1989

Nonpreemptive Ensemble Motion Planning on a Tree

Gred N. Frederickson
Purdue University, gnf@cs.purdue.edu

D. J. Guan

Report Number:
89-864

Frederickson, Gred N. and Guan, D. J., "Nonpreemptive Ensemble Motion Planning on a Tree" (1989).
Department of Computer Science Technical Reports. Paper 735.
<https://docs.lib.purdue.edu/cstech/735>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

NONPREEMPTIVE ENSEMBLE MOTION
PLANNING ON A TREE

Greg N. Frederickson
D. J. Guan

CSD-TR-864
February 1989
(Revised August 1989)

Correction sheet for:

"Nonpreemptive ensemble motion planning on a tree," by D. J. Guan and G. N. Frederickson.

1. The following sentence should be inserted after the period on the fourth line from the bottom on page 30: "For any component D_i , let $IP(D_i)$ represent the set of initial vertices (initial positions) of moves in D_i ."
2. The reference for the minimum spanning tree was inadvertantly omitted. It should be: H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs", *Combinatorica* 6, 2 (1986) pp. 109–122.

Nonpreemptive Ensemble Motion Planning on a Tree

Greg N. Frederickson *

D. J. Guan †

Department of Computer Science

Purdue University

West Lafayette, Indiana 47907

August 7, 1989

Abstract

Consider the problem of transporting a set of objects between the vertices of a tree by a vehicle that travels along the edges of the tree. The vehicle can carry only one object at a time, and it starts and finishes at the same vertex of the tree. It is shown that if each object must be carried directly from its initial vertex to its destination then finding a minimum cost transportation is NP-hard. Several fast approximation algorithms are presented for this problem. The fastest runs in $O(k + n)$ time and generates a transportation of cost at most $3/2$ times the cost of an optimal transportation, where n is the number of vertices in the tree, k is the number of objects to be moved. Another runs in $O(k + n \log \beta(n, q))$ time, and generates a transportation of cost at most $5/4$ times the cost of an optimal transportation, where $q \leq \min\{k, n\}$ is the number of nontrivial connected components in a related directed graph.

Keywords. Motion planning, vehicle routing, graph algorithms, NP-complete, approximation algorithm, Steiner tree problem, minimum spanning tree, transportation, stacker-crane problem.

*The research of this author was supported in part by the National Science Foundation under grant CCR-86202271 and by the Office of Naval Research under contract N00014-86-K-0689. A portion of this research was done while this author was on sabbatical leave at the International Computer Science Institute, Berkeley, CA.

†The research of this author was supported in part by the National Science Foundation under grant CCR-86202271 and by Purdue University under a David Ross Fellowship.

Correction sheet for:

"Nonpreemptive ensemble motion planning on a tree," by D. J. Guan and G. N. Frederickson.

1. The following sentence should be inserted after the period on the fourth line from the bottom on page 30: "For any component D_i , let $IP(D_i)$ represent the set of initial vertices (initial positions) of moves in D_i ."
2. The reference for the minimum spanning tree was inadvertently omitted. It should be: H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs", *Combinatorica* 6, 2 (1986) pp. 109-122.

1 Introduction

Consider an undirected weighted graph with objects located at various vertices. Associated with each object is a destination vertex, to which that object is to be moved by a vehicle that traverses the edges of the graph. A fundamental problem in motion planning is to determine a tour of minimum cost for the vehicle to transport all objects from their initial positions to their destinations. In the case of general graphs, the problem is NP-hard, even if the vehicle can transport only one object at a time [5]. However, for special applications such as those that arise in robotics, it is reasonable to consider more restricted classes of graphs. In this paper and a companion paper [4] we consider problems in which the graphs are trees, and the vehicle can transport only one object at a time. In this paper we focus on nonpreemptive object movement. By this we mean that once an object is picked up, it must be transported directly to its destination.

We show that determining a tour of minimum cost for nonpreemptive motion planning on a tree is NP-hard. Consequently we design several polynomial time approximation algorithms for the problem, with provably good performance bounds. The NP-completeness of the nonpreemptive problem contrasts with our results in the case that preemptive object movement is allowed [4]. For that problem, in which objects can be dropped at the intermediate vertices and picked up later, optimal tours can be found in polynomial time. Note that viewed from the context of discrete job scheduling problems, it is not so surprising that the preemptive version of the problem is polynomial while the nonpreemptive version is NP-hard. See for example the work on the problem of scheduling independent tasks on identical processors [12] [10] [7].

Let n be the number of vertices in the tree and k the number of objects to be transported. Our first approximation algorithm runs in $O(k + n)$ time and yields a solution of cost at most $3/2$ times the cost of an optimal solution. Our second algorithm runs in $O(k + n \log \beta(n, q))$ time and yields a solution of cost at most $4/3$ times the cost of an optimal solution, where q is the number of nontrivial strongly connected components in a related directed graph. Note that $q \leq \min\{k, n\}$. An interesting feature of the above two approximation algorithms is that each algorithm handles well all the instances that the other handles poorly. Thus, a better solution can be obtained by running both algorithms on the same instance and choosing the one with smaller cost. We show that the solution obtained in this way has a cost that is at most $5/4$ times the cost of an optimal transportation.

Our results compare with those of others as follows. For the case in which the graph is a general graph, Frederickson, Hecht and Kim have shown that the problem, which they termed the stacker-crane problem, is NP-hard. They also have given a mixed strategy approximation algorithm that runs in $O(k^3 + n^3)$ time and yields a solution of cost at most $9/5$ times the cost of an optimal solution [5]. For the cases in which the graph is either a simple path or a simple cycle, Atallah and Kosaraju have presented algorithms that find an optimal solution in $O(k + n \log \beta(n, q))$ and $O(k + n \log n)$ time, respectively [1]. Frederickson has improved the latter time bound to $O(k + n \log \beta(n, q))$ [3]. For the cases in which the graph is either a simple path or a simple cycle and preemption is allowed, Atallah and Kosaraju have shown that the problem can be solved in $O(k + n)$ time [1].

In a manner similar to that in [1], we make a number of observations about the structure of an optimal tour for the problem. Some of these observations are also

used in our companion paper. In addition, we identify a number of interesting graph properties that we take advantage of in the design of our approximation algorithms. We show that the nonpreemptive motion planning problem in trees and the Steiner tree problem in bipartite graphs are polynomially equivalent. We show that the graph obtained by identifying certain vertices of the tree is not homeomorphic to a complete graph on four vertices. The Steiner tree problem and the single source shortest path problem on such graphs can be solved in linear time, benefiting our first and second approximation algorithms. We use the construction of [11] and [13] to avoid constructing a complete graph on which to find a minimum spanning tree of a related undirected graph, and thus allow our second approximation algorithm to run in almost linear time.

The rest of the paper is organized as follows. In Section 2, we introduce notation and definitions, and show how to efficiently transform the problem to yield what we call a balanced version. In Section 3, we show that the problem is NP-complete. In Section 4, we present the approximation algorithms.

2 Generating a Balanced Problem

In this section we define the problem, along with the notion of moves, drops and a transportation. The structure of our approach is similar to in [1]. In a manner similar to that in [1], We define a balanced version of a problem, and show how to construct it in linear time. An optimal transportation for the original problem can be obtained by solving the balanced version of the problem. Standard terminology of graph theory, such as a directed graph and an Euler tour, is used in our paper, and can be found in Bondy and Murty [2].

An instance P of the motion planning problem on trees consists of a tree T , a non-negative cost $c(e)$ on each edge $e \in E$, a starting vertex $s \in V$, a set of objects O , and an *initial vertex* x_j and a *destination vertex* y_j for each object $j \in O$. Each object $j \in O$ is initially located at its initial vertex x_j and has to be moved to its destination vertex y_j by a vehicle that traverses the edges of the tree. The vehicle can carry only one object at a time, and the tour must start and finish at vertex s .

We observe that, for every instance P , there is an optimal transportation such that each object visits the vertices on the path from x_j to y_j exactly once and visits no other vertices. If this is not the case, then there is a cycle traversed by some object. We could eliminate the cycle traversed by that object and get a transportation of equal or smaller cost.

A *move* is designated by (x, y, c) , where x and y are vertices in V and c is an object $j \in O$ or 0. The vehicle moves along the unique path from x to y in the tree T , and carries an object c in the move if $c \neq 0$, and no object otherwise. Thus, a move with $c \neq 0$ is called a *carrying move*, and a move with $c = 0$ is called a *non-carrying move*.

Figure 1: An example of the motion planning problem in trees.

A *transportation* Q for P is a sequence of moves (v_i, v_{i+1}, c_i) , $0 \leq i \leq r$, such that $v_0 = v_r = s$, $v_{i+1} \neq v_i$, and for each object $j \in O$ there is a move (x_j, y_j, j) in Q . We assume that $c_{i+1} \neq c_i$ for $0 \leq i \leq r$. The cost of a transportation Q , $c(Q)$, is defined to be the total distance the vehicle traveled. The motion planning problem is to find a transportation of minimum cost for an input instance P .

An example of the motion planning problem in trees is given in Figure 1. There are eight vertices in T and four objects in O . The edges of the tree T are shown as straight lines. An object j that has to be moved from x_j to y_j is shown as a curved arc from x_j to y_j with label j . The starting vertex is 0. Let $Q = (0, 7, 0)(7, 6, 1)(6, 5, 0)(5, 4, 2)(4, 5, 0)(5, 8, 3)(8, 4, 4)(4, 0, 0)$. Then Q is a transportation for the problem.

We assume that every vertex of degree one or two in T is either s or x_j or y_j for some $j \in O$. A vertex of degree one and the edge incident at it can be deleted from T if it is not s nor x_j or y_j for some $j \in O$. It is easy to see that a vertex of

degree two and its adjacent edges can be replaced by a single edge with a cost the sum of the two edges deleted if it is not s nor x_j or y_j for some $j \in O$. Thus, the number of objects, $k = \Omega(n)$.

Because every vertex of degree one is either s or x_j or y_j for some $j \in O$, every edge of T must be traversed by a valid transportation at least once. Furthermore, the number of times an edge is traversed in one direction must be equal to the number of times that edge is traversed in the other direction, since the vehicle starts and finishes at s .

We represent a move (x, y, c) by an arc from x to y labeled c . Given an optimal transportation Q for a problem P , consider a directed graph $D'(Q)$ on the vertex set V such that there is an arc from x to y labeled c if and only if there is a move (x, y, c) in Q . It is easy to see that this graph is Eulerian and it contains the arc $\langle x_j, y_j \rangle$ with label j for each object $j \in O$.

On the other hand, given an instance P , consider a directed graph D_0 with vertex set V such that there is an arc from x_j to y_j labeled j if and only if there is an object $j \in O$ initially located at x_j and has to be moved to y_j . If this graph is Eulerian, then any Euler tour starting from s can easily be translated into an optimal transportation for P . Since each arc $\langle x, y \rangle$ in D_0 represents a move, we assign a cost $d(x, y)$, the distance from x to y in T , to it. In a manner similar to that in [1], the problem is reduced to a graph augmentation problem, that of finding a set of minimum cost non-carrying moves added to D_0 so that it is Eulerian.

One type of non-carrying moves added are the *balancing moves*. They are added so that every edge is traversed at least once and the number of times an edge is traversed in one direction is equal to the number of times that edge is traversed in

the other direction. In the remainder of this section we shall show how to compute a set of balancing moves. Suppose that a set of balancing moves B is given. For each balancing move $(x, y, 0) \in B$, add a *balancing arc* $\langle x, y \rangle$ with label 0 to D_0 , and let the resulting graph be D . It is easy to see that the in-degree is equal to the out-degree for every vertex in the graph D . We shall call the graph D the *balanced graph*. Note that the augmentation of the balancing arcs may not be sufficient to get a transportation, since the arcs of D may not be in one connected component.

A connected component of D is called a *trivial component* if it contains only one vertex and this vertex is not s . Otherwise, it is called *non-trivial component*. Note that a non-trivial component that contains x_j or y_j for some $j \in O$ must contain more than one vertex. Since each non-trivial component is Eulerian, no additional non-carrying moves between two vertices in the same non-trivial component are necessary. All additional non-carrying moves will be used to connect non-trivial components. We call these non-carrying moves the *linking moves*. We shall show how to find a set of linking moves with minimum cost in the following sections.

There are, in general, many sets of balancing moves of minimum cost that satisfy the above conditions. In the remainder of this section, we show how to construct a set of $O(k + n)$ balancing moves B with minimum cost, and such that the graph D will have minimum number of non-trivial components. We also prove that for every instance P , there is an optimal transportation Q that contains all the moves in B .

Let u and v be two adjacent vertices in T , and $\phi(u, v)$ be the number of objects in O that traverse the edge from u to v . Let $b(u, v)$ be the number of

balancing moves that must traverse the edge from u to v . Then,

$$b(u, v) = \begin{cases} 1, & \text{if } \phi(u, v) = \phi(v, u) = 0; \\ \phi(v, u) - \phi(u, v), & \text{if } \phi(v, u) - \phi(u, v) > 0; \\ 0, & \text{otherwise.} \end{cases}$$

A naive way to construct a set of balancing moves is to add $b(u, v)$ balancing moves from u to v , for every ordered pair of adjacent vertex u and v in T . It is easy to see that this set of balancing move has minimum cost. It is also clear that these balancing moves give a graph D with minimum number of non-trivial components. We shall use q to denote the number of non-trivial components in the graph D , when the balancing moves are constructed this way.

The cost of the transportation will not increase if these balancing moves are added, since every valid transportation must traverse the edge (u, v) , from u to v , $b(u, v)$ times without carrying any object. The only problem with this set of balancing moves is that the total number of moves could be $O(kn)$.

We shall construct a set of $O(k + n)$ balancing moves in $O(k + n)$ time as follows. First, we compute the function $b(u, v)$ for each pair of adjacent vertices u and v in T . Second, for each $b(u, v) > 0$ we generate a balancing move $(u, v, 0)$, and then subtract one from $b(u, v)$. This insures that the number of non-trivial components in D is q . Finally, if not all $b(u, v)$ are equal to zero then we call a procedure *generate-moves* to generate additional balancing moves.

We first show how to compute the function ϕ for all ordered pairs of adjacent vertices in $O(k + n)$ time. We preprocess the input as follows. Note that this preprocessing is only for the calculation of the function ϕ . The original copy of the input instance should not be changed. First, let s be the root of T . Second, for each object j , let t be the nearest common ancestor of x_j and y_j . If t is neither x_j

nor y_j , then this object is replaced by two objects, one with initial vertex x_j and destination vertex t and the other with initial vertex t and destination vertex y_j . It is easy to see that this replacement will not change the value of ϕ . Finally, partition the resulting objects into two sets A_1 and A_2 . The objects in A_1 are those that are to be transported towards s and the objects in A_2 are those to be transported away from s .

The value of $\phi(v, u)$, where u is a parent of v in the tree rooted at s , can be computed with the set of objects in A_1 by the following algorithm. A similar algorithm on the set A_2 can be used to calculate the value of $\phi(u, v)$.

Each vertex v in T is associated with two values $xcount(v)$ and $ycount(v)$. The value of $xcount(v)$ is the number of objects with v as the initial vertex. The value of $ycount(v)$ is the number of objects with v as the destination vertex. For each child w of s we call the recursive procedure *compute- ϕ* with parameters w and s .

The procedure *compute- $\phi(v, u)$* , where u is the parent of v , does the following. If v is a leaf then $\phi(v, u) = xcount(v)$. Otherwise, for each child w call *compute- $\phi(w, v)$* , and then let $\phi(v, u)$ be the sum of $\phi(w, v)$ for every child w , plus $xcount(v)$ minus $ycount(v)$.

It is easy to see that the above algorithm correctly computes the function ϕ . The time of the above algorithm is analyzed as follows. The tree can be rooted at s in $O(n)$ time. With $O(n)$ preprocessing time, the nearest common ancestor of any pair of vertices can be found in $O(1)$ time [8,14]. Thus, the preprocessing can be done in $O(k + n)$ time. The values of $xcount(v)$ and $ycount(v)$ can all be computed in $O(k)$ time. The computations at each vertex can be done in constant

time. Thus, the value of $\phi(u, v)$ for every ordered pair of adjacent vertices u and v in T can be computed in $O(k + n)$ time.

Once the function ϕ is computed, we can then compute the function $b(u, v)$ for every ordered pair of adjacent vertices u and v in T . This takes $O(n)$ time, since there are only $O(n)$ such pairs and it takes $O(1)$ time for each pair. After the function b is computed, we then generate a balancing move $(u, v, 0)$ for every $b(u, v) > 0$ and decrease $b(u, v)$ by one. This again takes $O(n)$ time. We now describe an algorithm that, given the modified function b , generates the remaining portion of a set of $O(k + n)$ balancing moves.

Again, we assume that the tree T is rooted at s . To avoid handling the root s as a special case, a new vertex s' is added. Let s' be the parent of s in T , and let $b(s, s') = b(s', s) = 0$. We first partition the $b(u, v)$'s into two sets B_1 and B_2 . Those $b(u, v)$'s with u a parent of v are in B_1 , and the others are in B_2 . We show how to generate a set of balancing moves with the set B_1 . A similar algorithm can be used to generate balancing moves with the set B_2 .

The procedure *generate-moves* builds a list of vertices, $L(v)$ for each vertex v in T . For each list $L(v)$, there is a counter associated with it. The counter, which is denoted by $|L(v)|$, is the number of vertices in that list. For each list operation, such as append, delete or merge, the value of the counter will be updated accordingly. This takes only $O(1)$ time for each list operation. We assume that this operation is included in the list operations and will not be explicitly stated. After B_1 is set up, we then call procedure *generate-moves* with parameters s and s' .

Procedure *generate-moves*(v, u), where u is the parent of v , does the following. If v is a leaf, then append $b(v, u)$ copies of v to $L(v)$. Otherwise, v is not a leaf, and

Figure 2: An example of the problem with a set of balancing moves.

we do the following. First, for each child w of v , call *generate-moves*(w, v). Second, let $L(v)$ be the concatenation of the list $L(w)$ for every child w of v . Third, if $b(v, u) > |L(v)|$ then append $b(v, u) - |L(v)|$ copies of v to $L(v)$. If $b(v, u) < |L(v)|$, then pop $|L(v)| - b(v, u)$ elements from the list $L(v)$, and for each element x popped from $L(v)$, generate a balancing move $(x, v, 0)$.

Figure 2 shows a set of balancing moves generated by the above algorithm for the example shown in Figure 1. Note that $\phi(1, 3) = \phi(3, 1) = 0$. Two balancing moves, one for each direction, are generated on this edge to make sure that each edge is traverse at least once. Since $\phi(7, 3) = \phi(3, 6) = 1$ and $\phi(3, 7) = \phi(6, 3) = 0$, balancing moves $(6, 3, 0)$ and $(3, 7, 0)$, are generated. Since $\phi(4, 1) = \phi(1, 0) = \phi(0, 2) = \phi(2, 5) = 0$ and $\phi(1, 4) = \phi(0, 1) = \phi(2, 0) = \phi(5, 2) = 2$, $b(4, 1) = b(1, 0) = b(0, 2) = b(2, 5) = 2$. One balancing move from u to v for each edge (u, v) in $\{(4, 1), (1, 0), (0, 2), (2, 5)\}$ are generated. Two additional balancing moves on these edges, one from 4 to 0 and the other from 0 to 5, are generated by the

procedure *generate-moves*.

Note that balancing moves $(4, 0, 0)$ and $(0, 5, 0)$ can be replaced by one balancing move $(4, 5, 0)$, but our algorithm did not do that. Since this can only reduce the number of balancing moves by one half, the order of the number of balancing moves generated is the same.

Lemma 1 *Given an instance P for the motion planning problem on trees, the balanced graph D for P can be computed in $O(k + n)$ time.*

Proof It is sufficient to show that there are only $O(k + n)$ balancing moves generated, and it can be done in $O(k + n)$ time. Note that there are only $O(n)$ balancing moves that are not generated by the algorithm *generate-moves*. We count the number of balancing moves generated by the algorithm *generate-moves*.

Since each connected component of D is Eulerian, we can list the arcs, hence the moves, in that component by finding an Euler tour. Let D_i be a connected component in D and k_i be the number of arcs in D_i . Then there are at most $\lceil k_i/3 \rceil$ balancing moves in D_i directed towards s , and at most $\lceil k_i/3 \rceil$ balancing moves in D_i directed away from s . Otherwise, there would be two balancing moves adjacent to each other such that both are directed towards s , or both are directed away from s , or the first is directed away from s and the second is directed towards s . The first two cases are excluded by the actions of algorithm *generate-moves*, and the latter by the fact that no cycles of balancing moves are generated by *generate-moves*. Note that there are $q \leq \min\{k, n\}$ strongly connected components in D , the number of carrying moves is k , and the number of balancing moves generated before the call to *generate-moves* is $O(n)$. Therefore, the number of balancing moves generated by *generate-moves* can be at most $O(k + n)$.

We analyze the time complexity of the above algorithm as follows. At each vertex, the algorithm spends $O(d_v)$ time in manipulating the lists, where d_v is the degree of v in T . It spends a time proportional to the number of elements added to $L(v)$ and the number of balancing moves generated at each vertex. Since the algorithm generates $O(k + n)$ balancing moves, the total time to generate the balancing moves is $O(k + n)$. Since every element in $L(v)$ is deleted when generating a balancing move, the total time to add all elements to $L(v)$ is $O(k + n)$. Therefore, the balancing moves can be generated in $O(k + n)$ time. ■

A motion planning problem is *balanced* if none of the moves in the balanced graph D are balancing moves. Given an instance P , we first construct a set of balancing moves B by the algorithm as described above. For each balancing move from x to y in B , add an object $o_{x,y}$ to O with initial vertex x and destination vertex y . Let the resulting set of objects be O' . The new instance P' with the objects O' is called the balanced version of the original problem P . We show that adding the balancing moves will not increase the cost of the transportation of the original problem.

Lemma 2 *The costs of optimal transportations for P and its balanced version P' are equal.*

Proof Given a transportation Q' for P' , we can obtain a transportation Q for P from Q' by replacing each move (x, y, c) such that $c \notin O$ with a non-carrying move $(x, y, 0)$.

On the other hand, let Q be an optimal transportation for P . A transportation Q' of P' can be obtained from Q as follows. First, construct a graph $D'(Q)$

with vertex set V such that there is an arc $\langle x, y \rangle$ labeled c if and only if there is a move (x, y, c) in Q . Since Q is a transportation, $D'(Q)$ is Eulerian. Second, replace every arc $\langle x, y \rangle$ in $D'(Q)$ with label 0 by a set of arcs $\langle v_i, v_{i+1} \rangle$ labeled 0, $0 \leq i \leq r-1$, where $x = v_0, v_1, \dots, v_r = y$ are the list of vertices on the path from x to y in T . The modified $D'(Q)$ will remain Eulerian. Third, for each balancing move from x to y , replace a set of moves $(u_i, u_{i+1}, 0)$, $0 \leq i \leq t$ by a move $(x, y, o_{x,y})$, where $x = u_0, u_1, \dots, u_{t+1} = y$ are the list of vertices on the path from x to y in T . These moves must exist since every valid transportation must traverse an edge (u, v) from u to v , $b(u, v)$ times without carrying an object. Note that $D'(Q)$ will remain Eulerian under this operation. It follows that an Euler tour of $D'(Q)$ starting with vertex s , will form the basis of a transportation for P' . Since the split and merge of moves will not change the total cost of the transportation, the cost of Q' is equal to the cost of Q . ■

We have shown how to generate a balanced version of the problem. The constructive proof of the lemma that the costs of the optimal transportations of P and its balanced version P' are equal gives a method to translate a transportation for P' into a transportation for P . That is, a transportation for the original problem P can be obtained from the transportation of the balanced problem P' by replacing each move (x, y, c) with $c \notin O$ by $(x, y, 0)$. In the following sections, we discuss how to compute a transportation for the balanced version of the problem.

3 Nonpreemptive Motion Planning in Trees is Hard

In this section, we show that motion planning problem in trees is NP-complete when objects cannot be dropped at the intermediate vertices during the transportation. Note that this version of the problem is a special case of the stacker-crane problem [5], in which an arc can have arbitrary positive weight.

Given an instance P , we first construct the balanced graph D . If D has only one nontrivial component then any Euler tour starting from s defines an optimal transportation for P . Note that every transportation must traverse all the carrying arcs and all the balancing arcs of the balanced graph D . Since these arcs are fixed for a given instance, we are interested in finding a set of linking arcs with minimum total weight to link the nontrivial components of D . We show that this is hard by reducing to it the Steiner tree problem in bipartite graphs [6]. We first define the two problems formally.

The Steiner tree problem in bipartite graphs is: Given a bipartite graph $G = (V, E)$ with bipartition R and $V - R$ and a positive integer $|R| - 1 \leq k_1 < |V|$, decide whether there is a subtree of G that spans at least the vertices in R and has at most k_1 edges [6].

The decision version of motion planning problem in trees is: Given an instance P and a positive integer k_2 , decide whether there is a transportation of cost at most k_2 for the instance P .

We first show that no optimal transportation of a balanced problem can traverse an edge more than once without carrying an object. This will be used to show that the motion planning problem on tree without drops is NP-complete.

Lemma 3 *No optimal transportation of a balanced problem can traverse an edge in the same direction more than once without carrying an object.*

Proof Given a balanced problem P , assume that there is an optimal transportation Q that traverses an edge (u, v) , in the direction from u to v , $r > 1$ times without carrying an object. Then it must also traverse the same edge in the opposite direction r times without carrying an object. We first decompose each of these moves into three moves—the move before the edge (u, v) , the move on the edge (u, v) and the move after the edge (u, v) . We then construct a directed graph $D'(Q)$ with vertex set V . There is an arc $\langle x, y \rangle$ labeled c if and only if (x, y, c) is a move of Q . It is clear that $D'(Q)$ is Eulerian. We then delete $r - 1$ copies of $\langle u, v \rangle$ and delete $r - 1$ copies of $\langle v, u \rangle$ from $D'(Q)$. Since we delete the same number of $\langle u, v \rangle$ and $\langle v, u \rangle$ arcs in $D'(Q)$, after deleting these arcs, $D'(Q)$ must be Eulerian. A transportation of smaller cost than Q can be obtained by finding an Euler tour of $D'(Q)$. This is a contradiction. ■

Theorem 1 *Nonpreemptive motion planning on tree is NP-complete.*

Proof Nonpreemptive motion planning on a tree is in NP since there is a polynomial time algorithm that nondeterministically generates a sequence of moves (u_i, v_i, c_i) , $0 \leq i \leq r$, and then checks whether this is a valid transportation. If it is, then the algorithm checks whether the cost of this transportation is at most k_2 .

To show that the problem is hard, we reduce the Steiner tree problem in bipartite graphs to it. Given an instance of the Steiner tree problem with underlying graph $G = (V, E)$, a subset of vertices R and a number k_1 , we construct an instance P of the motion planning problem in trees as follows.

The graph T is constructed by the following procedure. Each vertex v in G has associated with it a number $\#(v)$. At the end of the computation a vertex v in G will have $\#(v)$ copies, in addition to itself, in T . These vertices will be denoted by $v^1, v^2, \dots, v^{\#(v)}$, respectively. Initially, let T be a spanning tree of G and let $\#(v) = 0$ for each $v \in V$. For every edge $e = (u, v)$ of G not in T , since there are no edges joining two vertices in $V - R$, at least one end is in R . Assume, without loss of generality, that $v \in R$. Increase $\#(v)$ by one and add a new vertex $v^{\#(v)}$ and a new edge $(u, v^{\#(v)})$ to T . (We say that the vertex $v^{\#(v)}$ is added to T due to v .) At this point, for each edge in G , there is one and only one corresponding edge in T . We let the weights of these edges be 1.

We add additional vertices and edges to T . For each vertex v in R with $\#(v) = 0$, we let $\#(v) = 1$ and create a new vertex v^1 and a new edge (v, v^1) with weight 0 in T . We need these vertices and edges to set up the required moves to force the vehicle to visit these vertices.

The set of objects O , together with an initial vertex x_j and a destination vertex y_j for each object $j \in O$, is constructed as follows. First, set the object index j to 1. For each vertex $v \in R$, for $t = j, j + 1, \dots, j + \#(v)$, add an object t with initial vertex v^{t-j} , destination vertex v^{t-j+1} , where $v^0 = v^{\#(v)+1} = v$. Then, set $j = j + \#(v) + 1$.

Finally, let s be any vertex in R and $k_2 = 2k_1 + c(D)$, where $c(D)$ is the total weight of the arcs of the balanced graph D .

This completes the description of the transformation. Note that for each vertex v in T , if we identify it and all the vertices created due to v , then the resulting graph is isomorphic to G . In other words, we split vertices to get a tree

and set up required moves in M to force the vehicle to visit at least the vertices in components of T corresponding to the vertices R in G .

It is clear that T has at most $|E| + |R| + 1$ vertices and has at most $|E| + |R|$ edges. Thus the reduction can be done in polynomial time.

We claim that the graph G has a Steiner tree that spans at least the vertex set R with cost at most k_1 if and only if the motion planning problem P has a transportation of cost at most k_2 .

Let S be a subtree of G that spans at least the vertex set R and has cost $x \leq k_1$. We observe that the instance of the motion planning problem we generated is a balanced problem. Initialize D' to be the balanced graph for P . For each edge (u, v) of S we add two linking arcs $\langle u, v \rangle$ and $\langle v, u \rangle$ to D' . Since S spans at least all vertices in R , all components must be connected by these linking arcs. Furthermore, D' must be Eulerian, since we add the same number of arcs into a vertex as the number of arcs out of that vertex. Since s is originally in some connected component of D' , an Euler tour of D' starting at vertex s defines a transportation for P . The total cost of the transportation is $2x + c(D) \leq 2k_1 + c(D) = k_2$.

On the other hand, let Q be a transportation for P of cost $y \leq k_2$. It is sufficient to show that there is a subgraph that spans at least the vertex set R and has cost at most k_1 . Let E' be the set of edges in T such that the vehicle traverses them without carrying any object. Let E'' be the set of edges in G that correspond to E' , and let S be the induced graph $G[E'']$. Since every vertex in R corresponds to a connected component of D , and the connected components of D must be connected by some linking arcs to form a transportation, S must span at least the vertex set R . By Lemma 3, the vehicle traverses these edges exactly twice without

carrying any object. The total weight of S is $(y - c(D))/2 \leq (k_2 - c(D))/2 = k_1$. ■

The Steiner tree problem and the motion planning problem in trees are actually equivalent under polynomial reduction. Given a balanced motion planning problem P , let H be a graph obtained from T by identifying the vertices x_j and y_j for each object $j \in O$. We call the graph H the *arc-identified graph*. We partition the vertices of H into two subsets C' and C'' . Those vertices which are not s or x_i or y_i for some i will be in C'' and the others will be in C' . Let S^* be an optimal Steiner tree of H that spans at least the vertex set C' , and $c(S^*)$ be the cost of S^* .

Theorem 2 *The cost of optimal transportation of a balanced motion planning problem P is $c(D) + 2c(S^*)$.*

Proof Given an instance P , construct a graph H and partition the vertices of H into C' and C'' as described above. The argument that establishes the claim in Theorem 1 shows that P has a transportation of cost $c(D) + 2x$ if and only if there is a tree of cost x in H that spans at least the vertex set C' . ■

4 Approximation Algorithms

In this section we present three approximation algorithms for nonpreemptive motion planning in trees. We measure the performance of the approximation algorithm by its *worst-case performance ratio*, which is defined as the supremum over all instances of the ratio of the cost of the transportation generated by the algorithm to the cost of an optimal transportation.

The first algorithm, *use-Steiner-tree*, runs in $O(k + n)$ time, and generates a transportation whose cost is at most $3/2$ times the optimal cost. The second algorithm, *use-spanning-tree*, runs in $O(k + n \log \beta(n, q))$ time, where $\beta(n, q) = \min\{i \mid \log^i(n) \leq n/q\}$. It generates a transportation whose cost is at most $4/3$ times the optimal cost. The third approximation algorithm is a combination of the above two algorithms. That is, it runs both algorithms on the same instance and then chooses the transportation with smaller cost. We show that the cost of the transportation obtained in this way is within $5/4$ time the cost of an optimal transportation.

Our approximation algorithms use linear time algorithms for the computation of a Steiner tree and the shortest path in some special graphs. Let K_4 be a complete graph of four vertices. A graph G is homeomorphic to K_4 if we can obtain K_4 from G by repeatedly taking a subgraph of G or replacing a vertex v of degree two and its incident edges by a single edge joining the two vertices adjacent to v . There is a linear time algorithm for Steiner tree problem in graphs that are not homeomorphic to K_4 [15]. The single source shortest path problem in graphs that are not homeomorphic to K_4 can also be solved in linear time [9]. We first prove the following lemma, which shows that some graphs generated in the intermediate

steps of our approximation algorithms are not homeomorphic to K_4 .

One of the operations in our algorithms is to identify vertices together. For notational convenience, we shall let each vertex in the resulting graph be a set of vertices that have been identified together. A singleton set will represent a vertex participating in 0 identifications. Given a graph G and a subset U of vertices in G , let G' be a graph obtained from G by vertex identifications. For distinct vertices X and Y in G' , there is an edge (X, Y) in G' if and only if there are two vertices $x \in X$ and $y \in Y$ such that (x, y) is an edge in G .

We show that if we identify a subset of vertices in T then the resulting graph G is not homeomorphic to K_4 . In fact, we show a more general case than just identifying a set of vertices.

Lemma 4 *Let $V_1, V_2, \dots, V_r$, $r > 0$, be pairwise disjoint subsets of vertices in a tree T . Suppose that for any pair of distinct vertex sets V_i and V_j in $\{V_1, V_2, \dots, V_r\}$, for any $v_{i1}, v_{i2} \in V_i$ and $v_{j1}, v_{j2} \in V_j$, the path from v_{i1} to v_{i2} in T shares no edges in common with the path from v_{j1} to v_{j2} . Then the graph G obtained from T by identifying the vertices in V_k , $1 \leq k \leq r$, is not homeomorphic to K_4 .*

Proof Suppose that G is homeomorphic to K_4 . We color the edges in T with $r + 1$ colors, $0, 1, \dots, r$. An edge is colored i , $1 \leq i \leq r$, if and only if there are two vertices $v_{i1}, v_{i2} \in V_i$ such that e is in the path from v_{i1} to v_{i2} in T . Otherwise, it is colored 0.

Since T is a tree, every cycle of G must contain at least one of the vertices $V_1, V_2, \dots, V_r$ of G .

It is clear that for any color $i > 0$ the subgraph of T induced by the edges of color i is connected. Thus, any closed trail in G that contains edges of more than

one color must visit some vertex that is adjacent to edges of different colors more than once. Thus, any closed trail that consists of more than one color cannot be a simple cycle. In other words, every simple cycle of G must contain edges of the same color. Since every pair of cycles in K_4 share a common edge, all the edges of K_4 must have the same color. Without loss of generality, suppose it is color 1.

Consider the graph G' created from T by identifying only the vertices in V_1 . Then G' is homeomorphic to K_4 . The graph $G' - V_1$ is a subgraph of T . On the other hand, removing a vertex from K_4 yields a graph that is homeomorphic to K_3 . This is a contradiction. ■

4.1 An Algorithm Based on a Steiner Tree

In this subsection, we first show that a special case of the problem can be solved in linear time. We then present the first approximation algorithm *use-Steiner-tree*. The algorithm first transforms the input by contracting edges that are used by more than one nontrivial component. The resulting problem can then be solved in linear time. The union of the edges in the Steiner tree of the resulting problem and those edges that are used by more than one component is then used to approximate the Steiner tree of the original problem.

Recall that the arc-identified graph H is a graph obtained from T by identifying the vertices x_j and y_j for each object $j \in O$. We label each vertex in H by $\{w_1, w_2, \dots, w_i\}$ if it is the results of identifying the vertices $w_1, w_2, \dots, w_i$. Recall that we partition the vertices of H into two sets C' and C'' . A vertex $\{w_1, w_2, \dots, w_i\}$ of H is in the set C' if it contains at least one of the vertices s or x_j or y_j for some $j \in O$, otherwise, it is in the set C'' .

Given a problem P , let j be an object in O . Define $V_T(j)$ to be a set of vertices on the path from x_j to y_j , including x_j and y_j in T . For each nontrivial component D_i of the balanced graph D , let $V_T(D_i)$ be the union of $V_T(j)$ over all objects j in the component D_i . Define $E_T(D_i)$ to be the set of edges on the path from x_j to y_j for all objects j in the component D_i . We say that an edge e is used by a component D_i if $e \in E_T(D_i)$, and a vertex v is used by a component D_i if $v \in V_T(D_i)$.

Lemma 5 *Let P be an instance of nonpreemptive motion planning on trees such that every edge in T is used by only one nontrivial component. Then a Steiner tree spanning these components can be found in linear time.*

Proof The graph H can be computed in $O(k + n)$ time. Since every edge of T is used by only one component, H is not homeomorphic to K_4 , by Lemma 4. Thus, Wald and Colbourn's algorithm [15] can be used to compute S^* in $O(n)$ time. ■

If there are edges used by more than one component, then we first find this set X of edges. We then generate a new instance P' from P as follows. The graph T' is obtained from T by contracting edges in X . We denote a vertex in T' by a set of vertices in T that gets identified in the edge contractions. For each object j in P with initial vertex x_j and destination vertex y_j , let W_x and W_y be the two vertices in T' that contain the vertices x_j and y_j , respectively. Add an object j with initial vertex W_x and destination vertex W_y to P' if $W_x \neq W_y$. Finally let the starting vertex be W_s , the vertex in T' that contains the vertex s .

It is clear that the resulting problem satisfies the condition that each edge of T' is used by only one component. We then compute a Steiner tree spanning the

nontrivial components. We use the edges in the Steiner tree plus the set of edges used by more than one component to connect all the components of the original problem. The algorithm is summarized as follows.

ALGORITHM *use-Steiner-tree*

INPUT: An instance P of the motion planning problem on trees.

OUTPUT: A transportation Q' for P .

METHOD:

1. Compute the balanced graph D for P .
2. Compute a set of edges X in T that are used by more than one component.
3. Generate the instance P' from P that results from contracting edges in X .
4. Compute the arc-identified graph H for P' .
5. Compute an optimal Steiner tree S' for H that spans at least the set C' .
6. Initialize $D_{S'} = D$, and for each edge $(u, v) \in S' \cup X$, add two arcs $\langle u, v \rangle$ and $\langle v, u \rangle$ to $D_{S'}$.
7. Find an Euler tour of $D_{S'}$ starting from s , and output the result as a transportation Q' .

The rest of this subsection shows that the above algorithm can be implemented to run in linear time, and the cost of the transportation Q' is at most $3/2$ times the cost of an optimal transportation.

We first present a linear time algorithm to compute a set of edges that are used by more than one component. First, let s be the root of T . Second, decompose

each arc $\langle u, v \rangle$ of D into two arcs $\langle u, t \rangle$ and $\langle t, v \rangle$, if t , the nearest common ancestor of u and v , is not u or v . Third, delete each arc $\langle u, v \rangle$ if it is from a vertex u to its descendent v . Since each nontrivial component of the graph D is Eulerian, an edge will be used in more than one component by arcs in the set of remaining arcs if and only if it is used in more than one component by arcs in the original set of arcs. Fourth, call the following recursive procedure *find-arcs* with parameter s .

The algorithm *find-arcs*(v) decides whether each edge in the subtree rooted at v is used by more than one component by constructing in turn lists of arcs $L(w)$, for each vertex w in the subtree rooted at v . Let t be the parent of w . An arc $\langle x, y \rangle$ of D will be in $L(w)$ if and only if the edge (w, t) is in the path from x to y in T .

To decide whether edge (w, t) is used by more than one component in constant time, the arcs in $L(w)$ are grouped into lists with all arcs in one list belonging to the same components. A two level linked list is used to store this information, with lists on each level being doubly linked. Each entry in the top level list stores a component name and a pointer to the first entry of the second level list, which is a list of some of the arcs in this component. Each arc appears at most once in at most one second level list. A component name may appear more than once in the first level list, but no two adjacent entries in the first level list can have the same component name. With this data structure, the edge (v, u) is used by more than one component if and only if $L(v)$ contains at least two entries in its top level list. This can be determined in constant time once the list $L(v)$ has been computed.

The recursive procedure *find-arcs*(v) does the following. If v is a leaf, then initialize $L(v)$ to be empty list, and then for each arc $\langle v, x \rangle$ in the component D_i inserts the arc into $L(v)$. Note that when an arc is inserted into $L(v)$, the address of

that arc in $L(v)$ is associated with the arc inserted. With this address, the deletion of that arc can be done in constant time.

Otherwise, v is not a leaf. First, *find-arcs* initializes $L(v)$ to be an empty list. Second, for each child w of v , *find-arcs*(w) is called, and $L(w)$ is merged into $L(v)$. Note that if there are two adjacent top level entries with the same component name after merging $L(w)$ into $L(v)$, their second level lists should be merged and one of the top level entries should be deleted. Third, each arc $\langle x, v \rangle$ is deleted from $L(v)$. If this arc is the only arc in the list, then the top level entry that points to this list is deleted. The deletion may cause two adjacent top level entries to have the same component name. If this is the case, then the two entries are merged. Fourth, each arc $\langle v, x \rangle$ is inserted into $L(v)$.

In either case, the edge (v, u) is used by more than one component if and only if $L(v)$ has at least two entries in its top level list.

Lemma 6 *The set of edges that are used by more than one nontrivial component can be computed in $O(k + n)$ time.*

Proof The tree rooted at s can be computed in $O(n)$ time. With $O(n)$ preprocessing time, the nearest common ancestor of any pair of vertices in T can be determined in $O(1)$ time [8,14]. The insertion of an arc can be done in $O(1)$ time since we allow a component name to appear more than once in the top level link list as long as none of them are adjacent. The deletion of an arc can also be done in $O(1)$ time since we know the address of that arc in the list, and the list is doubly linked. The merge of two lists at a non-leaf vertex can be done in $O(1)$ time. To determine if an edge is used by more than one component can be done in $O(1)$ time. Therefore, the algorithm can be implemented to run in $O(k + n)$ time. ■

Lemma 7 *Let S' be an optimal Steiner tree for the arc-identified graph H that spans at least the vertex set C' . Then $c(S') \leq c(S^*)$.*

Proof It is sufficient to show that there is a subgraph T' in H that spans at least the vertex set C' and $c(T') \leq c(S^*)$. Let X be the set of edges that are used by more than one component. For each edge (u, v) in X such that both u and v are in S^* , we identify the vertex u and the vertex v in S^* . We then delete all loops as well as all multiple edges except the one with lowest cost in the contracted version of S^* . Let the resulting graph be T' . It is clear that T' spans at least the vertex set C' since S^* is a Steiner tree that spans the vertex set C' . The cost of T' is no more than the cost of S^* , since we only delete edges from it. ■

Theorem 3 *The approximation algorithm use-Steiner-tree can be implemented to compute a transportation Q' with $c(Q') \leq (3/2)c(Q^*)$ in $O(k + n)$ time.*

Proof The worst-case performance ratio is established as follows. Since X is the set of edges used by more than one component, the vehicle must traverse all edges in X at least four times, twice in each direction. Note that the edges in S' and the edges in X are disjoint. Therefore, $4c(X) + 2c(S') \leq c(D)$. Recall that an optimal transportation of P has cost $c(D) + 2c(S^*)$. Thus

$$\begin{aligned} \frac{c(Q')}{c(Q^*)} &= \frac{c(D) + 2c(X) + 2c(S')}{c(D) + 2c(S^*)} \leq \frac{c(D) + [c(D)/2 - c(S')] + 2c(S')}{c(D) + 2c(S^*)} \\ &\leq \frac{3c(D)/2 + c(S')}{c(D) + 2c(S^*)} \leq \frac{3c(D)/2 + c(S^*)}{c(D) + 2c(S^*)} \leq \frac{3}{2} \end{aligned}$$

By Lemma 6, the set X of edges that are used by more than one component can be computed in $O(k + n)$ time. By Lemma 1, the balanced graph for P , as well as the balanced graph for P' , can be computed in $O(k + n)$ time. It is easy to

Figure 3: An Example of $N = 4$.

see that the graph H can be computed in $O(k + n)$ time. The Steiner tree S' can be computed in $O(n)$ time. Finding and outputting Q' takes $O(k + n)$ time. All the other steps take $O(n)$ time. Therefore, the time complexity of the algorithm is $O(k + n)$. ■

An example for which the algorithm *use-Steiner-tree* realizes the bound $3/2$ can be constructed as follows. Let T be a tree of $3r + 1$ vertices, $0, 1, \dots, 3r$, where r is an even number greater than one. The vertex 0 is adjacent to each of the vertices $1, 2, \dots, r$ by an edge of cost a . Each of the vertices i , $1 \leq i \leq r$, is adjacent to two more vertices $r + 2i - 1$ and $r + 2i$ by edges of cost b . In this example, we assume that $b < (a/2)$.

The set of objects is $O = \{1, 2, \dots, 2r\}$. Assume that $v_{3r+1} = v_{r+1}$. For $1 \leq i < r$, the initial vertex for object $2i - 1$ is $r + 2i$, and the destination vertex is $r + 2i + 1$. For $1 \leq i < r$, the initial vertex for object $2i$ is $r + 2i + 1$, and the destination vertex is $r + 2i$. Finally, let $s = r + 1$. An example with $r = 4$ is shown in Figure 3.

The balanced graph D would have $2r + 1$ strongly connected components, namely $\{0\}, \{1\}, \dots, \{r\}, \{r + 2, r + 3\}, \{r + 4, r + 5\}, \dots, \{3r, r + 1\}$.

It is easy to see that $c(D) = 4(a + b)r$. An optimal Steiner tree of the arc-identified graph H that spans at least the vertex set C' would not include the Steiner vertex $\{0\}$, but would include vertices $\{i\}$, $1 \leq i < r$. The cost of the Steiner tree is $2b(r - 1)$. Hence the optimal solution Q^* would have a cost equal to $4(a + b)r + 4b(r - 1) = 4ar + 8br - 4b$.

The edges that are used by more than one component are the edges adjacent to 0, and have a total cost of ar . The optimal Steiner tree after the contraction of these edges has cost br . The solution Q' computed by *use-Steiner-tree* would have cost $4(a+b)r + 2ar + 2br = 6ar + 6br$. The ratio,

$$\frac{c(Q'')}{c(Q^*)} = \frac{6ar + 6br}{4ar + 8br - 4b},$$

approaches $3/2$ as b approaches 0.

4.2 An Algorithm Based on Minimum Spanning Tree

In this subsection, we present the algorithm *use-spanning-tree*. The algorithm first computes a graph on the vertex set consisting of the nontrivial components. Each edge of the graph represents the shortest path between a destination vertex in one component and an initial vertex in the other component. The minimum spanning tree of the graph is then computed and is used to approximate the Steiner tree of H . We use the construction of [11,13] in the minimum spanning tree computation so that we compute $O(n)$ edges rather than $\theta(q^2)$ edges of this graph whenever n is $o(q^2)$. Noting the special structure of our problem, we get a faster running time than that claimed in [11,13]. Finally, we show that the cost of the transportation generated by the algorithm *use-spanning-tree* is at most $4/3$ times the cost of an optimal transportation.

Given an instance P we define the *component graph* H' , a weighted complete graph, on vertex set C' as follows. For each pair of vertices $D_i \neq D_j$ in H' , the weight of the edge (D_i, D_j) is equal to the minimum distance $d(u, v)$ in T , where u is a vertex in $IP(D_i)$ and v is a vertex in $IP(D_j)$. (Recall that in a balanced problem, a destination vertex must also be an initial vertex.) The algorithm *use-*

spanning-tree uses the minimum spanning tree of H' to approximate the Steiner tree of H that spans at least the vertex set C' . To make the algorithm efficient, a graph H'' that has only $O(n)$ edges and contains a minimum spanning tree of H' is constructed. We call H'' the *sparse component graph*.

We describe briefly the procedure in [11,13] to compute graph H'' . First, construct a graph G from T by adding a new vertex s_0 and an edge (s_0, v) for each vertex v that is either s , or x_j or y_j for some object $j \in O$. Let the cost on each of these new edges be zero. Let U be the shortest path tree with s_0 as the source vertex. Without loss of generality, assume that each edge (s_0, v) is in U , for every edge added to T . Then delete from U the vertex s_0 and the edges incident on it. Now U is a forest, and each tree is rooted at a vertex v that is either s or x_j or y_j for some $j \in O$. For each such tree rooted at v , color the vertices and the edges in that tree with color i , if D_i is the nontrivial component that contains the vertex v . Those edges in T but not in U are colored with color 0.

Next generate the set of edges for H'' as follows. Let (x, y) be an edge of color 0, such that the vertices x and y have different colors i and j , respectively. Let u be the root of the subtree with color i and v be the root of the subtree with color j . For each such edge (x, y) , generate an edge (D_i, D_j) with cost $t = d(u, x) + c(x, y) + d(y, v)$, where $d(a, b)$ is the total costs on the edges from a to b . Since there are $O(n)$ edges of color 0, at most $O(n)$ edges are generated. Sort these edges by their endpoints lexicographically, and then scan the sorted edge list. For each edge (D_i, D_j) in the list, delete all but the one with smallest cost and add it to H'' .

Clearly, the graph H'' has only $O(n)$ edges. It is shown in [11,13] that H''

Figure 4: An Example of the graph H' and H'' .

contains a minimum spanning tree of H' .

An example of the graph H' and the graph H'' for an instance shown in Figure 4(a) are shown in Figure 4(b) and 4(c). There are four nontrivial components, namely, $D_1 = \{0\}$, $D_2 = \{1, 7\}$, $D_3 = \{5, 8\}$, $D_4 = \{4, 6, 9\}$. We use the notation $T[v_1, v_2, \dots, v_r]$ to denote a subtree of T induced by the vertices $v_1, v_2, \dots, v_r$. The forest U contains one tree of color 1, $T[0, 2]$, two trees of color 2, $T[1]$ and $T[7]$, two trees of color 3, $T[5, 3]$ and $T[8]$, and three trees of color 4, $T[4]$, $T[6]$ and $T[9]$. The edge $(0, 1)$ in T is adjacent to two trees of U with color 1 and color 2, respectively. An edge (D_1, D_2) of H'' is generated and the cost of this edge 9. The edge $(2, 3)$ of T is adjacent to two trees of U with color 1 and color 3, respectively. An edge (D_1, D_3) of H'' is generated, and the cost of this edge is 3. The other edges of H'' are generated in a similar way.

Lemma 8 *The graph H'' can be computed in $O(n)$ time.*

Proof Note that if we delete the vertex s_0 from G then the resulting graph is T . Thus, G is not homeomorphic to K_4 . Using an algorithm of Hassin and Tamir [9], the single-source shortest path problem in G can be solved in linear time. Each edge can be colored in $O(1)$ time. The lexicographic sorting of the edges can be done in $O(n)$ time. The elimination of parallel edges, except one of the minimum cost can be done in $O(n)$ time. ■

After the graph H'' is computed, we can then compute a minimum spanning tree T'' for H'' . Recall that each edge of H' and H'' represents a path in T . For each

edge e in T'' , let p_e be the corresponding path in T . The set of paths corresponding to the edges of T'' is then used to link the components. Since the paths are not necessarily edge disjoint, we do not want to add two arcs $\langle x, y \rangle$ and $\langle y, x \rangle$ for each path p_e with origin x and terminus y . Let S'' be the set of edges that are in some path that corresponds to some edge in T' . Adding two arcs $\langle u, v \rangle$ and $\langle v, u \rangle$ for each edge (u, v) in S'' is sufficient to connect all the components.

We show how to compute S'' in $O(n)$ time. First, let s be the root of T . Second, let A be the set of arcs such that there is an arc $\langle x, y \rangle$ in A if and only if there is an edge e in T'' such that the corresponding path in T of this path is from x to y . Third, for each arc $\langle x, y \rangle$ in A split it into two arcs $\langle x, t \rangle$ and $\langle t, y \rangle$ if the nearest common ancestor t of x and y is an intermediate vertex of the path from x to y in T . Fourth, compute the function $\phi(u, v)$ for each pair of adjacent vertices u and v . Finally, add edge (u, v) to S'' if $\phi(u, v) + \phi(v, u) > 0$. Since there are $O(q)$ arcs, the function $\phi(u, v)$ for all adjacent pair of vertices u and v can be computed in $O(n)$ time. Thus, S'' can be computed in $O(n)$ time.

We summarize the algorithm as follows.

ALGORITHM *use-spanning-tree*

INPUT: an instance P of motion planning problem on trees.

OUTPUT: a transportation Q'' for P .

METHOD:

1. Compute the balanced graph D for P .
2. Compute the sparse component graph H'' .
3. Compute a minimum spanning tree T'' of the graph H'' .

4. Compute the graph S'' for T'' .
5. Initialize $D_{S''} = D$, and then for each edge $e = (u, v)$ in S'' add two arcs $\langle u, v \rangle$ and $\langle v, u \rangle$ to $D_{S''}$.
6. Find an Euler tour of $D_{S''}$ starting with the vertex s , and output the resulting transportation Q'' .

In the rest of this subsection, we prove that the cost of the solution obtained by the above approximation algorithm is at most $4/3$ times the cost of an optimal solution.

Lemma 9 *Let S^* be an optimal Steiner tree of the graph H that spans at least the vertex set C' . Then $c(D) \geq 2c(T) \geq 2c(S'')$, and $c(S'') \leq 2c(S^*)$.*

Proof (1): Since each edge in T must be covered by at least one arc in each direction in a balanced problem, $c(D) \geq 2c(T)$. Since in the worst case every edge of T would be marked by step 4 of the algorithm, $c(T) \geq c(S'')$.

(2): It is sufficient to show that there is a spanning subgraph T''' in the component graph H' with $c(T''') \leq 2c(S^*)$. Let the vertices of the arc-identified graph H be $v_1, v_2, \dots, v_{n'}$ and the vertices of H' be $D_1, D_2, \dots, D_{n''}$. Note that for each vertex in H' there is a vertex in H corresponding to it, so that $n'' \leq n'$. Without loss of generality, we assume that v_i corresponds to D_i for $1 \leq i \leq |C'|$. We first do a depth-first search in the tree S^* . Every time we visit a vertex we list that vertex. Note that each vertex v_i is listed $d + 1$ times, where d is the number of children of v_i . Let $v_{j_1}, v_{j_2}, \dots, v_{j_t}$ be the list of vertices obtained by the above procedure. Note that $(v_{j_l}, v_{j_{l+1}})$, $1 \leq l < t$, is an edge in S^* . Therefore, the above

list of vertices defines a list of edges in which each edge in S^* appears twice. Thus, the total weight of the edges associated with the list is equal to $2c(S^*)$. We then delete those vertices not in C' from the above list, and let the resulting list be $v_{j'_1}, v_{j'_2}, \dots, v_{j'_t}$. Let $D_{j'_1}, D_{j'_2}, \dots, D_{j'_t}$ be the list of vertices in the component graph H' corresponding to the list of vertices $v_{j'_1}, v_{j'_2}, \dots, v_{j'_t}$ in the arc-identified graph H .

We observe that each time we delete a vertex $v_{j_i} \in C''$ from the list, there is a path from $v_{j_{i-1}}$ to $v_{j_{i+1}}$ in H , which is a union of the path from $v_{j_{i-1}}$ to v_{j_i} and the path from v_{j_i} to $v_{j_{i+1}}$. This is because each component in C'' is a single vertex in T . Therefore the vertex list $D_{j'_1}, D_{j'_2}, \dots, D_{j'_t}$ defines an edge list in H' and the total cost of these edges is no more than $2c(S^*)$. Let T''' be the subgraph induced by the edges as defined by the list $D_{j'_1}, D_{j'_2}, \dots, D_{j'_t}$. It is easy to see that T''' spans all the vertices of H' since we start with a list of vertices that contains at least all vertices in C' and we delete only vertices in C'' . Clearly, T''' is connected and $c(T''') \leq 2c(S^*)$. ■

Theorem 4 *Let n be the number of vertices in T , k be the number of objects to be moved and Q^* an optimal solution. Then the approximation algorithm *use-spanning-tree* computes a transportation Q'' with $c(Q'')/c(Q^*) \leq 4/3$ in $O(k + n \log \beta(n, q))$ time, where $\beta(n, q) = \min\{i \mid \log^i(n) \leq n/q\}$.*

Proof The algorithm uses a minimum spanning tree of H' , instead of the Steiner tree S^* . Thus, $c(Q'') = c(D) + 2c(S'')$. By Theorem 2, $c(Q^*) = c(D) + 2c(S^*)$. By applying Lemma 9, we have

$$\frac{c(Q'')}{c(Q^*)} = \frac{c(D) + 2c(S'')}{c(D) + 2c(S^*)} = 1 + \frac{2c(S'') - 2c(S^*)}{c(D) + 2c(S^*)} \leq 1 + \frac{2c(S'') - c(S'')}{2c(S'') + c(S'')} = 1 + \frac{1}{3} = \frac{4}{3}.$$

The time complexity of *use-spanning-tree* is analyzed as follows. By Lemma 1, the balanced graph D can be constructed in $O(k + n)$ time. By Lemma 8, the graph H'' can be computed in $O(n)$ time, and it has q vertices and $O(n)$ edges. A minimum spanning tree of H'' can be computed in $O(n \log \beta(n, q))$ time, where $\beta(n, q) = \min\{i \mid \log^i(n) \leq n/q\}$. The graph S'' can be computed in $O(n)$ time. The linking arcs of the balanced graph D can be added in $O(n)$ time. Since the total number of arcs in D is $O(k + n)$, the transportation Q'' can be output in $O(k + n)$ time by finding an Euler tour of D starting from s . Thus, the time bound for *use-spanning-tree* is $O(k + n \log \beta(n, q))$. ■

An example for which the algorithm *use-spanning-tree* realizes the bound $4/3$ can be constructed as follows. Consider the example whose description at the end of the preceding subsection, shown in Figure 3 with $r = 4$. For this example, now assume that a is a small positive number that is much smaller than b . An optimal Steiner tree of the arc-identified graph H that spans at least the vertex set C' would include the Steiner vertex $\{0\}$, and $\{2i - 1\}$, $1 \leq i \leq r/2$. The cost of the Steiner tree is $a(r/2) + br$. Hence the optimal solution Q^* would have a cost equal to $4(a + b)r + (ar + 2br) = 5ar + 6br$.

The minimum distance between each pair of vertices in C' is $2b$. Hence the minimum spanning tree of the component graph H' would not include the vertex $\{0\}$. It is easy to show that the minimum spanning tree of H' would have cost $2b(r - 1)$. The solution Q'' computed by *use-spanning-tree* would have cost equal to $4(a + b)r + 4b(r - 1) = 4ar + 8br - 4b$. The ratio,

$$\frac{c(Q'')}{c(Q^*)} = \frac{4ar + 8br - 4b}{5ar + 6br},$$

approaches $4/3$ as r approaches ∞ and a approaches 0 .

4.3 A Mixed-Strategy Algorithm

It is interesting to show that, for any instance P , if we run both algorithms described above and select the one with smaller cost, then the transportation obtained in this way has a better worst-case performance bound than either of its constituent algorithms. Such an approach was termed a *mixed strategy* approach in [5]. We call our combined algorithm *use-both-trees*.

Theorem 5 *The approximation algorithm use-both-trees generates a transportation Q such that $c(Q)/c(Q^*) \leq 5/4$.*

Proof If $c(D) + 2c(S^*) \geq 4c(S'')$ then

$$\begin{aligned} \frac{c(Q'')}{c(Q^*)} &= \frac{c(D) + 2c(S'')}{c(D) + 2c(S^*)} = 1 + \frac{2c(S'') - 2c(S^*)}{c(D) + 2c(S^*)} \\ &\leq 1 + \frac{2c(S'') - c(S'')}{4c(S'')} = \frac{5}{4}. \end{aligned}$$

Otherwise, $c(D) + 2c(S^*) \leq 4c(S'')$. Note that $4c(X) + 2c(S') \leq c(D)$.

$$\begin{aligned} \frac{c(Q')}{c(Q^*)} &= \frac{c(D) + 2c(X) + 2c(S')}{c(D) + 2c(S^*)} \leq \frac{c(D) + [c(D)/2 - c(S')] + 2c(S')}{c(D) + 2c(S^*)} \\ &= \frac{3c(D)/2 + c(S')}{c(D) + 2c(S^*)} = \frac{3}{2} - \frac{3c(S^*) - c(S')}{c(D) + 2c(S^*)} \leq \frac{3}{2} - \frac{3c(S^*) - c(S'')}{4c(S'')} \\ &= \frac{3}{2} - \frac{2c(S^*)}{4c(S'')} = \frac{3}{2} - \frac{c(S'')}{4c(S')} = \frac{5}{4}. \end{aligned}$$

Since $c(Q) = \min\{c(Q''), c(Q')\}$, it follows that $c(Q) \leq (5/4)c(Q^*)$. ■

Acknowledgements.

We would like to thank Dan Gusfield and Chip Martel for bringing two references to our attention.

References

- [1] M. J. Atallah and S. R. Kosaraju. Efficient solutions to some transportation problems with application to minimizing robot arm travel. *SIAM Journal on Computing*, 17(5):849–869, 1988.
- [2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. The MacMillan Press, Ltd, London, England, 1976.
- [3] G. N. Frederickson. *A Note on the Complexity of a Simple Transportation Problem*. Technical Report CSD-TR-809, Department of Computer Science, Purdue University, West Lafayette, Indiana, September 1988.
- [4] G. N. Frederickson and D. J. Guan. *Preemptive Ensemble Motion Planning on a Tree*. Technical Report CSD-TR-865, Department of Computer Science, Purdue University, West Lafayette, Indiana, February 1989.
- [5] G. N. Frederickson, M. S. Hecht, and C. E. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7(2):178–193, May 1978.
- [6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, 1976.
- [7] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

- [8] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
- [9] R. Hassin and A. Tamir. Efficient algorithms for optimization and selection on series-parallel graphs. *SIAM Journal on Algebraic and Discrete Methods*, 7(3):379–389, July 1986.
- [10] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, Plenum Press, New York, 1972.
- [11] L. T. Kou and Y. K. Makki. An even faster approximation algorithm for the steiner tree problem in graphs. *Proceedings of Congressus Numerantium, Eighteenth Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, Florida)*, 59:147–154, 1987.
- [12] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, October 1959.
- [13] Kurt Mehlhorn. A faster approximation algorithm for the steiner tree problem in graphs. *Information Processing Letters*, 27(3):125–128, March 1988.
- [14] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
- [15] J. A. Wald and C. J. Colbourn. Steiner trees, partial 2-trees and minimum IFI networks. *Networks, An International Journal*, 13(2):159–167, 1983.