Planning Tours of Robotic Arms Among Partitioned Goals

Mitul Saha(1) Gildardo Sánchez-Ante(2) Tim Roughgarden(1)
Jean-Claude Latombe(1)

(1) Computer Science Department, Stanford University, Stanford, CA, USA
(email:{mitul,tim,latombe}@cs.stanford.edu).

(2) Computer Science Department, National University of Singapore, Singapore
(email: ante@comp.nus.edu.sg).

Abstract

This paper considers a motion planning problem that occurs in tasks, such as spot welding, car painting, inspection, and measurement, where the end-effector of a robotic arm must reach successive goal placements given as inputs. The problem is to compute a near-optimal path of the arm so that the end-effector visits each goal once. It combines two notoriously hard sub-problems – the collision-free shortest-path and the traveling-salesman problems. It is further complicated by the fact that each goal placement of the end-effector may be achieved by several configurations of the arm (distinct solutions of the arm's inverse kinematics). This leads to considering a set of goal configurations of the robot that are partitioned into groups. The planner must compute a robot path that visits one configuration in each group and is near optimal over all configurations in every goal group and over all group orderings. The algorithm described in this paper operates under the assumption that finding a good tour in a graph with edges of given costs takes much less time than computing good paths between all pairs of goal configurations from different groups. So, the algorithm balances the time spent in computing paths between goal configurations and the time spent in computing tours. Although the algorithm still computes a quadratic number of such paths in the worst case, experimental results show that it is much faster in practice.

1 Introduction

Consider a robotic arm whose end-effector must reach several goal placements. These placements are defined in some workspace coordinate system, but the sequence in which they should be reached is not given. The planning problem studied in this paper is to compute a near-optimal robot path (a loop) that visits each goal once. This problem occurs often in practice, e.g., in spot-welding, car-painting, inspection, and measurement tasks. For example, Figure 1 shows a spot-welding workcell in an automotive body shop, where each robot arm brings a welding tool to successive placements. At each goal, the robot stops, while its end-effector performs some operation. The time taken by this operation is independent of the goal ordering. The aim of the planning problem is to minimize the total travel time spent in moving the end-effector between goals.

Using the robot's inverse kinematics (IK), we map each goal placement of the end-effector to a finite set of goal configurations of the robot. We call such a set a *goal group*. Figure 2 shows two configurations in a group (where the end-effector placement is defined by the position of its tip). To solve the above planning problem, we must compute a robot path that both visits one configuration in each group and has near minimal length over all configurations in every goal group and over all group orderings. This problem is illustrated in Figure 3. While in practice the configurations in a goal group often correspond to distinct IK solutions of the arm, this need not be the case, and the methods presented in this paper do not depend on how goal groups are obtained. So, we assume that the goal configurations are an input to our multi-goal planning algorithm.

This paper expands both the theoretical and experimental results that we previously introduced in [23]. In particular, in [23] we only considered the case where goals are not partitioned into groups (i.e., all goal groups are singletons). Here, we study the *generalized* multi-goal motion planning problem where goals are partitioned into groups. While this extension is critical for many applications, it is also much harder to solve than the original problem. Our solution makes use of a recent approximation algorithm for the group Steiner tree problem [5]. In this paper, we also present several improvements that significantly speed up the planning methods presented in [23].

The multi-goal motion planning problem combines two hard problems. One is to compute a shortest collision-free robot path between two goal configurations. This problem is at least as hard as finding a Euclidean shortest path between two points among polyhedral obstacles in 3-D space, which is NP-hard [4]. The other subproblem is a variant of the classical traveling-salesman problem (TSP), which requires computing an optimal tour through the vertices of a graph whose edges have known costs. Here, the vertices are partitioned into groups and the tour must only contain one member from

Figure 1: A spot-welding workcell

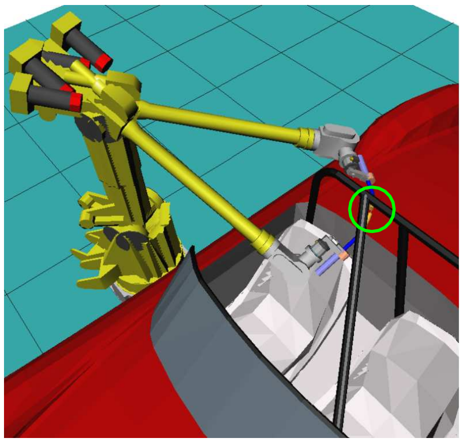


Figure 2: Two goal configurations in a group

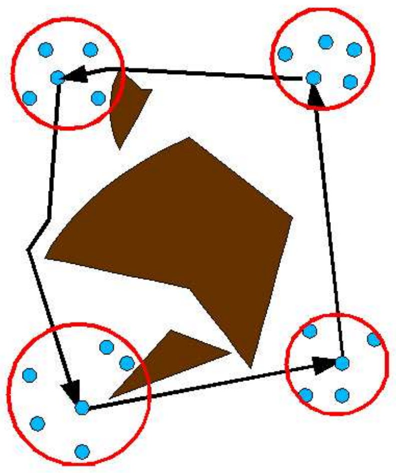


Figure 3: Two-dimensional generalized multi-goal motion planning problem

each group. This subproblem is at least as hard as finding the shortest tour through points in the Euclidean plane, which is NP-complete [18]. However, for both subproblems there exist reasonably efficient algorithms to compute sub-optimal, but still satisfactory solutions.

In this paper, we use such algorithms to build the following two functions:

- PATH – Given two robot configurations, this function returns a satisfactory collision-free path between them if they can be connected without collision. It indicates that no path exists otherwise.

- TOUR – Given a collection of points partitioned into groups, with given distances between all pairs of points in distinct groups, this function returns a near optimal tour that traverses one point in each group.

Given these two functions, the naive algorithm to solve the generalized multi-goal planning problem is simply to run PATH on all pairs of configurations from distinct goal groups – hence, a quadratic number of times – before running TOUR once. We will show that the length of the multi-goal path computed by this algorithm is within a factor 2β of the length of the optimal tour that can be computed given the goal-to-goal paths generated by the function PATH. If the goal configurations are non-partitioned (i.e., each group has size 1), then $\beta = 1$; otherwise $\beta = O(\frac{1}{\epsilon} \times (\log n)^{2+\epsilon} \times \log r)$ for every fixed constant $\epsilon > 0$, where n is the total number of goal configurations, and $r+1$ is the number of goal groups. The extra factor 2 comes from the fact that our function TOUR computes a tour by "doubling" a spanning tree [7].

However, in many robotics problems, the number of goal groups typically ranges between 5 and 50, while each group consists of few configurations (usually less than 10). These numbers are considered small for available TOUR algorithms. In contrast, robot paths must be computed in complex configuration spaces, so that the running time of PATH is high relative to that of TOUR. Hence, we wish to use PATH sparingly.

For a multi-goal problem of the size specified above, the naive algorithm may evaluate PATH several thousand times, or more, which takes prohibitive time. Instead, we propose a lazy algorithm that avoids calls to PATH by delaying them until they are needed. It does this by initializing the distance between every two configurations from distinct goal groups to an easily computed lower bound. Then, it successively runs PATH on pairs of configurations whose distances affect the length of the current best tour through the goal groups. It terminates when a tour has been found that is guaranteed to be within the factor 2β of the length of the optimal tour, where β is defined as above. Hence, both the naive and lazy algorithms have the same approximation factor. In the worst case, the lazy algorithm calls PATH a quadratic number of times, but our experimental results (Section 6) show that it is usually much faster than the naive algorithm. Moreover, the paths computed by the two algorithms have similar lengths.

The rest of this paper is organized as follows. Section 2 relates our work to previous work on multi-goal planning. Section 3 gives a precise statement of the generalized multi-goal planning problem. Section 4 describes and analyzes our planning algorithm. Section 5 presents in detail the implemented function PATH. Section 6 analyzes the experimental performance of the implemented planner on several examples.

2 Relation to Previous Work

The multi-goal motion planning problem was previously introduced in [8, 22, 24]. We review these works below and relate them to ours.

a) In [22] the multi-goal planning problem is considered in the context of coordinate measuring machines (CMM). In this work, goals do not occur in groups (i.e., are non-partitioned). Each goal defines a position of the CMM's measuring probe. The proposed planner computes goal-to-goal paths using a Probabilistic Road-Map (PRM) planner [17]. A roadmap is pre-computed over the probe's free space. Its vertices are all the goal configurations, plus additional configurations successively sampled uniformly at random over the entire configuration space, until all the goals are in the same connected component of the roadmap. Next, a search algorithm extracts the shortest path in the roadmap between every two goals. This yields a reduced, complete graph in which each vertex is a goal configuration and each edge is labelled by the length of the shortest path extracted between the two goals it connects. A TSP algorithm then computes a near-optimal tour from this graph. Since the lengths labelling the edges satisfy the triangle inequality, polynomial-time algorithms are available to compute a tour within a constant ratio of the optimal tour [7, 19].

This planner is based on a set of coherent choices. The CMM problems considered in [22] are formulated in a quasi-planar, hence low-dimensional, configuration space (where a configuration is the position of the tip of the measuring probe). In the two examples given in [22], the generated roadmaps only contain 100 vertices (including the goals). Thus, pre-computing them does not take much time. Moreover, once a roadmap is available, the search of a shortest path between every two goals is very fast. This justifies that all goal-to-goal paths are computed before a tour is extracted.

Our multi-goal planner is based on different assumptions. In many applications (like spot welding), the cost of pre-computing a roadmap would be too high, especially if one uses a uniform sampling strategy, due to both the dimensionality of the configuration space and the geometric complexity of the obstacles and the robot. For this reason, the PATH function in our multi-goal planner is based on the bi-directional tree-expansion PRM algorithm introduced in [16] and refined in [21] (more details are given in Section 5). This algorithm makes it possible to construct goal-to-goal paths one at a time until the multi-goal planning problem is solved. As our experimental results will show (Section 6), it is often the case that only a relatively small number of goal-to-goal paths need to be computed before a good robot tour is obtained. However, to avoid repeating work done at previous evaluations of PATH, each new evaluation of PATH is expedited by re-using PRM trees previously constructed.

b) The multi-goal planner described in [24] tries to avoid computing a collision-free path between every pair of goals. But, unlike ours, it neither uses lower bounds on path lengths, nor the information contained in intermediate tours computed by the TSP algorithm to guide the selection of the pairs of goals between which paths are computed. Instead, this planner loops on the following operations: select pairs of goals using a technique like random or nearest-pair selection, compute a collision-free path between each pair, run the TSP algorithm. Once the TSP algorithm has returned a complete tour, the planner can stop at any time. If more computing time is allocated, each new tour generated by the planner is at least as good as the previous one. But, if the planner is stopped before all goal-to-goal paths have been generated, then there is no formal guarantee on the goodness of the last computed tour. Like ours, the planner in [24] takes goal groups as inputs. A genetic algorithm generates optimized tours through the input groups. Goal-to-goal paths are computed by performing a best-first search on a regular grid in configuration space. This technique is inherently limited to searching low-dimensional configuration spaces, hence restricts the range of problems that the planner can handle. The planner was tested on pin-assembly and spot-welding examples comprising up to 22 goals.

c) The multi-goal planner in [8] is aimed at generating inspection tours, for example finding cracks in large structures. The robot, which is modeled as a point in a polygonal free space F, must go to successive goal positions such that it can see every point in the boundary of F from at least one of these positions (goal groups are not considered). This problem is also known as the "watchman-route" problem. Unlike in our multi-goal problem, the goals are not given. Instead, in [8], they are computed using the approximate randomized "art-gallery" algorithm given in [13]. Next, the visibility graph of these goals and the vertices of F's boundary are computed, as well as the shortest path between every two goals [19]. An approximate TSP algorithm then extracts a tour that is at most twice as long as the optimal tour. An extension to 3-D workspace is discussed in [8].

d) Other variants of multi-goal robot planning problems, with no or few collision-avoidance constraints, have been addressed using general optimization techniques such as simulated annealing [6] and genetic algorithms [3].

3 Problem Formulation

3.1 Overview

Let C denote the configuration space of a robot, $F \subseteq C$ its collision-free subset, and ℓ the measure that maps any given path τ in C to its length $\ell(\tau)$. Let $\tilde{g}_0, \tilde{g}_1, ..., \tilde{g}_r$ be $r+1$ input groups – called *goal groups* – of distinct configurations in F, $|\tilde{g}_i|$ denotes the size of $\tilde{g}_i$ and g_{ik} its k^{th} element.

Any path in F joining two goals g_{ik} and g_{jl} from two distinct groups i and j is called a *goal-to-goal* path. Any loop path in F starting and ending at the same configuration in $\tilde{g}_0$, and passing through one configuration in each goal group, is called a *multi-goal path.* So, a multi-goal path τ is the concatenation of $r + 1$ goal-to-goal paths. The length of τ is the sum of the lengths (measured by ℓ) of the goal-to-goal paths it contains. This assumption is reasonable since in most applications the robot stops at each goal; so, the goals divide the multi-goal path into "independent" pieces.

The *generalized multi-goal planning problem* is to find the shortest multi-goal path, or a good approximation of it.

3.2 Function PATH

We assume that the function PATH defined as follows is given. For any two goals g_{ik} and g_{jl}, PATH(g_{ik}, g_{jl}) returns a path in F joining g_{ik} and g_{jl}, if these configurations are in the same component of F. For general robot arms, no efficient algorithm is available to compute a goal-to-goal path guaranteed to be within some approximation factor of the shortest path. So, by necessity, PATH is a heuristic algorithm. In our implementation, PATH is a bi-directional tree-expansion PRM planner [21], augmented with an optimization post-processing step. This planner is made deterministic by resetting the seed of the pseudo-random source at each call to the value of a given function of the two input goal configurations. (See Section 5 for more detail.) Without loss of generality, we assume that the three paths returned by PATH between any three goal configurations satisfy the triangle inequality. This assumption can always be trivially enforced, if needed, by replacing one path by the concatenation of the other two.

Note that if the goals are non-partitioned, then for a multi-goal path to exist, all goal configurations must lie in the same connected component of F. So, a path must exist between *every* two goal configurations. If the goals are partitioned, then there must exist a component of F that contains at least one member of each goal group. It is then possible for two goals from two different groups to belong to two distinct components of F.

In the following, we let $\mathcal{L}_{Opt}$ stand for the length of the optimal multi-goal path through $\tilde{g}_0, \tilde{g}_1, ..., \tilde{g}_r$ when every goal-to-goal path is computed by PATH.

3.3 Function TOUR

We define the *goal graph* to be the undirected graph $G = (V, E)$, in which the set of vertices is $V = \cup_i \tilde{g}_i$ and the set of edges, E, contains one edge connecting every pair of configurations from two distinct goal groups. When all goal groups have size 1, i.e., $|\tilde{g}_i| = 1$ for all $i = 0, 1, ..., r$, G is said to be

```
Algorithm TOUR($G_c$)
1. $T \leftarrow$ GSTREE($G_c$)
2. Return PREORDER-WALK($T$)
```

Figure 4: TOUR algorithm

non-partitioned. A *tour* of G is any list $\pi = (g_{0k_0}, g_{i_1k_1}, ..., g_{i_rk_r}, g_{0k_0})$ such that $< i_1, ..., i_r >$ is a permutation of $\{1, ..., r\}$.

A *weighted* goal graph G_c is identical to G, except that each edge $\{g_{ik}, g_{jl}\}$ is now weighted by a positive real number $c(g_{ik}, g_{jl})$, the edge's *cost.* Then the *cost* $c(\pi)$ of a tour π of G is the sum of the costs of the edges traversed by π, hence:

$$c(\pi) = \sum_{m=1}^{m=r+1} c(g_{i_{m-1}k_{m-1}}, g_{i_mk_m})$$

where $i_0 = i_{r+1} = 0$.

In particular, when the cost of every edge $\{g_{ik}, g_{jl}\}$ in E is equal to ℓ(PATH($\{g_{ik}, g_{jl}\}$)), we denote the weighted goal graph by G_ℓ. So, the cost of the optimal tour of G_ℓ is $\mathcal{L}_{Opt}$.

A *group-spanning tree* T is a tree contained in G such that T has exactly $r + 1$ vertices, one from each goal group. In a weighted goal graph G_c, the cost of T is the sum of the costs of the edges contained in T. The function TOUR defined in Figure 4 first computes a group-spanning tree T. Then it computes a tour by performing a preorder walk of T, which recursively visits every vertex in the tree, starting at the root, listing a vertex when it is first encountered. Under the assumption that the edge costs satisfy the triangle inequality, the cost of the computed tour is at most twice the cost of T [7].

More precisely, when G is non-partitioned, T is simply a spanning tree of G and GSTREE(G_c) computes the minimum-cost spanning tree of G in time $O(r^2)$ using the Prim's algorithm [7]. In this case, TOUR is a classical TSP algorithm, which has also been used in the multi-goal planners described in [8, 22, 23]. Then TOUR(G_ℓ) generates a multi-goal path of length $\mathcal{L}_N \leq 2 \times \mathcal{L}_{Opt}$. Indeed, the cost of the optimal spanning tree of G_ℓ is not greater than $\mathcal{L}_{Opt}$ and $\mathcal{L}_N$ is at most twice the cost of the optimal spanning tree.

When G is partitioned, i.e., goal groups have sizes greater than 1, we are facing the problem of computing the optimal group-spanning tree of G_c, a particular case of the so-called group Steiner problem studied in circuit and network design [5, 20]. This problem is provably hard, even to approximate closely [14], but the polynomial-time algorithm given in [5] allows us to compute a group-spanning tree of G_c whose cost is at most $\beta = O(\frac{1}{\epsilon} \times (\log |V|)^{2+\epsilon} \times \log r)$ times the minimum cost, for any fixed constant[1] $\epsilon > 0$. (A slightly better worst-case ratio is possible using linear programming [12], but this approach would be too computationally expensive for our purposes.) When G is partitioned, our function GSTREE uses this algorithm. Then TOUR(G_ℓ) computes a multi-goal path of length $\mathcal{L}_N \leq 2\beta \times \mathcal{L}_{Opt}$.

In the following, we let $\beta = 1$ if G is non-partitioned, and $\beta = O(\frac{1}{\epsilon} \times (\log |V|)^{2+\epsilon} \times \log r)$ otherwise.

3.4 Objective

The naive multi-goal planning algorithm NAIVE-GMGP first computes G_ℓ by weighting every edge $\{g_{ik}, g_{jl}\}$ of G by $\ell(\text{PATH}(\{g_{ik}, g_{jl}\}))$, then evaluates TOUR($G_\ell$) once. The resulting multi-goal path is the concatenation of the goal-to-goal paths between every two successive goals in the tour returned by TOUR(G_ℓ). Its length is $\mathcal{L}_N \leq 2\beta \times \mathcal{L}_{Opt}$.

This algorithm evaluates PATH for all the edges in G. However, in most robotics applications, the number $r + 1$ of goal groups is relatively small (a few dozens at most). The size of each goal group is usually upper bounded by a small constant. In contrast, the robot's collision-free space F has high complexity. So, here, we are interested in instances of the multi-goal planning problem where the running time of PATH is high relative to that of TOUR and we wish to use PATH sparingly, even if this requires evaluating TOUR (or just GSTREE) more often. But, simultaneously, we would like the length of the generated multi-goal path to remain within a small factor of $\mathcal{L}_{Opt}$. The lazy planning algorithm described in the following section attempts to achieve this twofold objective.

4 Lazy Planning Algorithm

4.1 Algorithm

Our algorithm first creates a weighted goal graph G_c in which the cost of every edge $\{g_{ik}, g_{jl}\}$ is set to the length of the shortest path joining g_{ik} and g_{jl} in C. Since the shortest path in C may not be collision-free, this length is a lower-bound approximation of $\ell(\text{PATH}(g_{ik}, g_{jl}))$. Usually, this bound is very easy to calculate, since for many measures ℓ the shortest path in C between two configurations is the straight line segment joining them. Then, the algorithm iteratively modifies weights in G_c, by evaluating PATH

[1]Note that ϵ is not an input of our algorithm.

```
Algorithm LAZY-GMGP
1. Initialize $G_c$ to the weighted goal graph in which the cost of every edge
       $\{g_{ik}, g_{jl}\}$ is set to the length of the shortest path in $C$ between
       $g_{ik}$ and $g_{jl}$
2. Repeat
   a. $T \leftarrow$ GSTREE($G_c$)
   b. $\kappa \leftarrow c(T)$
   c. Repeat while $c(T) \leq \alpha \times \kappa$
      i. If all edges in $T$ are exact then return multi-goal path defined
            by PREORDER-WALK($T$)
      ii. Pick a non-exact edge $e = \{g_{ik}, g_{jl}\}$ in $T$
      iii. Modify $G_c$ by resetting the cost of $e$ to $\ell$(PATH($g_{ik}, g_{jl}$))
```

Figure 5: Lazy multi-goal planning algorithm

for edges picked in the group-spanning tree computed by GSTREE(G_c) and updating their costs accordingly. Once the cost of an edge $\{g_{ik}, g_{jl}\}$ has been updated to ℓ(PATH(g_{ik}, g_{jl})), we say that this edge is *exact*. This yields the lazy algorithm LAZY-GMGP shown in Figure 5 and discussed below.

Step 1 initializes the weighted goal graph G_c. Next, each iteration of Step 2 first computes a group-spanning tree T of the current G_c (the optimal spanning tree if G_c is non-partitioned, or a near optimal group-spanning tree if the graph is partitioned). Throughout the rest of Step 2, $c(T)$ denotes the current cost of T. Each iteration of Step 2.c leads to evaluating PATH for a non-exact edge picked in T and updating the cost of this edge in G_c. During this iteration, the topology of T remains fixed, but its cost $c(T)$ varies, as long as it remains within factor α of the initial cost recorded in κ, where $\alpha \geq 1$ is an input number. If $c(T)$ ever grows greater than $\alpha \times \kappa$, then a new iteration of Step 2 is performed which recomputes a new tree T from the current G_c. Step 2 exits with a solution when all edges of T are exact.

In addition, whenever PATH fails to find a path at Step 2.c.iii, the corresponding edge is removed from G (this is equivalent to resetting its cost to infinity) and a new iteration of Step 2 is performed. If GSTREE fails to find a group-spanning tree at Step 2.a, then LAZY-GMGP returns *failure*. Note that if the goals are not partitioned, LAZY-GMGP returns *failure* at the first failure of PATH. For simplicity, these cases are not shown in the algorithm of Figure 5.

4.2 Analysis

Let $\mathcal{L}_N$ and $\mathcal{L}_G$ denote the lengths of the tours respectively computed by NAIVE-GMGP and LAZY-GMGP for a given instance of the multi-goal planning problem. We assume that, whenever these two algorithms evaluate PATH for the same two goals g_{ik} and g_{jl}, each evaluation yields the same goal-to-goal path. Recall from Subsection 3.3 that $\mathcal{L}_N \leq 2\beta \times \mathcal{L}_{Opt}$, where $\mathcal{L}_{Opt}$ is the length of the optimal tour in G_ℓ. The following theorem relates $\mathcal{L}_G$ and $\mathcal{L}_{Opt}$.

Theorem 1: *The length $\mathcal{L}_G$ of the multi-goal path computed by* LAZY-GMGP *satisfies:*

$$\mathcal{L}_G \quad \leq \quad 2\alpha\beta \times \mathcal{L}_{Opt}.$$

Proof: Let $T_\ell^\star$ and $T_c^\star$ denote the optimal (group-)spanning trees of graphs G_ℓ and G_c, respectively. Let T_c be the tree returned by GSTREE(G_c). We have:

$$c(T_c^\star) \leq c(T_\ell^\star)$$

and:

$$c(T_c) \leq \beta \times c(T_c^\star).$$

So: $c(T_c) \leq \beta \times c(T_\ell^\star)$.

Therefore, at Step 2.b, we have $\kappa \leq \beta \times c(T_\ell^\star)$. Step 2.c computes goal-to-goal paths for non-exact edges in T, until either the cost of T grows larger than $\alpha \times \kappa$, or all edges in T are exact. When this second condition turns true, we have $c(T) \leq \alpha \times \kappa \leq \alpha\beta \times c(T_\ell^\star)$. Since the length $\mathcal{L}_G$ of the multi-goal path determined by PREORDER-WALK(T) at Step 2.c.i is within factor 2 of $c(T)$, we have $\mathcal{L}_G \leq 2\alpha\beta \times c(T_\ell^\star)$. We also have $c(T_\ell^\star) \leq \mathcal{L}_{Opt}$, hence $\mathcal{L}_G \leq 2\alpha\beta \times \mathcal{L}_{Opt}$. ■

Note that when α is set to 1, the lengths of the paths respectively computed by NAIVE-GMGP and LAZY-GMGP are within the same factor 2β of $\mathcal{L}_{Opt}$. Note also that, in the case of non-partitioned goals (then, $\beta = 1$), the upper bound $2\alpha \times \mathcal{L}_{Opt}$ on $\mathcal{L}_G$ is tighter than the bound $2\alpha \times \mathcal{L}_N$ established in [23].

The tour returned by PREORDER-WALK(T) at Step 2.c.i may contain edges of G_c that are still non-exact. To transform the tour into a multi-goal path, it is necessary to evaluate PATH on every such edges. The weighted goal graph after this computation may still contain edges that are non-exact, so that the triangle inequality may not be satisfied by every triplet of vertices. But as long as the triangle inequality holds for the edges in T

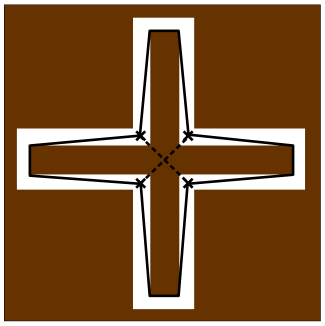


Figure 6: A worst-case example for LAZY-GMGP in a two-dimensional configuration space C. The free space F is shown in white and each of the 4 cross-marks denotes a goal configuration. The dashed and plain lines depict the shortest paths between the goals in C and in F, respectively. The length of the shortest path in C between any two goals is much smaller than the length of the shortest path in F between any two goals.

and its preorder walk, the tour returned by PREORDER-WALK(T) has cost within a factor 2 of that of T.

The experimental results of Section 6 will show that LAZY-GMGP is usually much faster than NAIVE-GMGP, but that the lengths of the paths respectively computed by the two algorithms are comparable. As one would expect, the running time of LAZY-GMGP also decreases as α gets larger, while the lengths of the paths only increase slightly.

However, for any given value of α, in the worst case, LAZY-GMGP evaluates PATH for all the $\Theta(|V|^2)$ edges of the goal graph. A worst-case instance with non-partitioned goals can easily be constructed as follows. Let the $r+1$ goal configurations $g_0, g_1, ..., g_r$ be very close to each other in C. More precisely, let the length of the shortest path in C between any two of them be less than some small ε. Assume that F is made of "pipes" connecting the goal configurations so that the length of the shortest path in F between any two of them is greater than some sufficiently large L (see Figure 6). Then, each execution of Step 2.c (except the last one) evaluates PATH exactly once, because the increment $L - \varepsilon$ in the cost of the spanning tree T causes $c(T)$ to become greater than $\alpha \times \kappa$. Moreover, until all edges of G_c are exact, the minimal spanning tree at Step 2.a necessarily includes at least one non-exact edge. Therefore, LAZY-GMGP will terminate only when all goal-to-goal paths have been computed.

```
• $\lambda \leftarrow$ cost of $e$ in $G_c$
• Modify $G_c$ by resetting the cost of $e$ to $\lambda' = \ell(\text{PATH}(g_{ik}, g_{jl}))$
• If $(\lambda' - \lambda)/\lambda > \gamma$ then for every $g_{st} \in V$ other than $g_{ik}$ and $g_{jl}$ do
    - If edge $\{g_{jl}, g_{st}\}$ is exact and edge $\{g_{ik}, g_{st}\}$ is not exact, then reset the
      cost of $\{g_{ik}, g_{st}\}$ to $\max\{c(g_{ik}, g_{st}), |c(g_{ik}, g_{jl}) - c(g_{jl}, g_{st})|\}$.
    - If edge $\{g_{ik}, g_{st}\}$ is exact and edge $\{g_{jl}, g_{st}\}$ is not exact, then reset the
      cost of $\{g_{jl}, g_{st}\}$ to $\max\{c(g_{jl}, g_{st}), |c(g_{ik}, g_{jl}) - c(g_{ik}, g_{st})|\}$.
```

Figure 7: Improvement of Step 2.c.iii of LAZY-GMGP

4.3 Improvements

Below we briefly discuss three possible improvements of the LAZY-GMGP algorithm shown in Figure 5. Only the first two have been implemented.

a) In general, one may expect LAZY-GMGP to make fewer calls to PATH if the edge costs of G_c approximate the exact values more tightly. This leads our planner to exploit the triangle inequality among goal-to-goal paths, as follows. If the edges $\{g_{ik}, g_{jl}\}$ and $\{g_{jl}, g_{st}\}$ in G_c are exact, but $\{g_{ik}, g_{st}\}$ is still non-exact, then $c(g_{ik}, g_{st})$ should not be smaller than $|c(g_{ik}, g_{jl}) - c(g_{jl}, g_{st})|$. So, we can replace Step 2.c.iii of LAZY-GMGP by the steps given in Figure 7, where $\gamma \geq 0$ is a real constant aimed at skipping the update when PATH(g_{ik}, g_{jl}) is not longer, or only marginally longer than the lower-bound cost weighting edge $\{g_{ik}, g_{jl}\}$.

b) In order to avoid unnecessary iterations at Step 2.c of LAZY-GMGP, we can try to pick, at Step 2.c.ii, the non-exact edge that is likely to yield the maximum increase of the cost of T. A heuristic used in our implemented planner is to pick the longest non-exact edge, since it is the most likely to require a long detour around obstacles (see [21]).

c) Step 2.a of LAZY-GMGP re-computes a new group-spanning tree T at each iteration. However, between two successive computations of T, it is often the case that only a small number of edges of G_c (at most r) have changed cost. This suggests saving time by updating the tree computed at the previous iteration. For non-partioned graphs, if k is the number of edges in G_c that have changed costs, then the optimal spanning tree of G_c can be updated in time $O(k\sqrt{r})$ [9, 10]. We are not aware of a similar algorithm for updating group-spanning trees.

5 Implementation of PATH

For robot arms, no algorithm is available to efficiently compute a goal-to-goal path guaranteed to be within a given factor of shortest paths. So, our implementation of PATH uses a heuristic two-phase approach: first, compute a collision-free path; next, optimize this path. This approach was introduced in [2].

We use a bi-directional tree-expansion PRM planner (more precisely, the SBL planner presented in [21]) to generate an initial collision-free path between two given goals g_{ik} and g_{jl}. This planner grows two trees of sampled configurations, called milestones, that are rooted at g_{ik} and g_{jl}, respectively. At every iteration, it picks a milestone m in one of the two trees and samples configurations at random in a neighborhood of m until one, m', tests collision-free. It installs m' as a child of m in its tree and creates a connection between m' and the closest milestone in the other tree, thus establishing a path of milestones between g_{ik} and g_{jl}. If all connections in a path of milestones are collision-free, the planner returns the path (a polygonal line in configuration space), otherwise it removes the colliding connection from the trees and samples more milestones. The planner exits with failure if it has not found a path after generating a given maximum number of milestones. The planner has been shown, both theoretically and empirically, to have fast convergence rate [16, 21]. Failure to find a path, when one exists, has not been an issue in our experiments.

SBL samples configurations by using a pseudo-random source of numbers parameterized by a seed. For a given seed, such a source produces a deterministic sequence of numbers that approximates the statistical properties of a random sequence. At each evaluation, our function PATH resets the seed to the value of a function of the two input goal configurations. By doing so, we guaraantee that an important assumption for Theorem 1 – that if NAIVE-GMGP and LAZY-GMGP evaluate PATH for the same two goals, then each evaluation yields the same goal-to-goal path – is satisfied.

The trees grown by the planner are not biased in any particular direction. So, in both the naive and the lazy algorithms, rather than discarding the two trees produced by a run of PATH, we store them. When PATH is invoked again, if any of the two goals in the new pair of goals has already been considered before, then the tree rooted at this goal is retrieved and re-used by the planner, thus saving considerable amount of work. As more evaluations of PATH are performed, each evaluation, on average, takes less time. This implementation of PATH is reminiscent of the Probabilistic Roadmaps of Trees (PRT) described in [1], except that in PATH all trees are rooted at

```
Algorithm OPTIMIZE($\tau$)
Repeat $N$ times
1. Pick two configurations $q$ and $q'$ on $\tau$ at random
2. $\tau \leftarrow$ SHORTEN($q, q', \tau$)
```

```
Algorithm SHORTEN($q, q', \tau$)
If the straight segment between $q$ and $q'$ is collision-free
1. Then return the path obtained by replacing the sub-path of $\tau$ between
        $q$ and $q'$ by this segment
2. Else
    2.1. Let $q''$ be the midpoint on the sub-path of $\tau$ between $q$ and $q'$
    2.2. $\tau \leftarrow$ SHORTEN($q, q'', \tau$)
    2.3. Return SHORTEN($q'', q', \tau$)
```

Figure 8: Path optimizer

goal configurations.

Several optimizers can be used to improve a path generated by the PRM planner. For instance, the variational optimization technique used in [2] iteratively deforms a path to minimize the time that the robot will take to execute the path. It takes the robot's dynamic model and torque limits in the joints into account. Our implementation of PATH uses a simpler (and faster) optimizer described in Figure 8 [15]. This optimizer assumes that the shortest path in configuration space between two arbitrary configurations is the straight segment. The input path, τ, is a polygonal line in configuration space. The optimizer repeatedly replaces sub-paths of τ by collision-free straight segments. Though the outcome is only locally optimal at best, it is usually quite satisfactory.

6 Experimental Results

In this section we report on some of the experiments we have performed using our implementations of both the NAIVE-GMGP and LAZY-GMGP algorithms. All results given below have been obtained on a 1-GHz Pentium-III computer with 1Gb of memory running Linux. All times are in seconds, with a resolution of 0.01 (hence, a few times are reported to be 0.00). All performance data are averaged over 50 runs of each of the two planners, such that in each pair of runs PATH computes the seed of the pseudo-random source with a different function (see Section 5).

For PATH, we used the implementation of SBL available at: http://robotics.stanford.edu/~mitul/mpk. When goals are not partitioned, a minimum-cost spanning tree is computed using the Prim's algorithm [7]. Otherwise, a near-optimal group-spanning tree is computed using an implementation of the algorithm given in [5] (since the input to this algorithm must be a tree, as suggested in [5], we use the algorithm proposed in [11] to approximate a weighted goal graph with a tree). We further optimize the output tree by replacing it with the optimal spanning tree of the subgraph of G_c reduced to the nodes in the tree. Our implementation of LAZY-GMGP incorporates the two improvements described in Subsections 4.3 *a*) and *b*).

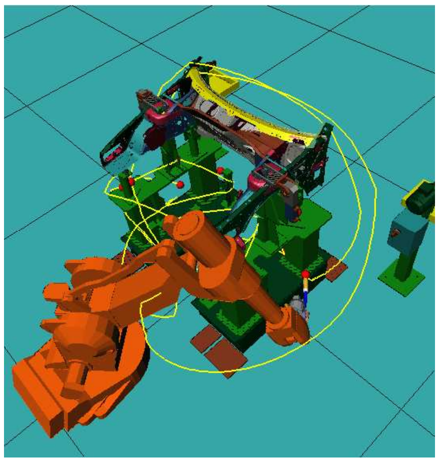
Throughout this section, we consider the three examples depicted in Figures 9, 10, and 11, in which there are 10, 31, and 50 goal groups, respectively. All three examples use a six-degree-of-freedom arm. The arm in Example 1 is modeled by 3,791 triangles and the arms in Examples 2 and 3 by 2,502 triangles. The workspaces (obstacles) of Examples 1, 2, and 3 are modeled by 74,681, 19,668, and 31,184 triangles, respectively. Each goal group corresponds to a given position of the end-effector's tip. Since the arm has six degrees of freedom, such a position yields an infinity of IK solutions forming a continuous IK subset of C. To construct a goal group of size p, we sample a large number of IK solutions at random and we cluster them into p clusters, in order to select p very distinct configurations providing a good coverage of the IK subset. A goal group of size 1 simply consists of one of these configurations.

6.1 Non-partitioned case

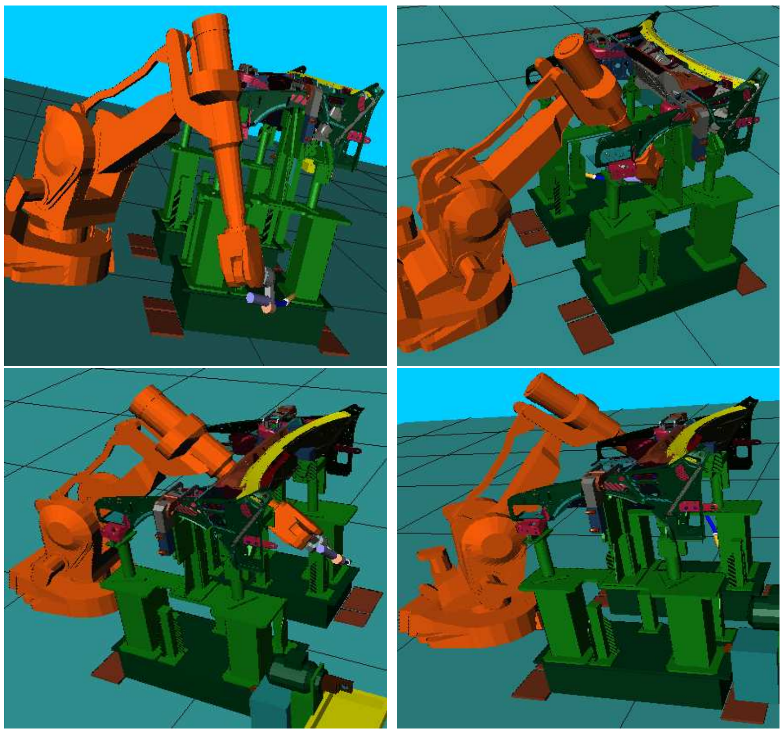
Here, we compare the performances of LAZY-GMGP and NAIVE-GMGP on the three examples when each group has size 1.

Table 1 lists performance data with the parameter α set to the minimum value of 1. The columns correspond to the three examples. The rows successively indicate the total running time of a planner, the length of the generated solution (using the $\ell = L_2$ metric in configuration space), the number of evaluations of PATH, the number of evaluations of GSTREE, and the times spent in those evaluations.

On all three examples, LAZY-GMGP is much faster than NAIVE-GMGP – about ten to twenty times faster in Examples 2 and 3, respectively. Figure 12, which plots the running times of LAZY-GMGP and NAIVE-GMGP when

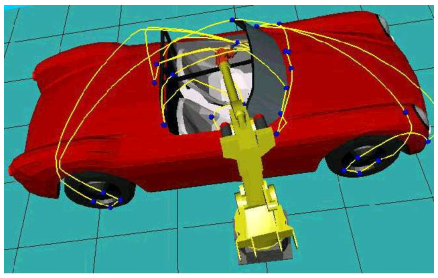


(a)



(b)

Figure 9: (a) Example 1 (10 goal groups); (b) some goal configurations


(a)

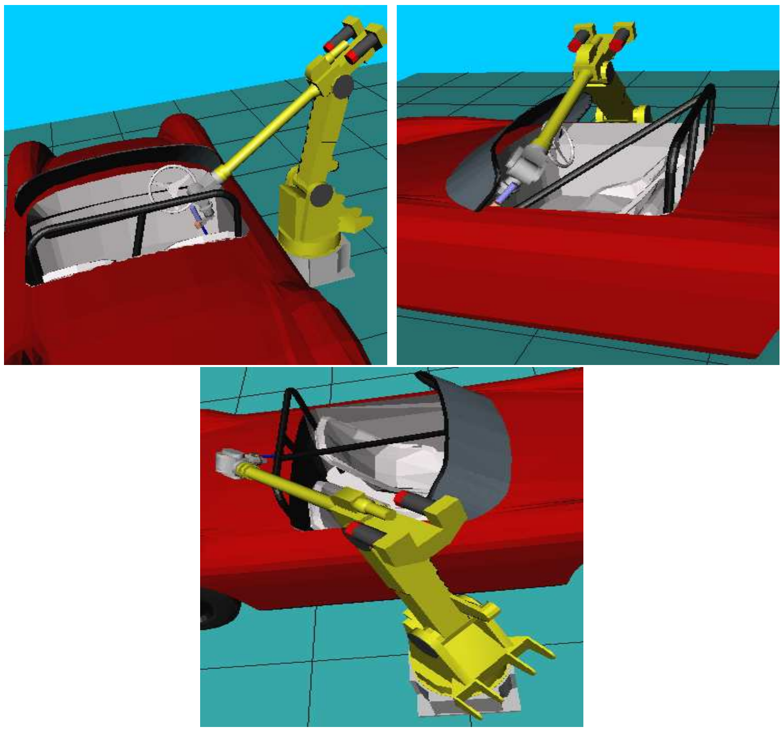
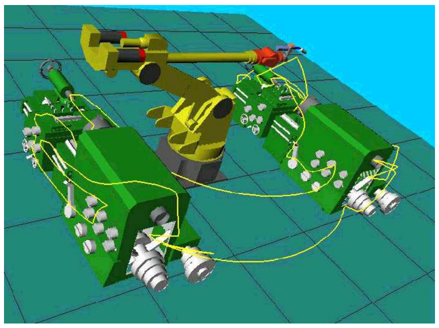
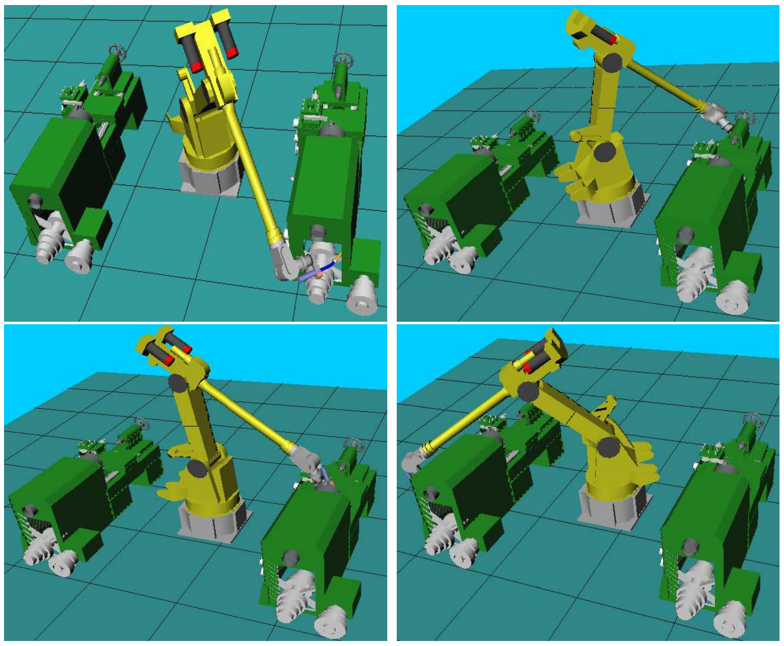

(b)

Figure 10: (a) Example 2 (31 goal groups); (b) some goal configurations



(a)



(b)

Figure 11: Example 3 (50 goal groups); (b) some goal configurations

	NAIVE-GMGP			LAZY-GMGP		
	Ex.1	Ex.2	Ex.3	Ex.1	Ex.2	Ex.3
Total-time	229.98	2509.60	5358.19	116.71	230.93	218.49
Total-length	12.79	27.34	39.79	12.79	27.34	39.79
#PATH	45	465	1225	26	57	75
#GSTREE	1	1	1	27	50	69
Time-PATH	229.98	2509.60	5358.19	116.71	230.25	216.45
Time-GSTREE	0.00	0.00	0.00	0.00	0.68	2.04

Table 1: Comparison of NAIVE-GMGP and LAZY-GMGP, when goal configurations are non-partitioned and $\alpha = 1$ (all times are in seconds)

the number of goal configurations increases from 5 to 50 in Example 3, further indicates that the speedup of LAZY-GMGP over NAIVE-GMGP increases with the number of goals. Additional tests not shown here indicate that the speedups achieved by LAZY-GMGP over NAIVE-GMGP are almost cut in half when the improvements described in Subsection 4.3 *a*) and *b*) are not included.

In each example, the two planners generate the same multi-goal path. This is not surprising. Indeed, when G is non-partitioned and α is set to 1, LAZY-GMGP terminates at Step 2.c.i with a spanning tree T that is optimal in both G_c and G_ℓ. So, if G_ℓ has a unique optimal spanning tree, then both NAIVE-GMGP and LAZY-GMGP return the same multi-goal path. This is because our implementation of PREORDER-WALK returns a unique preorder walk for an input tree (by choosing the same root and the same ordering on the nodes). It is easy to see that in almost all practical cases, G_ℓ has a unique optimal spanning tree. This is the case in our three examples.

Table 1 indicates clearly that on all three examples LAZY-GMGP obtains the optimal spanning tree from a graph G_c that still contains many non-exact edges, hence without having run PATH on all pairs of goals. In particular, in Example 3, it runs PATH on only 6% (on average) of the 1225 pairs of goal configurations.

Note that NAIVE-GMGP actually pre-computes a multi-tree roadmap that connects all pairs of goal configurations. We compared the cost of this pre-computation with that of running our own implementation of the roadmap-construction algorithm proposed in [22] (see Section 2 *a*). On each of the three examples, NAIVE-GMGP pre-computes a roadmap 2 to 3 times faster than this algorithm. But, of course, the main reason for implementing PATH as a bidirectional tree-expansion PRM planner is to generate goal-to-goal paths incrementally, one at a time, a capability that LAZY-GMGP exploits to

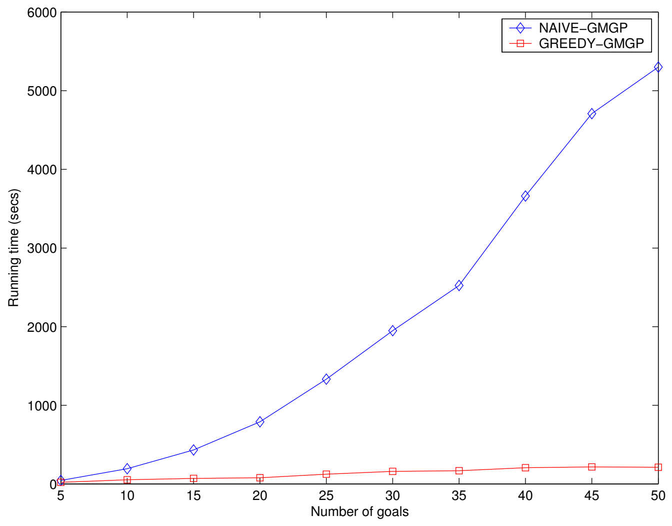


Figure 12: Running times of NAIVE-GMGP and LAZY-GMGP as the number of goal configurations grows from 5 to 50 in Example 3 (non-partitioned case)

avoid computing un-necessary goal-to-goal paths.

6.2 Partitioned case

Table 2 compares the performances of LAZY-GMGP and NAIVE-GMGP on the three examples when each goal group consists of 5 configurations. In Examples 1 and 2, the speedups achieved by LAZY-GMGP over NAIVE-GMGP are even more impressive than in the non-partitioned case. In particular, in Example 2, LAZY-GMGP is 100 times faster than NAIVE-GMGP. However, in the third example, the speedup is more modest (about 2). This observation will motivate the improved lazy algorithm presented in Section 6.3.

We note that, unlike in the non-partitioned case, the paths computed by NAIVE-GMGP and LAZY-GMGP in each example are different, but have comparable lengths. In Example 3, the paths computed by LAZY-GMGP are even shorter (on average). There are two reasons for this. First, even though edge costs in G_ℓ are greater than or equal to those in G_c, GSTREE(G_c) may be luckier than GSTREE(G_ℓ) and compute a tree T_c whose cost is less than that of the tree T_ℓ computed by GSTREE(G_ℓ). Secondly, even if $c(T_c) > c(T_\ell)$, it is still possible that T_c leads to a shorter pre-order walk than T_ℓ.

Since GSTREE is exact in the non-partitioned case, but only approximate in the non-partitioned case, it is interesting to compare the lengths of the

	NAIVE-GMGP			LAZY-GMGP		
	Ex.1	Ex.2	Ex.3	Ex.1	Ex.2	Ex.3
Total-time	5828.59	49238.58	87322.75	215.91	478.38	40036.51
Total-length	10.33	18.01	21.53	11.67	19.28	16.94
#PATH	1225	11935	31125	50	81	169
#GSTREE	1	1	1	51	75	159
Time-PATH	5828.48	49212.14	87103.98	208.27	223.46	346.77
Time-GSTREE	0.11	26.44	218.77	7.64	254.92	39689.74

Table 2: Comparison of NAIVE-GMGP with LAZY-GMGP when each goal group contains 5 configurations and $\alpha = 1$ (all times are in seconds)

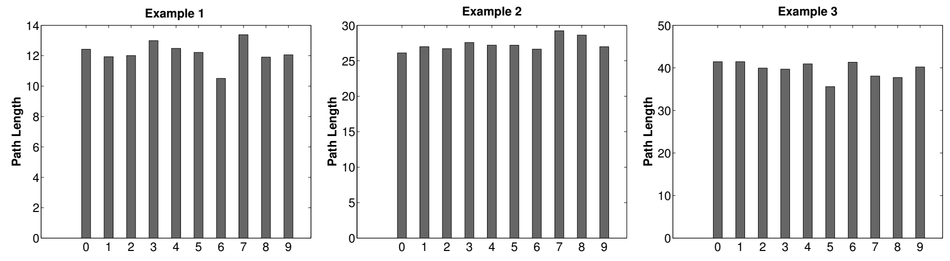


Figure 13: Lengths of paths returned by LAZY-GMGP for ten different sets of non-partitioned goals for Examples 1, 2 and 3

multi-goal paths obtained with LAZY-GMGP in the partitioned case with the lengths of paths also computed by LAZY-GMGP for non-partitioned problems generated by picking one goal at random from each goal groups. In fact, the three non-partitioned examples used to produce Table 1 were generated in this way, and all three lengths reported in Table 2 by solving the partitioned problem are significantly better. We conducted more experiments summarized in the three bar graphs of Figure 13. Each graph reports the lengths of the paths computed by LAZY-GMGP on 10 non-partitioned problems for Example 1, 2, or 3. In each of these problems, every goal was picked at random from the corresponding group of five goals used in the partitioned problem. In most cases, the paths are significantly longer than the corresponding paths computed by LAZY-GMGP on the partitioned problems.

6.3 Improved lazy algorithm

We noted above that in the partitioned case, the speedup achieved by LAZY-GMGP over NAIVE-GMGP is much smaller in Example 3 than in the other two examples. Table 2 reveals that, in Example 3, evaluating GSTREE takes

```
2.c. $\delta_{gstree} \leftarrow$ TIME(GSTREE)
2.d. $\delta_{path} \leftarrow 0$
2.e. Repeat while $c(T) \leq \alpha \times \kappa$ or $\delta_{path} \leq \delta_{gstree}$
  i.     If all edges in $T$ are exact
  ii.        then if $c(T) \leq \alpha \times \kappa$
  iii.         then return multi-goal path defined
             by PREORDER-WALK($T$)
  iv.          else exit loop 2.e
  v.     else
  vi.        Pick a non-exact edge $e = \{g_{ik}, g_{jl}\}$ in $T$
  vii.       Modify $G_c$ by resetting the cost of $e$ to $\ell$(PATH($g_{ik}, g_{jl}$))
  viii.  $\delta_{path} \leftarrow \delta_{path}$+TIME(PATH)
```

Figure 14: Replacing Step 2.c of LAZY-GMGP by the above steps results in LAZY-GMGP-2, an improved version of the lazy planner when the running time of TOUR is greater than that of GSTREE

more than 99% of the total running time of LAZY-GMGP. In fact, because the total number of goal configurations (250) is much larger than in the other examples, the running time of GSTREE dominates that of PATH, which violates the assumption on which the design of LAZY-GMGP is based.

This observation led us to create a variant of LAZY-GMGP – we call it LAZY-GMGP-2 – in which we balance the total times spent evaluating GSTREE and PATH. We obtain LAZY-GMGP-2 by replacing Step 2.c of LAZY-GMGP with the steps shown in Figure 14. The function TIME(x), where $x =$ PATH or GSTREE, returns the time spent in the last evaluation of x. Thanks to the test at Step 2.e.ii, Theorem 1 still holds for LAZY-GMGP-2. While LAZY-GMGP assumes that finding a good tour in G with given edge costs is faster than computing the exact cost of an edge in G by running PATH, LAZY-GMGP-2 operates under the more relaxed assumption that finding a tour in G is only faster than computing the exact costs of *all* the edges in G.

Table 3 reports the results obtained with LAZY-GMGP-2 on the three examples. As expected, LAZY-GMGP-2 is much faster than LAZY-GMGP in the third example, and marginally faster than LAZY-GMGP on the other two examples. Hence, it is always much faster than NAIVE-GMGP. In all the three examples, the lengths of the paths computed by LAZY-GMGP-2 are similar to those computed by LAZY-GMGP.

LAZY-GMGP-2 has an interesting additional property over LAZY-GMGP.

	LAZY-GMGP-2		
	Ex.1	Ex.2	Ex.3
Total-time	214.69	419.95	9621.64
Total-length	11.61	19.41	17.22
#PATH	50	86	198
#GSTREE	51	37	32
Time-PATH	207.17	242.98	427.94
Time-GSTREE	7.52	176.97	9193.70

Table 3: Results obtained with lazy-mgp-2 when each goal group contains 5 configurations and $\alpha = 1$ (all times are in seconds)

In a worst-case example, like the one shown in Figure 6, GSTREE is called $O(r^2)$ times. If the number $r + 1$ of goal groups grows very large, then each run of GSTREE may get much more costly than a run of PATH, so that LAZY-GMGP may become much slower than NAIVE-GMGP. Instead, in such an example, LAZY-GMGP-2 spends about the same total time evaluating GSTREE and PATH. Hence, in the worst case, LAZY-GMGP-2 can only be at most twice slower than NAIVE-GMGP. However, such a case is very unlikely in practice, and in all the examples on which we have experimented our planners, NAIVE-GMGP is significantly slower than both LAZY-GMGP and LAZY-GMGP-2.

6.4 Influence of parameter α

Here, we measure the impact of the parameter α on the performance of LAZY-GMGP. The three plots in Figure 15 correspond to the three examples of Figures 9, 10, and 11, with non-partitioned goals. Each plot shows three curves that respectively represent the total running time of LAZY-GMGP (solid), the length of the computed multi-goal path (dashed), and the number of evaluations of PATH (dotted) when α grows from 1 to 3. Again, each value is averaged over 50 independent runs.

These plots indicate that the number of calls to PATH and consequently the running time of LAZY-GMGP first decrease sharply when α increases, but then level out. In fact, when the value of α gets greater than a certain threshold α_0, Step 2 of LAZY-GMGP returns a solution at the first cycle. Then, the average running time of LAZY-GMGP stops improving. The value α_0 depends on how well the lengths of the straight line segments joining all pairs of goals approximate the lengths of the collision-free paths computed by PATH. The tighter the approximations, the lower the threshold. In our

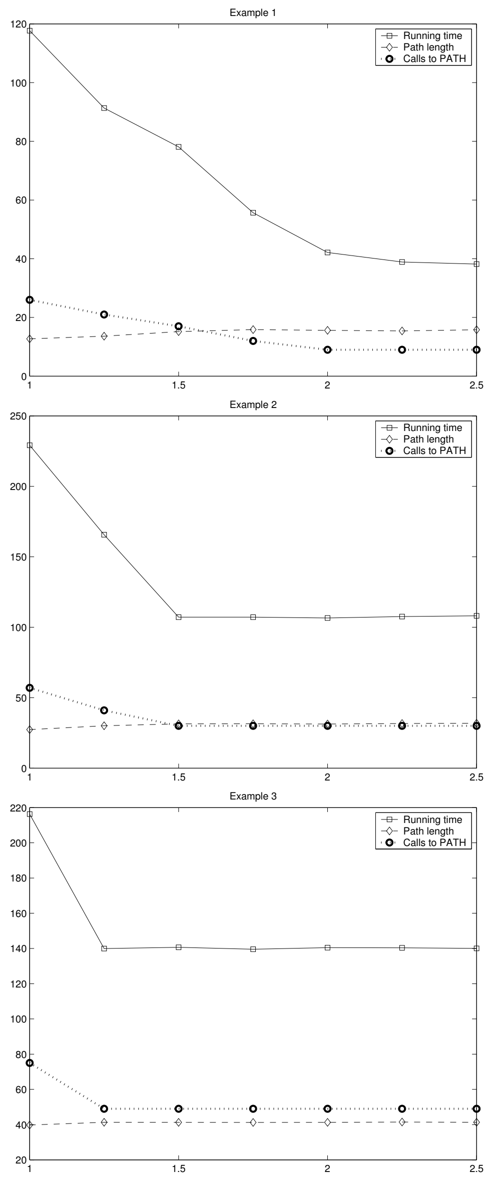


Figure 15: Running time of LAZY-GMGP, length of solution, and number of calls to PATH on Examples 1, 2, and 3 when α grows from 1 to 3

examples, α_0 ranges between 1.2 and 2.3.

7 Conclusion

This paper describes a new multi-goal motion planning algorithm for a robot arm whose task requires visiting multiple goal configurations. This algorithm generates a near-optimal tour through these configurations. It operates under the assumption that finding a tour in a goal graph with edges of given cost is much faster than finding the exact cost of all the edges in the graph. The algorithm can handle both the case where the goals are non-partitioned (then each goal must be visited once) and the case where the goals are partitioned into groups (then each group must be visited once). The partitioned case typically occurs when each goal is specified by the placement of the robot's end-effector and the robot's IK gives several solutions. It is much harder than the non-partitioned case. Our solution makes use of a recent approximation algorithm for the group Steiner tree problem [5].

Our lazy algorithm tries to compute as few goal-to-goal paths as possible. For this purpose, it uses the minimum spanning tree of a weighted goal graph (non-partitioned case) or a near-minimum group-spanning tree (partitioned case) to decide which goal-to-goal paths to compute. It returns a solution guaranteed to be within the same approximation factor of the optimal path as the solution returned by the naive algorithm that first computes all goal-to-goal paths. Experiments show that in general the lazy algorithm is much faster than the naive algorithm, while producing paths of similar lengths.

There are several directions in which this work could be extended in the future. In some applications, a partial ordering is imposed on the input goals. How to efficiently incorporate this ordering into the multi-goal planner? When the robot kinematics is redundant, there may be infinitely many IK solutions. In our experiments, we pre-sampled the continuous IK space. Could a multi-goal planner deal directly with continuous goal groups? When goals are partitioned, for a tour to exist, it is only necessary that one component of F contains at least one goal from each group. So, many pairs of goals may not belong to the same component, making it more difficult to set a "good" time limit for PATH. Too small, and PATH may fail to find critical goal-to-goal paths. Too large, and PATH may waste time trying to connect pairs of goals lying in different components of F. This issue did not arise in our experiments because in each example F had a single component. But it deserves more attention in the future. Finally, it would be interesting to investigate the multi-goal problem for multi-robot systems, when each goal may be reached by several robots.

Acknowledgment

This research has greatly benefited from valuable discussions with Prof. Fritz Prinz and Dr. Fabian Schwarzer, and comments from Phil Fong and Ankur Dhanik. Jean-Claude Latombe's research has been funded by gifts from General Motors and ABB, and by NSF grants ACI-0205671, IIS-0412884, and DMS-0443939, and NIH grant 5R33 LM007295. Tim Roughgarden's research is funded by DARPA grant W911NF-04-9-0001, ONR grant N00014-04-1-0725, and an NSF CAREER award.

References

[1] M. Akinc, K.E. Bekris, B.Y. Chen, A.M. Ladd, E. Plaku, and L.E. Kavraki. Probabilistic Roadmaps of Trees for Parallel Computation of Multiple Query Roadmaps. In *Robotics Research*, P. Dario and R. Chatila (eds.), Springer, pp. 80-89, 2005.

[2] J.E. Bobrow. Optimal Robot Path Planning Using the Minimum-Time Criterion. *IEEE J. Robotics and Automation*, 4(4):443-450, August 1988.

[3] M. Bonert, L.H. Shu, and B. Benhabib. Motion Planning for Multi-Robot Assembly Systems. *Proc. ASME Design Engineering Technical Conferences*, Las Vegas, NV, Sept. 1999.

[4] J.F. Canny and J. Reif. New Lower Bound Techniques for Robot Motion Planning Problems. *Proc. IEEE Conf. on Foundations of Computer Science*, pp. 39-48, 1987.

[5] C. Chekuri, G. Even, and G. Kortsarz. A lazy Approximation Algorithm for the Group Steiner Problem. *To appear in Discrete Applied Mathematics.*

[6] B. Cao, G.I.Dodds, and G.W. Irwin. A Practical Approach to Near Time-Optimal Inspection-Task-Sequence Planning for Two Cooperative Industrial Robot Arms. *Int. J. of Robotics Research*, 17(8):858-867, 1998.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990.

[8] T. Danner and L.E. Kavraki. Randomized Planning for Short Inspection Paths. *Proc. IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, April 2000.

[9] D. Eppstein, Z Galil, G.F. Italiano, and A. Nissenzweig. Sparsification–Technique for Speeding up Dynamic Graph Algorithms *J. of the ACM*, 44(5):669-696, 1997.

[10] G.N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *The SIAM J. on Computing*, 14(4):781-798, 1985.

[11] J. Fakcharoenphol, S. Rao, and K. Talwar. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. *Proc. ACM Symp. on Theory of Computing*, pp. 448-455, 2003.

[12] N. Garg, G.Konjevod, and R. Ravi. A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem. *J. of Algorithms*, 37:66-84, 2000.

[13] H.H. Gonzalez-Baños and J.C. Latombe. A Randomized Art-Gallery Algorithm for Sensor Placement. *Proc. ACM Symp. on Computational Geometry*, pp. 232-240, 2000.

[14] E. Halperin, R. Krauthgamer. Polylogarithmic Inapproximability. *Proc. ACM Symp. on Theory of Computing*, pp. 585-594, 2003.

[15] D. Hsu. *Randomized Single-Query Motion Planning in Expansive Space.* Ph.D. Thesis, Computer Science Dept., Stanford University, Stanford, CA, May 2000.

[16] D. Hsu, J.C. Latombe, and R. Motwani. Path Planning in Expansive Configuration Spaces, *Proc. IEEE Int. Conf. on Robotics and Automation*, pp. 2719–2726, 1997.

[17] L.E. Kavraki, P. Svetska, J.C. Latombe, and M. Overmars. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Tr. on Robotics and Automation*, 12(4):566–580, 1996.

[18] C. Papadimitriou. The Euclidean Traveling Salesman Problem is NP-complete. *Theoretical Computer Science* 4:237–244, 1977.

[19] J.S.B. Mitchell. Shortest Paths and Networks. In *Discrete and Computational Geometry*, J.E. Goodman and J. O'Rourke (eds.), CRC Press, Boca Raton, FL, pp. 445–466, 1997.

[20] G. Reich and P. Widmayer. Beyond Steiner's Problem: A VLSI Oriented Generalization. *Proc. Graph-Theoretic Concepts in Computer Science*, LNCS vol.411, pp. 196-210, 1990.

[21] G. Sánchez and J.C. Latombe. On Delaying Collision Checking in PRM Planning – Application to Multi-Robot Coordination. *Int. J. of Robotics Research*, 21(1):5-26, 2002.

[22] S.N. Spitz and A.A.G. Requicha. Multiple-Goals Path Planning for Coordinate Measuring Machines. *Proc. IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, 2000.

[23] M. Saha, G. Sánchez, and J.C. Latombe. Planning Multi-Goal Tours for Robot Arms. *Proc. IEEE Int. Conf. on Robotics and Automation*, Taipei, Taiwan, 2003.

[24] C. Wurll, D. Henrich, and H. Wörn. Multi-Goal Path Planning for Industrial Robots. *Proc. Int. Conf. on Robotics and Applications*, Santa Barbara, CA, 1999.