()  <<  endl;
#endif
                  return  TRUE;
            }
            else  return  FALSE;
      }
      else  return  FALSE;
}

/**********************************************************************
                                                template_Q2
**********************************************************************/

template<class  T,class  X,class  Y>
```

```
int  pqtree<T,X,Y>::template_Q2(node<T,X,Y>  *node_ptr,  int  isRoot)
{
	node<T,X,Y>      *_fullNode          =  NULL;
	node<T,X,Y>      *_sequenceBegin  =  NULL;
	node<T,X,Y>      *_sequenceEnd      =  NULL;
	node<T,X,Y>      *_partialChild     =  NULL;
	boundary          *partialBound      =  NULL;
	chain             *partialChain      =  NULL;
	element           *elem                  =  NULL;
	int                    _sequenceCons    =  FALSE;
	int                    numb                   =  0;

	if  (node_ptr->type()  !=  Q_NODE  ||  node_ptr->partialChildren->count()  >  1)
		return  FALSE;
	else
	{
		if  (node_ptr->fullChildren->count()  >  0)
		{
			if  (node_ptr->_leftEndmost  !=  NULL)
			{
				_fullNode  =  clientLeftEndmost(node_ptr);
				if  (_fullNode->status()  !=  FULL)  _fullNode  =  NULL;
			}
			if  (node_ptr->_rightEndmost  !=  NULL    &&  _fullNode  ==  NULL)
			{
				_fullNode  =  clientRightEndmost(node_ptr);
				if  (_fullNode->status()  !=  FULL)  _fullNode  =  NULL;
			}

			if  (_fullNode  !=  NULL)
				_sequenceCons  =  checkChain(node_ptr,_fullNode,
								&_sequenceBegin,&_sequenceEnd);
			if  (_sequenceCons  &&  (node_ptr->partialChildren->count()  ==  1))
			{
				node_ptr->partialChildren->startAtBottom();
				_partialChild  =  node_ptr->partialChildren->readNext();
				_sequenceCons  =  FALSE;

				if  (clientSibLeft(_sequenceEnd)  ==  _partialChild  ||
					clientSibRight(_sequenceEnd)  ==  _partialChild)
					_sequenceCons  =  TRUE;
			}
		}
		else
		{
			if  (!node_ptr->partialChildren->stackEmpty())
			{
				node_ptr->partialChildren->startAtBottom();
				_partialChild  =  node_ptr->partialChildren->readNext();
				if  ((clientLeftEndmost(node_ptr)  ==  _partialChild)  ||
					(clientRightEndmost(node_ptr)  ==  _partialChild))
					_sequenceCons  =  TRUE;
			}
		}

		if  (_sequenceCons)
		{
			removeBlock(node_ptr,isRoot);
			if  (_partialChild  !=  NULL)
			{
				if  (_partialChild->getNodeInfo()  !=  NULL)
				{
```

```
                    partialChain = _partialChild->getNodeInfo()->userStructInfo();
                    _partialChild->setNodeInfo(NULL);
                    /* The chain will be thrown away */
                }
                partialBound = _partialChild->getInternal()->userStructInternal();
            }
            if (isRoot)
                node_ptr->getInternal()->userStructInternal()->
                    rootQ2(partialBound, partialChain, iterationNumber);
            else
            {
                node_ptr->getInternal()->userStructInternal()->
                    nonRootQ2(partialBound, partialChain);
                if (node_ptr->getNodeInfo() != NULL)
                    numb = node_ptr->getNodeInfo()->userStructInfo()->first();
                else numb = node_ptr->getInternal()->
                         userStructInternal()->joint()->number();
                if (node_ptr->_parent->identificationNumber() != -1)
                    elem = node_ptr->_parent->getInternal()->
                        userStructInternal()->updateBound(numb, PARTIAL, NULL);
                if (elem != NULL)
                    pertinentElements->push(elem);
            }
#ifdef PRINT
        cout << "Template Q2 succesfull for node "
             << node_ptr->identificationNumber() << endl;
#endif
        }
        return _sequenceCons;
    }
}

/*********************************************************************
                              template_Q3
*********************************************************************/

template<class T,class X,class Y>
int pqtree<T,X,Y>::template_Q3(node<T,X,Y> *node_ptr)

{
    node<T,X,Y>      *_fullChild      = NULL;
    node<T,X,Y>      *_fullStart      = NULL;
    node<T,X,Y>      *_fullEnd        = NULL;
    node<T,X,Y>      *partial_1       = NULL;
    node<T,X,Y>      *partial_2       = NULL;
    boundary         *bound_1         = NULL;
    boundary         *bound_2         = NULL;
    chain            *chain_1         = NULL;
    chain            *chain_2         = NULL;
    int              cons_sequence    = FALSE;
    int              found            = FALSE;
    int              numberOfPartials = 0;

    if (node_ptr->type() != Q_NODE || node_ptr->partialChildren->count() >= 3)
        return FALSE;
    else
    {
        numberOfPartials = node_ptr->partialChildren->count();

        if (!node_ptr->fullChildren->stackEmpty())
        {
            node_ptr->fullChildren->startAtBottom();
```

```
        _fullChild = node_ptr->fullChildren->readNext();
        cons_sequence = checkChain(node_ptr,_fullChild,&_fullStart,&_fullEnd);
        if (cons_sequence)
        {
                node_ptr->partialChildren->startAtBottom();
                while (!node_ptr->partialChildren->readLast())
                {
                        partial_1 = node_ptr->partialChildren->readNext();
                        found = FALSE;
                        if ( (clientSibLeft(_fullStart) == partial_1)   ||
                               (clientSibRight(_fullStart) == partial_1) ||
                               (clientSibLeft(_fullEnd) == partial_1)     ||
                               (clientSibRight(_fullEnd) == partial_1)       )
                                found = TRUE;
                        if (!found)
                                cons_sequence = found;
                }
        }
}

else if (node_ptr->partialChildren->count() == 2)
{
        node_ptr->partialChildren->startAtBottom();
        partial_1 = node_ptr->partialChildren->readNext();
        partial_2 = node_ptr->partialChildren->readNext();
        if ( (clientSibLeft(partial_1) == partial_2)   ||
               (clientSibRight(partial_1) == partial_2)     )
                found = TRUE;
        cons_sequence = found;
}

int isRoot = TRUE;
if (cons_sequence)
{
        if (node_ptr != _pseudoRoot)
        {
                if (numberOfPartials > 0)
                {
                        node_ptr->partialChildren->startAtBottom();
                        partial_1 = node_ptr->partialChildren->readNext();
                        bound_1 = partial_1->getInternal()->userStructInternal();
                        if (partial_1->getNodeInfo() != NULL)
                        {
                                chain_1 = partial_1->getNodeInfo()->userStructInfo();
                                partial_1->setNodeInfo(NULL);
                                /* The chain will be thrown away */
                        }
                        if (numberOfPartials == 2)
                        {
                                partial_2 = node_ptr->partialChildren->readNext();
                                bound_2 = partial_2->getInternal()->userStructInternal();
                                if (partial_2->getNodeInfo() != NULL)
                                {
                                        chain_2 = partial_2->getNodeInfo()->userStructInfo();
                                        partial_2->setNodeInfo(NULL);
                                        /* The chain will be thrown away */
                                }
                        }
                }

                removeBlock(node_ptr,isRoot);
```

```
                    node_ptr->getInternal()->userStructInternal()->
                         rootQ3(bound_1, bound_2, chain_1, chain_2, iterationNumber,
                                   numberOfPartials);
               }
#ifdef PRINT
          cout << "Template Q3 succesfull for node "
                 << node_ptr->identificationNumber() << endl;
#endif
          }
          return cons_sequence;
     }
}

#endif
```

A.3 Code files for parameter types

A.3.1 edge

```
/*****************************************************
                        File: edge.h

                        Author: Gørril Vollen
                        Last update: 18/02/1998
*****************************************************/
#ifndef EDGE_H
#define EDGE_H

/*****************************************************
class edge
Definition of class edge. Class edge is teh type of the
information stored by every key/leaf in the tree, type T
*****************************************************/

#include <iostream.h>

class edge
{
private:
     int _from;          /* number of tail vertex */
     int _to;            /* number of head vertex */
     int _mapNumber; /* number used to handle edges outside of PQ-tree */

public:
     edge() {}           /* default constructor */
     edge(int count, int f, int t)
     {
          _from = f;
          _to = t;
          _mapNumber = count;
     }
     ~edge() {}          /* default destructor */
     int from() { return _from; }
     int to() { return _to; }
     void print() {cout << "(" << _from << "," << _to << ")";}

};

#endif
```

A.3.2 Chain

```
/*******************************************************************
                                    File: chain.h

                                    Author: Gørril Vollen
                                    Last update: 20/02/1998
*******************************************************************/
#ifndef CHAIN_H
#define CHAIN_H
#include <strstream.h>
#include <iostream.h>
#include "element.h"
#include "stack.h"
#include "nodeInfo.h"
#include "node.h"

class edge;
class boundary;

/***********************************************************************
class chain
Class chain is the type of the information stored at every
node in the tree, type X.
Attributes: _chainSt - stack of integers, the numbers of the
                              vertices in the chain this chain represents.
               info_ptr - pointer to the nodeInfo object that points to
                              this chain object.
***********************************************************************/
class chain
{
     /*Invariant: When read, the first call is first(), then getNext
        while return-value > 0. If used by insertChain, ALWAYS call
        last() before reading the rest. */

private:
     stack<int>* _chainSt;
     nodeInfo<edge, chain, boundary>* info_ptr;

public:
     chain() { _chainSt = new stack<int>; }
     chain(int from, int to)
     {
          _chainSt = new stack<int>;
          insert(from); insert(to);
     }
     chain(int from, int number, int to)
     {
          _chainSt = new stack<int>;
          insert(from); insert(number); insert(to);
     }
     chain(chain &c) { _chainSt = c._chainSt; }
     ~chain()
     {
          delete _chainSt;
          _chainSt = NULL;
          delete info_ptr;
          info_ptr = NULL;
     }
```

```
void setInfo(nodeInfo<edge, chain, boundary>* info)
{
      info_ptr = info;
}

void insert(int number) /*insert new number into stack;*/
{
      _chainSt->push(number);
}

chain* getChain() /*returns this object if _chainSt not empty, else NULL*/
{
      if (!_chainSt->stackEmpty()) return this;
      else return NULL;
}

int first() /*returns the bottom number on the stack*/
{
      _chainSt->startAtBottom();
      return _chainSt->readNext();
}

int last() /*removes and returns top number on the stack*/
{
      return _chainSt->pop();
}

int getNext() /*returns the next read number (bottom-up), -1 if all read*/
{
      if (!_chainSt->readLast()) return _chainSt->readNext();
      else return -1;
}

void addChain(chain* addition) /* stack of addition is placed */
{                                             /* on top of this stack. */
      int i;
      if (_chainSt->stackEmpty)
            cout << "ERROR in chain::addChain: this chain is empty!" << endl;
      else
      {
            i = last();
            if (i != addition->first()) /*Make sure this number is not*/
                  insert(i);                  /*doubly on the stack*/
            i = addition->first();
            while (i != -1)
            {
                  insert(i);
                  i = addition->getNext();
            }
      }
      delete addition;
}

char* print()
{
      ostrstream txt;
      char out[INFOSIZE];
      int i;

      if (!_chainSt->stackEmpty())
      {
```

```
            txt << " Chain: ";
            _chainSt->startAtBottom();
            i = _chainSt->readNext();
            txt << i;
            while (!_chainSt->readLast())
            {
                  i = _chainSt->readNext();
                  txt  << "-" << i;
            }
            txt << ends;
        }
        strcpy(out, txt.str());
        strstreambuf * buf_ptr = txt.rdbuf();
        buf_ptr->freeze(0);
        return out;
    }
};

#endif
```

A.3.3 Boundary

```
/*********************************************************************
                                           File: boundary.h

                                           Author: Gørril Vollen
                                           Last update: 20/02/1998
*********************************************************************/

/*****************************************************
Class boundary is the type of the internal information,
designed for P and Q nodes. Type Y.
*****************************************************/
#ifndef BOUNDARY_H
#define BOUNDARY_H
#include "element.h"
#include "node.inc"
#include "internal.h"
//#define DEBUG

class edge;
class chain;

class boundary
{
private:
    int _vertexNumber;

    element* _joint;

    element* _end1; /* Following the 'left' pointer out
                       of 'joint' leads to 'end1' first */
    element* _end2; /* Following the 'right' pointer out
                       of 'joint' leads to 'end2' first */
    element* _begFull; /* When set, begFull will be encountered */
    element* _endFull; /* before endFull, walking from left to  */
    /* right out of joint. */
    element* _begEmpty; /* The element "to the left of" begFull */
    element* _endEmpty; /* The element "to the right of" endFull */

    stack<chain*> *_fullChains;
```

```
    internal<edge,chain,boundary>* internal_ptr;

public:
    boundary()
    {
        _vertexNumber = 0;
        _joint = NULL; _end1 = NULL; _end2 = NULL;
        _endEmpty = NULL; _begEmpty = NULL;
        _begFull = NULL; _endFull = NULL;
        _fullChains = NULL; internal_ptr = NULL;
    }
    boundary(int vertexNumber)
    {
        _vertexNumber = vertexNumber;
        _joint = NULL; _end1 = NULL; _end2 = NULL;
        _endEmpty = NULL; _begEmpty = NULL;
        _begFull = NULL; _endFull = NULL;
        _fullChains = NULL; internal_ptr = NULL;
#ifdef DEBUG
        cout << "boundary " << _vertexNumber << endl;
#endif
    }
    boundary(int vertexNumber, internal<edge,chain,boundary>* ptr)
    {
        _vertexNumber = vertexNumber;
        internal_ptr = ptr;
        _joint = NULL; _end1 = NULL; _end2 = NULL;
        _fullChains = new stack<chain*>;
#ifdef DEBUG
        cout << "boundary " << _vertexNumber << endl;
#endif
    }
    ~boundary() {delete _fullChains;}

    int vertexNumber() {return _vertexNumber;}
    void vertexNumber(int number) {_vertexNumber = number;}

    element* joint() {return _joint;}

    element* end1() {return _end1;}
    void end1(element* elem) {_end1 = elem;}

    element* end2() {return _end2;}
    void end2(element* elem) {_end2 = elem;}

    element* begFull() {return _begFull;}
    void begFull(element* elem) {_begFull = elem;}

    element* endFull() {return _endFull;}
    void endFull(element* elem) {_endFull = elem;}

    element* begEmpty() {return _begEmpty;}
    element* endEmpty() {return _endEmpty;}

    int fullChains() { return _fullChains ? _fullChains->count() : 0; }

    void removeFullCh() {if (_fullChains != NULL) _fullChains->cleanupStack();}

    internal<edge,chain,boundary>* getInternal() {return internal_ptr;}
    void setInternal(internal<edge,chain,boundary>* ptr) {internal_ptr = ptr;}

    void setFull();
```

```
    element* updateBound(int number, int status, chain* ch_ptr);

    element* findElem(int number);

    void removeElements(element *from, element *first, element *to);

    void insertChain(element *end1, element *other1, element *end2,
                     element *other2, chain *ch_ptr);

    void partialConnect(element* emptyEnd, element* oldEmpty,
                        element* fullEnd, element* oldFull,
                        chain* emptyChain, chain* fullChain,
                        int i, element **e1, element **e2);
    void partialConnect(element* emptyEnd, element* oldEmpty,
                            element* fullEnd, element* oldFull,
                            chain* partialChain, element **e1, element **e2);

    void doubleConnect(boundary *bound1, boundary *bound2,
                       chain *chain1, chain *chain2, int i);

    void connectToJoint(element* elem, int direction);

    void properP1(int i, int from, int count, chain **ch_ptr);

    void P2(int count, chain **ch_ptr);

    element* properP3(int count);

    void properP4(boundary* bound, int count, int i, chain* partialChain);
    void partialP4(int jointNo, int i, chain* nodeChain, chain* partialChain);

    void nonProperP4(boundary* bound, int i, chain* partialChain);

    void properP5(boundary* bound, int hasEmpty, chain* chain1);
    void partialP5(int jointNo, int hasEmpty, chain* chain1);

    void nonProperP5(boundary* bound, int hasEmpty, chain* chain1);

    void properP6(boundary* partial1, boundary* partial2, chain* chain1,
                  chain* chain2, int i);
    void partialP6(int jointNo, chain* chain1, int i, boundary* partial2, chain* chain2);

    void nonProperP6(boundary* bound1, boundary* bound2, chain* chain1,
                     chain* chain2, int i);

    void Q1(int i, chain **ch_ptr);

    void rootQ2(boundary* bound, chain* partialChain, int i);

    void nonRootQ2(boundary* bound, chain* partialChain);

    void rootQ3(boundary* bound1, boundary* bound2, chain* chain1,
                chain* chain2, int i, int numberOfPartials);

    char* print();

};

#endif

/*****************************************************************
```

```
                                    File: boundary.cc

                                    Author: Gørril Vollen
                                    Last update: 20/02/1998
*********************************************************************/
#include <iostream.h>
#include <fstream.h>
#include "def.inc"
#include "stack.h"
#include "boundary.h"
#include "element.h"
#include "chain.h"

/* *************** setFull *************** */

void boundary::setFull()
{
    element* elem, *oldEl, *nextEl;
#ifdef DEBUG
    cout << "    setFull" << endl;
#endif
    elem = _joint->left();
    oldEl = _joint;
    while (elem != _joint && elem->status() == EMPTY)
    {
        nextEl = elem->getNextEl(oldEl);
        oldEl = elem;
        elem = nextEl;
    }
    _begFull = elem;
    _endEmpty = oldEl;
    /* For å finne _endFull, må jeg gå ut fra joint i den andre retningen! */
    elem = _joint->right();
    oldEl = _joint;
    while (elem != _joint && elem->status() == EMPTY)
    {
        cout << elem->number() << " ";
        nextEl = elem->getNextEl(oldEl);
        oldEl = elem;
        elem = nextEl;
    }
    cout << endl;
    _endFull = elem;
    _begEmpty = oldEl;
} //setFull

/* *************** updateBound *************** */

element* boundary::updateBound(int number, int stat, chain* ch_ptr)
{
    element *elem = NULL;

    if (_joint != NULL) /* There is a boundary that needs updating */
    {
        elem = findElem(number);
        if (elem != NULL) /* elem were found */
        {
            elem->setChain(ch_ptr);
            elem->status(stat);
            if (stat == FULL) elem->child(FALSE);
            /* P3 is FULL,TRUE after inclusion in boundary, */
            /* while FULL nodes will be idle */
```

```
            }
        }
            if (stat == FULL && ch_ptr != NULL)
            {
                if (!_fullChains) _fullChains = new stack<chain*>;
                _fullChains->push(ch_ptr);
            }
        return elem;
} //updateBound

/* *************** findElement *************** */

element* boundary::findElem(int number)
    /* Used by extendBoundary and addChild */
{
        element *elem, *oldEl, *nextEl;

        elem = _joint;
        if (elem->number() == number) return elem;
        else {
            oldEl = elem;
            elem = elem->getNextEl(NULL);
            while (elem->number() != number && elem != _joint)
            {
                nextEl = elem->getNextEl(oldEl);
                oldEl = elem;
                elem = nextEl;
            }
            if (elem->number() == number) return elem;
            else
            {
                cerr << "ERROR in boundary::findElem: No element found!" << endl;
                return NULL;
            }
        };
} //findElement

/* *************** removeElements *************** */

void boundary::removeElements(element *from, element *first, element *to)
{
        element *elem, *nextEl, *oldEl;

        if (from != to)
        {
            elem = first;
            oldEl = from;
            while (elem != to)
            {
                elem->status(TO_BE_DELETED);
                nextEl = elem->getNextEl(oldEl);
                oldEl = elem;
                elem = nextEl;
            }
        }
} //removeElements

/* *************** insertChain *************** */

void boundary::insertChain(element *end1, element *old1, element *end2,
                                        element *old2, chain *ch_ptr)
{
```

```
    element *elem = NULL;
    element *oldEl = NULL;
    int _first, _last, i;

    if (ch_ptr == NULL) {/*MARKER;*/}
    else
    {
        _first = ch_ptr->first();
        _last = ch_ptr->last();
        if ((end1->number() == _first) && (end2->number() == _last))
        {
            i = ch_ptr->getNext();
            if (i != -1)
            {
                elem = new element(i, EMPTY, FALSE, end1, NULL);
                end1->changeElements(old1, elem);
                i = ch_ptr->getNext();
                while (i != -1)
                {
                    oldEl = elem;
                    elem = new element(i, EMPTY, FALSE, oldEl, NULL);
                    oldEl->setNextEl(elem);
                    i = ch_ptr->getNext();
                }
                elem->setNextEl(end2);
                end2->changeElements(old2, elem);
            }
        }
        else if ((end2->number() == _first) && (end1->number() == _last))
        {
            i = ch_ptr->getNext();
            if (i != -1)
            {
                elem = new element(i, EMPTY, FALSE, end2, NULL);
                end2->changeElements(old2, elem);
            }
            i = ch_ptr->getNext();
            while (i != -1)
            {
                oldEl = elem;
                elem = new element(i, EMPTY, FALSE, oldEl, NULL);
                oldEl->setNextEl(elem);
                i = ch_ptr->getNext();
            }
            elem->setNextEl(end1);
            end1->changeElements(old1, elem);
        }
        else cerr << "ERROR in boundary::insertChain: "
                  << "first and/or last number in chain did not match "
                  << "with connection-points!" << endl;

        delete ch_ptr;
    }

} //insertChain

/* **************** partialConnect **************** */
/*          (Parent root of pertinent subtree)          */
/* Connect this boundary to emptyEnd and fullEnd,      */
/* insert new element i, clean up.                      */
/* oldEmpty is partial element on parents boundary. */
```

```
void  boundary::partialConnect(element* emptyEnd, element* oldEmpty,
                                element* fullEnd, element* oldFull,
                                chain* emptyChain, chain* fullChain,
                                int i, element **e1, element **e2)
{
      element  *elem, *other, *newEl;
      chain    *ch_ptr;

      if (oldEmpty != NULL) removeElements(emptyEnd, oldEmpty, fullEnd);

      if (_joint == NULL) /* Only _end1 exists */
      {
            if (emptyChain != NULL)
            {     /* oldEmpty will still be part of boundary*/
                  _joint = new element(oldEmpty->number(), EMPTY, FALSE,
                                       _end1, emptyEnd);
                  _end1->right(_joint);
                  emptyEnd->changeElements(oldEmpty, _joint);
                  insertChain(_joint, _end1, _end1, _joint, emptyChain);
            }
            else
            {
                  emptyEnd->changeElements(oldEmpty, _end1);
                  _end1->right(emptyEnd);
            }
            elem = new element(i, EMPTY, TRUE, fullEnd, _end1);
            _end1->left(elem);
            ch_ptr = _end1->getChain();
            if (ch_ptr != NULL)
                  insertChain(_end1, elem, elem, _end1, ch_ptr);
            if (fullChain != NULL)
                  insertChain(fullEnd, oldFull, elem, fullEnd, fullChain);
            else
                  fullEnd->changeElements(oldFull, elem);
            _end1->status(FULL); /* end1 is no longer PARTIAL */
            *e1 = _end1; //e1 is empty end
            *e2 = elem;  //e2 is full end
      }
      else  /* Full boundary exists, setFull() already called */
      {
            /* Need to know the direction of this boundary, in order to
               connect it correctly */
            if (_end1->status() == EMPTY)
            {
                  if (emptyChain != NULL)
                  {     /*oldEmpty will still be part of boundary*/
                        newEl = new element(oldEmpty->number(), EMPTY, FALSE,
                                            _joint, emptyEnd);
                        _joint->right(newEl);
                        emptyEnd->changeElements(oldEmpty, newEl);
                        insertChain(newEl, _joint, _joint, newEl, emptyChain);
                        if (_joint->child() == TRUE) _end1 = _joint;
                        /* partial was proper P5, _joint is real empty end. */
                  }
                  else
                  {
                        _joint->right(emptyEnd);
                        emptyEnd->changeElements(oldEmpty, _joint);
                  }
                  elem = new element(i, EMPTY, TRUE, fullEnd, _begFull);
                  other = _begFull->getNextEl(_endEmpty);
                  _begFull->changeElements(other, elem);
```

```
                removeElements(_begFull, other, _joint);

                ch_ptr = _begFull->getChain();
                if (ch_ptr != NULL)
                      insertChain(_begFull, elem, elem, _begFull, ch_ptr);
                if (fullChain != NULL)
                      insertChain(fullEnd, oldFull, elem, fullEnd, fullChain);
                else
                      fullEnd->changeElements(oldFull, elem);
                *e1 = _end1; //e1 is empty end
                *e2 = elem;  //e2 is full end
          }
          else if (_end2->status() == EMPTY) /*must be complete boundary*/
          {
                if (emptyChain != NULL)
                {     /*oldEmpty will still be part of boundary*/
                      newEl = new element(oldEmpty->number(), EMPTY, FALSE,
                                                emptyEnd, _joint);
                      _joint->left(newEl);
                      emptyEnd->changeElements(oldEmpty, newEl);
                      insertChain(newEl, _joint, _joint, newEl, emptyChain);
                      if (_joint->child() == TRUE) _end2 = _joint;
                      /* partial was proper P5, _joint is real empty end. */
                }
                else
                {
                      _joint->left(emptyEnd);
                      emptyEnd->changeElements(oldEmpty, _joint);
                }
                elem = new element(i, EMPTY, TRUE, _endFull, fullEnd);
                other = _endFull->getNextEl(_begEmpty);
                _endFull->changeElements(other, elem);
                removeElements(_endFull, other, _joint);

                ch_ptr = _endFull->getChain();
                if (ch_ptr != NULL)
                      insertChain(_endFull, elem, elem, _endFull, ch_ptr);
                if (fullChain != NULL)
                      insertChain(fullEnd, oldFull, elem, fullEnd, fullChain);
                else
                      fullEnd->changeElements(oldFull, elem);
                *e1 = _end2; //e1 is empty end
                *e2 = elem;  //e2 is full end
          }
          else cerr << "ERROR in boundary::partialConnect(pertinentRoot): "
                      << "No empty end of boundary found!" << endl;
    }
} //partialConnect (pertinentRoot)

/* **************** partialConnect **************** */
/*          (Parent not root of pertinent subtree)          */

/* Will need to be redesigned when cases are made.   */
/* How much of partial boundary should substitute     */
/* the partial element, and how much of parents full*/
/* sequence should be swallowed?                          */

/* P5 and Q2 can have only partial, so fullEnd might be idle. */
/* e1 and e2 wil be full and empty end of this boudary.          */

void boundary::partialConnect(element* emptyEnd, element* oldEmpty,
                                    element* fullEnd, element* oldFull,
```

```
                                        chain* partialChain,
                                        element** e1, element** e2)
{
    element *newEl;

    removeElements(emptyEnd, oldEmpty, fullEnd);
    if (_joint == NULL) /* Only _end1 exists */
    {
        if (partialChain != NULL)
        {   /* oldEmpty will still be part of boundary*/
            _joint = new element(oldEmpty->number(), EMPTY, FALSE,
                                 _end1, emptyEnd);
            _end1->right(_joint);
            emptyEnd->changeElements(oldEmpty, _joint);
            insertChain(_joint, _end1, _end1, _joint, partialChain);
        }
        else
        {
            _end1->right(emptyEnd);
            emptyEnd->changeElements(oldEmpty, _end1);
        }
        _end1->left(fullEnd);
        fullEnd->changeElements(oldFull, _end1);
        _end1->status(FULL); /* end1 is no longer PARTIAL */
        *e1 = _end1;        /* _end1 is partial and only element, */
        *e2 = _end1;        /* therefore both empty and full end. */
        /* Up to caller to pick which one (if any) to update. */
    }
    else  /* Full boundary exists, setFull() already called */
    {
        /* Need to know the direction of this boundary, in order to
           connect it correctly */
        if (_end1->status() == EMPTY)
        {
            if (partialChain != NULL)
            {   /*oldEmpty will still be part of boundary*/
                newEl = new element(oldEmpty->number(), EMPTY, FALSE,
                                    _joint, emptyEnd);
                _joint->right(newEl);
                emptyEnd->changeElements(oldEmpty, newEl);
                insertChain(newEl, _joint, _joint, newEl, partialChain);
                if (_joint->child() == TRUE) _end1 = _joint;
                /* partial was proper P5, _joint is real empty end. */
            }
            else
            {
                _joint->right(emptyEnd);
                emptyEnd->changeElements(oldEmpty, _joint);
            }
            fullEnd->changeElements(oldFull, _endFull);
            _endFull->changeElements(_begEmpty, fullEnd);
            removeElements(_endFull, _begEmpty, _joint);
            *e1 = _end1;       /* empty end */
            *e2 = _endFull; /* _endFull == _end2 */
        }
        else if (_end2->status() == EMPTY)
        {
            if (partialChain != NULL)
            {
                newEl = new element(oldEmpty->number(), EMPTY, FALSE,
                                    emptyEnd, _joint);
                _joint->left(newEl);
```

```
                emptyEnd->changeElements(oldEmpty, newEl);
                insertChain(newEl, _joint, _joint, newEl, partialChain);
                if (_joint->child() == TRUE) _end2 = _joint;
                /* partial was proper P5, _joint is real empty end. */
            }
            else
            {
                _joint->left(emptyEnd);
                emptyEnd->changeElements(oldEmpty, _joint);
            }
            fullEnd->changeElements(oldFull, _begFull);
            _begFull->changeElements(_begEmpty, fullEnd);
            removeElements(_begFull, _endEmpty, _joint);
            *e1 = _end2;      /* empty end */
            *e2 = _begFull; /* _begFull == _end1 */
        }
        else
        {
            cerr << "ERROR in boundary::partialConnect(pertinentRoot): "
                 << "No empty end of boundary found!" << endl;
        }
    }
} //partialConnect (not pertinentRoot)

/* *************** doubleConnect *************** */

void boundary::doubleConnect(boundary *bound1, boundary *bound2,
                             chain *chain1, chain *chain2, int i)
{
    /* bound1 is the boundary located at _begFull, and bound2 at _endFull */
    element *other, *e1, *e2;
    chain *ch_ptr;

    /* Place the first boundary where the first partial is */
    other = _endFull->getNextEl(_begEmpty);
    if (_end1 == _begFull) /* _end1 will change */
        bound1->partialConnect(_endEmpty, _begFull, _endFull, other,
                               chain1, &_end1, &e2);
    else /* _end1 will not change */
        bound1->partialConnect(_endEmpty, _begFull, _endFull, other, chain1, &e1, &e2);

    /* Take care of the second partial */

    other = _endFull->getNextEl(_begEmpty);
    ch_ptr = _endFull->getChain();
    if (_end2 == _endFull) /* _end2 will change */
        bound2->partialConnect(_begEmpty, _endFull, other, _endFull, chain2,
                               ch_ptr, i, &_end2, &e2);
    else /* _end2 will not change */
        bound2->partialConnect(_begEmpty, _endFull, other, _endFull, chain2,
                               ch_ptr, i, &e1, &e2);
} //doubleConnect

/* *************** P1 *************** */
/* Extends chain if it exists, and builds a new one if not. */

void boundary::properP1(int i, int from, int count, chain **ch_ptr)
{
    if (*ch_ptr != NULL)
    {
        if (_fullChains && count == _fullChains->count())
            (*ch_ptr)->addChain(_fullChains->pop());
```

```
		else
			(*ch_ptr)->insert(i);
	}
	else if (from != 0)
	{	/* This node is child of a proper P-node, */
		/* and chain should start there */
		*ch_ptr = new chain();
		(*ch_ptr)->insert(from);
		if (_fullChains && count == _fullChains->count())
			(*ch_ptr)->addChain(_fullChains->pop());
		else
		{
			(*ch_ptr)->insert(_vertexNumber);
			(*ch_ptr)->insert(i);
		}
	}
	else
	{
		if (_fullChains && count == _fullChains->count())
			*ch_ptr = _fullChains->pop();
		else
			*ch_ptr = new chain(_vertexNumber, i);
	}
	/* Since this boundary will be deleted by the end of this iteration,*/
	/* there is no need to clean up _fullChains */
} //properP1

/* **************** P2 **************** */
/* Returns a chain in ch_ptr if every full child of P2 has one */

void boundary::P2(int count, chain **ch_ptr)
{
	if (_fullChains)
	{
		if (_fullChains->count() == count)
			*ch_ptr = _fullChains->pop();
		_fullChains->cleanupStack();
	}
} //P2

/* **************** properP3 **************** */
/* Instantiates the boundary with one element */
/* (_end1), to be able to take care of its     */
/* chain, if it exists.                         */

element* boundary::properP3(int count)
{
	element *elem;

	elem = new element(_vertexNumber, FULL, TRUE);
	if (_fullChains)
	{
		if (_fullChains->count() == count)
			elem->setChain(_fullChains->pop());
		_fullChains->cleanupStack();
	}
	_end1 = elem;
	return elem;
} //properP3

/* ***************** properP4 ***************** */
/* Check if new full child of partial has chain.*/
```

```
/* Let partial extend its own boundary.              */

void boundary::properP4(boundary* partialBound, int count, int i,
                                    chain* partialChain)
{
      chain* ch_ptr = NULL;

      if (_fullChains)
      {
            if (_fullChains->count() == count)
                  ch_ptr = _fullChains->pop();
            _fullChains->cleanupStack();
      }
      partialBound->partialP4(_vertexNumber, i, ch_ptr, partialChain);
} //properP4

/* **************** partialP4 **************** */
/* Cleans up full end. Makes a complete            */
/* boundary for the new tree by including i.     */

void boundary::partialP4(int jointNo, int i, chain* nodeChain,
                                     chain* partialChain)
{
      chain *ch_ptr =NULL;
      element *elem, *other;

      if (_joint == NULL) /* Partial is proper P3, has only _end1 */
      {
            _joint = new element(jointNo, EMPTY, FALSE, _end1, NULL);
            _end2 = new element(i, EMPTY, TRUE, _joint, _end1);
            _joint->right(_end2);
            _end1->left(_end2);
            _end1->right(_joint);

            if (partialChain != NULL)
                  insertChain(_joint, _end1, _end1, _joint, partialChain);

            /* properP3 has chain between end1 and end2 if end1 has chain */
            ch_ptr = _end1->getChain();
            if (ch_ptr != NULL)
                  insertChain(_end1, _end2, _end2, _end1, ch_ptr);
            if (nodeChain != NULL)
                  insertChain(_joint, _end2, _end2, _joint, nodeChain);
      }
      else   /*Boundary has at least two elements*/
      {        /*joint, end1 and end2 are all valid pointers.*/

            if (_end1->status() == EMPTY)
            {
                  if (jointNo != _joint->number())
                  {
                        /*If this boundary is proper P5 (joint==end1),
                           it will always hit here.*/

                        elem = new element(jointNo, EMPTY, FALSE, _joint, _joint->right());
                        _joint->right(elem);
                        elem->right()->changeElements(_joint, elem);
                        /* Partial chain can only exist if new and old joint differ*/
                        if (partialChain != NULL)
                              insertChain(elem, _joint, _joint, elem, partialChain);
                        if (_end1->child() == FALSE) _end1 = _end2;
                        /* Old end2 is only active element on boundary. */
```

```
                _joint = elem;
            }
            else if (_joint == _end1)
                cerr << "ERROR in boundary::partialP4: _joint==_end1 "
                     << "&& jointNo == _joint->number()" << endl;

            elem = new element(i, EMPTY, TRUE, _joint, _begFull);
            _joint->right(elem);
            _end2 = elem;

            /* Joint is now linked to _begFull via elem, the new _end2. */
            /* Any elements thus removed from the boundary must be deleted.*/
            other = _begFull->getNextEl(_endEmpty);
            removeElements(_begFull, other, _joint);
            _begFull->changeElements(other, elem);
            ch_ptr = _begFull->getChain();
            if (ch_ptr != NULL)
                insertChain(_begFull, elem, elem, _begFull, ch_ptr);
            if (nodeChain != NULL)
                insertChain(_joint, elem, elem, _joint, nodeChain);
        }
        else if (_end2->status() == EMPTY)
        {
            if (jointNo != _joint->number())
            {
                /*If this boundary is proper P5 (joint==end2), it will always hit here.*/
                elem = new element(jointNo, EMPTY, FALSE, _joint->left(), _joint);
                _joint->left(elem);
                elem->left()->changeElements(_joint, elem);
                /* Partial chain can only exist if new and old joint differ*/
                if (partialChain != NULL)
                    insertChain(elem, _joint, _joint, elem, partialChain);
                _joint = elem;
            }
            elem = new element(i, EMPTY, TRUE, _endFull, _joint);
            _joint->left(elem);
            _end1 = elem;

            /* Joint is now linked to _endFull via elem, the new _end1. */
            /* Any elements thus removed from the boundary must be deleted.*/
            other = _endFull->getNextEl(_begEmpty);
            removeElements(_endFull, other, _joint);
            ch_ptr = _endFull->getChain();
            if (ch_ptr != NULL)
                insertChain(_endFull, other, elem, _endFull, ch_ptr);
            else
                _endFull->changeElements(other, elem);
            if (nodeChain != NULL)
                insertChain(_joint, _joint->left(), elem, _joint, nodeChain);
        }
        else cerr << "ERROR in boundary::partialP4: no endmost were EMPTY!"
                  << endl;
    }
} //partialP4

/* **************** properP5 **************** */
/*   */

void boundary::properP5(boundary *partialBoundary, int hasEmpty, chain* partialChain)
{
    partialBoundary->partialP5(_vertexNumber, hasEmpty, partialChain);
    if (_fullChains) _fullChains->cleanupStack();
```

```
}

/* **************** partialP5 **************** */
/* Extends partial boundary with new joint if necessary */

void boundary::partialP5(int jointNo, int hasEmpty, chain* partialChain)
{
      element* elem;

      if (_joint == NULL)
      {
            /* boundary will now consist of two elements only,                    */
            /* _joint and _end1 points to new and _end2 to old one (old _end1) */
            _end2 = _end1;
            _joint = new element(jointNo, EMPTY, hasEmpty);
            _joint->left(_end2);
            _end2->right(_joint);
            if (partialChain != NULL)
                  insertChain(_joint, _end2, _end2, _joint, partialChain);
            _end2->left(_joint);
            _joint->right(_end2);

            _end1 = _joint;
            _begFull = _end2;
            _endFull = _end2;
            _begEmpty = _joint;
            _endEmpty = _end2->getNextEl(_begEmpty);
                                          /*will not be end1 if chain were inserted*/
      }
      else /* Boundary has at least two elements */
      {
            /* The existing boundary is extended if necessary */
            if (_end1->status() == EMPTY)
            {
                  if (_joint->number() != jointNo)
                  {
                        /*If this boundary is proper P5 (joint==end1),*/
                        /*it will always hit here.*/
                        elem = new element(jointNo, EMPTY, hasEmpty, _joint, _joint->right());
                        _joint->right(elem);
                        elem->right()->changeElements(_joint, elem);
                        if (partialChain != NULL)
                              insertChain(elem, _joint, _joint, elem, partialChain);
                        _joint = elem;
                  }
                  else
                  {
                        /* If new joint was not needed, there cannot be a chain */
                        if (partialChain != NULL)
                              cerr << "ERROR in boundary::partialP5: Received "
                                    << "partialChain althoug joint was already "
                                    << "connected!" << endl;
                        _joint->child(hasEmpty);
                        /* _joint represents the vertex of P5, and must have the
                           correct child-value */
                  }
                  if (hasEmpty) _end1 = _joint; /*joint is empty end*/
            }
            else if (_end2->status() == EMPTY)
            {
                  if (_joint->number() != jointNo)
                  {
```

```
                elem = new element(jointNo, EMPTY, hasEmpty, _joint->left(), _joint);
                _joint->left(elem);
                elem->left()->changeElements(_joint, elem);
                if (partialChain != NULL)
                    insertChain(elem, _joint, _joint, elem, partialChain);
                _joint = elem;
            }
            else
            {
                /* If new joint was not needed, there cannot be a chain */
                if (partialChain != NULL)
                    cerr << "ERROR in boundary::partialP5: Received "
                         << "partialChain althoug joint was already "
                         << "connected!" << endl;
                _joint->child(hasEmpty);
                /* _joint represents the vertex of P5, and must have the
                   correct value for child */
            }
            if (hasEmpty) _end2 = _joint; /*joint is empty end*/
        }
        setFull();
    }
} //partialP5

void boundary::nonProperP5(boundary* bound, int hasEmpty, chain* partialChain)
{
    nonRootQ2(bound, partialChain);
    if (hasEmpty)
    {
        _joint->child(TRUE);
        /* nonRootQ2 returns boundary with EMPTY and FULL end*/
        if (_end1->status() == EMPTY) _end1 = _joint;
        else _end2 = _joint;
    }
} //nonProperP5

/* **************** properP6 **************** */
/* Cleans up this node. Leaves all work to partial1 */

void boundary::properP6(boundary* partial1, boundary* partial2, chain* chain1,
                        chain* chain2, int i)
{
    partial1->partialP6(_vertexNumber, chain1, i, partial2, chain2);
    if (_fullChains) _fullChains->cleanupStack();
} //properP6

/* **************** partialP6 **************** */
/* Connects this boundary to joint.            */
/* Includes partial2 and i by calling          */
/* partialConnect.                             */

void boundary::partialP6(int jointNo, chain* chain1, int i, boundary* partial2,
                         chain* chain2)
{
    element *elem, *other, *e1;
    chain *ch_ptr;

    if (_joint == NULL) /* Partial is properP3, has only end1 */
    {
        /* jointNo will be different than end1->number() */
        _joint = new element(jointNo, EMPTY, FALSE, _end1, NULL);
        _end1->right(_joint);
```

```
        if (chain1 != NULL)
            insertChain(_joint, _end1, _end1, _joint, chain1);
        ch_ptr = _end1->getChain();
        partial2->partialConnect(_joint, NULL, _end1, NULL,
                                 chain2, ch_ptr, i, &_end2, &e1);
    }
    else /* Partial has complete boundary */
    {
        if (_end1->status() == EMPTY)
        {
            if (jointNo != _joint->number())
            {
                elem = new element(jointNo, EMPTY, FALSE, _joint, _joint->right());
                _joint->right(elem);
                elem->right()->changeElements(_joint, elem);
                if (chain1 != NULL)
                    insertChain(elem, _joint, _joint, elem, chain1);
                _joint = elem;
            }
            ch_ptr = _begFull-> getChain();
            other = _begFull->getNextEl(_endEmpty);
            partial2->partialConnect(_joint, _joint->right(), _begFull, other,
                                     chain2, ch_ptr, i, &_end2, &e1);
        }
        else if (_end2->status() == EMPTY)
        {
            if (jointNo != _joint->number())
            {
                elem = new element(jointNo, EMPTY, FALSE, _joint->left(), _joint);
                _joint->left(elem);
                elem->left()->changeElements(_joint, elem);
                if (chain1 != NULL)
                    insertChain(elem, _joint, _joint, elem, chain1);
                _joint = elem;
            }
            ch_ptr = _endFull-> getChain();
            other = _endFull->getNextEl(_begEmpty);
            partial2->partialConnect(_joint, _joint->left(), _endFull, other,
                                     chain2, ch_ptr, i, &_end1, &e1);
        }
        else cerr << "ERROR in boundary::partialP6: no endmost were empty!"
                  << endl;
    }

} //partialP6

/* **************** Q1 **************** */
/* If ROOT(T,S), builds chain of component of degree two, */
/* else in case it becomes part of the boundary of the new component */
void boundary::Q1(int i, chain **ch_ptr)
{
    chain *chain1 = NULL;
    chain *chain2 = NULL;
    element* elem, *oldEl, *nextEl;

    chain1 = _end1->getChain();
    chain2 = _end2->getChain();

    if (*ch_ptr == NULL)
    {
        *ch_ptr = new chain();
        (*ch_ptr)->insert(_joint->number());
```

```
	}
	/* _joint is the top element of the chain */
	if ((chain1 != NULL && chain2 != NULL) || chain1 == NULL)
	{
		/* Either both endmost has chain, or end1 don't */
		/* Uses left side of boundary from joint to end1 */
		if (_joint->left() != _end1)
		{
			elem = _joint->left();
			oldEl = _joint;
			while (elem != _end1)
			{
				(*ch_ptr)->insert(elem->number());
				nextEl = elem->getNextEl(oldEl);
				oldEl = elem;
				elem = nextEl;
			}
		}
		/* _end1 is next element to enter chain */
		if (chain1 != NULL)
			(*ch_ptr)->addChain(chain1);
		else
		{
			(*ch_ptr)->insert(_end1->number());
			(*ch_ptr)->insert(i);
		}
	}
	else
	{
		/* end1 has chain, end2 don't */
		/* Uses right side of boundary form joint to end2 */
		if (_joint->right() != _end2)
		{
			elem = _joint->right();
			oldEl = _joint;
			while (elem != _end2)
			{
				(*ch_ptr)->insert(elem->number());
				nextEl = elem->getNextEl(oldEl);
				oldEl = elem;
				elem = nextEl;
			}
		}
		/* _end2 is next element to enter chain */
		if (chain2 != NULL)
			(*ch_ptr)->addChain(chain2);
		else
		{
			(*ch_ptr)->insert(_end2->number());
			(*ch_ptr)->insert(i);
		}

	if (_fullChains) _fullChains->cleanupStack();
	}
} //Q1

/* ******************* rootQ2 ******************* */
/* RootQ2 must have at least two pertinent children,*/
/* and a full end. If it has a partial child,        */
/* it must be interior. Partial boundary and i are   */
/* included by calling partialConnect.               */
```

```
void boundary::rootQ2(boundary* bound, chain* partialChain, int i)
{
    element *other, *e1, *e2, *elem;
    chain    *ch_ptr = NULL;

    setFull();
    if (bound != NULL)
    {   /* All FULL elements will be idle after this, so full end must get new endmost */
        if (_end1->status() == EMPTY && _end2->status() == FULL)
        {   /* end1 empty end, begFull == PARTIAL, end2 == endFull */
            other = _endFull->getNextEl(_begEmpty);
            ch_ptr = _endFull->getChain();
            bound->partialConnect(_endEmpty, _begFull, _endFull, other,
                                  partialChain, ch_ptr, i, &e1, &_end2);
        }
        else if (_end1->status() == PARTIAL && _end2->status() == FULL)
        {   /* end1 == begFull & PARTIAL, end2 == endFull */
            other = _endFull->getNextEl(_begEmpty);
            ch_ptr = _endFull->getChain();
            bound->partialConnect(_endEmpty, _begFull, _endFull, other,
                                  partialChain, ch_ptr, i, &_end1, &_end2);
        }
        else if (_end2->status() == EMPTY && _end1->status() == FULL)
        {   /* end2 empty end, endFull == PARTIAL, end1 == begFull */
            other = _begFull->getNextEl(_endEmpty);
            ch_ptr = _begFull->getChain();
            bound->partialConnect(_begEmpty, _endFull, _begFull, other,
                                  partialChain, ch_ptr, i, &e2, &_end1);
        }
        else if (_end2->status() == PARTIAL && _end1->status() == FULL)
        {   /* end2 == endFull & PARTIAL, end1 == begFull */
            other = _begFull->getNextEl(_endEmpty);
            ch_ptr = _begFull->getChain();
            bound->partialConnect(_begEmpty, _endFull, _begFull, other,
                                  partialChain, ch_ptr, i, &_end2, &_end1);
        }
        else cerr << "ERROR in boundary::rootQ2: Q-node did not have full and "
                  << "empty end!" << endl;
    }
    else /* No partial child */
    {   /*  Include i in boundary, and clean up. */

        if (_end1->status() == FULL)
            _end1 = new element(i, EMPTY, TRUE, _endFull, _begFull);
        else /* end2 has status FULL */
            _end2 = new element(i, EMPTY, TRUE, _endFull, _begFull);

        /* The new endmost will be linked to endFull and begFull. */
        /* Any element thus removed from the boundary, must be deleted. */
        other = _begFull->getNextEl(_endEmpty);
        removeElements(_begFull, other, _endFull);
        ch_ptr = _begFull->getChain();
        if (ch_ptr != NULL)
            insertChain(_begFull, other, elem, _begFull, ch_ptr);
        else
            _begFull->changeElements(other, elem);
        other = _endFull->getNextEl(_begEmpty);
        ch_ptr = _endFull->getChain();
        if (ch_ptr != NULL)
            insertChain(_endFull, other, elem, _endFull, ch_ptr);
        else
            _endFull->changeElements(other, elem);
```

```
    }
    if (_fullChains) _fullChains->cleanupStack();
} //rootQ2

/* **************** nonRootQ2 **************** */

void boundary::nonRootQ2(boundary* bound, chain* partialChain)
{
    element *other, *e1, *e2;

    setFull();
    if (bound != NULL)
    {
        if (_end1->status() == EMPTY && _end2->status() == PARTIAL)
        {   /* end1 empty end, end2 full end & PARTIAL (only pertinent element) */
            bound->partialConnect(_endEmpty, _begFull, _joint, _joint->right(),
                                  partialChain, &e1, &_end2);
        }
        else if (_end1->status() == EMPTY && _end2->status() == FULL)
        {   /* end1 empty end, begFull == PARTIAL, end2 full end */
            other = _begFull->getNextEl(_endEmpty);
            bound->partialConnect(_endEmpty, _begFull, other, _begFull,
                                  partialChain, &e1, &e2);
        }
        else if (_end1->status() == PARTIAL && _end2->status() == FULL)
        {   /* end1 == begFull (empty end), end2 full end */
            other = _begFull->getNextEl(_endEmpty);
            bound->partialConnect(_endEmpty, _begFull, other, _begFull,
                                  partialChain, &_end1, &e2);
        }
        else if (_end2->status() == EMPTY && _end1->status() == PARTIAL)
        {   /* end2 empty end, end1 full end & PARTIAL (only pertinent element) */
            bound->partialConnect(_begEmpty, _endFull, _joint, _joint->left(),
                                  partialChain, &e2, &_end1);
        }
        else if (_end2->status() == EMPTY && _end1->status() == FULL)
        {   /* end2 empty end, end1 full end */
            other = _endFull->getNextEl(_begEmpty);
            bound->partialConnect(_begEmpty, _endFull, other, _endFull,
                                  partialChain, &e2, &e1);
        }
        else if (_end2->status() == PARTIAL && _end1->status() == FULL)
        {   /* end2 == endFull (empty end), end1 full end */
            other = _endFull->getNextEl(_begEmpty);
            bound->partialConnect(_begEmpty, _endFull, other, _endFull,
                                  partialChain, &_end2, &e1);
        }
        else cerr << "ERROR in boundary::nonRootQ2: "
                  << "no legal status for endmosts!" << endl;
        setFull();
    }

    if (_fullChains) _fullChains->cleanupStack();
} //nonRootQ2

/* **************** rootQ3 **************** */

void boundary::rootQ3(boundary* bound1, boundary* bound2, chain* chain1,
                      chain* chain2, int i, int numberOfPartials)
{
    int number1 = 0, number2 = 0;
    element *elem, *other, *e1, *e2;
```

```
chain *ch_ptr;

setFull();
/* _begFull and _endFull will always be different, since if there
    was only one pertinent child, that child would be pertient root! */

if (numberOfPartials == 0)
{   /* bound1,bound2,chain1,chain2 == NULL   */
    /* insert i between begFull and endFull */
    elem = new element(i, EMPTY, TRUE, _begFull, _endFull);
    ch_ptr = _begFull->getChain();
    other = _begFull->getNextEl(_endEmpty);
    if (ch_ptr != NULL)
        insertChain(_begFull, other, elem, _begFull, ch_ptr);
    else
        _begFull->changeElements(other, elem);
    ch_ptr = _endFull->getChain();
    other = _endFull->getNextEl(_begEmpty);
    if (ch_ptr != NULL)
        insertChain(_endFull, other, elem, _endFull, ch_ptr);
    else
        _endFull->changeElements(other, elem);
}
else if (numberOfPartials == 1)
{   /* Only one of bound1 and bound2 is != NULL */
    if (bound1 != NULL)
    {
        if (_begFull->status() == PARTIAL)
        {   /* begFull is empty end, endFull is full end */
            ch_ptr = _endFull->getChain();
            other = _endFull->getNextEl(_begEmpty);
            bound1->partialConnect(_endEmpty, _begFull, _endFull, other,
                                   chain1, ch_ptr, i, &e1, &e2);
            if (_end1 == _begFull) _end1 = e1; /* empty end */
            if (_end2 == _endFull) _end2 = e2; /* full end */
        }
        else if (_endFull->status() == PARTIAL)
        {   /* endFull is empty end, begFull is full end */
            ch_ptr = _begFull->getChain();
            other = _begFull->getNextEl(_endEmpty);
            bound1->partialConnect(_begEmpty, _endFull, _begFull, other,
                                   chain1, ch_ptr, i, &e1, &e2);
            if (_end2 == _endFull) _end2 = e1; /* empty end */
            if (_end1 == _begFull) _end1 = e2; /* full end */
        }
        else cerr << "ERROR in boundary::rootQ3: the partial element were "
                  << "not found!" << endl;
    }
    else if (bound2 != NULL)
    {
        if (_begFull->status() == PARTIAL)
        {   /* begFull is empty end, endFull is full end */
            ch_ptr = _endFull->getChain();
            other = _endFull->getNextEl(_begEmpty);
            bound2->partialConnect(_endEmpty, _begFull, _endFull, other,
                                   chain2, ch_ptr, i, &e1, &e2);
            if (_end1 == _begFull) _end1 = e1; /* empty end */
            if (_end2 == _endFull) _end2 = e2; /* full end */
        }
        else if (_endFull->status() == PARTIAL)
        {   /* endFull is empty end, begFull is full end */
            ch_ptr = _begFull->getChain();
```

```
                    other = _begFull->getNextEl(_endEmpty);
                    bound2->partialConnect(_begEmpty, _endFull, _begFull, other,
                                           chain2, ch_ptr, i, &e1, &e2);
                    if (_end2 == _endFull) _end2 = e1; /* empty end */
                    if (_end1 == _begFull) _end1 = e2; /* full end */
                }
                else cerr << "ERROR in boundary::rootQ3: the partial element were "
                          << "not found!" << endl;
            }
            else cerr << "ERROR in boundary::rootQ3: no partial child when "
                      << "numberOfPartials == 1!" << endl;
        }
        else if (numberOfPartials == 2)
        {   /* both begFull and endFull is partial */
            /* doubleConnect will do the job, but must prepare the attributes */

            /* Find connection point of bound1 */
            if (chain1 != NULL) number1 = chain1->first();
            else if (bound1->joint() != NULL)
                number1 = bound1->joint()->number();
            else number1 = bound1->vertexNumber();

            /* Find connection point of bound2 */
            if (chain2 != NULL) number2 = chain2->first();
            else if (bound2->joint() != NULL)
                number2 = bound2->joint()->number();
            else number2 = bound2->vertexNumber();


            if (_begFull->number() == number1 && _endFull->number() == number2)
                doubleConnect(bound1, bound2, chain1, chain2, i);
            else if (_begFull->number() == number2 && _endFull->number() == number1)
                doubleConnect(bound2, bound1, chain2, chain1, i);
            else cerr << "ERROR in boundary::rootQ3: two partial were not at "
                      << "ends of full sequence!" << endl;
        }
        else cerr << "ERROR in boundary::rootQ3: numberOfPartials had illegal "
                  << "value!" << endl;

        if (_fullChains) _fullChains->cleanupStack();
} //rootQ3

/* *************** print *************** */

char* boundary::print()
{
        ostrstream txt;
        char out[INFOSIZE];
        element *elem, *nextEl, *oldEl;

        txt << ", Vertex " << _vertexNumber;
        if (_joint != NULL)
        {
            txt << " Bound: " << _joint->number() << "(j)";
            if (_joint == _end1) txt << "(1)";

            if (_joint->status() == FULL) txt << "(F)";
            else if (_joint->status() == PARTIAL) txt << "(P)";
            else if (_joint->status() == PERTINENT) txt << "(Pe)";
            else if (_joint->status() == EMPTY) txt << "(E)";
            else if (_joint->status() == TO_BE_DELETED) txt << "(D)";
            else cout << "Unknown status in element: " << _joint->status()
```

```
                << endl;

    if (_joint->child()) txt << "(T)";
    else txt << "(F)";

    if (_joint->chainQ())  txt << "(ch)";

    oldEl = _joint;
    elem = _joint->left();
    while (elem != _joint)
    {
        if (elem == NULL)
        {
            cerr << "  ERROR in aboundary::print: elem == NULL" << endl;
            elem = _joint;
        }
        else
        {
            txt << " - " << elem->number();
            if (elem == _end1) txt << "(l)";
            else if (elem == _end2) txt << "(r)";

            if (elem->status() == FULL) txt << "(F)";
            else if (elem->status() == PERTINENT) txt << "(Pe)";
            else if (elem->status() == PARTIAL) txt << "(P)";
            else if (elem->status() == EMPTY) txt << "(E)";
            else if (elem->status() == TO_BE_DELETED) txt << "(D)";
            else cout << "Unknown status in element: "
                      << elem->status()<< endl;

            if (elem->child()) txt << "(T)";
            else  txt << "(F)";

            if (elem->chainQ())  txt << "(ch)";

            nextEl = elem->getNextEl(oldEl);
            oldEl = elem;
            elem = nextEl;
        }
    }
}
else if (_end1 != NULL)
{
    txt << " Bound: " << _end1->number() << "(l)";

    if (_end1->status() == FULL) txt << "(F)";
    else if (_end1->status() == PARTIAL) txt << "(P)";
    else if (_end1->status() == PERTINENT) txt << "(Pe)";
    else if (_end1->status() == EMPTY) txt << "(E)";
    else if (_end1->status() == TO_BE_DELETED) txt << "(D)";
    else cout << "Unknown status in element: " << _end1->status()
                << endl;

    if (_end1->child()) txt << "(T)";
    else txt << "(F)";

    if (_end1->chainQ())  txt << "(ch)";
}
txt << " fChs: " << fullChains() << ends;
strcpy(out, txt.str());
strstreambuf * buf_ptr = txt.rdbuf();
buf_ptr->freeze(0);
```

```
    return out;
}
```

A.3.4 element

`Class element` is used by `Boundary` to build the doubly linked list of boundary elements.

```
/*********************************************************************
                                    File: element.h

                                    Author: Gørril Vollen
                                    Last update: 20/02/1998
*********************************************************************/
#ifndef ELEMENT_H
#define ELEMENT_H
#include "def.inc"
#include "node.inc"
//#define DEBUG

class chain;

class element
{
private:
    int _number;
    int _status;        /* EMPTY, FULL or PARTIAL */
    int _child;         /* TRUE if element has child, else FALSE */
    element* _left;
    element* _right;
    chain* _chain;

public:
    element() {}
    element(int number)
    {
        _number = number;
        _status = EMPTY; _child = FALSE;
        _left = NULL;
        _right = NULL;
        _chain = NULL;
    }
    element(int number, int status, int child)
    {
        _number = number;
        _status = status;
        _child = child;
        _left = NULL;
        _right = NULL;
        _chain = NULL;
    }
    element(int number, int status, int child, element* left, element* right)
    {
        _number = number;
        _status = status;
        _child = child;
        _left = left;
        _right = right;
        _chain = NULL;
    }

    ~element();
```

```
int number() const
{
	return _number;
}
int status() const
{
	return _status;
}
void status(int i)
{
	_status = i;
}
int child() const
{
	return _child;
}
void child(int b)
{
	_child = b;
}
element* right()
{
	return _right;
}
void right(element* elem)
{
	_right = elem;
}
element* left()
{
	return _left;
}
void left(element* elem)
{
	_left = elem;
}

void reset(int stat, int child, chain* ch_ptr, int del);

int chainQ() /* Brukes av print i internal.h */
{
	if (_chain == NULL) return FALSE;
	else return TRUE;
}

void setChain(chain* ch_ptr)
{
	if (ch_ptr != NULL) _chain = ch_ptr;
}

chain* getChain()
{
	chain* ptr;
	ptr = _chain;
	_chain = NULL;
	return ptr;
}

element* getNextEl(element* other)
{
	if (_left != other)
```

```
			return _left;
		else
			return _right;
	}

	int setNextEl(element* elem)
	{
		if (_left == NULL){
			_left = elem;
			return TRUE;
		}
		else if (_right == NULL){
			_right = elem;
			return TRUE;
		}
		else return FALSE;
	}

	int changeElements(element* oldEl, element* newEl)
	{
		if (oldEl == _left)
		{
			_left = newEl;
			return TRUE;
		}
		else if (_right == oldEl)
		{
			_right = newEl;
			return TRUE;
		}
		else
		{
			cout << "ERROR: element::changeElements: old element not found!"
				<< " (number = " << oldEl->number() << ")" << endl;
			return FALSE;
		}
	}
};

#endif


/*****************************************************************
				File: element.cc

				Author: Gørril Vollen
				Last update: 20/02/1998
*****************************************************************/
#include "chain.h"
#include "element.h"

element::~element()
{
	if (_chain != NULL) delete _chain;
}

void element::reset(int stat, int child, chain* ch_ptr, int del)
{
	_status = stat;
	_child = child;
	if (del && _chain != NULL)
	{
```

```
            delete _chain;
            _chain = NULL;
        }
        if (ch_ptr != NULL) _chain = ch_ptr;
}
```