A Combinatorial Optimization Algorithm for Solving the Branchwidth Problem

[1]J. Cole Smith, [2]Elif Ulusal, [3]Illya V. Hicks

Abstract

In this paper, we consider the problem of computing an optimal branch decomposition of a graph. Branch decompositions and branchwidth were introduced by Robertson and Seymour in their series of papers that proved the Graph Minors Theorem. Branch decompositions have proven to be useful in solving many NP-hard problems, such as the traveling salesman, independent set, and ring routing problems, by means of combinatorial algorithms that operate on branch decompositions. We develop an implicit enumeration algorithm for the optimal branch decomposition problem and examine its performance on a set of classical graph instances.

Keywords. Branch decomposition; branchwidth; implicit enumeration; partitioning.

1 Introduction

Branch and tree decompositions, and their connectivity invariants branchwidth and treewidth, were introduced by Robertson and Seymour in 1991 [15]. It has been shown that tree decompositions are beneficial for solving NP-complete problems that are modeled on graphs with bounded treewidth, using dynamic programming techniques [1, 6]. The benefits of identifying a graph's treewidth also apply to branchwidth, since branchwidth and treewidth of a graph bound each other by constants [15]. The algorithms that utilize branch decompositions to solve NP-complete problems are called branch decomposition-based algorithms.

[1]University of Florida, Industrial & Systems Engineering, cole@ise.ufl.edu
[2]Texas A & M University, Industrial & Systems Engineering, elif@tamu.edu
[3]Rice University, Computational & Applied Mathematics, ivhicks@rice.edu

In this paper, we investigate exact optimization techniques to find optimal branch decompositions. Testing if a general graph has branchwidth at most some input integer k was proved to be NP-complete by Seymour and Thomas [17]. We develop an alternative algorithm based on implicit enumeration and compare its performance against a branch decomposition-based algorithm [12] for a set of classical graph instances.

Branch decomposition-based algorithms have been proposed for ring routing [4], traveling salesman [5], general minor containment [11], optimal branch decomposition [12], and independent set [8] problems. Branch decomposition-based algorithms for combinatorial optimization problems first identify a branch decomposition of small width, and then use the branch decomposition in a dynamic programming fashion to solve the problems of interest. This procedure is promising for finding fast algorithms, but encounters two important limitations. One is that not all optimization problems can be solved in polynomial time using this procedure, e.g., the bandwidth minimization problem [9, 14]. In addition, the run time and space strongly depend on the width of the branch decomposition found in the first step, since the worst-case complexity of these algorithms is an exponential function of the width.

Although computing a general graph's branchwidth is NP-hard, Seymour and Thomas [17] developed a polynomial-time algorithm for finding an optimal branch decomposition of a planar graph. Another algorithm [13] based on the framework of Seymour and Thomas [17] improved these algorithms in terms of memory usage and practical run time. Later, Bian et al. [2] offered efficient implementations of the procedure presented by Seymour and Thomas.

For general graphs, Robertson and Seymour [16] offered the first heuristic that estimates branchwidth within a factor of 3. Cook and Seymour [4] offered a heuristic called the eigenvector method and showed that branch decompositions produced by this method can be practically used for solving the traveling salesman problem [5]. Hicks [10] offered two other heuristics for computing branchwidth, one called the diameter method and the other

a hybrid of the diameter method and the eigenvector method. We will utilize the hybrid method to generate upper bounds in solving our formulations.

Section 2 reviews necessary graph theory concepts for this paper and defines the branchwidth problem. Section 3 presents various bounding algorithms and properties of branch decompositions that we use in our approach. Section 4 describes our implicit enumeration algorithm for finding the branchwidth of a graph. Section 5 offers computational results of our proposed models, and Section 6 concludes the paper with directions for further research.

2 Problem Statement

2.1 Basic definitions

A *graph* is an ordered pair (V, E) where V is a nonempty set of *vertices* or *nodes*; and where E, the set of *edges*, is an unordered binary relation on V. A *hypergraph* is an ordered pair of node and edge sets, and an incidence relationship between them that is not restricted to two ends for each edge. A *loop* is an edge that starts and ends on the same vertex. *Multiple edges* are two or more edges that are incident to the same two vertices. A *simple graph* is a graph with no loops and no multiple edges. All of our graphs are simple, unless stated otherwise.

A *walk* in a graph is a sequence of vertices such that an edge exists between each adjacent pair of vertices in the sequence. A *path* is a walk with no repeated vertices. A *cycle* is a closed walk with no repeated vertices aside from the starting and ending vertices. A graph is *connected* if every pair of vertices can be joined by a path. A connected graph that does not contain any cycle is called a *tree*. The *degree* of a vertex v, $deg(v)$, is the number of edges incident to v. The *leaves* of a tree are the vertices of degree one. The nodes that are not leaves are called *non-leaf nodes*. A graph is called *complete* if there is an edge between every pair of vertices.

A graph $\hat{G} = (\hat{V}, \hat{E})$ is a *subgraph* of the graph $G = (V, E)$ if $\hat{V} \subseteq V$ and $\hat{E} \subseteq E$. A graph obtained by a set of edges F and a set of vertices consisting of the ends of the edges in F is referred to as the graph induced by edge set F. *Contraction* of an edge e means deleting e and identifying its ends into one node. A graph H is a *minor* of a graph G if H can be obtained from a subgraph of G by a series of edge contraction operations. The *connectivity* of a graph is the smallest number of vertices that can be removed to disconnect the graph. A *biconnected* graph is a connected graph that has connectivity at least two. Similarly, the *edge connectivity* of a graph is the smallest number of edges that can be removed to disconnect the graph. A graph is called *2-edge connected* if its edge connectivity is two or more.

2.2 Branch decompositions

Let $G = (V_G, E_G)$ be a hypergraph and T be a ternary tree (a tree where every non-leaf node has degree three) with $|E_G|$ leaves and edge set E_T. Let ν be a bijection from the edges of G to the leaves of T. The pair (T, ν) is defined to be a *branch decomposition* of G [15]. A *separation* of G is a pair (G_1, G_2) of subgraphs with $G_1 \cup G_2 = G$ and $E_{G_1 \cap G_2} = \emptyset$. The *order* of this separation is defined as $|V_{G_1 \cap G_2}|$. (For simplicity, we will refer to an edge's "order" directly, rather than to the order of its separation.) Let (T, ν) be a branch decomposition. Removing an edge q from T partitions the edges of G into two subsets A_q and B_q. A branch decomposition (T, ν) is called *connected* if for every edge $q \in E_T$, the two edge-induced subgraphs of G corresponding to the components of $T \backslash q$ are both connected. Figure 1 shows a graph G, a branch decomposition tree T of G, and the separation generated by removing edge q from T. The *width* of a branch decomposition (T, ν) is the maximum order among all edges of the branch decomposition. The *branchwidth* of G, $\beta(G)$, is the minimum width over all branch decompositions of G. A branch decomposition of G with width equal to the branchwidth is an *optimal branch decomposition* of G. The branchwidth problem is to find the branchwidth and an optimal branch decomposition of a given graph.

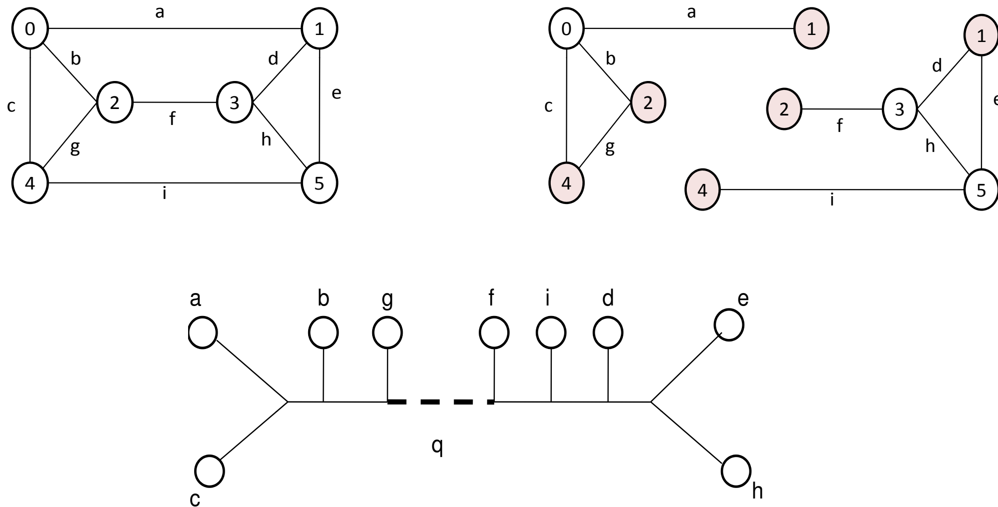

Figure 1: G (top-left), a branch decomposition tree T of G (bottom), and a separation {1,2,4} generated by removing q from T (top-right)

3 Branch Decomposition Properties and Heuristics

3.1 Preprocessing

We begin the description of our optimization algorithms for the branchwidth problem by discussing a simple preprocessing step that helps reduce the number of graph edges. Assume that we are given a biconnected graph G. In addition, suppose there exists a vertex, w, in G with degree equal to two. Let $e = (w, q)$ and $f = (w, r)$ be the edges incident to w. By a result of Cook and Seymour [4, 5], there exists an optimal branch decomposition such that the leaves mapped to the edges incident to w are adjacent to the same non-leaf node. In addition, one can visualize this procedure by replacing w and its incident edges with the edge qr. Figure 2 shows a graph and two steps of the reduction process, leading to the connection of three leaf nodes to two non-leaf nodes.

Note that edge e or f may itself represent a subgraph of G, if it corresponds to a set of edges that have already gone through the reduction process. In this case, the subtrees corresponding to e and f are connected to a designated non-leaf node. Hence, in the reduction procedure, at each step we search for a vertex of degree two, perform a reduction on this

vertex, update the graph, and repeat this procedure until we have no more degree-two vertices.

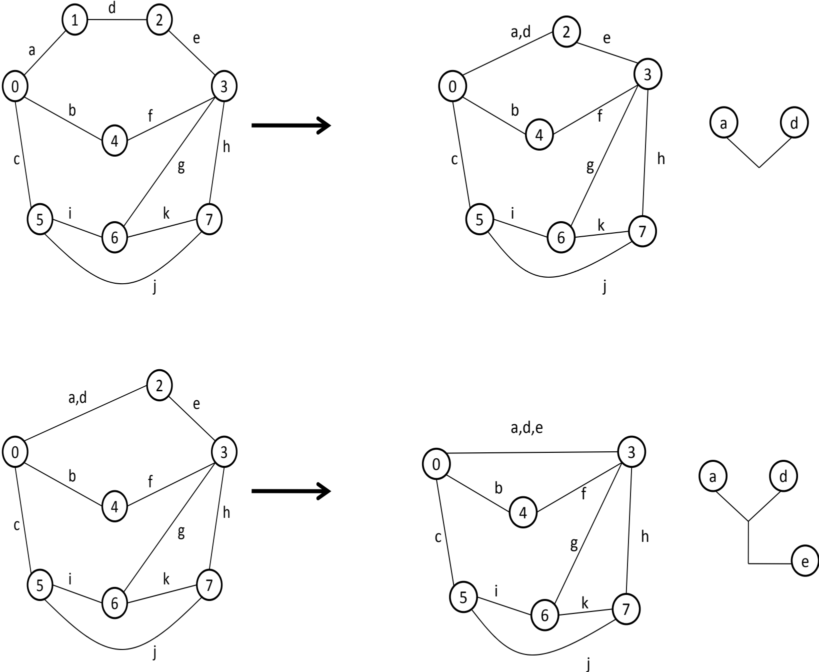

Figure 2: A set of reductions on G

3.2 Heuristic bounds

Hicks [10] implemented two heuristics for finding good branch decompositions called the diameter and hybrid methods, based partially on the ideas of Cook and Seymour [4]. We use the hybrid method and add the width obtained using this heuristic as an upper bound for the objective function.

To compute a lower bound on branchwidth, we note that the treewidth and branchwidth of a graph bound each other by constants. Robertson and Seymour [15] proved that for a graph $G = (V, E)$ with $E \neq \emptyset$, $\max\{\beta(G), 2\} \leq \tau(G) + 1 \leq \max\{\lfloor \frac{3}{2}\beta(G) \rfloor, 2\}$, where $\tau(G)$ is the treewidth of G. Thus, any lower bound for treewidth leads to a lower bound for branchwidth of a graph. The lower bound heuristic we use is based on a treewidth lower bound offered by Bodlaender et al. [3], which is called contraction degeneracy.

Define the degree of graph G, $\delta(G)$, to be the minimum degree of any vertex in G. The *contraction degeneracy*, $\delta C(G)$, of a graph G is defined as the maximum degree of any (non-empty) minor of G. To determine if the contraction degeneracy of a graph G is at least some integer k is NP-complete [3]. Bodlaender et al. [3] implemented several heuristics with different contraction strategies to find lower bounds for the contraction degeneracy. We use a slight modification of the one that performs best experimentally. Given an input graph $G = (V, E)$, the contraction degeneracy heuristic is summarized in Figure 3.

Step 1: Make a copy G' of G, and set the lower bound on the contraction degeneracy to be $\delta_{LB}C(G) := \delta(G')$.

Step 2: Find a minimum degree vertex v of G', breaking ties randomly.

Step 3: Select a vertex u adjacent to v in G', such that u and v have the fewest common neighbors.

Step 4: Contract edge (u, v), update G', and compute $\delta(G')$.

Step 5: If $\delta(G') > \delta_{LB}C(G)$, then set $\delta_{LB}C(G) := \delta(G')$.

Step 6: Repeat steps 2–5 until G' has no more than $\delta_{LB}C(G) + 1$ vertices.

Step 7: Repeat steps 1–6 $\lceil \frac{|V|}{\delta(G)} \rceil$ times, and select the highest obtained contraction degeneracy.

Figure 3: Contraction degeneracy heuristic

Note in Step 6 that when there are $\delta_{LB}C(G) + 1$ vertices remaining in the graph, the maximum degree of any node is $\delta_{LB}C(G)$, and the heuristic cannot improve the contraction degeneracy bound.

Usually, in Steps 2 and 3 of this algorithm, we encounter ties, which we break randomly in our implementation (Bodlaender et al. [3] do not prescribe a method to break ties). Bodlaender et al. [3] also mention that this heuristic may not perform very well for sparse graphs. However, we obtained good lower bounds for sparse graphs in our experiments, which we credit to the random selection of minimum-degree vertices, and repeating this randomized algorithm $\lceil \frac{|V|}{\delta(G)} \rceil$ times to get an improved bound.

Since contraction degeneracy is a lower bound for treewidth and treewidth is a lower

bound for branchwidth, we have the following relationship:

$$\begin{aligned} \delta_{LB}C(G)+1 \leq \tau(G)+1 &\leq \max(\lfloor \frac{3}{2}\beta(G) \rfloor, 2) \\ \Rightarrow \lceil \frac{2}{3}(\delta_{LB}C(G)+1) \rceil &\leq \beta(G). \end{aligned} \quad (1)$$

We thus use $\lceil \frac{2}{3}(\delta_{LB}C(G)+1) \rceil$ as a lower bound on the branchwidth for a graph.

3.3 Formulation on hypergraphs

As a final note to this section, we describe a generic transformation algorithm that allows all of the foregoing procedures to be applied to hypergraphs as well, as given by Seymour and Thomas [17]. Let H be a hypergraph, and G be a graph constructed by replacing each hyperedge of form $\{u_1, \ldots, u_t\}$ with a complete bipartite graph with vertex set $\{u_1, \ldots, u_t, v_1, \ldots, v_t\}$, i.e., there is an edge connecting every u_i and v_j, $\{i, j\} \in \{1, \ldots, t\}$. Then the branchwidth of H is equal to the branchwidth of G.

4 Implicit Enumeration Algorithm

We consider a technique for solving the branchwidth problem via an implicit enumeration procedure. We begin the procedure by executing the heuristics described in Section 3.2, obtaining an upper bound UB (along with an *incumbent* branch decomposition solution whose width equals UB), and lower bound LB. We terminate if $LB = UB$, with the incumbent solution being optimal. Otherwise, we seek a branch decomposition having width less than UB. The key idea in this section is to observe that the removal of a non-leaf node from a branch decomposition tree results in three tree components, and that the original graph edges are partitioned among the three components. We thus seek a *tripartition* of edges into these components.

Consider the deletion of some non-leaf node, which we call the *primary* node, from

a branch decomposition tree T. Denote the set of original graph edges contained as leaf nodes in those components as $\mathcal{C}^j$, and label the edges that connected the components to the primary node as ξ^j, for $j = 1, 2, 3$. Next, we say that component j *occupies* an original graph node $i \in V$ if at least one edge in $\mathcal{C}^j$ is incident to i. If at least two components occupy $i \in V$, then i contributes to the order of ξ^j for each $j = 1, 2, 3$ such that component j occupies i. Let λ^j denote the order of edge ξ^j for the branch decomposition tree.

An naive search could enumerate all possible edge tripartitions, and determine an upper and lower bound on the width of any branch decomposition tree that contains the tripartition. We update the incumbent solution and value of UB if a new best solution is found during the search. The lower bound on the branchwidth of G is the maximum of the initial LB value and the smallest lower bound identified during the search. However, this scheme will not be practically useful unless effective bounding strategies are used, and unless we provide search and fathoming rules that limit the set of tripartitions that are examined. Sections 4.1 and 4.2 respectively describe the bounding and fathoming strategies that we employ in our search.

4.1 Bounds obtained from edge tripartitioning

Given a tripartitioning of edges into sets $\mathcal{C}^1$, $\mathcal{C}^2$, and $\mathcal{C}^3$, it is possible to establish upper and lower bounds on the width of any branch decomposition tree that contains this tripartition. To find an upper bound, we utilize a modification of the hybrid algorithm of Hicks [10] in which edges are partitioned into sets as given by $\mathcal{C}^1$, $\mathcal{C}^2$, and $\mathcal{C}^3$, and take the largest of these three upper bounds.

Alternatively, for $j = 1, 2, 3$, consider the hypergraph consisting of all edges in $\mathcal{C}^j$, plus a hyperedge that is incident to every node in ξ^j, say $\mathcal{H}^j$. Let μ^j be the width of a branch decomposition for each corresponding hypergraph. It is easy to see that the branchwidth of the original graph is at most the maximum of μ^1, μ^2, and μ^3. Therefore, we take the smaller of the bound provided by the modified algorithm of Hicks [10], and $\max\{\mu^1, \mu^2, \mu^3\}$,

as an upper bound on the optimal branch decomposition of a tree that contains the given tripartition.

To obtain lower bounds on the width of a branch decomposition containing the given tripartition, we establish the following lemma, whose proof is straightforward and is omitted for brevity.

Lemma 1 *Consider a tripartition of original graph edges into three nonempty sets* $\mathcal{C}^j$*,* $j = 1, 2, 3$*. Let* ℓ^j *be a lower bound on the branchwidth of the corresponding* $\mathcal{H}^j$*,* $j = 1, 2, 3$*, where* $\ell^j = 2$ *if* $|\mathcal{C}^j| < 4$*. Then* $max\{\ell^j, \lambda^j : j = 1, 2, 3\}$ *is a lower bound on the width of any branch decomposition that contains the tripartition.*

As an implementation note, we compute each ℓ^j by executing our contraction degeneracy heuristic on the hypergraphs $\mathcal{H}^j$ as described above, and then on the graph in which the hyperedge containing nodes in ξ^j is ignored. (The *optimal* branchwidth of the former graph will not be less than that of the second graph, but the contraction degeneracy heuristic may provide a larger bound on the latter graph.) We take ℓ^j to be the maximum of these two bounds.

4.2 Search and fathoming rules

In order to reduce the computational effort required by our procedure, we first observe that any feasible branch decomposition has $|E| - 2$ different tripartitions that could be selected by simply taking each non-leaf node as the primary node. Furthermore, we anticipate that our algorithm will become less effective if one of the tripartition sets is almost as large as the original graph, because the bounds for this large component are likely to be the same as those for the original graph. Given these considerations, we utilize the following lemma in our algorithm.

Lemma 2 *Consider any feasible branch decomposition of a graph* $G = (V, E)$*. There exists a primary (non-leaf) node in this tripartition such that* $max_{j=1,2,3}\{|\mathcal{C}^j|\} \leq \lceil (|E| - 1)/2 \rceil$*.*

Proof. First, if $|E| = 4$, this bound trivially holds since tripartitions are nonempty. For $|E| \geq 5$, consider a branch decomposition tree and a primary node $i^\star$ whose associated tripartition minimizes $\max_{j=1,2,3}\{|\mathcal{C}^j|\}$. Without loss of generality, let $|\mathcal{C}^1|$ equal this maximum value, and by contradiction, suppose that $|\mathcal{C}^1| > \lceil (|E| - 1)/2 \rceil$. Consider the tripartition induced by the non-leaf node $\bar{\imath}$ adjacent to $i^\star$, such that the path from $i^\star$ to any leaf node corresponding to an edge in $\mathcal{C}^1$ goes immediately to $\bar{\imath}$. (Such a node must exist since $|\mathcal{C}^1| \geq 3$ given that $|E| \geq 5$.) Now, observe that the tripartition $\bar{\mathcal{C}}^j$, $j = 1, 2, 3$ given by deleting $\bar{\imath}$ obeys the properties that $\bar{\mathcal{C}}^1 \bigcup \bar{\mathcal{C}}^2 = \mathcal{C}^1$, and $\bar{\mathcal{C}}^3 = \mathcal{C}^2 \bigcup \mathcal{C}^3$. Noting that $\bar{\mathcal{C}}^1$ and $\bar{\mathcal{C}}^2$ are both nonempty, we have that $\bar{\mathcal{C}}^1 \subset \mathcal{C}^1$ and $\bar{\mathcal{C}}^2 \subset \mathcal{C}^1$. Furthermore, $|\bar{\mathcal{C}}^3| = |E| - |\mathcal{C}^1| \leq |E| - \lceil (|E| - 1)/2 \rceil - 1 \leq \lceil (|E| - 1)/2 \rceil$. Therefore, $\max_{j=1,2,3}\{|\bar{\mathcal{C}}^j|\} < \max_{j=1,2,3}\{|\mathcal{C}^j|\}$, which contradicts the assumption that $i^\star$ was a non-leaf node that minimizes the maximum cardinality of its edge tripartition sets. $\square$

Note that the bound given in this lemma is tight for the case in which $|E| \geq 4$ and exactly two non-leaf nodes are connected to two leaf nodes.

We thus consider the following three rules that guide our search procedure. First, based on Lemma 2, we consider only tripartitions such that the maximum cardinality of any edge partition does not exceed the limit $\lceil (|E| - 1)/2 \rceil$. Second, indexing the edges in E as $e_1, \ldots, e_{|E|}$, we disallow symmetric tripartitions by insisting that the lowest-indexed edge in E is always assigned to $\mathcal{C}^1$, and that the lowest-indexed edge assigned to $\mathcal{C}^2$ is less than the lowest-indexed edge assigned to $\mathcal{C}^3$. Finally, we incorporate a result of Fomin et al. [7] that states that there always exists an optimal connected branch decomposition for 2-edge-connected graphs. Hence, we require that the subgraphs induced by $\mathcal{C}^j$ for each $j = 1, 2, 3$ are connected in an optimal solution.

Our search procedure consists of three primary phases, plus an initialization phase. The initialization phase obtains upper and lower bounds UB and LB on the graph's branchwidth as described in Section 3.2, and stops if the bounds are equal. Else, a candidate list of solutions $\mathcal{S}$ is created and initialized to be empty. The algorithm will proceed by assigning

edges to some components, and by prohibiting the assignment of edges to some components. We initialize the algorithm with no such assignments (or prohibited assignments) in place.

The first phase begins by assigning e_1 to $\mathcal{C}^1$, and then examines an edge e that is incident to some node occupied by $\mathcal{C}^1$ and has not been prohibited from being assigned to $\mathcal{C}^1$ at a previous step. The algorithm assigns e to $\mathcal{C}^1$ and recursively calls the first phase algorithm, and then prohibits e from being assigned to $\mathcal{C}^1$ followed by another recursive call to the first phase. These recursive calls continue until $\lceil(|E|-1)/2\rceil$ edges are assigned to $\mathcal{C}^1$, or until all edges incident to any node occupied by $\mathcal{C}^1$ have been marked as prohibited. At this point, if the graph G^P induced by edges that are not assigned to $\mathcal{C}^1$ contains three or more components, then any tripartition that includes edges in $\mathcal{C}^1$ will not be connected, and we cease searching tripartitions that include $\mathcal{C}^1$. If G^P has two components, then $\mathcal{C}^2$ is assigned the edges of one component, and $\mathcal{C}^3$ is assigned the edges of another component, and the third phase of the algorithm is invoked (as described below). Otherwise, G^P has one component, and the algorithm proceeds to phase two.

Phase two is analogous to phase one for building $\mathcal{C}^2$ and $\mathcal{C}^3$, having fixed edges in $\mathcal{C}^1$. In this second phase, we assign to $\mathcal{C}^2$ the lowest-indexed edge from among those not assigned to $\mathcal{C}^1$. Then, we proceed in a similar fashion as the first phase: For each unassigned edge e that is incident to a node in $\mathcal{C}^2$, we try assigning e to $\mathcal{C}^2$, and then try assigning e to $\mathcal{C}^3$, recursively calling the phase two algorithm each time. We stop the second phase either when:

(a) all edges have been assigned to a component,

(b) $|\mathcal{C}^2|$ (or $|\mathcal{C}^3|$) equals $\lceil(|E|-1)/2\rceil$, in which case all remaining edges are assigned to $\mathcal{C}^3$ (or $\mathcal{C}^2$), or

(c) there are no more edges incident to a node occupied by $\mathcal{C}^2$, in which case all remaining edges are assigned to $\mathcal{C}^3$.

At this point, a tripartition of edges is sent to phase three, which computes bounds

described in Section 4.1. (Note that the bound-computation steps for $\mathcal{C}^j$ can be skipped if $|\mathcal{C}^j| \leq LB$, since the lower bounds and upper bounds obtained over any such subgraph cannot exceed LB.) However, while the graphs induced by $\mathcal{C}^1$ and $\mathcal{C}^2$ were explicitly guaranteed to be connected, $\mathcal{C}^3$ may not induce a connected graph. The phase three algorithm first checks the connectivity of the graph induced by $\mathcal{C}^3$, and if this graph is connected, checks the upper and lower bounds given in Section 4.1. If an upper bound is obtained that is strictly better than UB, then UB is updated to this new value, and its corresponding solution is stored as the incumbent. Furthermore, all solutions in $\mathcal{S}$ that have a lower bound at least as large as this new upper bound are deleted. Else, if the lower bound given by the procedures in Section 4.1 is not less than the current value of UB, then the tripartition is discarded. Otherwise, the tripartition is stored in the candidate list $\mathcal{S}$ along with its lower bound. At the end of the algorithm, if $\mathcal{S}$ is empty, then the incumbent solution is optimal. Else, we take LB to be the minimum lower bound from among all solutions in $\mathcal{S}$.

One important consideration in reducing the computational effort required by our algorithm is in *edge fixing*. In phases one and two of our algorithm, we update the current order of edges ξ^j, $j = 1, 2, 3$, that are incident to the primary node. In phase one, node $i \in V$ contributes to the order of ξ^1 if i is incident to some edge in $\mathcal{C}^1$, and is also incident to some edge that is prohibited from being assigned to $\mathcal{C}^1$. Suppose that at some point during the first phase, we determine that the order of ξ^1 equals $UB-1$. Since we are not interested in finding any solution whose width is not less than UB, then for every unassigned edge e, we make the following checks.

1. Suppose that e is incident to some node $i \in V$ that is occupied by component 1, such that i is not in the separation induced by deleting ξ^i (i.e., i does not contribute to the order of ξ^i). Then edge e must assigned to component 1, or else assigning e to another component would make the order of ξ^1 equal to UB due to the inclusion of i in ξ^1.

2. Suppose that e is incident to some node $i \in V$ that is not occupied by component 1, but will be occupied by either components 2 or 3. (Nodes that will be occupied

by components 2 or 3 are recorded when edges are prohibited from being assigned to component 1.) Then edge e is prohibited from being assigned to component 1, or else assigning e to component 1 would make the order of ξ^1 equal to UB due to the inclusion of i in ξ^1.

3. Suppose that both conditions hold, such that one incident node obeys the first case, and the other incident node obeys the second case. Then the current assignment of edges to partitions cannot result in a new incumbent branch decomposition, and this search can be pruned.

A similar check is made in the second phase, but is done for both components 2 and 3. Note that these checks can be made for every unassigned edge, and if any assignments are made, the edges can be searched again given the updated set of node occupancies in each component. However, our computational experience indicates that executing one such edge-fixing pass is the most effective implementation.

Finally, recall that upon termination of the algorithm, we may still have a positive optimality gap, since candidate tripartitions in $\mathcal{S}$ may lead to solutions having a smaller width than UB. To eliminate this optimality gap, we must determine the minimum width of each tree that contains a tripartition in $\mathcal{S}$. Accordingly, we can recursively call the implicit enumeration algorithm individually on each of the three partitions given by an element of $\mathcal{S}$. To execute this approach for component $\mathcal{C}^j$ of a tripartition in $\mathcal{S}$, we would create a new instance given by the edges in $\mathcal{C}^j$, along with a dummy set of edges which connects nodes that contribute to the order of ξ^j. We must now establish a tripartition $\widehat{\mathcal{C}}^1$, $\widehat{\mathcal{C}}^2$, and $\widehat{\mathcal{C}}^3$ of these edges. The set of dummy edges would be fixed to $\widehat{\mathcal{C}}^1$, while the edges of $\mathcal{C}^j$ would be assigned to $\widehat{\mathcal{C}}^2$ and $\widehat{\mathcal{C}}^3$ starting with phase two of the implicit enumeration algorithm. After executing this procedure for each of $\mathcal{C}^1$, $\mathcal{C}^2$, and $\mathcal{C}^3$, the maximum upper bound (lower bound) obtained over these three procedures is the upper bound (lower bound) on the width of an optimal branch decomposition containing the tripartition.

5 Computational Results

In this section, we test the ability of the foregoing procedures to determine optimal branchwidth values on a test bed of graph instances.

Our test instances are the biconnected graphs among compiler graphs, some well known graphs such as K_6, the Petersen graph, and the webs $W_{6,2}$, $W_{8,3}$, $W_{10,4}$, and $W_{12,5}$. Compiler graphs are the test instances of control-flow graphs from actual C compilations, provided by Keith Cooper at Rice University. In addition, graphs we have named as "g#", "g#.cimi", and "t#", or "class#" and "bazaraa#", are graph instances from maximum planar subgraphs problems provided by Mauricio Resende at AT&T Labs and Brett Peters at Texas A&M, respectively. A web $W_{n,i}$ is a graph of n labeled vertices such that the edge set consists of the edges of the form $\{(j,(j+i)\text{Mod } n),\ldots,(j,(j-i)\text{Mod } n)\}$. We also demonstrate run times of our algorithms on the graphs K_5, M_6, M_8, and Q_3, and on a set of graphs we generate by deleting one edge from these graphs. (Since the latter four graphs are *edge transitive*, deleting one edge is equivalent to deleting any other edge.) Finally, we consider the performance of our algorithms on randomly generated biconnected graphs we have named as "rg$n$_m" where n and m denote the number of nodes and edges in the instance, respectively.

In our computations, several instances are solved by finding that the initial upper bound equals the initial lower bound that we compute in Section 3.2. We display these instances in Table 1, along the numbers of edges and nodes for each instance, the optimal branchwidth (labeled as β), and the CPU seconds (Time) required to compute the initial bounds for each of these instances. For the remaining instances, the initial lower bound is less than the initial upper bound, and our proposed optimization technique is invoked to find the branchwidth. Table 2 displays those instances with the same columns as in Table 1, along with the initial upper (UB) and lower (LB) bounds obtained.

The implicit enumeration presented in Section 4 is effective in solving many of these instances. Table 3 gives the CPU time (Opt Time) required to solve each of these instances by the implicit enumeration procedure given in Section 4. In these experiments, candidate

Graph	$(\lvert V\rvert, \lvert E\rvert)$	β	Time
dens	(6,7)	2	0
$W_{6,2}$	(6,9)	3	0
K_5^-	(5,9)	3	0
K_5	(5,10)	4	0
class6	(6,10)	3	0
cosqb	(8,10)	2	0
M_8^-	(8,11)	3	0
Q_3^-	(8,11)	3	0
M_6	(6,12)	4	0
coeray	(11,14)	2	0
sudtbl	(11,14)	3	0
K_6	(6,15)	4	0
Petersen	(10,15)	4	0
arret	(12,15)	2	0
supp	(12,15)	2	0
class7	(7,16)	4	0
class8	(8,18)	4	0
$W_{12,5}$	(12,18)	4	0
laspow	(14,18)	3	0
tpart	(14,18)	2	0
inter	(16,21)	2	0
t1	(11,22)	4	0
nprio	(17,22)	3	0
saturr	(17,23)	2	0
setinj	(18,23)	2	0
g3	(10,24)	3	0
genb	(18,24)	2	0

Graph	$(\lvert V\rvert, \lvert E\rvert)$	β	Time
g4	(10,25)	4	0
laser	(19,25)	2	0
g5	(10,26)	4	0
g6	(10,27)	4	0
drigl	(21,29)	3	0
si	(23,31)	3	0
seval	(24,31)	3	0
bcndb	(25,34)	2	0
dyeh	(27,35)	3	0
linj	(27,35)	3	0
recre	(27,35)	3	0
injchk	(33,44)	2	0
bilan	(37,49)	3	0.01
tcomp	(38,50)	3	0.01
colbur	(40,54)	3	0.01
sortie	(44,62)	3	0.01
cardeb	(48,64)	3	0.02
prophy	(48,65)	2	0.02
yeh	(50,68)	3	0.02
g8	(25,69)	3	0
g9	(25,70)	4	0
verify	(55,73)	2	0.02
bcnd	(55,77)	2	0.03
tele56	(56,85)	3	0.02
heat	(69,95)	3	0.05
ddeflu	(92,124)	2	0.09
pastem	(158,214)	3	0.45

Table 1: Graph instances for which the initial upper and lower bounds prove optimality

Graph	$(\|V\|, \|E\|)$	β	UB	LB	Time
M_6^-	(6,11)	3	4	3	0
M_8	(8,12)	4	4	3	0
Q_3	(8,12)	4	4	3	0
$W_{8,3}$	(8,12)	4	4	3	0
$W_{10,4}$	(10,15)	4	4	3	0
g1.cimi	(10,21)	4	5	4	0
g1	(10,22)	5	5	4	0
g4.cimi	(10,22)	5	5	4	0
rg11_23	(11,23)	5	5	4	0
t2	(17,25)	5	5	4	0
starfish	(20,30)	5	6	4	0
rg19_33	(19,31)	5	5	4	0
g7	(10,34)	6	6	5	0
class12ex	(12,37)	6	6	5	0
rg14_40	(14,40)	6	6	5	0
class12	(12,45)	7	8	6	0
bazaraa13	(13,57)	8	9	6	0
bazaraa12	(12,59)	8	8	7	0
g6.cimi	(43,63)	5	5	4	0.01
rg28_64	(28,64)	7	7	6	0.01
class15	(15,65)	NA	9	6	0

NA: Optimal branchwidth is not known.

Table 2: Graph instances for which the initial upper and lower bounds are not equal

tripartitions in $\mathcal{S}$ are not examined further. Instead, we focus on the problem of enumerating and evaluating candidate tripartitions, and report the best upper and lower bounds (given by the Range column, which is a singleton if optimality is proven). Obviously, the procedure consumes more CPU time on graphs that have more edges, but a secondary consideration also regards the graph density. For graphs having a high density, the initial gaps tend to be greater, optimal branchwidth values tend to be higher, and less fathoming can be performed due to restrictions on component connectivity. The difficulties imposed by graph density are exemplified by observing the performance of the implicit enumeration algorithm on g6.cimi, rg28_64, and class15. The 63-edge graph g6.cimi is very sparse and requires just over three seconds to solve, while the slightly more dense 64-edge instance rg28_64 can be solved in just over four minutes. However, the 65-edge instance class15 is considerably more dense

than either g6.cimi or rg28_64, and cannot be solved within three hours with the implicit enumeration algorithm.

Graph	$(\lvert V\rvert, \lvert E\rvert)$	Range	Opt Time
M_6^-	(6,11)	3	0
M_8	(8,12)	4	0
Q_3	(8,12)	4	0
$W_{8,3}$	(8,12)	4	0
$W_{10,4}$	(10,15)	4	0
g1.cimi	(10,21)	[4,5]	0.12
g1	(10,22)	5	0.14
g4.cimi	(10,22)	5	0.04
rg11_23	(11,23)	5	0.04
t2	(17,25)	5	0.02
starfish	(20,30)	5	0.63
rg19_33	(19,31)	5	0.06
g7	(10,34)	[5,6]	6.19
class12ex	(12,37)	[5,6]	11.46
rg14_40	(14,40)	6	11.42
class12	(12,45)	[8,6]	$>$ 10800
bazaraa13	(13,57)	[9,6]	$>$ 10800
bazaraa12	(12,59)	[8,7]	$>$ 10800
g6.cimi	(43,63)	5	1.68
rg28_64	(28,64)	7	263.38
class15	(15,65)	[9,6]	$>$ 10800

Table 3: Implicit enumeration algorithm results

For instances g1.cimi, g7, and class12ex, the implicit enumeration program terminated within a few seconds, but did not eliminate the optimality gap. For these instances, we must evaluate the candidate tripartitions in the set $\mathcal{S}$ at the end of the algorithm. To evaluate these candidates, we implemented the recursive implicit enumeration procedure, as described at the end of Section 4.2. For g1.cimi, there were 65 elements in $\mathcal{S}$, and our algorithm found a branch decomposition of width 4, matching the lower bound, on the fourth candidate. The procedure required less than 0.01 additional CPU seconds for this instance. For g7, there were 414 candidates in $\mathcal{S}$, and all were proven to have a width of at least 6. The procedure required 0.67 additional CPU seconds to examine these candidates. Finally, for

class12ex, our algorithm required 0.18 additional CPU seconds to prove that none of the 85 candidate tripartitions in $\mathcal{S}$ improve the initial upper bound, and hence that the branchwidth of class12ex is 6.

Next, note that our algorithm does not narrow the initial optimality gap on any of the four unsolved instances within three hours. We ran an experiment in which class12, bazaraa13, and class15 were fed upper bounds of 7 to see whether the lower bounds on those instances (which all equal 6) could be improved to 7. Given this bound, we were able to determine in 4033.49 seconds that no solution with branchwidth 6 exists for class12, thus indicating that its optimal branchwidth is either 7 or 8. (Note that the optimality gap for bazaraa12 is already equal to one, and so this experiment does not apply for that instance.)

We also ran an experiment to assess the computational effectiveness of some of our search strategies on the five solvable instances in Table 3 that consume more than one CPU second. We tried omitting the following individual search and fathoming strategies: Replacing the upper bound on the maximum component size with the trivial bound of $|E|-2$ ("No max"), removing edge fixing checks ("No edge"), and skipping the check on the number of connected components in the graph G^P of edges not assigned to $\mathcal{C}^1$ in phase one ("No connectivity"). For this experiment, we again skip the further investigation of tripartitions in $\mathcal{S}$. The results of these tests compared to the strategy with each of these strategies enabled ("Full") is given in Table 4, in which computational time was limited to 1800 CPU seconds. These results indicate that limiting the maximum component cardinality and performing edge fixing checks are critical to reducing computational time. Checking for connectivity of G^P after the first phase provides a more modest computational benefit on these instances, but is still significant enough to warrant the inclusion of the strategy in our overall algorithm.

Finally, recall that Hicks [12] provided an algorithm to compute optimal branch decompositions, which is to our knowledge the only other practical algorithm capable of optimally computing a graph's branchwidth. The idea behind this algorithm is to find a branch decomposition with order k (for some positive integer k) along with a *tangle basis*

Graph	No max	No edge	No connectivity	Full
g6.cimi	89.14	13.11	2.44	1.68
g7	> 1800	64.56	8.99	6.19
rg14_40	> 1800	143.38	11.51	11.42
class12ex	1620.66	78.36	11.59	11.46
rg28_64	> 1800	> 1800	319.49	263.38

Table 4: CPU seconds consumed by variations of the implicit enumeration algorithm

of order k, which then proves that the graph's branchwidth equals k. In this last set of experiments, we refer to our algorithm as **implicit** and to the algorithm of Hicks [12] as **tangle**. We noted that the **tangle** and **implicit** algorithms performed similarly on most of the above instances, with six exceptions as reported in Table 5. Here, we see that **tangle** is able to solve one instance that **implicit** could not solve within the time limit, and is significantly faster on three other instances. On the other hand, **implicit** is more efficient on the starfish and rg28_64 instances. The underlying reason for these differences seems to lie in the density of these instances. As noted above, the fathoming strategies in the **implicit** algorithm make it far more effective on sparse graphs than it is on dense graphs. We thus hypothesize that **implicit** is generally preferable to **tangle** for sparse graphs.

Graph	$(\|V\|, \|E\|)$	**implicit**	**tangle**
starfish	(20,30)	0.63	2.28
g7	(10,34)	6.19	0.47
class12ex	(12,37)	11.46	2.87
rg14_40	(14,40)	11.42	6.93
class12	(12,45)	> 10800	99.13
g28_64	(28,64)	263.38	1996.21

Table 5: CPU seconds required by **implicit** and **tangle** on the test instances

To test this hypothesis, we generated three new sets of graphs. The graph instance labeled "sn_m_i" is the ith instance of a randomly generated graph having n nodes and m edges. We generated ten instances for (n, m) = (20,40), (25,50), and (30,60). The results of these experiments are given in Tables 6, 7, and 8. In each table we provide the CPU time (in seconds) required for the two algorithms to solve each instance, again with a three-

hour (10800-second) time limit. These tables demonstrate that the **implicit** algorithm is significantly more effective than the **tangle** algorithm on the given sparse instances. In particular, Table 8 shows that **implicit** solves five 30-node instances that **tangle** cannot within the three-hour time limit, and is substantially faster than **tangle** on two other 30-node instances in which both algorithms find optimal solutions. By comparison, **tangle** solves one 30-node instance that **implicit** cannot within the time limit, and is faster on one in which both algorithms find optimal solutions. The fact that **implicit** is effective on these sparse instances is of particular interest because most algorithms that benefit from branch decompositions are effective only for graphs that have a small branchwidth, and are thus more likely to be sparse.

Graph	**implicit**	**tangle**
s20_40_1	7.03	7.24
s20_40_2	0.70	0.36
s20_40_3	0.58	9.60
s20_40_4	0.27	14.76
s20_40_5	8.73	12.11
s20_40_6	0.36	7.03
s20_40_7	11.07	14.81
s20_40_8	20.41	10.17
s20_40_9	4.22	8.83
s20_40_10	0.30	0.55

Table 6: Computational comparison of **implicit** and **tangle** on 20-node, 40-edge instances

6 Conclusions

We consider the problem of optimizing the branchwidth of a graph. We examined the use of strong upper and lower bounding techniques within an implicit enumeration algorithm, which is capable of solving most of the graph instances considered in this paper to optimality. Our algorithm outperforms the approach given in [12] for sparse graphs, although it is markedly less efficient for dense graphs.

Graph	**implicit**	**tangle**
s25_50_1	20.33	544.75
s25_50_2	21.67	36.94
s25_50_3	22.65	48.71
s25_50_4	65.82	419.99
s25_50_5	20.85	38.61
s25_50_6	94.21	899.18
s25_50_7	0.72	15.53
s25_50_8	56.16	66.40
s25_50_9	2.99	0.81
s25_50_10	22.75	1588.03

Table 7: Computational comparison of **implicit** and **tangle** on 25-node, 50-edge instances

Graph	**implicit**	**tangle**
s30_60_1	1402.12	>10800
s30_60_2	2180.34	4901.10
s30_60_3	4339.17	>10800
s30_60_4	6666.56	>10800
s30_60_5	4951.58	>10800
s30_60_6	3930.79	>10800
s30_60_7	1445.01	2928.00
s30_60_8	>10800	4805.03
s30_60_9	>10800	>10800
s30_60_10	262.53	88.48

Table 8: Computational comparison of **implicit** and **tangle** on 30-node, 60-edge instances

Our investigation in this paper underscores the difficulty of determining a graph's branchwidth. For larger graphs (e.g., those having more than 100 edges) that are not sparse enough for our bounding techniques to immediately optimize the branchwidth problem, there is still a need to explore low-complexity algorithms for obtaining strong upper bounds on a graph's branchwidth. We recommend the study of exact strategies on large-scale graphs and exact strategies for general branchwidth for future research.

References

[1] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[2] Z. Bian, Q. Gu, M. Marzban, H. Tamaki, and Y. Yoshitake. Empirical study on branchwidth and branch decomposition of planar graphs. In *Proceedings of the 2008 SIAM Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 152–165. SIAM, 2008.

[3] H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. *Journal of Graph Algorithms and Applications*, 10(1):5–49, 2006.

[4] W. J. Cook and P. D. Seymour. An algorithm for the ring-router problem. Technical report, Bellcore, 1994.

[5] W. J. Cook and P. D. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[6] B. Courcelle. The monadic second-order logic of graphs I: Recognizable set of finite graphs. *Information and Computation*, 85:12–75, 1990.

[7] F. V. Fomin, P. Fraigniaud, and D. M. Thilikos. The price of connectedness in expansions. Technical Report 273, Department of Informatics, University of Bergen, Bergen, Norway, May 2004.

[8] F. V. Fomin and D. M. Thilikos. A simple and fast approach for solving problems on planar graphs. *Lecture Notes in Computer Science*, 2996:56–67, 2004.

[9] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal of Applied Mathematics*, 34:477–495, 1978.

[10] I. V. Hicks. Branchwidth heuristics. *Congressus Numerantium*, 159:31–50, 2002.

[11] I. V. Hicks. Branch decompositions and minor containment. *Networks*, 43(1):1–9, 2004.

[12] I. V. Hicks. Graphs, branchwidth, and tangles! Oh my! *Networks*, 45(2):55–60, 2005.

[13] I. V. Hicks. Planar branch decompositions II: The cycle method. *INFORMS Journal on Computing*, 17(4):413–421, 2005.

[14] B. Monien. The bandwidth minimization problem for caterpillars with hair length 3 is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 7:505–512, 1986.

[15] N. Robertson and P. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52:153–190, 1991.

[16] N. Robertson and P. D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

[17] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.