Tour Merging via Branch-decomposition

William Cook*
Industrial and Systems Engineering
Georgia Institute of Technology

Paul Seymour†
Applied and Computational Mathematics
Princeton University

December 18, 2002

Abstract

Robertson and Seymour introduced branch-width as a new connectivity invariant of graphs in their proof of the Wagner conjecture. Decompositions based on this invariant provide a natural framework for implementing dynamic-programming algorithms to solve graph optimization problems. We describe a heuristic method for finding branch-decompositions; the method is based on the eigenvector technique for finding graph separators. We use this as a tool to obtain high-quality tours for the traveling salesman problem by merging collections of tours produced by standard traveling salesman heuristics.

1 Tour Merging

For many discrete optimization problems, heuristic algorithms are both a practical technique for obtaining acceptable solutions, as well as a key ingredient in exact solution methods such as branch-and-bound. In the later case, in particular, it is important to have solutions that are of near-optimal quality, in order to guide the exact search.

A common approach taken to improve the quality of solutions provided by heuristics is to apply them repeatedly, using pseudo-random numbers or other methods to alter the behavior of the algorithms on each run, and selecting the best among the solutions that are produced. A valid criticism of such multiple-run methods is that we may be discarding a great deal of valuable information; a challenge is to utilize the combined content of the solutions to produce an overall result that is superior to any single member of the collection.

In this paper, we consider the case of the *traveling salesman problem* (TSP), where the travel costs are symmetric (the cost to travel between city x and city y does not depend on the direction we are traveling). A TSP instance can be specified as a edge-weighted complete-graph, where nodes represent cities and edge weights give the cost of travel between pairs of nodes. A solution to the TSP is a *tour* (or Hamiltonian circuit) in the graph, that is, a circuit through the entire set of nodes.

*Supported by ONR Grant N0001 4-01-1-0608
†Supported by ONR Grant N0001 4-01-1-0608

For the TSP, multiple-run heuristics have long been the method of choice when very high quality solutions are required. In the classic work of Lin and Kernighan (1973), pseudo-random starting tours are used to permit repeated application of their local-search procedure. Besides just taking the best of the tours that are produced, Lin and Kernighan propose to use the intersection of the edge sets of the tours as a means to guide further runs of their algorithm. Their idea is to modify the basic procedure so that it will not delete any edge that has appeared in each of the tours that has been found up to that point (they start this restricted search after a small number of tours have been found [between two and five in their tests]). Variations of this idea have been explored recently by Helsgaun (2000), Schilham (2001), and Tamaki (2002).

A general technique to utilize the collective information in a set of tours $\mathcal{T}$ is to assemble a graph consisting of the union of the edge sets of the tours, and look for a tour in this restricted graph G; an optimal tour in G would give the best combination of tour-segments from $\mathcal{T}$. This type of *tour-merging* can improve the tour-quality of even the best available heuristic algorithms. For example, the graph in Figure 1 consists of the union of ten tours for the 5,934-city TSP instance rl5934 (from the TSPLIB test instances, collected by Reinelt 1991); the ten tours were obtained by running the `LKH` code of Helsgaun (2000), a powerful variant of the Lin-Kernighan heuristic. While the cost of the best of the ten LKH tours is only 0.006% above the optimal value for rl5934, the best tour in the union is in fact an optimal solution for this instance.

The difficulty with tour-merging is, of course, solving the TSP over the restricted graph G. In general, optimizing over G can be as hard as the original TSP instance. The main purpose of this paper is to propose a "branch-width" algorithm as a practical means for merging collections of high-quality tours.

Branch-width (defined below in Section 2) is a graph invariant introduced by Robertson and Seymour (1991); it is closely related to the more widely studied notion of tree-width. Graph decompositions based on branch-width (or on tree-width) provide a natural framework for implementing dynamic-programming algorithms to solve graph optimization problems; theoretical studies of this approach can be found in Bern et al. (1987), Courcelle (1990), Arnborg et al. (1991), Borie et al. (1992), and elsewhere. Despite a large body of theoretical work, little in the way of practical computation has been reported in the literature. We use the TSP as a means to demonstrate that branch-width can be a practical tool in large-scale optimization, reporting computational results on tour merging for graphs having well over 10,000 nodes.

Returning to the rl5934 example, the average cost of the ten tours is 0.089% above optimum, and `LKH` required an average of 1,973 seconds on a Compaq Alphaserver ES40 (500 MHz EV6 Alpha processor) to produce each tour. The union of the ten tours has 6,296 edges, and the branch-width algorithm required 2.10 seconds to determine the best solution in the restricted graph (and thus produce an optimal solution for the instance). This example is typical of the performance of the merging algorithm on `LKH`-generated tours, where the consistent high quality of the LKH heuristic produces a very sparse union of edges. In general, the branch-width algorithm is sensitive to the density of the graph G, although the practical performance exceeds that which would be predicted by a worst-case analysis of the underlying dynamic-programming method.

The paper is organized as follows. In Section 2 we define branch-decompositions and

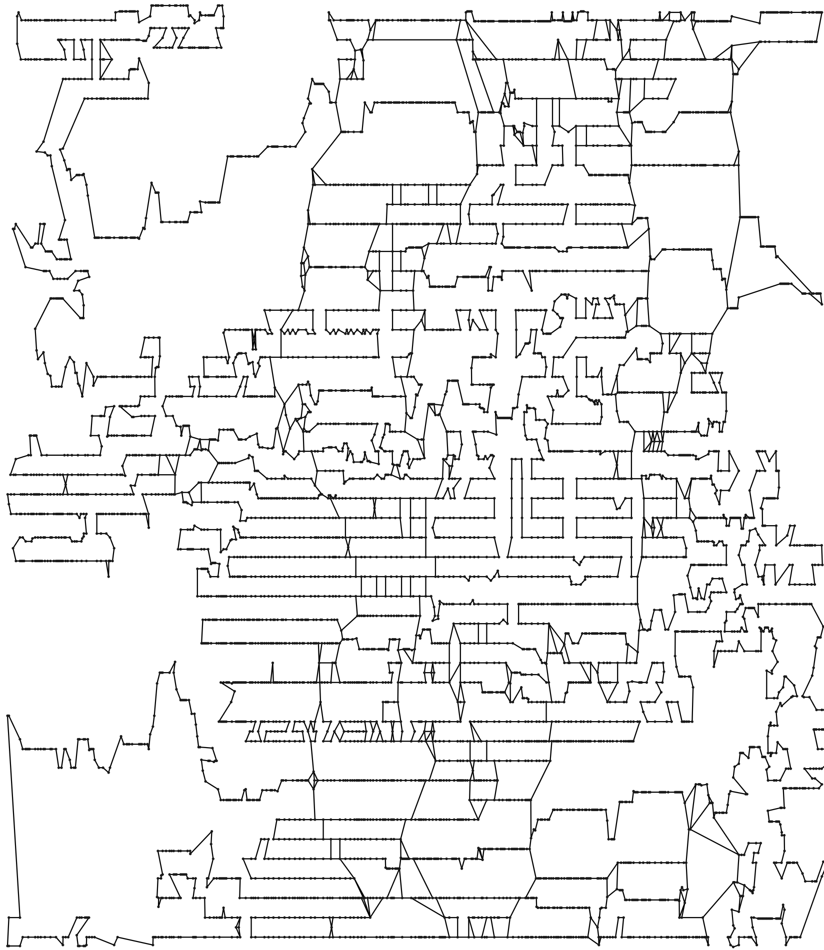

Figure 1: Union of ten LKH Tours for rl5934

the branch-width of a graph, and in Sections 3 and 4 we describe our heuristic for finding branch-decompositions. Computational results for the branch-decomposition heuristic are given in Section 5. In Section 6 we describe the dynamic-programming algorithm for the TSP, and in Section 7 we present computational results using this algorithm in a tour-merging heuristic. Some concluding remarks are given in Section 8.

2 Branch Decompositions

Let G be a graph, with node set $V(G)$ and edge set $E(G)$. Any partition of $E(G)$ into two sets (A, B) is called a *separation* of G. Let us assume that each node in G has degree at least one (the degree of a node is the number of edges it meets). Then any separation (A, B) partitions the nodes of G into three classes: those that meet only edges in A, those that meet only edges in B, and those that meet both edges in A and edges in B. We call these three classes the $left(A, B)$ nodes, the $right(A, B)$ nodes, and the $middle(A, B)$ nodes, respectively. Note that no node in left(A, B) has a neighbor in right(A, B). Consequently, middle(A, B) does indeed "separate" left(A, B) from right(A, B). The cardinality of middle(A, B) is called the *order* of the separation.

Let T be a tree (not necessarily a subtree of G) having $|E(G)|$ leaves and in which every non-leaf node has degree at least three. Associate with each leaf v of T one of the edges of G, say $\nu(v)$, in such a way that every edge of G is associated with a distinct leaf. The tree T, together with the function ν, is called a *partial branch-decomposition* of G. If each non-leaf node of T has degree exactly three, then (T, ν) is a *branch-decomposition.*

Let (T, ν) be a partial branch-decomposition of G. If e is an edge of T, then the graph obtained from T by deleting e has exactly two connected components, and consequently the set of leaves of T is partitioned into two subsets. Since each leaf of T is associated with an edge of G, there corresponds a separation (A_e, B_e) of G. The *width* of (T, ν) is the maximum of the order of (A_e, B_e) over all edges e of T. The smallest k such that G has a branch-decomposition of width k is called the *branch-width* of G.

Branch-width was introduced in Robertson and Seymour (1991) as part of their graph minors project and it played a fundamental role in their proof of the Wagner conjecture. Our interest here is not in the connection of branch-width to graph minors, but rather in the algorithmic possibilities it opens up via dynamic programming. To motivate the definition of branch-width in this context, we will describe a possible algorithm for the TSP.

Suppose our graph G has a separation (A, B) of small order, say three. Then there are only a small number of different ways in which a tour in G can hit middle(A, B). See Figure 2. For a given pattern of hitting middle(A, B), we can independently solve the problems of finding collections of paths in A and B that realize the pattern, since the only thing one side needs to know about the other side is the way it behaves on middle(A, B). This implies that we can reduce the problem of finding a minimum-cost tour to that of solving a list of subproblems in A and B. Of course, the question now is: how do we solve the subproblems? One approach is to try to repeat the separation process on each of the two pieces. Suppose we can find two separations of G that together split A into two smaller pieces, as indicated in Figure 3. Using these separations, we could solve problems in the smaller pieces in order to create the solution for the larger piece. Now to solve these smaller problems, we could split the pieces again, as indicated in Figure 4. These steps can be

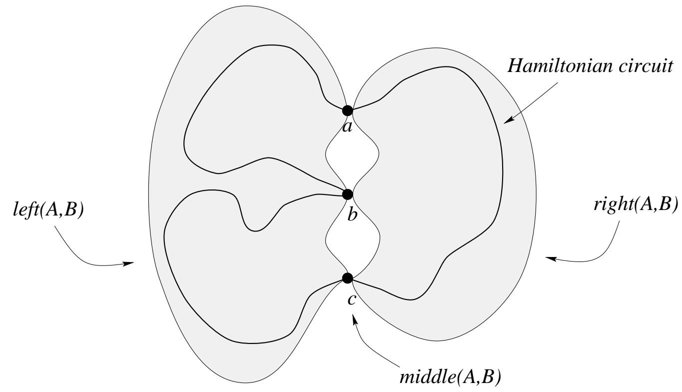

Figure 2: A Separation of Order Three

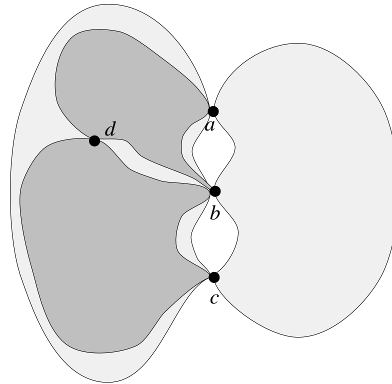

Figure 3: A Second Pair of Separations

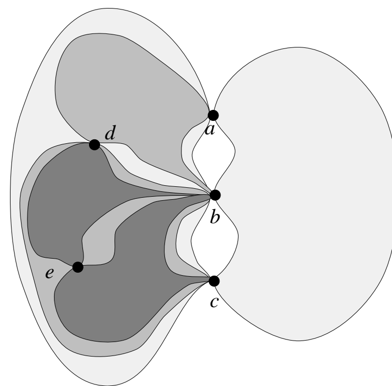

Figure 4: A Third Pair of Separations

summarized by means of a “separation tree” as indicated in Figure 5. In this figure, we

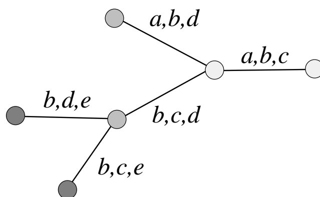

Figure 5: The Separation Tree

have labeled the edges with the middle of the corresponding separation. If we continued the splitting process until each piece consists of a single edge, then the separation tree would represent the tree T in a branch-decomposition of the graph, where ν is given by the edges in the final pieces of the decomposition.

This description makes it clear how we use branch-decompositions in an optimization procedure: starting at the leafs of the tree, we work our way through the nodes, gluing together solutions to the pieces as we go along. This is a dynamic-programming procedure. To carry it out, we need to be able to encode the hitting patterns and to be able to combine two hitting patterns into a pattern for the middle of the next separation.

Branch-width is closely related to another graph decomposition scheme known as “tree-width”. Suppose we have a graph G and a tree T. Associate with each node v of T a subset $\omega(v)$ of the nodes of G. The pair (T, ω) is called a *tree-decomposition* of G if

(*i*) $\bigcup\{\omega(v) : v \in V(T)\} = V(G)$;

(*ii*) for each edge $(v, w) \in E(G)$, there exists a node $t \in V(T)$ such that $\{v, w\} \subseteq \omega(t)$;

(*iii*) for all $u, v, w \in V(T)$, if v is on the path from u to w in T, then $\omega(u) \cap \omega(w) \subseteq \omega(v)$.

The *width* of a tree-decomposition is the maximum of $|\omega(v)| - 1$, taken over all $v \in V(T)$. The smallest k such that there exists some tree-decomposition of G with width k is called the *tree-width* of G. (A number of equivalent characterizations of tree-width can be found in Bodlaender 1998.)

As we mentioned, the concepts of tree-width and branch-width are closely related. Indeed, Robertson and Seymour (1991) have shown that every graph of branch-width at most k has tree-width at most $3k/2$, and every graph of tree-width at most k has branch-width at most $k + 1$. So having small branch-width is equivalent to having small tree-width.

Tree-width is studied in a great many papers; surveys of the literature can be found in Bodlaender (1993, 1998) and computational studies can be found in Koster et al. (1999) and in Koster et al. (2001). For many discrete optimization problems that deal with edge sets of graphs, however, branch-width is a more natural framework for carrying out dynamic programming.

3 The Eigenvector Heuristic

There are very many problems that are NP-hard on general graphs and that can (at least in theory) be solved quickly on graphs that have branch-decompositions of small width

k. For any constant k, it is known that the dynamic-programming algorithms for these problems have running time proportional to the size of the input graph, where the constant of proportionality depends on k. Consequently, for fixed k, dynamic programming provides linear-time algorithms for these problems. It must be noted, however, that the constant of proportionality usually grows very quickly with k, and therefore these algorithms cannot be practically implemented to run for even moderate values of k. It is therefore important that there should be very little error in our calculation of the branch-width of a graph. For instance, in Robertson and Seymour (1991) there is a fast algorithm to estimate branch-width, within an error factor of three (that is, it would decide either that a graph has branch-width at least 10 or it would find a branch-decomposition of width at most 30), but this method is of little value in a practical implementation. This approximation algorithm was improved in Matoušek and Thomas (1991), Lagergren (1990), Reed (1992), Bodlaender (1996), and Bodlaender and Kloks (1996), but all of these methods are either impractical or yield too large an error. There is no known method that can be practically implemented, that runs quickly, and that will decide if a graph has branch-width at most 8.

We describe a heuristic method for finding decompositions; one that attempts to construct a branch-decomposition of the input graph which is close to optimal, but for which good performance guarantees cannot be made. We will assume that the input graph is simple and 2-connected. The method proceeds as follows.

Take a tree T_1 which is a "star" (that is, one of its nodes is adjacent to all the others) with $|E(G)|$ leaves, and with each leaf v associate some edge $\nu(v)$ of G in a one-to-one way. Then (T, ν) is a partial branch-decomposition of order at most two. It is not yet a branch-decomposition because the interior node of T_1 does not have degree exactly three. We shall repeatedly choose a node of the tree of degree at least four and "split" it into two adjacent nodes of degree at least three; when this terminates every interior node has degree three and we have a branch-decomposition. At each iteration, one new separation is introduced, and we shall choose the splitting to keep this of small order if we can.

Let us state this more precisely. At the start of the ith iteration, we have a partial branch-decomposition (T_i, ν) such that T_i has precisely i internal nodes. If every internal node has degree three we stop. Otherwise, we choose an internal node v of T_i of degree at least four. Let D be the set of edges of T_i incident with v, and partition D into two sets X, Y both of cardinality at least two. In T_i, replace the node v by two new nodes x and y, so that x is an end of each edge in X, and y is an end of each edge in Y, and there is an edge xy. See Figure 6. This forms T_{i+1}. We see that in T_{i+1}, every internal node has

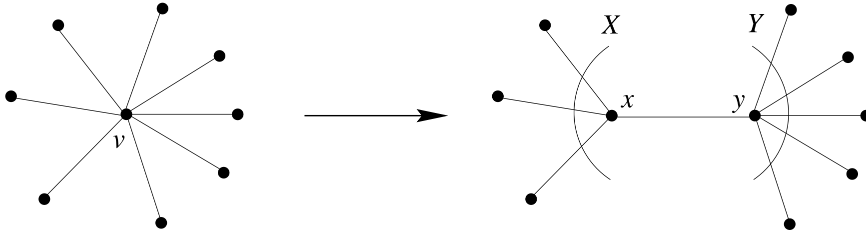

Figure 6: Splitting a Node

degree at least three, since $|X|, |Y| \geq 2$; and so (T_{i+1}, ν) is a partial branch-decomposition

of G. This completes the iteration.

Let us say that (T, ν) is *extendible* if there is some way to (repeatedly) split the nodes of T having degree greater than three, to obtain a branch-decomposition of width equal to the branch-width of G. Certainly (T_1, ν) is extendible, and on the other hand, if the process terminates with an extendible branch-decomposition then this is optimal. Thus, we need to choose X and Y at each stage so that if (T_i, ν) is extendible then so is (T_{i+1}, ν). Unfortunately, we do not know how to check directly if a given (T, ν) is extendible. We therefore need some indirect method to maintain extendibility.

The first, most obvious, method to try to preserve extendibility is the greedy one; we just choose X, Y at each step so that (T_{i+1}, ν) has small width, if we can. Every separation of G arising from (T_{i+1}, ν) also arises from (T_i, ν) except for one, the separation arising from the new edge xy. The order of this separation depends on the choice of X and Y. For each $e \in D$, let us define M_e to be the middle of the separation of G arising from e in (T_i, ν). (We recall that D is the set of edges of T_i incident with v.) Define

$$M(X, Y) = |\bigcup(M_e : e \in X) \cap \bigcup(M_e : e \in Y)|.$$

One can easily check that the new separation has order $M(X, Y)$. Therefore, we certainly want to choose X, Y at each stage so that $M(X, Y)$ is small.

Unfortunately, repeatedly choosing X, Y with $M(X, Y)$ small, without regard to other considerations, is too short-sighted, and does not work well in practice. If we are more careful in the choice of X, Y we can do better. Indeed, there is often an "obviously safe" way to split v. By "safe" we mean that if the current decomposition is extendible, then the new decomposition will be extendible as well. Our computer code uses a number of algorithms that search for obviously safe splits; we describe these algorithms in the next section.

If we cannot find an obviously safe split, then we do not in general know how to choose a partition (X, Y) such that the derived decomposition (T_{i+1}, ν) is extendible if (T_i, ν) is extendible. However, we have made the following empirical observation: if we choose (X, Y) with $M(X, Y)$ small, in such a way that $|X|, |Y| \geq |D|/3$, then often the (T_{i+1}, ν) we produce will be extendible. Conversely, we can show that if (T_i, ν) is extendible then there always is a choice of (X, Y) with $M(X, Y)$ small and with $|X|, |Y| \geq |D|/3$. The problem of finding such an (X, Y) is called the *separator problem*.

We do not know how to solve the separator problem efficiently, but we do have a good heuristic method. It is based on the eigenvector work of Alon (1986) and others, and proceeds as follows.

Let S denote $\bigcup(M_e : e \in D)$, and for each $v \in S$ let $N_v = \{e \in D : v \in M_e\}$. Construct a $|D| \times |D|$ matrix $F = (f_{ij} : i, j \in D)$ as follows. For distinct $i, j \in D$, let

$$f_{ij} = \sum\left(\frac{-1}{|N_v| - 1} : v \in S, \{i, j\} \subseteq N_v\right).$$

For each $i \in D$, let

$$f_{ii} = |\{v \in S : i \in N_v\}|.$$

Then for each i, we have $\sum(f_{ij} : j \in D) = 0$, and every eigenvalue of F is nonnegative.

We compute the eigenvector $x = (x_i : i \in D)$ of F, corresponding to the second smallest eigenvalue of F. We then order D so that the numbers x_i are in nondecreasing order, and we let A be the $\lceil |D|/3 \rceil$ first terms and we let B be the $\lceil |D|/3 \rceil$ last terms.

Given A and B, we use network flow theory to compute a partition (X, Y) of D with $A \subseteq X$ and $B \subseteq Y$ such that $M(X, Y)$ is minimum. These sets X, Y are the solution to the separator problem. They are not always the optimal answer, but in practice it seems that they are a often a good approximation.

Our overall decomposition algorithm is then the following. At any stage, if there is an obviously safe split we make it. Otherwise, we choose a node v having degree at least four and we split it using the above eigenvector method. (After the use of the eigenvector method, our search for obviously safe splits is restricted to the neighborhood of the new edge that we introduced into the tree, since this is the only portion of the tree that was altered.)

4 Safe Splits

We describe two techniques for producing safe splits during our decomposition algorithm. The first technique uses pairs of separations from the current partial branch-decomposition, and the second technique uses 2-separations and 3-separations that satisfy some simple conditions.

4.1 Pushing

Let (T_i, ν) be a partial branch-decomposition of G and let v be a node of T_i with degree at least four. As in the previous section, let D be the set of edges of T_i incident with v and for each $e \in D$, let M_e be the middle of the separation of G arising from e.

Lemma 1 *Let $e_1, e_2 \in D$ be distinct edges and suppose*

$$|(M_{e_1} \cup M_{e_2}) \cap \bigcup (M_e : e \in D \setminus \{e_1, e_2\})| \leq max(|M_{e_1}|, |M_{e_2}|) \tag{1}$$

holds. Then taking $X = \{e_1, e_2\}, Y = D \setminus X$ yields a tree T_{i+1} which is extendible if T_i is extendible.

Proof: Since the partial branch-decomposition (T_i, ν) is extendible, there is a branch-decomposition (T, ν) extending it, of width the branch-width of G. We need to show that there is such an extension so that the edges e_1, e_2 have a common end in T. Let P be the path of T with first edge e_1 and last edge e_2, with vertices $p_1, \ldots, p_n$ say. (We may assume that $n \geq 4$.) So e_1 is incident with p_1, p_2 in T, and so on. Let S_1, S_2, S_3 be the three components of $T \setminus \{e_1, e_2\}$, where $p_1 \in V(S_1)$ and $p_n \in V(S_2)$; and for $i = 1, 2, 3$ let E_i be the set of all edges $e \in E(G)$ with $\nu(e) \in V(S_i)$. So E_1, E_2, E_3 is a partition of $E(G)$. For $1 \leq i \leq 3$ let N_i be the set of vertices of G incident with an edge of E_i; so

$$|M_{e_1}| = |N_1 \cap (N_2 \cup N_3)|$$

and so on. From the given inequality, we may assume that

$$|N_1 \cap (N_2 \cup N_3)| \geq |(N_1 \cup N_2) \cap N_3|,$$

and so

$$|(N_1 \cap N_2) \setminus N_3| \geq |(N_2 \cap N_3) \setminus N_1|.$$

Now let us make a new tree as follows. Contract the edge $p_{n-2}p_{n-1}$ from S_3, forming S_3' say. Take a new vertex w, and the three disjoint trees S_1, S_2, S_3', and add three edges $e_1 = wp_1, e_2 = wp_n$ and a new edge $f = wp_2$ say. This produces a tree in which every vertex has degree 1 or 3. Then (T', ν) is a branch-decomposition, extending (T_i, ν), and in it, e_1, e_2 have a common end. So it suffices to show that its width is the branch-width of G. This was true for (T, ν), and the only edges that make separations whose orders might differ in the two branch-decompositions, are the edges of the path $p_2, \ldots, p_{n-1}$, and the new edge f. But f is fine, since it makes the separation that we wanted to add to (T_i, ν) in the first place. For an edge of $p_2, \ldots, p_{n-1}$, it make a separation in both (T, ν) and (T', ν). But in the order of the second separation, we no longer have to count the vertices in $(N_1 \cap N_2) \setminus N_3$, and the only new vertices that we might need to count belong to $(N_2 \cap N_3) \setminus N_1$. Since the first set is at least as large as the second, this is a net improvement, and shows that (T', ν) has width the branch-width of G, as required. ∎

We call (1) the *pushing inequality.* Whenever there is a pair e_1, e_2 that satisfies the pushing inequality, we say that the separations made by e_1 and e_2 can be *pushed* towards v, and we say T_{i+1} is obtained by *pushing.*

In our algorithm, whenever we introduce a new separation into the partial branch-decomposition, we immediately try to push it towards both ends of the corresponding edge of the tree. If we succeed in pushing the separation, we obtain more separations to try to push, and it can happen that one good initial choice of a separation and then pushing alone will completely decompose the graph.

We mention a few details of our pushing implementation. First, we wish at every stage to be sure that every separation arising from the tree cannot be pushed in either direction, or if it can, we wish to immediately do the corresponding splitting. It is not necessary to keep rechecking old separations, however; the only pairs that might become pushable after introducing some new separations each involve at least one of the new separations.

To check if the separation arising from an edge $e_1 \in D$ can be pushed towards v (with the notation as above) we need to examine all edges $e_2 \in D \setminus \{e_1\}$ and check the pushing inequality. However, for any edge e_2 for which the pushing inequality holds, there will be a node $w \in M_{e_1} \cap M_{e_2}$ such that $w \notin M_e$ for all $e \in D \setminus \{e_1, e_2\}$; and this observation eliminates most possibilities for e_2 at a stroke.

4.2 2-separations and 3-separations

In our algorithm, we construct the star T_1 as before and then we push all separations as much as possible. This results in a partial branch-decomposition (T, ν) in which one node of T has large degree, but all others have degree one or three; the decomposition has order two, and is extendible, and no separation arising can be pushed. (Recall that we have assumed the input graph in simple and 2-connected.)

Now we wish to introduce more separations of order two if possible. In general, we have a partial branch-decomposition (T, ν) of order two, and in it several internal nodes have degree greater than three. For each such node v, let D be as before; we test if there

is a partition X, Y of D with $|X|, |Y| \geq 2$ such that $|M(X,Y)| = 2$. If we find one, we make the corresponding split of T, push in both directions and repeat. The new tree we obtain is guaranteed to be extendible. To see this, note that in a 2-connected graph, every separation of order 2 is "titanic", in the terminology of Robertson and Seymour (1991). It follows that for any 2-separation with middle $\{u, v\}$, we can add an edge joining u, v, without changing the branch-width, because of theorems (4.3) and (8.3) of that paper. But then the claim follows. To test if there is such a partition, we construct the graph H with node set $\bigcup(M_e : e \in D)$, in which two nodes are adjacent if some M_e contains them both. This graph H is necessarily 2-connected, and we are looking for a separation (A, B) of H of order two with left$(A, B) \neq \emptyset$ and right$(A, B) \neq \emptyset$. To test for this, we choose a node x of H and three edges e, f, g of H incident with x. First we look for (A, B) with $e \in A$ and $f \in B$. To do so, we choose a separation (A, B) with $e, g \in H$ and $f \in B$ of order two with A minimal (using network flows), and see if right$(A, B) \neq \emptyset$. Failing this, we repeat with $e \in A$ and $f, g \in B$, and now choose the separation with B minimal, and see if left$(A, B) \neq \emptyset$. If this also fails, then e and f are on the same side of every separation (A, B) of H of order two with left$(A, B) \neq \emptyset$, right$(A, B) \neq \emptyset$; so we choose (A, B) of H of order two with $e, f \in A$ and with A minimal, and see if right$(A, B) \neq \emptyset$. If this fails there is no choice of (A, B) possible, and no splitting of v is possible with 2-separations. If one of these three succeeds, we find a way to split v with a 2-separation; we construct the corresponding new partial branch-decomposition and push the new separation in both directions as much as possible. We continue this process until no further splitting with 2-separations is possible.

The next step is the introduction of 3-separations. At the start of the iteration we have a partial branch-decomposition (T, ν), of width at most three, and we try to split each internal node of T with a 3-separation. We construct H as before; H is now 3-connected. First we test if some $x \in V(H)$ has degree three and has two adjacent neighbors. If so, let $X = \{e \in D : x \in M_e\}$ and $Y = D \setminus X$; then we split according to X, Y, and push in both directions and repeat. If there is no such x, we choose $x \in V(H)$ arbitrarily, let w and y be distinct neighbors of x and let z be a neighbor of y different from w, x. We test if there is a 3-separation (A, B) of H such that left(A, B) and right(A, B) both meet $\{w, x, y, z\}$ and left(A, B), right(A, B) both have size at least two; this is done by enumerating the different possibilities for which nodes of w, y, y, z are on the left and right, and solving a max-flow problem in each case. If we find such a 3-separation, we split using it, and push and repeat. If not, then we choose a 3-separation (A, B) of H with $\{wx, xy, yz\} \supseteq A$ and with A minimal, and see if right(A, B) has size at least two; if so we split, push and repeat. If not then we abandon trying to split v with 3-separations, and move on to the next node of T.

This process concludes with a partial branch-decomposition of order at most three, still guaranteed to be extendible, such that for every internal node of the tree, if H is the corresponding graph then H is 4-connected except for nodes of degree three, and no node of degree three is in a triangle. Showing that these splits are safe is another application of theorems (4.3) and (8.3) of Robertson and Seymour (1991), and we omit the details. The main point is that, since H is 3-connected, *every* 3-separation of H (that corresponds to a split) is safe unless its middle is a stable set *and* one of the left and right sets only contains one vertex. The algorithm above is just a fast way of looking for such a 3-separation.

At this stage, if there is a node of the tree with degree at least four, we apply the eigenvector method to find a way to split; then we push, and repeat. It is not necessary to keep checking for 2-separations and 3-separations; none can appear.

5 Decomposition Results

To be effective in applications, branch-decomposition heuristics must be able to produce near-optimal decompositions. This criterion is unfortunately difficult to evaluate, since Seymour and Thomas (1994) proved that determining if a graph has branch-width $\leq k$ is $\mathcal{NP}$-complete (when k is part of the input to the problem). This makes it difficult to obtain a reliable set of benchmark results for instances appropriate to our TSP setting. (Note, moreover, that the width of a decomposition is only a rough measure of its quality, a more more refined approach would be to consider the distribution of middle sets of the various sizes.)

One thing we can establish (within the context of the TSP tour-merging application), is that is does appear to be important to consider some form of the separator problem where we ask for splits having a substantial number of tree edges on both sides. In Section 3, we proposed to require that each side in a split have at least $|D|/3$ edges, and we have incorporated this into our implementation of the eigenvector heuristic. In Table 1 we compare our implementation with one obtained by relaxing the separator problem to require only that we have two or more edges on each side of the split; the column labeled "Default" contains the widths of the decompositions found by our default implementation, and the column labeled "≥ 2" contains the results obtained with the modified algorithm. The graphs reported in Table 1 were obtained by taking the union of ten tours found by an implementation of the Chained Lin-Kernighan heuristic (see Section 7) and shrinking any induced path containing more than three edges; the TSP instances are taken from the TSPLIB collection of Reinelt (1991).

The default version of our heuristic produced substantially better decompositions in the tests reported in Table 1. It should be noted, however, that it is not clear that the decompositions cannot be improved further by a more effective heuristic. (Indeed, we have found decompositions of width 14 for the rl5915 instance and of width 15 for rl5934. The five test graphs are available for further study at `www.isye.gatech/edu/~wcook/bwidth.`)

Table 1: Decompositions of TSP Graphs

Name	Nodes	Edges	Default	≥ 2
pcb3038	1,985	3,109	15	18
fl3795	2,103	3,973	9	10
fnl4461	3,326	5,147	19	28
rl5915	1,939	2,935	16	18
rl5934	2,048	3,087	16	25

If we move away from the TSP application, it is possible to test the quality of the heuristic decompositions by considering general planar graph instances. Seymour and

Thomas (1994) have shown that the branch-width of planar graphs can be computed in polynomial time, and Hicks (2000) has developed an implementation of the Seymour-Thomas algorithm that is practical for instances having up to several thousand nodes.

Hicks (2000) reports the branch-width of Delaunay triangulations for many of the geometric instances from the TSPLIB. In our tests, we consider all instances studied by Hicks having at least 200 nodes and having branch-width less than 20 (the graphs of higher branch-width are less relevant in studies of optimization algorithms). The results are presented in Table 2; the "Edges" column gives the number of edges in each graph, the "Branch-width" column gives the branch-width computed by Hicks (2000), the "Heuristic" column gives the width of the decomposition found by our heuristic, and the "Time" column gives the CPU time (in seconds) used by the heuristic on a 500 MHz EV6 Compaq Alpha processor.

Table 2: Decompositions of Planar Graphs

Name	Edges	Branch-width	Heuristic	Time
kroA200	586	11	11	4.31
kroB200	580	12	13	4.20
tsp225	622	12	13	4.42
pr226	586	7	7	2.36
pr264	772	13	14	8.33
gil262	773	15	16	8.57
a280	788	13	14	4.10
pr299	872	11	12	10.05
rd400	1,183	17	19	21.38
fl417	1,179	9	9	20.03
fl417	1,179	9	9	20.03
pr439	1,297	16	18	25.66
pcb442	1,286	17	22	31.02
u574	1,708	17	19	45.55
rat575	1,699	17	19	45.90
p654	1,806	10	10	65.60
u724	2,117	18	22	76.34
vm1084	2,869	15	16	90.06
rl1304	3,879	19	21	279.93
fl1400	4,138	13	14	314.92
fl1577	4,637	16	19	209.44

For the smaller instances in Table 2, the width of the decomposition found by the heuristic algorithm is usually within one of the branch-width of the graph. For the larger instances, however, the heuristic results are as much as five away from the optimal results. This indicates that there is certainly room to improve the practical performance of the algorithm, especially in applications where the dynamic-programming portion of an optimization task is very sensitive to the width of the decomposition.

6 Dynamic Programming

With a branch-decomposition in hand, it is easy to construct a dynamic-programming algorithm to solve the TSP on the corresponding graph. We give below some details of our TSP implementation, following the sketch we presented in Section 2.

6.1 General Description

Let G be a simple, 2-connected graph with edge costs $(c_e : e \in E(G))$ and let (T, ν) be a branch-decomposition of G. The idea of the dynamic-programming algorithm is to start at the leaves of the tree T and work "inwards", processing the corresponding separations of G. To make the notion of "inwards" precise, we *root* the branch-decomposition by selecting an arbitrary edge (a, b) of the tree and adding new tree nodes r and s, and new edges (a, s), (s, b), and (s, r), and removing the edge (a, b); let T' denote the tree we obtain in this way and call r the root node of T'. No edge of G is assigned to node r, so the separations corresponding to (a, s) and (s, b) are the same as the separation for the old edge (a, b); the separation corresponding to (s, r) is $(E(G), \emptyset)$.

Each node v of T', other than r, meets a unique edge that lies on the path from v to r in the tree, and if v is not a leaf it also meets two edges that do not lie on this path; we refer to the unique edge as the *root edge* of v and we refer to the other two edges as the *left* and *right* edges of v (the choice of which is left and which is right is arbitrary). Since each tree edge e is the root edge of precisely one node v, when we have processed the separation corresponding to e we say that we have processed the node v. We say that a tree node is *ready* to be processed if either it is a leaf of the tree (and it is not the root node), or both its left edge and its right edge have already been processed. The overall procedure is then to select any ready node and process it, stopping when r is the only remaining node. To specialize this general algorithm, we need to describe how to "process" a separation when solving the TSP.

6.2 Encoding Partial Tours

Let $(A, \bar{A})$ be a separation of G, where $\bar{A} = E(G) \setminus A$. A *partial tour* in A is a subset $R \subseteq A$ consisting of the union of a collection of edge sets of paths having both ends in the middle of $(A, \bar{A})$, such that every node in left$(A, \bar{A})$ is included in exactly one of the paths and every node in middle$(A, \bar{A})$ is included in at most one of the paths.

Let R be a partial tour in A, and let $n_0, n_1, \ldots, n_k$ be the nodes in the middle set of $(A, \bar{A})$. A middle node n_i meets either zero, one, or two edges in R. We say that n_i is *free* if it meets zero edges, *paired* if it meets one edge, and *used* if it meets two edges. If we follow the path in R meeting a paired middle node n_i, we will reach another paired middle node n_k. We can specify how R "hits" the middle of $(A, \bar{A})$ by listing the pairs (n_i, n_k) of paired nodes and listing the free nodes and the used nodes. We refer to such an encoding as a *matching*. To process the separation $(A, \bar{A})$, we find the list of matchings corresponding to the partial tours in A, and for each matching we record the corresponding partial tour having the least cost, that is, the partial tour S that realizes the matching and minimizes $\sum(c_e : e \in S)$.

6.3 Processing Separations

Let us first consider a separation corresponding to a leaf v of T' (other than the root node). Since G is simple and 2-connected, the middle set of the separation consists of two nodes, n_0 and n_1 (the ends of the edge $\nu(v)$). The only matchings in this case are either to have both n_0 and n_1 free (of total cost 0), or to have n_0 paired with n_1 (of total cost $c_{\nu(v)}$).

Now consider an internal node v of T'. The graph obtained by deleting v from T' has three connected components, giving a partition of the leaves of T' into three sets; let L, R, and N be the corresponding subsets of $E(G)$, where L corresponds to the component of $T' \setminus v$ that meets v's left edge, R corresponds to the component that meets v's right edge, and N corresponds to the component that meets v's root edge. So $(L, \bar{L})$ is the separation associated with the left edge of v, $(R, \bar{R})$ is the separation associated with the right edge of v, and $(L \cup R, \overline{L \cup R})$ is the separation associated with the root edge of v. (Note that $N = \overline{L \cup R}$.)

Each node in middle$(L \cup R, \overline{L \cup R})$ is either in middle$(L, \bar{L})$ or in middle$(R, \bar{R})$. This makes it possible to compute the matchings for $L \cup R$ by studying only the matchings for L and the matchings for R (we do not need to know anything else about the graph G). Any partial tour in $L \cup R$ must arise as the disjoint union of a partial tour in L and a partial tour in R, but not all such unions give partial tours in $L \cup R$. It easy to check, however, if a pair of matchings is *compatible* (that is, the union of their corresponding partial tours gives a partial tour in $L \cup R$) by examining only the nodes in the middle sets of $(L, \bar{L})$ and $(R, \bar{R})$. So we can proceed by running through all pairs of a matching for L and a matching for R and computing if the pair gives a valid matching for $L \cup R$. This procedure involves some analysis (following paths to find the new paired nodes), but as long as the middle sets are small it can be carried out quickly.

6.4 Merging Lists of Matchings

The main computational difficulty in the dynamic-programming algorithm arises from the fact that the lists of matchings for the left and right edges of a node may be quite long, and we need to run through each possible left-right pair in order to obtain the list for the root edge of the node. The worst-case complexity (counting the number of possible matchings on a middle set of cardinality k) indicates that we cannot hope to to solve every instance of width, say, 20 or higher. We attempt to solve, however, at least a reasonable portion of the examples that arise in practice by taking some simple steps to speed up the list merging, as we indicate below. The computational tests reported in the next section indicate mixed results—although the code was usually successful, in some cases we hit time or memory limits even when the width was under 20. (The smallest case that failed was an instance of width 18, but we also report on a case of a width 13 decomposition that took over 600,000 seconds to execute the dynamic-programming algorithm.)

Let k denote the width of the branch decomposition (T, ν) and suppose we have lists of matchings for L and R (using the notation as above) and we need to compute the implied list for $L \cup R$. The set Q of *local nodes* that we use in the computation consists of the union of the middle sets for the three separations $(L, \bar{L})$, $(R, \bar{R})$, and $(L \cup R, \overline{L \cup R})$. Since each node in Q appears in at least two of the three middle sets, we have $|Q| \leq 3k/2$. Let us order the nodes in Q as $q_0, q_1, \ldots, q_{t-1}$, where $t = |Q|$.

For a matching γ, let $d(\gamma)$ denote the *degree sequence*

$$(d_0(\gamma), d_1(\gamma), \ldots, d_{t-1}(\gamma))$$

where for each $i = 0, \ldots, t-1$, we have

$$d_i(\gamma) = \begin{cases} 0 & \text{if } q_i \text{ is free in } \gamma \text{ or if } q_i \text{ is not in the middle set for the separation.} \\ 1 & \text{if } q_i \text{ is paired in } \gamma. \\ 2 & \text{if } q_i \text{ is used in } \gamma. \end{cases}$$

If a matching α for L and a matching β for R are compatible, then for each node q_i that is in both middle$(L, \bar{L})$ and middle$(R, \bar{R})$ but not in middle$(L \cup R, \overline{L \cup R})$ we must have

$$d_i(\alpha) + d_i(\beta) = 2$$

since q_i will be in the left-set of $(L \cup R, \overline{L \cup R})$. Also, for each node q_i that is in all three middle sets we must have

$$d_i(\alpha) + d_i(\beta) \leq 2.$$

(This condition on the remaining local nodes will hold trivially.) Therefore, we can avoid explicitly checking many incompatible left-right pairs by keeping the matching lists grouped according to their degree sequences; to merge the lists we run through the lists of degree sequences and check the sub-lists of matchings only if the degree sequences are compatible.

As we build the list of matchings for $L \cup R$, we use a hash table to store the collection of degree sequences, making it simple to determine if we have already encountered a sequence in the list-merging process. We encode each degree sequence as a single number (by considering the sequence as ternary digits) to make it easy to compare if two sequences are identical (when they hash to the same value).

If we determine that the degree sequence for the matching we are considering is already in our collection, then we need to determine if the matching itself is also one we have previously encountered. To handle this, it is possible to use a hash table to store the sub-lists of matchings for each degree sequence, but we found it satisfactory just to keep the sub-lists sorted by a numerical encoding of the matchings. If the current matching is not in the sub-list, we add it. Otherwise, we compare the cost of the current matching to the cost of the identical matching already in the sub-list. (The cost of the matching is just the sum of the costs of the matchings in the left-right pair we are merging.) If the current matching has lower cost, we record it in place of the existing copy. The record for a matching is its cost together with links to the left matching and to the right matching we are merging; the left-right links are used to work backwards to gather the edges in the optimal tour after we process the entire tree, permitting us to avoid explicitly storing the partial tours associated with the matchings in our list.

7 Tour-Merging Results

Our primary application of the TSP dynamic-programming algorithm is to implement the tour-merging idea we described in Section 1. The sensitivity of dynamic programming to the width of the branch-decompositions makes it clear that care needs to be taken in generating

the tours to be merged. A simple rule is that as the problem size increases, the quality of the generated tours must also increase. We will follow this in our computational study, using Chained Lin-Kernighan tours for small to medium instances, and using Helsgaun's LKH heuristic for larger instances.

7.1 Merging Chained Lin-Kernighan Tours

Martin et al. (1991) introduced a TSP heuristic that uses the basic Lin-Kernighan (1973) algorithm in an iterative fashion. Their general method is called *Chained Local Optimization* in Martin and Otto (1996), and we refer to their TSP algorithm as *Chained Lin-Kernighan.* Johnson (1990), Johnson and McGeoch (1997, 2002), and Applegate et al. (2002) present results showing the effectiveness of Chained Lin-Kernighan over a wide range of problem instances.

In our tests, we build the pool $\mathcal{T}$ of tours using the `linkern` implementation of Chained Lin-Kernighan included in the *Concorde* TSP package of Applegate et al. (1998). The `linkern` implementation is described in Applegate et al. (2002) and it can be obtained at `www.math.princeton.edu/tsp`.

We test the combination of Chained Lin-Kernighan and tour-merging on all TSPLIB instances having at least 1,000 cities and at most 10,000 cities. In each case, we report the average results over four trials, where we merge ten tours in each trial. The tours are generated using the default parameters of `linkern` (including n iterations, where n is the number of cities in the test instance).

The test results are presented in Table 3. The column "Best CLK" gives the % gap (to the optimal tour value) for the best of the ten Chained Lin-Kernighan tours, and the "Merged" column gives the % gap for the merged tour. In the "Failures" column we report the number of times that the merge-step failed in the four trials. The running times are again given in seconds on a 500 MHz EV6 Compaq Alpha processor; the "CLK time" is the total time to generate the ten tours; the "Merge Time" includes both the time to find the branch-decomposition and the time to run the dynamic-programming algorithm. The average width of the branch-decompositions is reported in the "Width" column.

In two of the instances, u2319 and pla7397, the tour-merging code failed on all four trials. In both cases, the failures were due to decompositions of width greater than 20 (the maximum allowed in our dynamic-programming algorithm). For fnl4461, the code failed in three of the four trials; in the three failures, decompositions of width 19 were found, but the run of the dynamic-programming algorithm exceeded the 2GByte memory limit of our computer. (In Table 3, we report the "Width" and "Merge Time" only for the successful fn4461 trial.)

Averaging over all trials in our tests, tour-merging reduced the % gap (to the optimal tour value) from 0.21% down to 0.07%.

Our tests give just one sample point of the many choices for $|\mathcal{T}|$ and the many possible settings for `linkern`, but the results are typical of the type of improvements that are possible with tour-merging.

Table 3: Merging ten Chained-LK Tours - Average over four Trials

Name	Best CLK	Merged	Failures	CLK Time	Merge Time	Width
dsj1000	0.08%	0.02%	0	56.67	1.04	6.8
pr1002	0.22%	0.05%	0	22.05	1.20	8.0
si1032	0.09%	0.04%	0	18.78	0.14	6.8
u1060	0.14%	0.01%	0	33.65	1.70	7.5
vm1084	0.02%	0.00%	0	29.18	0.37	7.2
pcb1173	0.21%	0.01%	0	18.19	3.64	11.0
d1291	0.22%	0.11%	0	33.90	1.33	8.5
rl1304	0.34%	0.30%	0	40.80	0.42	8.5
rl1323	0.18%	0.07%	0	32.52	0.42	8.0
nrw1379	0.13%	0.03%	0	21.97	3.28	10.8
fl1400	0.00%	0.00%	0	199.62	1582.47	9.5
u1432	0.20%	0.03%	0	34.49	282.46	8.8
fl1577	0.03%	0.01%	0	116.42	2.07	8.0
d1655	0.37%	0.12%	0	45.28	4.39	9.8
vm1748	0.10%	0.00%	0	55.22	1.47	8.5
u1817	0.38%	0.12%	0	32.33	10.72	11.5
rl1889	0.25%	0.05%	0	74.02	1.94	10.2
d2103	0.47%	0.41%	0	81.06	11.88	10.2
u2152	0.24%	0.14%	0	38.57	67.33	11.8
u2319	0.11%	-	4	168.15	-	> 20
pr2392	0.25%	0.04%	0	44.94	12.65	11.5
pcb3038	0.16%	0.03%	0	55.50	346.28	14.0
fl3795	0.41%	0.11%	0	271.61	65.19	9.5
fnl4461	0.17%	0.06%	3	96.46	(38.94)	(14)
rl5915	0.30%	0.03%	0	210.47	475.99	13.5
rl5934	0.27%	0.02%	0	228.74	47.11	15.2
pla7397	0.25%	-	4	404.04	-	> 20

7.2 Merging Lin-Kernighan-Helsgaun Tours

We use Helsgaun's LKH code in our tests on large-scale instances—results reported in Helsgaun (2000) and in Johnson and McGeoch (2002) indicate that LKH finds significantly better tours than does multiple runs of Chained Lin-Kernighan. Other recent high-end TSP heuristics that could also be considered include Balas and Simonetti (2001), Schilham (2001), and Walshaw (2000), but LKH appears to exceed the performance of all TSP heuristic algorithms proposed to date.

7.2.1 TSPLIB Instances

In our merge tests, we consider only those TSPLIB instances having at least 5,000 cities. For the smaller instances, Helsgaun (2000) reports that the default version of LKH finds optimal solutions in at least one run out of ten in each case. We also exclude the instance pla7397 since LKH routinely finds optimal solutions in this case. Finally, we exclude pla85900 since multiple runs of LKH on this instance would exceed our available computing time.

Our first test on the remaining eight TSPLIB instances is summarized in Table 4. Each run in this test consists of merging ten LKH tours generated with the default settings (n repeated trials for each n-city TSP). For each instance we made four of these runs, and we report the average results in Table 4. The information in the table is organized in the same manner as in Table 3. The column "Best LKH" gives the % gap (to the best available lower bound) for the best of the ten LKH tours, and the "Merged" column gives the % gap for the merged tour (in the case of rl11849, the optimal solution was found in each of the four trials). In the "Failures" column we report that the merge step failed in two of the four d18512 trials as well as in all four pla33810 trials; the failures were due to decompositions of width greater than 20. The running times are again given in seconds on a 500 MHz EV6 Compaq Alpha processor.

Table 4: Merging ten LKH Tours - Average over four Trials

Name	Best LKH	Merged	Failures	LKH Time	Merge Time	Width
rl5915	0.0166%	0.0054%	0	16,298	1.55	7.50
rl5934	0.0133%	0.0081%	0	23,134	2.07	7.75
rl11849	0.0051%	Optimal	0	161,621	15.49	7.25
usa13509	0.0047%	0.0010%	0	242,118	29.22	9.25
brd14051	0.0086%	0.0036%	0	419,079	156.93	14.25
d15112	0.0047%	0.0007%	0	494,044	134.20	15.50
d18512	0.0161%	0.0075%	2	926,215	(311.71)	(17.50)
pla33810	0.1024%	-	4	10,908,095	-	> 20

In Table 5 we report the results of merging forty LKH tours for each of the test instances (only a single trial). The two largest instances failed, but in four of the remaining six instances an optimal solution was found. Note that although the total LKH time is quite high, merging the tours requires only a modest amount of additional computing time. The time for merging can be reduced, moreover, if we merge only the ten best from each set of forty tours, as we indicate in Table 6. This ten-out-of-forty test produced the same

tours as the all-forty tests in the six smaller instances, and it also produced a solution to d18512 (but still failed on pla33810). The average branch-width was reduced by 26% in the ten-out-of-forty test, resulting in a large decrease in the merging time.

Table 5: Merging forty LKH Tours

Name	Best LKH	Merged	Total LKH Time	Merge Time	Width
rl5915	0.0087%	Optimal	63,954	2.21	8
rl5934	0.0065%	Optimal	91,264	3.15	8
rl11849	0.0023%	Optimal	646,483	36.26	7
usa13509	0.0031%	0.0001%	968,473	62.06	13
brd14051	0.0060%	0.0030%	1,676,314	444.34	17
d15112	0.0029%	Optimal	1,976,174	3944.32	19
d18512	0.0139%	Failed	3,704,858	-	> 20
pla33810	0.0998%	Failed	43,632,379	-	> 20

Table 6: Merging best ten out of forty LKH Tours

Name	Merged	Merge Time	Width
rl5915	Optimal	1.68	7
rl5934	Optimal	0.17	5
rl11849	Optimal	4.64	6
usa13509	0.0001%	16.54	8
brd14051	0.0030%	103.48	12
d15112	Optimal	107.21	15
d18512	0.0071%	278.90	13
pla33810	Failed	-	> 20

Optimal values are not known for brd14051 and d18512, but in both cases the tour found in the ten-out-of-forty merge improved the best reported result (Reinelt 1991) for the given instance. We list the tour lengths in Table 7, together with the best reported lower bounds for the instances (found by Applegate et al. 2001).

Table 7: Unsolved TSPLIB Instances

Name	Best Tour	Lower Bound	Gap
brd14051	469,388	469,374	0.003%
d18512	645,244	645,198	0.007%

7.2.2 World Instances

To provide further tests of LKH tour merging, we consider ten instances from the "World" test set available at `www.math.princeton.edu/tsp/world/`. We study all instances in the

range of 6,000 cities through 11,000 cities. For most of the instances in this test set, we do not have available optimal values or good lower bounds. We therefore report only the final tour lengths and the % improvement tour merging provided over the best of the LKH tours.

In Table 8 we report the averages over four runs, merging ten LKH tours in each test. The % improvements are quite small, but they may well account for a large portion of the gap between the LKH tours and the (unknown) optimal values. (In the case of gr9882, the tour length 300,899 is indeed optimal, as verified by the Applegate et al. code.)

Table 8: Merging ten LKH Tours – World Instances – Four Trials

Name	Tour Length	Merge Improvement	LKH Time	Merge Time	Width
tz6117	394,722	0.0022%	42,758	17.18	8.25
eg7146	172,740	0.0017%	93,352	15.23	8.25
ym7663	238,315	0.0008%	59,719	16.35	8.5
pm8079	114,876	0.0183%	647,351	2,076.20	10
ei8246	206,171	0.0019%	76,844	26.01	9.25
ar9152	837,638	0.0070%	369,531	183.19	8.5
ja9847	492,007	0.0169%	119,425	14.76	8.75
gr9882	300,899	0.0003%	105,013	18.06	7.75
kz9976	1,061,883	0.0057%	105,300	15.79	8.25
fi10639	520,535	0.0029%	156,039	121.12	11.75

Table 9: Merging forty LKH Tours – World Instances

Name	Tour Length	Merge Improvement	LKH Time	Merge Time	Width
tz6117	394,718	0.0030%	169,679	23.12	6
eg7146	172,738	0.0017%	371,422	58.56	11
ym7663	238,314	0.0004%	236,494	30.15	9
pm8079	114,872	0.0139%	2,586,483	620,597.08	13
ei8246	206,171	0.0000%	304,886	52.99	10
ar9152	837,556	0.0094%	1,474,264	532.78	14
ja9847	-	-	473,571	$> 1,000,000$	18
gr9882	300,899	0.0000%	415,948	49.72	10
kz9976	1,061,882	0.0009%	417,212	34.70	10
fi10639	520,527	0.0025%	619,082	167.73	14

The results for merging forty LKH tours are presented in Table 9. The merging procedure improved the best of the forty LKH tours in all but two of the cases (`LKH` actually delivered the optimal solution for gr9882 and it is quite possible that the ei8246 tour is optimal as well). The ja9847 test failed, even though the width of the decomposition was within the range allowed in the optimization routine. In this case, the dynamic-programming procedure exhibited poor behavior and exceeded a 1,000,000-second time limit. Note also the exceptionally long running time for the merge portion of the pm8069 run; the union of the

forty tours for this 8,069-city instance contained 21,352 edges and the decomposition had three middle sets of order 13. This example (as well as that of ja9847) emphasizes the need for care in applying dynamic programming in instances where the width is relatively high—a running time limit of several thousand seconds may be appropriate in many applications.

In our final test, we again consider merging the best ten out of forty LKH tours as we did for the TSPLIB instances. The results reported in Table 10 indicate a sharp reduction in computing time, permitting the algorithm to produce tours for all ten instances. In six of the cases, the merged tour has the same length as in the all-forty test, and in three cases the tour is slightly worse than the corresponding all-forty tour (the all-forty test did not produce a tour in the remaining case). (Note that the width of the decomposition for tz6117 actually increased over the width in the all-forty test—this points out the heuristic nature of our decomposition algorithm.)

Table 10: Merging best ten out of forty LKH Tours – World Instances

Name	Merged	Merge Time	Width
tz6117	394,718	6.66	9
eg7146	172,738	4.85	9
ym7663	238,314	3.78	8
pm8079	114,882	314.54	10
ei8246	206,171	7.76	7
ar9152	837,611	94.36	7
ja9847	491,947	28.52	9
gr9882	300,899	8.13	8
kz9976	1,061,882	1.09	5
fi10639	520,528	69.79	10

8 Conclusions

Our TSP results provide an example of a successful application of branch-decomposition in discrete optimization. We do not claim to have a combination of heuristics and tour-merging that is consistently better than other high-end TSP methods. Our LKH tests show, however, that results from even the best current heuristics can often be improved in a nominal amount of extra CPU time (if multiple runs of the heuristic have been made). The branch-width based algorithm can also be used in a stand-alone fashion, finding the best tour through sets of well-chosen edges. This idea was employed in the linear programming (LP) based TSP code of Applegate et al. (1995), where the edge sets were provided by the solution of the LP relaxation (each edge assigned value greater than $\epsilon = 0.001$ was placed into the edge set). This idea should have applications to other network optimization problems where good LP relaxations are available.

Our results make use of the eigenvector-based heuristic for finding branch-decompositions. This method appears to work well in practice, but we have not made a study of other classes of heuristics. One comparison was carried out by Hicks (2000), who described a branch-decomposition heuristic that uses the diameter of the graph to find separations. Hicks'

results show that slightly better decompositions can be found at the expense of additional CPU time. It would be interesting to see how branch-width versions of the tree-width heuristics described in the theoretical papers of Reed (1992) and others perform in practice.

Acknowledgements

We would like to thank Noga Alon for suggesting the use of eigenvectors to find separators in our branch-decomposition heuristic and to thank David Applegate for discussions on tour-merging strategies.

References

Alon, N. 1986. Eigenvalues and expanders. *Combinatorica* **6** 83–96.

Applegate, D., R. Bixby, V. Chvátal, W. Cook. 1995. Finding cuts in the TSP (a preliminary report). DIMACS Technical Report 95-05, DIMACS, Rutgers University, New Brunswick, New Jersey.

Applegate, D., R. Bixby, V. Chvátal, W. Cook. 1998. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians.* 645–656.

Applegate, D., R. Bixby, V. Chvátal, W. Cook. 2001. TSP cuts which do not conform to the template paradigm. M. Jünger, D. Naddef, eds. *Computational Combinatorial Optimization.* Springer-Verlag, Heidelberg, Germany. 261–304.

Applegate, D., W. Cook, A. Rohe. 2002. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing.* To appear.

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

Balas, E., N. Simonetti. 2001. Linear time dynamic programming algorithms for some new classes of restricted TSPs: a computational study. *INFORMS Journal on Computing* **13** 56–75.

Bern, M. W., E. L. Lawler, A. L. Wong. 1987. Linear time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms* **8** 216–235.

Bodlaender, H. L. 1993. A tourist guide through treewidth. *Acta Cybernetica* **11** 1–21.

Bodlaender, H. L. 1996. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing* **25** 1305–1317.

Bodlaender, H. L. 1998. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science* **209** 1–45.

Bodlaender, H. L., T. Kloks. 1996. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms* **21** 358–402.

Borie, R. B., R. G. Parker, C. A. Tovey. 1992. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica* **7** 555–581.

Courcelle, B. 1990. The monadic second-order logic of graphs I: recognizable sets of finite graphs. *Information and Computation* **85** 12–75.

Helsgaun, K. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research* **126** 106–130. The `LKH` code is available at `www.dat.ruc.dk/~keld/research/LKH/` .

Hicks, I. V. 2000. *Branch Decompositions and their Applications.* Ph.D. Thesis, Computational and Applied Mathematics, Rice University, Houston, Texas.

Johnson, D. S. 1990. Local optimization and the traveling salesman problem. *Proceedings 17th Colloquium of Automata, Languages, and Programming*, Lecture Notes in Computer Science **443**. Springer-Verlag, Berlin, Germany. 446–461.

Johnson, D. S., L. A. McGeoch. 1997. The traveling salesman problem: a case study in local optimization. E. H. L. Aarts, J. K. Lenstra, eds. *Local Search in Combinatorial Optimization.* John Wiley & Sons, New York. 215–310.

Johnson, D. S., L. A. McGeoch. 2002. Experimental analysis of heuristics for the STSP. G. Gutin, A. Punnen, eds. *The Traveling Salesman Problem and its Variations.* Kluwer Academic Publishers, Dordrecht, The Netherlands. 369–443.

Koster, A. M. C. A., H. L. Bodlaender, S. P. M. van Hoesel. 2001. Treewidth: computational experiments. *Electronic Notes in Discrete Mathematics* **8**. Elsevier Science Publishers, Amsterdam, The Netherlands.

Koster, A. M. C. A., S. P. M. van Hoesel, A. W. J. Kolen. 1999. Solving frequency assignment problems via tree-decomposition. Research Memorandum 99011, Maastricht Research School of Economics of Technology and Organizations, Universiteit Maastricht, Maastricht, The Netherlands.

Lagergren, J. 1990. Efficient parallel algorithms for tree-decomposition and related problems. *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science.* ACM Press, New York. 173–182.

Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* **21** 498–516.

Martin, O. C., S. W. Otto. 1996. Combining simulated annealing with local search heuristics. *Annals of Operations Research* **63** 57–75.

Martin, O., S. W. Otto, E. W. Felten. 1991. Large-step Markov chains for the traveling salesman problem. *Complex Systems* **5** 299–326.

Matoušek, J., R. Thomas. 1991. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms* **12** 1–22.

Reed, B. 1992. Finding approximate separators and computing tree-width quickly. *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing.* ACM Press, New York. 221–228.

Reinelt, G. 1991. TSPLIB—A traveling salesman problem library. *ORSA Journal on Computing* **3** 376–384. An updated version is available at `www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/` .

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

Schilham, R. M. F. 2001. *Commonalities in Local Search.* Ph.D. Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands.

Seymour, P. D., R. Thomas. 1994. Call routing and the ratcather. *Combinatorica* **14** 217–241.

Tamaki, H. 2002. Alternating cycles contribution: a tour merging strategy for the traveling salesman problem. Submitted.

Walshaw, C. 2000. A multilevel approach to the travelling salesman problem. Technical Report 01/IM/80, Computing and Mathematical Sciences Department, University of Greenwich, London, UK.