

Empirical Study on Branchwidth and Branch Decomposition of Planar Graphs

Zhengbing Bian* Qian-Ping Gu* Marjan Marzban* Hisao Tamaki† Yumi Yoshitake†

Abstract

We propose efficient implementations of Seymour and Thomas algorithm which, given a planar graph and an integer β , decides whether the graph has the branchwidth at least β . The computational results of our implementations show that the branchwidth of a planar graph can be computed in a practical time and memory space for some instances of size about one hundred thousand edges. Previous studies report that a straightforward implementation of the algorithm is memory consuming, which could be a bottleneck for solving instances with more than a few thousands edges. Our results suggest that with efficient implementations, the memory space required by the algorithm may not be a bottleneck in practice. Applying our implementations, an optimal branch decomposition of a planar graph of practical size can be computed in a reasonable time. Branch-decomposition based algorithms have been explored as an approach for solving many NP-hard problems on graphs. The results of this paper suggest that the approach could be practical.

Key words: Graph algorithms, branch-decomposition, planar graphs, algorithm engineering, computational study.

1 Introduction.

The notions of branchwidth and branch decompositions are introduced by Robertson and Seymour [20] in relation to the more celebrated notions of treewidth and tree decompositions [18, 19]. A graph of small branchwidth (or treewidth) admits efficient dynamic programming algorithms for a vast class of problems on the graph [4, 6]. There are two major steps in a branch/tree-decomposition based algorithm for solving a problem: (1) computing a branch/tree decomposition with a small width and (2) applying a dynamic programming algorithm based on the decomposition to solve the problem. Step (2) usually runs in exponential time in the width of the branch/tree decomposition computed in Step (1). So it is extremely important to decide the branch/tree-width and compute the optimal decompositions. It is

NP-complete to decide whether the width of a given general graph is at least an integer β if β is part of the input, both for branchwidth [22] and treewidth [3]. When the branchwidth (treewidth) is bounded by a constant, both the branchwidth and the optimal branch decomposition (treewidth and optimal tree decomposition) can be computed in linear time [7, 9]. However, the huge constants behind the Big-Oh make the linear time algorithms only theoretically interesting.

One hurdle for applying branch/tree-decomposition based algorithms in practice is the difficulty of computing a good branch/tree decomposition because of the NP-hardness and huge hidden constants problems. Recently, the branch-decomposition based algorithms with practical importance for problems in planar graphs have been receiving increased attention [10, 11]. This is motivated by the fact that an optimal branch decomposition of a planar graph can be computed in polynomial time by Seymour and Thomas algorithm [22] and the algorithm is reported efficient in practice [13, 14]. Notice that it is open whether computing the treewidth of a planar graph is NP-hard or not. The result of the branchwidth implies a 1.5-approximation algorithm for the treewidth of planar graphs. Readers may refer to the recent papers by Bodlaender [8] and Hicks et al. [15] for extensive literature in the theory and application of branch/tree-decompositions.

Given a planar graph G of n vertices and an integer β , Seymour and Thomas give a decision algorithm (called ST Procedure for short in what follows) which decides if G has a branchwidth at least β in $O(n^2)$ time [22]. Using ST Procedure as a subroutine, they also give an algorithm which constructs an optimal branch decomposition of G . The algorithm calls ST Procedure $O(n^2)$ times and runs in $O(n^4)$ time. Gu and Tamaki [12] give an improved algorithm which calls ST Procedure $O(n)$ times and runs in $O(n^3)$ time to construct the branch decomposition. Hicks proposes a divide and conquer heuristic algorithm to reduce the number of calls for ST Procedure [14]. Computational studies show that the heuristic is effective in reducing the calls but has the time complexity of $O(n^4)$ [14]. All known algorithms for computing the optimal branch decomposition of a planar graph rely on ST Procedure; thus, an efficient implementation of the procedure plays a key role in com-

*School of Computing Science, Simon Fraser University, Burnaby BC, Canada V5A 1S6. {zbian/qgu/mmarzban}@cs.sfu.ca.

†Department of Computer Science, Meiji University, Kawasaki, 214-8571 Japan. {tamaki/yyoshi}@cs.meiji.ac.jp.

puting the branch decompositions. A straightforward implementation of ST Procedure requires $O(n^2)$ bytes of memory which is reported in [13] a bottleneck for solving large instances with more than 5,000 edges. Hicks proposes memory friendly implementations in the cost of performing re-calculations and increasing the running time of ST Procedure to $O(n^3)$ [13].

In this paper, we propose efficient implementations of ST Procedure. Our implementations can be classified into two groups. Group (1) does not perform re-calculations and runs in $O(n^2)$ time. The most memory efficient implementation in this group can compute the branchwidth of some instance of size up to one hundred thousand edges with 500Mbytes memory and in a couple of hours. Group (2) performs re-calculations and can compute the branchwidth of the instance of one hundred thousand edges with 200Mbytes of memory. The implementations in Group (2) may run in $O(n^3)$ time in the worst case. All of our implementations still use $O(n^2)$ bytes of memory. However, the constants behind the Big-Oh are much smaller than those in a straightforward implementation. In contrast, the results of this paper and those of [13] show that straightforward implementations can only handle instances of size up to about 5,000 edges within 1Gbytes of memory. Our most time efficient implementation is faster than the straightforward one by a factor of $3 \sim 15$. Compared with the previous memory friendly implementations of [13], our most memory efficient implementations of Group (1) and Group (2) use at most 1/4 memory and 1/8 memory and run faster by a factor of $100 \sim 400$ and a factor of $100 \sim 200$, respectively. Notice that the CPU used in [13] has frequency 194MHz and the CPU used for testing our implementations has frequency 3.06GHz, so we need to keep in mind this difference of speed when we compare the running time.

The results of this paper suggest that the memory size required by ST Procedure may not be a bottleneck for computing the branchwidth and optimal branch decomposition of a planar graph in practice. Our implementations also imply more efficient algorithms which call ST Procedure to find the optimal branch decompositions. We implemented the $O(n^4)$ time algorithm of Seymour and Thomas and the $O(n^3)$ time algorithm of Gu and Tamaki. The computational results show that the optimal branch decompositions of planar graphs with a few thousands edges can be computed in a couple of hours.

The rest of the paper is organized as follows. In Section 2, we give the preliminaries of the paper. We review ST Procedure and give some observations which provide the base of our efficient implementations in Section 3. Section 4 describes our implementations.

Computational results are presented in Section 5. The final section concludes the paper.

2 Preliminaries.

Readers may refer to a textbook on graph theory (e.g., the one by West [24]) for basic definitions and terminology on graphs. In this paper, graphs are unweighted undirected graphs (i.e., each edge has a unit length) unless otherwise stated. Let G be a graph. We use $V(G)$ for the vertex set of G and $E(G)$ for the edge set of G . A *branch decomposition* of G is a tree T_B such that the set of leaves of T_B is $E(G)$ and each internal node of T_B has node degree 3. For each edge e of T_B , removing e separates T_B into two sub-trees. Let E' and E'' be the sets of leaves of the subtrees. The width of e is the number of vertices of G incident to both an edge in E' and an edge in E'' . The width of T_B is the maximum width of all edges of T_B . The *branchwidth* of G is the minimum width of all branch-decompositions of G .

The algorithms of Seymour and Thomas [22] for branchwidth and branch decomposition are based on another type of decompositions called *carving decompositions*.

A carving decomposition of G is a tree T_C such that the set of leaves of T_C is $V(G)$ and each internal node of T_C has node degree 3. For each edge e of T_C , removing e separates T_C into two sub-trees and the two sets of the leaves of the sub-trees are denoted by V' and V'' . The width of e is the number of edges of G with both an end vertex in V' and an end vertex in V'' . The width of T_C is the maximum width of all edges of T_C . The *carvingwidth* of G is the minimum width of all carving decompositions of G . Notice that the carving decomposition is defined for more general graphs in [22]. The definition allows positive integer lengths on edges of the graphs. The width of e in T_C for the weighted graph is defined as the sum of lengths of edges with an end vertex in V' and an end vertex in V'' .

Let G be a planar graph with a fixed embedding. Let $R(G)$ be the set of faces of G . The *medial graph* [22] $M(G)$ of G is a planar graph with an embedding such that $V(M(G)) = \{u_e | e \in E(G)\}$, $R(M(G)) = \{r_s | s \in R(G)\} \cup \{r_v | v \in V(G)\}$, and there is an edge $\{u_e, u_{e'}\}$ in $E(M(G))$ if the edges e and e' of G are incident to a same vertex v of G and they are consecutive in the clockwise (or counter clockwise) order around v . $M(G)$ in general is a multigraph but has $O(|V(G)|)$ edges. Seymour and Thomas [22] show that the carvingwidth of $M(G)$ is exactly twice the branchwidth of G and an optimal carving decomposition of $M(G)$ can be translated into an optimal branch decomposition of G in linear time. To decide whether a planar graph G has

the branchwidth at least an integer β , ST Procedure actually decides whether $M(G)$ has the carvingwidth at least 2β .

A face $r \in R(G)$ and an edge $e \in E(G)$ are incident to each other if e is a boundary of r in the embedding. Notice that an edge e is incident to exactly two faces. For a face $r \in R(G)$, a vertex v is incident to r if v is an end vertex of an edge incident to r . For a face $r \in R(G)$, let $V(r)$ and $E(r)$ be the sets of vertices and edges incident to r , respectively. For a vertex $v \in V(G)$, let $E(v)$ be the set of edges incident to v .

The *planar dual* G^* of G is defined as that for each vertex $v \in V(G)$, there is a unique face $r_v^* \in R(G^*)$; for each face $r \in R(G)$, there is a unique vertex $v_r^* \in V(G^*)$; and for each edge $e \in E(G)$ incident to r and r' , there is a unique edge $e^* = \{v_r^*, v_{r'}^*\} \in E(G^*)$ which crosses e .

A *walk* in a graph G is a sequence of edges e_1, e_2, \dots, e_k of G , where $e_i = \{v_{i-1}, v_i\}$ for $1 \leq i \leq k$. A walk is *closed* if $v_0 = v_k$. The length of a walk is the number of edges in the walk. For two vertices u and v in a graph G , the distance $d(u, v)$ is the minimum length of all walks between u and v . The walk with distance $d(u, v)$ is a shortest path between u and v .

3 Seymour and Thomas procedure.

We give a brief review of ST Procedure and readers may refer to [22] for more details of the decision procedure. ST Procedure is often called the *rat-catching* algorithm because it can be intuitively described by a rat catching game introduced in [22]. We first review the game and then give a formal description of ST Procedure.

3.1 Rat catching game. In this game, there are two players, a rat and a rat-catcher. The game is on a planar graph G of a fixed embedding, with a face and an edge of G interpreted as a room and a wall of a room, respectively. The rules for the game are as follows.

- (R1) The rat-catcher selects a room.
- (R2) The rat selects a corner of a room (a vertex of G).
- (R3) The rat-catcher selects a room adjacent to the current room and moves to the wall between the two rooms (the edge of G incident to the current face and the selected face). The rat-catcher generates a noise of a fixed level that may make walls noisy. The condition of making a wall noisy will be given later.
- (R4) The rat moves to a different corner via walls or stays at the current corner. The rat can not use

a noisy wall but can use as many quiet walls as possible in one move.

- (R5) The rat-catcher moves to the room it selected and can not change its mind to move back to the previous room. The rat-catcher keeps making noise.
- (R6) If the rat is in a corner, all walls incident to the corner are noisy, and the rat-catcher is in a room with this corner, then the rat-catcher catches the rat and wins the game. Otherwise goto (R3).

Now we give the condition on a wall becoming noisy. For the planar dual G^* of G , let v_r^* and e^* be the vertex and edge of G^* corresponding to the face r and edge e of G , respectively. Let k be the noise level produced by the rat-catcher. When the rat-catcher is on edge e , edge f is noisy if and only if there is a closed walk of length smaller than k containing edges e^* and f^* in G^* . Similarly, when the rat-catcher is in face r , edge f is noisy if and only if there is a closed walk of length smaller than k containing vertex v_r^* and edge f^* in G^* . The rat-catcher wins the game if the rat is at a vertex v with node degree smaller than k and the rat-catcher is in a face incident to v . The rat wins the game if there is a scheme by which the rat can escape from the rat-catcher for ever. We use $RC(G, k)$ to denote the rat catching game on G and k . Seymour and Thomas show that the rat wins the game $RC(G, k)$ if and only if G has carvingwidth at least k and give ST Procedure which, given G and k , computes the outcome of the game $RC(G, k)$ [22].

3.2 ST Procedure. Now we present ST Procedure using the language of the game $RC(G, k)$. Our presentation is different from the original one which is based on a notion called antipodality [22]. For a graph G with maximum node degree at least k , the rat always wins the game $RC(G, k)$ because the rat will never get caught if it stays at a vertex with node degree at least k . So we assume that G has maximum node degree smaller than k in the following discussion. Given G and k , ST Procedure computes an escaping scheme for the rat or decides no such scheme exists. The escaping scheme is represented by a collection of vertex subsets and subgraphs of G by which the rat can escape from the rat-catcher for ever. The collection contains a non-empty subset of vertices of G (a subset of corners) for every face and a non-empty subgraph of G (a subset of corners and quiet walls) for every edge.

Given G and k , we define G_e to be the subgraph of G obtained by deleting noisy edges from G when the rat-catcher is on edge e . More specifically, $V(G_e) = V(G)$

and

$$E(G_e) = \{f \mid \text{every closed walk of } G^* \text{ containing } e^* \text{ and } f^* \text{ has length at least } k\}.$$

Notice that every edge of G_e is quiet when the rat-catcher is on e . For each face $r \in R(G)$, we define

$$S_r = \{(r, v) \mid v \in V(G)\} \text{ and } S = \cup_{r \in R(G)} S_r.$$

For each edge $e \in E(G)$, we define

$$T_e = \{(e, C) \mid C \text{ is a connected component of } G_e\}.$$

Let $T = \cup_{e \in E(G)} T_e$. Then the game $R(G, k)$ can be described by a bipartite graph $H(G, k)$, where the vertex set of $H(G, k)$ is $S \cup T$ and there is an edge between $(r, v) \in S$ and $(e, C) \in T$ if face r is incident to edge e and v is a vertex of C . The vertices of $H(G, k)$ can be interpreted as the states of the game: a $(r, v) \in S$ represents that the rat-catcher is in face r and the rat is at vertex v , and a $(e, C) \in T$ expresses that the rat-catcher is on edge e and the rat is at a vertex of C and can use the edges of C to move. The edge between $(r, v) \in S$ and $(e, C) \in T$ indicates the possible state transitions of the game: when the rat is at v and the rat-catcher moves from face r to edge e , the game state transits from (r, v) to (e, C) ; or when the rat is at some vertex of C and moves to v , and the rat-catcher moves from edge e to face r , the game state transits from (e, C) to (r, v) .

A game state of $R(G, k)$ is called a *losing state* if the rat will lose the game at the state. To compute an escaping scheme for the rat, ST Procedure deletes the losing states from $H(G, k)$. For a face $r \in R(G)$ and a vertex v incident to r ($v \in V(r)$), (r, v) is a losing state because the rat gets caught if the rat is at v and the rat-catcher is in r . For an edge e incident to face r , (e, C) is a losing state if for every vertex v of C , (r, v) is a losing state. To see this, assume that the rat-catcher is on edge e and the rat is at some vertex of C . The rat-catcher may move to r or r' , the other face incident to e , in the next step. In either of the moves, the rat can only move to a vertex v of C . If the rat-catcher moves to r then the game transits to (r, v) at which the rat will get caught. If the rat-catcher moves to r' then the rat is at some vertex of C . The rat-catcher can move back to e and then to r , and the rat will get caught. Similarly, if (e, C) is a losing state then for every face r incident to e and every vertex v of C , (r, v) is a losing state.

For every edge $e \in E(G)$, ST Procedure initializes set X_e to include all states of T_e . For every face $r \in R(G)$, ST Procedure initializes set X_r to include

all states of S_r and then deletes (r, v) from X_r for every $v \in V(r)$. After this initial deletion step, for each face r and each edge e incident to r , if there is a state (e, C) such that for every vertex v of C state (r, v) has been deleted, then the state (e, C) is deleted from X_e . If this deletion is done then for the other face r' incident to e , state (r', v) is deleted from $X_{r'}$ for every vertex v of C . This deletion may result in further deletions of losing states. The deletion process is repeated until no further deletion is possible. It is shown in [22] that graph G has carvingwidth at least k if and only if after the deletion process finishes, X_r and X_e are not empty for every $r \in R(G)$ and every $e \in E(G)$. The collection of non-empty X_r and X_e for every face r and every edge e is an escaping scheme for the rat. Below is a simplified version of the formal description of ST Procedure [22]. We remark that ST Procedure decides if the carvingwidth is at least k for more general planar graphs. It allows weighted input graphs with positive integer lengths on edges.

ST Procedure

Input: A non-null connected planar graph G with a fixed embedding, a planar dual G^* of G , an integer $k \geq 0$.

Output: Decides if G has carvingwidth at least k .

1. If the maximum node degree of G is at least k then output G has carvingwidth at least k and terminate.
2. For each face $r \in R(G)$, let $X_r = S_r$.

For each edge $e \in E(G)$, compute G_e and let $X_e = T_e$. For each $(e, C) \in X_e$ and the faces r and r' incident to e , let $c(r, e, C) = |V(C)|$ and $c(r', e, C) = |V(C)|$, where $V(C)$ is the set of vertices of C .

3. For each face r and each state $(r, v) \in X_r$ with $v \in V(r)$, put (r, v) to a stack L and delete (r, v) from X_r .
4. If L is empty then goto the next step.

Otherwise, remove a state x from L .

Assume that $x = (r, v)$ is a state for a face ($x \in S$). For each edge e incident to r , find the state $(e, C) \in X_e$ such that C contains v . Decrease $c(r, e, C)$ by one. If $c(r, e, C)$ becomes 0 and $(e, C) \in X_e$ then put (e, C) to L and delete (e, C) from X_e .

Assume that $x = (e, C)$ is a state for an edge ($x \in T$). If there is a face r incident to e such that $c(r, e, C) > 0$ then for each vertex v of C and $(r, v) \in X_r$ put (r, v) to L and delete (r, v) from X_r .

Repeat this step.

5. If X_r is non-empty for every $r \in R(G)$ and X_e is non-empty for every $e \in E(G)$ then output G has carvingwidth at least k , otherwise output G has carvingwidth smaller than k .

Notice that we can stop ST Procedure and conclude that the rat loses the game when some X_r becomes empty. The reason is that if all states of X_r are deleted, all states of X_e for e incident to r will be deleted; then all states for face r' incident to e will be deleted; and finally all states for every face and edge will be deleted. Similarly, the rat loses the game if some X_e becomes empty.

To compute G_e for each e , ST Procedure needs to find the quiet edges when the rat-catcher is on edge e . An edge f is quiet and will be included in G_e if every closed walk in G^* that contains e^* and f^* has length at least k . More specifically, let $e^* = \{u^*, v^*\}$ and $f^* = \{x^*, y^*\}$. Edge f is included in G_e if and only if $d(u^*, x^*) + d(v^*, y^*) + 2 \geq k$ and $d(u^*, y^*) + d(v^*, x^*) + 2 \geq k$. A solution for the all-pairs shortest path problem of G^* will suffice for the distances required in computing G_e for all $e \in E(G)$.

THEOREM 3.1. (*Seymour and Thomas [22]*)
Given a planar graph G of n vertices and integer $k \geq 0$, ST Procedure decides if G has carvingwidth at least k or not using graph $H(G, k)$ in $O(n^2)$ time and $O(n^2)$ bytes of memory.

To decide the branchwidth of G , the input to ST Procedure is the medial graph $M(G)$ and the branchwidth of G is $k/2$ if the carvingwidth of $M(G)$ is k .

3.3 Observations for efficient implementations.

We give some observations on the game $RC(G, k)$ that can be used for efficient implementations of ST Procedure. By the definition of the game $RC(G, k)$, a state (r, v) is a losing state if $v \in V(r)$ for G with maximum node degree smaller than k . ST Procedure makes use of this sufficient condition to delete the losing states at the initial step of the deletion process for each face r . We observe that if we can find and delete more losing states at the initial step for each face r , then ST Procedure may run faster and use less memory. We prove the following sufficient condition for finding more losing states.

LEMMA 3.1. *For a face r and a vertex v in graph G with maximum node degree smaller than k , (r, v) is a losing state if there exist two faces s and t incident to v such that there are*

- (1) *a closed walk W_1 in G^* with length smaller than k that consists of the shortest path from v_r^* to v_s^* , the clockwise walk from v_s^* to v_t^* around r_v^* , and the shortest*

path from v_t^ to v_r^* ; and (2) a closed walk W_2 in G^* with length smaller than k that consists of the shortest path from v_r^* to v_s^* , the counter-clockwise walk from v_s^* to v_t^* around r_v^* , and the shortest path from v_t^* to v_r^* .*

Proof. Assume that W_1 and W_2 exist. Then for every edge e incident to v in G , e^* is either in W_1 or W_2 and e is noisy when the rat-catcher is in r . Let e_1^*, \dots, e_j^* be the edges in the shortest path from v_r^* to v_s^* . Assume that the rat is at v and the rat-catcher is in r . Since all edges incident to v are noisy, the rat can not move away from v . Next, the rat-catcher can move to edge e_1 . Since all edges incident to v are noisy when the rat-catcher is on e_1 , the rat has to stay at v . Similarly, the rat has to stay at v when the rat-catcher is on edge e_i , $1 \leq i \leq j$. So the rat-catcher can move to face s using edges e_1, \dots, e_j and catch the rat at v . \square

Once the shortest paths from v_r^* to all other vertices of G^* have been computed, it is easy to see the time for checking if (r, v) is a losing state by the condition of Lemma 3.1 is proportional to the node degree of v . Therefore, it takes $O(n)$ time to check (r, v) for a face r and all $v \in V(G)$. For each face r , let $U(r)$ be the set of vertices that for every $v \in U(r)$, (r, v) is a losing state computed by the sufficient condition of Lemma 3.1. From Theorem 3.1, we have the following result.

THEOREM 3.2. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure decides if G has carvingwidth at least k in $O(n^2)$ time and $O(n^2)$ bytes of memory when the losing states (r, v) , $v \in U(r)$, are deleted at the initial step of the deletion process for each face r .*

For each face $r \in R(G)$, we define G_r to be the subgraph of G obtained by deleting the noisy edges from G when the rat-catcher is in face r . That is, $V(G_r) = V(G)$ and

$$E(G_r) = \{f \mid \text{every closed walk of } G^* \text{ containing } v_r^* \text{ and } f^* \text{ has length at least } k\}.$$

Notice that every edge of G_r is quiet when the rat-catcher is in r . Recall that G_e is the quiet subgraph of G when the rat-catcher is on edge e . Our next observation is that for every edge e incident to face r , $E(G_r) \subseteq E(G_e)$, because v_r^* is an end vertex of e^* and therefore the set of closed walks of G^* containing vertex v_r^* and edge f^* includes all closed walks of G^* containing edges e^* and f^* . From this, a component of G_r is a subgraph of some component of G_e . Hence, when the rat-catcher moves from face r to edge e and the rat is at any vertex of some component D of G_r , the component of G_e on which the rat can move is the

same one which contains D as a subgraph. Thus, when the rat-catcher is in face r , the states of the game can be expressed by

$$S'_r = \{(r, D) | D \text{ is a connected component of } G_r\}.$$

Let $S' = \cup_{r \in R(G)} S'_r$. The game $RC(G, k)$ can be described by a bipartite graph $H'(G, k)$, where the vertex set of $H'(G, k)$ is $S' \cup T$ and there is an edge between $(r, D) \in S'$ and $(e, C) \in T$ if face r is incident to edge e and D is a subgraph of C . For a face r and a component D of G_r , (r, D) is a losing state if for every vertex v of D , (r, v) is a losing state. For an edge e incident to face r , state (e, C) is a losing state if for every component D of G_r that is a subgraph of C , (r, D) is a losing state. Similarly, if (e, C) is a losing state then for every face r incident to e and every component D of G_r that is a subgraph of C , (r, D) is a losing state. Summarizing the above and from Theorem 3.1, the following result holds.

THEOREM 3.3. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure decides if G has carvingwidth at least k using graph $H'(G, k)$ in $O(n^2)$ time and $O(n^2)$ bytes of memory.*

When graph $H'(G, k)$ is used for the game $RC(G, k)$, X_r is initialized as S'_r for each face $r \in R(G)$ in ST Procedure. Compared with S_r , S'_r may have less game states and thus require less memory.

During the deletion process of ST Procedure, losing states are deleted from sets X_r and X_e . Our another observation is that the elements of X_e for an edge e incident to faces r and r' at a step of ST Procedure can be computed in $O(n)$ time from the elements of X_r and $X_{r'}$ at that step. This gives an option for implementing ST Procedure that does not keep but dynamically computes X_e from X_r and $X_{r'}$ during the deletion process.

THEOREM 3.4. *Given a planar graph G of n vertices and $k \geq 0$, ST Procedure can decide if G has carvingwidth at least k or not in $O(n^3)$ time and $O(n^2)$ bytes of memory if for each edge e , X_e is not kept but dynamically computed during the deletion process.*

Proof. For an edge e incident to faces r and r' , the set X_e is needed when an element of X_r or $X_{r'}$ is deleted during the computation and when ST Procedure terminates. So, we can compute X_e in $O(n)$ time once there is an element deleted from X_r or $X_{r'}$. Since there are $O(n)$ elements in $X_r \cup X_{r'}$, X_e is computed $O(n)$ times. The total time for computing X_e for all $e \in E(G)$ is $O(n^3)$. From Theorem 3.1, the theorem holds. \square

The re-calculation of edge data considered in this paper is different from the re-calculation in the previous study of [13], where face data are re-calculated for some faces and each re-calculation for a face r involves a computation of T_e for each e incident to r .

Finally, it is easy to see that if all states of S_r (or S'_r) for some face r are losing states then for every face r' , all states of $S_{r'}$ ($S'_{r'}$) are losing states and the rat loses the game.

OBSERVATION 3.1. *If X_r becomes empty for some face r during the deletion process then graph G has carvingwidth smaller than k .*

By this observation we can terminate ST Procedure when some X_r becomes empty. This may save the computation time when the rat loses the game.

4 Efficient implementations.

Let G be a connected planar graph with a given embedding and $V(G) = \{v_1, \dots, v_n\}$. We first describe a straightforward implementation (called *Naive*) of ST Procedure and then propose several improvements on the implementations of ST Procedure. Those improvements try to reduce both the memory space and running time of ST procedure.

4.1 Naive implementation. A straightforward implementation of ST Procedure would use graph $H(G, k)$ for deciding the outcome of the game $RC(G, k)$. We use the following data structure for graph $H(G, k)$ in Naive.

- For each face $r \in R(G)$, a Boolean array B_r (of n elements) is assigned such that $B_r[i]$ is used to indicate if $(r, v_i) \in X_r$ or not. A list of $|E(r)|$ elements is used to keep the edges incident to r .
- For each edge $e \in E(G)$, the two faces r and r' incident to e are kept. All components of G_e are kept in a list. Each component of G_e is given an index and component C_j is kept in the j th element of the list. The element of the list for C_j contains the set of vertices of C_j , $c(r, e, C_j)$, $c(r', e, C_j)$, and a Boolean variable indicating if (e, C_j) has been deleted from X_e or not. An integer array I_e (of n elements) is used to indicate which component a vertex is in. If v_i is a vertex of C_j then $I_e[i]$ is set to j .
- In addition to the face and edge data, a stack L is used and a distance matrix is kept for the all pairs shortest distances in the dual graph G^* of G .

It is easy to check that the Naive implementation runs in $O(n^2)$ time. A simple calculation shows that Naive

implementation requires about $40n^2$ bytes of memory when G is a medial graph. Since there are many single vertex components in G_e and the operating system may have a minimum memory allocation size of 16 bytes, the memory usage in practice is close to $50n^2$ bytes.

4.2 Common improvements. We first describe two common improvements which are used in all of our efficient implementations. When we say *processing* a face r or an edge e , we mean deleting a losing state from X_r or X_e .

The first common improvement is that we define a processing order of the faces in our implementations. We put losing states (r, v) to the stack for only one face at a time. When there are losing states of multiple faces to be included to the stack, we group the losing states according to the faces, and give an order on the groups to be put to the stack. Only the group at the top of the order is put to the stack at a time. A face which has been processed is given a higher priority to be put to the stack. The processing order on the faces is used to define a subset $Q \subseteq R(G)$ and to restrict the rat-catcher moving within the faces of Q . Given a subset Q of $R(G)$, let $S_Q = \cup_{r \in Q} S_r$, $S'_Q = \cup_{r \in Q} S'_r$, and $T_Q = \cup_{e \in E(r), r \in Q} T_e$. We start with a small Q and perform the deletion process for the subgraph of $H(G, k)$ induced by the vertices of $S_Q \cup T_Q$ (or the subgraph of $H'(G, k)$ induced by the vertices of $S'_Q \cup T_Q$) until no deletion is possible. Then we enlarge Q by including a new face and repeat the deletion process. Q is enlarged gradually until $Q = R(G)$. By Observation 3.1, ST Procedure may stop at a small Q when the rat-catcher wins the game. Also, for a given subset Q , the losing states are deleted from X_r and X_e ($r \in Q, e \in E(r)$), and after the deletion, the data for X_r and X_e can be compressed before Q is enlarged. This helps in reducing the time and memory of ST Procedure.

The second common improvement is that we use a parsimonious data structure for edge data. We observe that there are many single vertex components in edge data. This makes the list of components for each edge very big. We keep the same face data as those in Naive. For each edge e , a component of G_e is called *non-trivial* if it has at least one edge otherwise called *trivial*. We only assign an index to a non-trivial component and keep a list of non-trivial components. We decide the integer type for I_e based on the number of non-trivial components in G_e . The length of the integer type for I_e is just big enough to encode the indices of non-trivial components of G_e . A trivial component $C = \{v_i\}$ is not kept in the list and $I_e[i]$ is used to indicate if (e, C) has been deleted from X_e or not. Further, if a non-trivial C_j has at least a constant fraction of n (δn) vertices

then the set of vertices of C_j is not kept in the list. If there are at most a constant number (c) of non-trivial components then the sets of vertices of the components are not kept in the list. In these cases, when an access to vertices of a non-trivial component is needed, we check I_e to find the vertices of the component. It is easy to see that this does not increase the order of the time complexity of the implementation. A smaller δ saves more memory but may give a larger running time. Similarly, a larger c saves more memory but may increase the running time. We have chosen $\delta = 1/100$ and $c = 100$ in this study. A distance matrix is used to keep the all pairs shortest distances. We decide the integer type for the distance matrix based on the input integer k to ST Procedure. When G is a medial graph, we can reduce the required memory size to about $4n^2$ bytes if one-byte integer arrays are used for each I_e and the distance matrix, and to about $7n^2$ bytes if two-byte integer arrays are used.

4.3 More improvements.

Improvement A_1 This improvement is based on Theorem 3.2. In A_1 , the elements (r, v) , $v \in U(r)$, are deleted from X_r and put to the stack at the initial step of the deletion process for face r . From Lemma 3.1, $U(r) \supseteq V(r)$ and computational studies show that $|U(r)|$ is usually much larger than $|V(r)|$. Therefore, A_1 gives a room for improving both the running time and memory space.

Improvement D_1 The features of D_1 can be expressed by dynamic data creation and data compression. In D_1 the data for a face (edge) are created only when ST Procedure starts to process the face (edge). When some losing states are deleted, the face/edge data are compressed. More specifically, when ST Procedure is to perform the first deletion for a face r , $U(r)$ is computed and array B_r of n elements is created. After the losing states (r, v_i) , $v_i \in U(r)$, are deleted from X_r , vertices of $V(G) \setminus U(r)$ are re-indexed and array B_r is compressed to indicate if (r, v_i) has been deleted for vertices v_i of $V(G) \setminus U(r)$ only. Similarly, when ST Procedure is to perform the first deletion for an edge e , G_e is computed and the edge data are created. Let r and r' be the two faces incident to e . We create two integer arrays I_e and I'_e for e . If the vertices of $V(G) \setminus U(r)$ have been re-indexed and B_r has been compressed then I_e is compressed accordingly. Similarly, array I'_e is compressed for the vertices of $V(G) \setminus U(r')$.

To calculate $U(r)$ and G_e , the shortest distances from vertex v_r^* to all other vertices in the planar dual graph G^* of G are needed for each face r in G .

When a distance matrix is used to keep the shortest distances, we need to solve $|R(G)|$ single source shortest path problems. In D_1 , the distance matrix is discarded. When we process a face r , we create the data for r and the data for I_e for all e incident to r . Since each edge is incident to two faces r and r' , the total number of single source shortest path calculations is bounded by $2|E(G)|$. When G has n vertices and is a medial graph, $|R(G)| = n + 2$ and $|E(G)| = 2n$. From this, if the distance matrix is used, we need to solve $n + 2$ single source shortest path problems while we need to solve at most $4n$ such path problems if D_1 is applied.

Combining D_1 with A_1 , the required memory size is now about $5n \times q$ bytes if one-byte integer arrays are used for I_e and I'_e and about $9n \times q$ if two-byte integer arrays are used, where q is the average of $|V(G) \setminus U(r)|$. For the Delaunay triangulation instances tested, q is less than $0.3n$ (instances dependent).

Improvement A_2 This improvement is based on Theorem 3.3. For each face r , instead of S_r , A_2 initializes X_r to include all states of S'_r .

Improvement A_3 This improvement is based on Theorem 3.4 and performs re-calculation for edge data. A_3 keeps the face data once they are created but keeps the edge data for only a pre-defined maximum number of edges. Once this number is reached A_3 starts to delete the entire X_e for some edge e . If a deleted X_e is needed again, X_e is re-computed from X_r , where r is incident to e .

Improvement D_2 In D_2 , we use a bit vector B_r for the data of face r , with one bit for one element of X_r . The memory size for face data is 1/8 of that when a one-byte Boolean array is used. But more complex bit operations have to be used.

It is easy to check that all improvements except A_3 do not change the order of running time of ST Procedure. However, applying A_3 , the running time of ST Procedure may become $O(n^3)$.

5 Computational results.

All of our efficient implementations use common improvements. In our implementations with any of improvements A_2, A_3 and D_2 , improvements A_1 and D_1 are always used. We do not mention A_1 and D_1 explicitly in those implementations. We test Naive and Implementations $A_1, A_1D_1, A_2, A_2D_2, A_3, A_3D_2, A_2A_3$, and $A_2A_3D_2$. Three classes of instances are used in the test. Class (1) of instances includes Delaunay triangulations of point sets taken from TSPLIB [17]. Those

instances are used as test instances in the previous studies [13, 14]. The instances in Class (2) are generated by the LEDA library [2, 16]. LEDA generates two types of planar graphs. One type of the graphs are the randomly generated maximal planar graphs and their subgraphs obtained from deleting some edges. Since the maximal planar graphs generated by LEDA always have branchwidth four, the subgraphs obtained by deleting edges from the maximal graphs have branchwidth at most four. The graphs of this type are not interesting for the study of branchwidth and branch decompositions. The other type of planar graphs are those generated based on some geometric properties, including Delaunay triangulations and triangulations of points uniformly distributed in a two-dimensional plane, and the intersection graphs of segments uniformly distributed in a two-dimensional plane. We will report the results on the intersection graphs. The instances in Class (3) are generated by the PIGALE library [1]. PIGALE randomly generates one of all possible planar graphs with a given number of edges based on the algorithms of [21]. We use Naive and our implementations to compute the carvingwidth of the medial graphs of the instances (i.e., the input graph to ST Procedure is not an instance itself but the medial graph of the instance). Our implementations are tested on a computer with Intel(R) Xeon(TM) 3.06GHz CPU, 2Gbytes physical memory and 8Gbytes swap memory. The operating system is SUSE LINUX 10.0, and the programming language we used is C++.

We compute an upper bound on the carvingwidth as the initial guessed input integer k to call ST Procedure. It is known that the branchwidth of a planar graph of n vertices is at most $\sqrt{4.5n}$ [11]. From this, $2\sqrt{4.5n}$ is an upper bound on the carvingwidth of the medial graph of an instance of n vertices. We follow a similar approach in [13] to compute another upper bound l : Let $M(G)$ be a medial graph of a planar graph G of n vertices. For each face r of $M(G)$ which corresponds to a vertex in G , we compute the eccentricity of v_r^* (the length of the longest shortest paths from v_r^* to all other vertices) in the planar dual $M(G)^*$. We initialize l as twice as the minimum eccentricity among all v_r^* 's. Finally, we take $k = \min\{2\sqrt{4.5n}, l\}$. Either the linear search or the binary search can be used to find the carvingwidth starting from the initial guessed k . In the linear search, when the rat-catcher wins, k is decreased by two and ST Procedure is called again until the rat wins the game. In the binary search, we call ST Procedure to search for the carvingwidth between k (upper bound) and the node degree of $M(G)$ (which is four and a lower bound). For the instances in classes (1) and (2), the eccentricity-based guess is very close to the carvingwidth and k always takes the value of l . The linear search uses a

smaller number of iterations to find the carvingwidth than the binary search. For instances in Class (3), the eccentricity-based guess could be very large and k may take $2\sqrt{4.5n}$ for large instances. Since $2\sqrt{4.5n}$ is still far away from the carvingwidth, the binary search does a better job. One may run the linear search and binary search in parallel and take the results from the one which finishes earlier.

5.1 Computation time and memory. Table 1 shows the computation time of Naive and efficient implementations for the carvingwidth of the medial graphs of the instances in Class (1). In the table, Itr is the number of iterations in the linear search. Table 2 shows the memory size (in megabytes) of those implementations. Only the data for relatively large instances are given in the tables.

For the instances in Class (1), one-byte integer arrays are used for each edge and the distance matrix. The most time efficient implementation is A_1 which is faster than Naive by a factor of at least 10 and uses at most 1/10 memory of Naive. With more improvements, the memory requirement is further reduced but the running time is slightly increased. The effect of data compression in Improvement D_1 is significant. The memory used by A_1D_1 is only about $1/3 \sim 1/4$ of that by A_1 . Improvement A_2 is effective in reducing the memory size. In general, the number of non-trivial components is small for both faces and edges, and thus the memory saving is big. The memory used by Improvement D_2 for face data is 1/8 of that when one-byte Boolean arrays are used. When the memory for face data becomes dominating, Improvement D_2 reduces memory requirement significantly. The most memory efficient implementation without re-calculation is A_2D_2 which is faster than Naive by a factor of $8 \sim 9$. For Instance *pla33810* which has 101,367 edges (corresponding to 101,367 vertices in the input medial graph), A_2D_2 uses about 500Mbytes memory, which is about $\frac{1}{20}n^2$ bytes, where n is the number of vertices in the input medial graph. Compared with Naive, the memory saving is by a factor of about 1000.

Improvement A_3 performs re-calculation for edge data. The performance depends on the maximum number of edges that are kept. This maximum number can be chosen based on the size of available memory. In general, a larger maximum number gives an implementation which uses more memory but runs faster (less re-calculations). The maximum number of kept edges is 500 for the results in the paper unless otherwise stated explicitly. Among all implementations, $A_2A_3D_2$ is the most memory efficient one. $A_2A_3D_2$ is faster than Naive by a factor of $6 \sim 7$. For Instance *pla33810*, $A_2A_3D_2$

uses less than 200Mbytes memory, which is about $\frac{1}{50}n^2$ bytes. Compared with Naive, the memory saving is by a factor of about 2500. The memory used by Implementation $A_2A_3D_2$ can be further reduced to about 155Mbytes for Instance *pla33810* with a slightly increase in the running time, if we keep at most 50 edges.

The instances in Class (2) are generated by the LEDA function *random_planar_graph* [2]. We have tested our implementations on instances of Delaunay triangulations and triangulations of points randomly distributed in a two-dimensional plane, and intersection graphs of segments. Our implementations have similar performances for the Delaunay triangulations and triangulations instances as those for the instances in Class (1). Table 3 gives the computation time and memory of Naive and Implementations A_1 , A_2D_2 , and $A_2A_3D_2$ for instances of intersection graphs of segments. In the table, Itr is the number of iterations in the linear search. The instances of intersection graphs of segments may have a large number of non-trivial components for edges and faces, and two-byte integer arrays are used to represent the edge data. Therefore, the memory usage is considerably larger than the Delaunay instances of the same size. As shown in the table, our efficient implementations are faster and use much less memory than Naive.

Instances of Class (3) are generated by the PIGALE library [1]. PIGALE provides a number of planar graph generators. Since 2-connected planar graphs are the most interesting class of graphs in the study of branchwidth and branch decompositions, we selected the function for generating 2-connected planar graphs. The function, given the number m of edges, randomly generates one of all possible 2-connected planar graphs of m edges. The output graph is usually a multi-graph with parallel edges. Since parallel edges are not interesting for branchwidth finding, we specify the function to produce simple 2-connected graphs. With a given m , the function outputs a 2-connected random planar graph with m' edges. Normally m' is smaller than m , since parallel edges are not kept and there are performance considerations [1]. Table 3 gives the computation time and memory of Naive and Implementations A_1 , A_2D_2 , and $A_2A_3D_2$. In the table, Itr is the number of iterations in the binary search. The instances in Class (3) may have a small number of non-trivial edge components, but we still use two-byte integer arrays for the edge data. For this class of instances, the eccentricity-based guess is usually bad. For example, the medial graph of Instance *PI37730* has carvingwidth 12, but the eccentricity-based guess is 8974. For large instances tested, the binary search always finishes earlier than the linear search. The

Table 1: Computation time (in seconds) of Naive and efficient implementations for Class (1) instances.

Instances	Number of edges	bw	Itr	Computation time (in seconds)								
				Naive	A_1	A_1D_1	A_2	A_2D_2	A_3	A_3D_2	A_2A_3	$A_2A_3D_2$
pr1002	2,972	21	2	51.2	4.23	5.38	6.07	6.35	5.58	5.83	6.52	6.98
rl1323	3,950	22	2	95.7	6.47	8.0	9.19	9.56	8.65	9.05	12.3	13.1
d1655	4,890	29	2	158	11.3	15.0	17.3	17.6	15.7	16.7	21.6	22.3
rl1889	5,631	22	2	195	13.8	16.6	20.1	20.8	19.2	20.4	29.5	30.5
u2152	6,312	31	3	X	24.5	35.5	41.8	42.4	39.9	42	61.6	62.4
pr2392	7,125	29	2	X	21.1	25.8	32.1	32.8	31.8	31.3	49.1	50.4
pcb3038	9,101	40	2	X	31.6	41.3	50.8	51.9	44.6	49.7	83.2	74.2
fl3795	11,326	25	2	X	63.7	80.2	99	104	86.3	98	155	163
fnl4461	13,359	48	2	X	67.4	92.4	116	119	97.1	110	185	185
rl5934	17,770	41	2	X	151	197	245	249	213	241	385	391
pla7397	21,865	33	2	X	246	296	376	385	393	453	606	629
usa13509	40,503	63	4	X	X	1,061	1,359	1,371	1,241	1,386	2,154	2,165
brd14051	42,128	68	2	X	X	1,061	1,418	1,417	1,226	1,361	2,274	2,282
d15112	45,310	78	3	X	X	2,070	2,810	2,852	2,379	2,598	4,549	4,603
d18512	55,510	88	2	X	X	X	2,315	2,321	2,100	2,241	3,752	3,756
pla33810	101,367	100	5	X	X	X	12,379	12,614	X	14,747	20,482	21,734

number of iterations in the binary search is about 10 for large instances, and this prohibits us from solving very large instances in a reasonable time. The memory usage for the PIGALE instances is very small, compared to the instances of Classes (1) and (2) even two-byte integer arrays are used for edge data.

For the instances in Class (3), Implementation A_2D_2 is very memory efficient. This indicates that the numbers of non-trivial components for faces in those instances are small. The gap between the running time of Implementation A_1 and that of Implementation A_2D_2 is a little bigger for instances of this class than the gap for instances in the other two classes. This can be explained by the following reasons. Naive and Implementation A_1 keep the shortest distance matrix, while A_2D_2 discards the matrix. As analyzed in Improvement D_1 , Naive and A_1 need to solve $n + 2$ single source shortest path problems while A_2D_2 may need to solve $4n$ such problems, where n is the number of vertices of the input medial graph. A_2D_2 also needs to calculate the components of G_r for every face r and there is no such computation in Naive and A_1 . Notice that each of the shortest distances calculation, the computation of components of G_r for all r , and the deletion process takes $O(n^2)$ time. For instances of Classes (1) and (2), the time of deletion process is larger than the sum of the other two. However, the deletion process runs faster for the instances of Class (3) than for instances of Classes (1) and (2). In this case, the shortest distances calculation and the computation of components of G_r may become a dom-

inating part of the total running time (see [5] for more details).

Among all implementations, the most time efficient one is A_1 . Compared with Naive, A_1 is faster by a factor of $3 \sim 15$. The memory saving of A_1 is also significant. A_1 can solve an instance of about 20,000 edges in Class (1) and instances of about 15,000 edges in Classes (2) and (3), while Naive can only solve instances of size up to about 5,000 edges for all three classes. The most memory efficient implementation without re-calculation is A_2D_2 . It can solve an instance of about 1,000,000 edges in Class (1) by about 3.5 hours and 500 Mbytes, an instance of about 60,000 edges in Class (2) by about 1.5 hours and 1.5Gbytes memory, and an instance of about 1,000,000 edges in Class (3) by about 14 hours and 200 Mbytes. Implementation $A_2A_3D_2$ is the most memory efficient one among all implementations. It can solve an instance of about 1,000,000 edges in Class (1) by about 6 hours and 200 Mbytes, an instance of about 1,000,000 edges in Class (2) by about 6 hours and 1.4Gbytes, and an instance of about 700,000 edges in Class (3) by by about 14 hours and 160 Mbytes. All implementations without using A_3 have time complexity $O(n^2)$ since no re-calculation for edge data is performed. In the worst case, Implementation $A_2A_3D_2$ may perform re-calculation repeatedly for some edges and has time complexity $O(n^3)$. However, the worst case scenario has not been observed and the running time of Implementation $A_2A_3D_2$ is at most as twice as that of Implementation A_2D_2 for most instances.

Table 2: Memory usage (in megabytes) of Naive and efficient implementations for Class (1) instances.

Instances	Number of edges	Maximum Memory Usage (Mbyte)								
		Naive	A_1	A_1D_1	A_2	A_2D_2	A_3	A_3D_2	A_2A_3	$A_2A_3D_2$
pr1002	2,972	413	39	16	8	8	10	9	8	7
rl1323	3,950	734	66	23	11	10	15	13	11	9
d1655	4,890	1,188	99	30	14	11	17	14	13	10
rl1889	5,631	1,424	130	46	18	14	28	21	16	12
u2152	6,312	X	161	41	17	14	26	20	16	13
pr2392	7,125	X	204	66	21	17	35	26	19	15
pcb3038	9,101	X	328	76	25	21	36	27	22	19
fl3795	11,326	X	504	132	58	42	66	63	40	23
fnl4461	13,359	X	698	158	39	32	66	43	33	26
rl5934	17,770	X	1,226	358	67	51	155	86	50	35
pla7397	21,865	X	1,850	436	123	85	238	144	83	44
usa13509	40,503	X	X	1,534	220	153	498	271	149	79
brd14051	42,128	X	X	1,600	215	149	580	283	149	82
d15112	45,310	X	X	1,795	227	156	508	256	156	86
d18512	55,510	X	X	X	284	198	706	328	194	106
pla33810	101,367	X	X	X	814	508	X	876	507	198

5.2 Comparison with previous works. Hicks proposes a straightforward implementation *rat* and two memory friendly implementations *comprat* and *memrat* of ST Procedure [13]. The implementations are tested using instances of Class (1) on a SGI Power Challenge with 6×194 MHz processors, 1Gbytes of physical memory, and 1Gbytes of swap space. To compare our results with Hicks', we quote some data of [13] in Table 4. An M in the table indicates that the implementation runs out of 2Gbyte memory for that instance. From the table, *rat* runs out of 2Gbyte memory for instances of rl1889 (5,631 edges) and larger. Naive of this paper can solve rl1889 but runs out of 2Gbyte memory for instances of u2152 (6,312 edges) and larger. This confirms that straightforward implementations of ST Procedure are memory consuming. The memory used by *memrat* for Instance brd14051 (the largest one reported in [13]) is about 600Mbytes. For the same instance, A_2D_2 uses about 1/4 and $A_2A_3D_2$ uses about 1/8 of the memory of *memrat*. Implementation A_1 is faster by a factor of $200 \sim 500$, Implementation A_2D_2 is faster by a factor of $100 \sim 400$, and Implementation $A_2A_3D_2$ is faster by a factor of $100 \sim 200$ than *comprat* and *memrat* for large instances. Notice that the CPU used in [13] has frequency 194MHz and the CPU used in this paper has frequency 3.06GHz, so we need to keep in mind this difference of speed when we compare the running time.

5.3 Computing branch decompositions. Seymour and Thomas give an algorithm, which is known

as *edge contraction method*, for computing an optimal branch decomposition of a planar graph [22]. The contraction of an edge e in a graph G is to remove e from G , identify the two end vertices of e by a new vertex, and make all edges incident to e incident to the new vertex. We denote by G/e the graph obtained by contracting e in G . Given a 2-connected planar graph G , the algorithm of Seymour and Thomas computes an optimal branch decomposition of G by a sequence of edge contractions of the medial graph $M(G)$ of G as follows: First the carvingwidth cw of $M(G)$ is computed by ST Procedure. An edge e of $M(G)$ is *contractible* if the carvingwidth of $M(G)/e$ is at most cw and $M(G)/e$ is 2-connected. Next, a contractible edge e of $M(G)$ is found by ST Procedure and $M(G)$ is contracted to graph $M(G)/e$. The contraction is repeated on $M(G)/e$ until the graph becomes one with three vertices. A carving decomposition of $M(G)$ with width at most cw is constructed based on the sequence of edge contractions. Finally, the branch decomposition of G is obtained from the carving decomposition of $M(G)$. It is proved in [22] that for any 2-connected planar graph there is a contractible edge and for a 2-connected planar graph G , $M(G)$ is 2-connected. To check if an edge is contractible, ST Procedure is used to test if $M(G)/e$ has carvingwidth at most cw . In the worst case, all edges may be checked to find a contractible one and for a graph of n vertices, the algorithm of Seymour and Thomas may call ST Procedure $O(n)$ times for one contraction and $O(n^2)$ times in total. So the time complex-

Table 3: Computation time and required memory of Naive and Implementations A_1 , A_2D_2 , and $A_2A_3D_2$.

Class	Instances	Number of edges	bw	Itr	Time (seconds)				Memory (Mbyte)			
					Naive	A_1	A_2D_2	$A_2A_3D_2$	Naive	A_1	A_2D_2	$A_2A_3D_2$
(2)	rand3050	5,032	9	4	283	32.5	34.2	61.5	1,179	180	32	22
	rand6000	10,261	12	2	X	95.5	101	147	X	724	92	41
	rand8700	15,090	14	3	X	292	370	849	X	1,559	160	71
	rand11500	20,279	13	2	X	X	557	2,002	X	X	269	120
	rand33000	60,398	20	2	X	X	5,633	8,540	X	X	1,472	624
	rand54000	100,037	22	2	X	X	X	21,417	X	X	X	1,382
(3)	PI1180	2,022	7	5	23	6.8	9.1	10.4	196	32	8	7
	PI2995	5,043	7	6	156	58.7	93.3	96.5	1,034	178	14	13
	PI5940	10,016	7	8	X	289	522	563	X	683	28	26
	PI8950	15,097	10	9	X	907	1,771	2,089	X	1,541	48	39
	PI11974	20,071	9	9	X	X	3,646	3,702	X	X	46	44
	PI37730	70,022	6	10	X	X	49,136	49,180	X	X	188	160

ity of the algorithm is $O(n^4)$.

We call a contractibility test on an edge a *positive one* if the edge is tested contractible, otherwise a *negative one*. Gu and Tamaki give an algorithm which uses a better strategy to find positive tests [12]. When a negative test is obtained on an edge then the edge will not be tested again unless a necessary condition for that edge to be contractible is satisfied. By this improvement, the algorithm of Gu and Tamaki avoids the repeated negative tests on a same edge, calls ST Procedure $O(n)$ times, and has time complexity $O(n^3)$ for computing an optimal branch decomposition of a planar graph.

We test the algorithm of Seymour and Thomas and the algorithm of Gu and Tamaki for instances in three classes using a number of heuristics to select edges for testing the contractibility. The instances of Class (2) are generated by LEDA function *SEGMENT_INTERSECTION* [2]. Implementation A_1 , the most time efficient one, is used as ST Procedure. Both the algorithms have the minimum number of negative calls and running time when the round robin edge selection heuristic is used. The computer used for testing has an AMD Athlon(tm) 64 X2 Dual Core Processor 4600+ (2.4GHz) and has a similar performance with the PC for testing ST Procedure. Table 5 gives the number of calls of ST Procedure and computation time for the algorithms with the round robin edge selection heuristic. The data in the table show that optimal branch decompositions of planar graphs of a few thousands edges can be computed in a practical time. For most instances tested, repeated negative tests are not observed on any

edge in the algorithm of Seymour and Thomas. So the advantage of the algorithm of Gu and Tamaki is not shown by those instances when the round robin edge selection heuristic is used. On some other edge selection heuristic, more repeated negative tests are observed in the algorithm of Seymour and Thomas. In this case, the algorithm of Gu and Tamaki has much less negative calls and runs faster than the algorithm of Seymour and Thomas. We omit the details here.

6 Concluding remarks.

We tested our implementations on instances of size up to one hundred thousand edges. The results of this paper show that the branchwidth of those instances can be computed within a reasonable time and memory size. This suggests that the required memory may not be a bottleneck for computing branchwidth and optimal branch decompositions of planar graphs in practice. Our implementations require $O(n^2)$ bytes memory, although the constant behind the Big-Oh may be small. For the instances of n edges (n vertices in the input medial graphs), the memory required for face data could be as large as $n^2/8$ bytes. So they may not be able to solve extremely large instances with a few hundred thousands or more edges within a practical memory size. One approach to solve such instances is to have external memory implementations of ST Procedure. Another approach is to perform re-calculations for face data as well. How to bound the time complexity for re-calculating face data would be a key for this approach.

We have an upper bound $O(n^3)$ on the time complexity of the implementations with re-calculations for

Table 4: Computation time (in seconds) of *rat*, *comprat*, and *memrat* quoted from Table 1 of [13].

Instances	Number of edges	<i>bw</i>	<i>Itr</i>	Computation time (in seconds).		
				<i>rat</i>	<i>comprat</i>	<i>memrat</i>
pr1002	2,972	21	2	338	448	562
rl1323	3,950	22	3	876	1,519	1,590
d1655	4,890	29	3	1,318	1,608	2,206
rl1889	5,631	22	3	M	3,931	4,012
u2152	6,312	31	4	M	3,207	4,704
pr2392	7,125	29	3	M	3,813	5,167
pcb3038	9,101	40	4	M	13,817	15,865
fl3795	11,326	25	3	M	18,469	17,142
fnl4461	13,359	48	4	M	35,933	51,305
rl5934	17,770	41	3	M	73,468	66,461
pla7397	21,865	33	2	M	65,197	53,564
usa13509	40,503	63	1/2	M	M	413,861
brd14051	42,128	68	3	M	M	594,468

edge data. Let p be the maximum number of edges kept in those implementations. This bound is true for any $p \geq 1$. In general, a larger p results in a faster running time of the implementations. It is interesting to prove a better upper bound related to p , say $O(n^2(n/p))$, on the time complexity of those implementations.

We have used the eccentricity-based heuristic to get an initial guess for the starting value of k . This heuristic runs fast and gives a guess very close to the carvingwidth for the geometric instances. However, the guess is poor for instances generated by the PIGALE library. The heuristic of [23] can also be used for the guess but it has a performance very similar to the eccentricity-based one: good for geometric instances but poor for the PIGALE instances. The poor guess for the PIGALE instances results in a large number of iterations in the computation, which may become a bottleneck for computing the branchwidth for large instances in this class. To reduce the number of iterations, it is interesting to develop a performance guaranteed and yet efficient heuristic for computing a good approximation of the carvingwidth.

For large instances, computing optimal branch decompositions is still time consuming by the edge contraction method. Divide-and-conquer is a more efficient approach in practice for computing branch decompositions. In this approach, roughly speaking, we partition the medial graph of an instance into two (or more) subgraphs such that the size of the cut set between the two subgraphs is bounded by the carvingwidth of the medial graph. Then we use ST Procedure to test if each subgraph has carvingwidth at most that of the medial graph. If both answers are

positive (such a partition is called valid), we recursively find the carving decomposition of each subgraph. A heuristic based on this divide-and-conquer approach is reported in [14]. When the heuristic can not find a valid partition in a recursive step, it uses the edge-contraction method to make a progress. In the worst case, the heuristic may become the edge-contraction algorithm and have time complexity $O(n^4)$. It is worth to explore efficient heuristics to find a valid partition in every recursive step.

Acknowledgment

The authors thank Dr. I.V. Hicks for providing the test instances of Class (1). The work was partially supported by the NSERC Research Grant of Canada and the Japan Society for the Promotion of Science (JSPS) Grant-In-Aid for Scientific Research. The authors also thank anonymous reviewers for their constructive comments.

References

- [1] Public Implementation of a Graph Algorithm Library and Editor, 2007. <http://pigale.sourceforge.net/>.
- [2] The LEDA User Manual, Algorithmic Solutions, Version 4.2.1, 2007. <http://www.mpi-inf.mpg.de/LEDA/MANUAL/MANUAL.html>.
- [3] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embedding in a k -tree. *SIAM J. on Discrete Mathematics*, 8:277–284, 1987.
- [4] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

