

MOTION PLANNING WITH  
PROBABILISTIC ROADMAPS

A DISSERTATION  
SUBMITTED TO THE DEPARTMENT OF MECHANICAL  
ENGINEERING  
AND THE COMMITTEE ON GRADUATE STUDIES  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

Mitul Saha  
August 2006

© Copyright by Mitul Saha 2006

All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

---

Jean-Claude Latombe Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

---

Fritz B. Prinz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

---

Bernard Roth

Approved for the University Committee on Graduate Studies.

# Abstract

Over the last few years, Probabilistic Roadmaps (PRMs) have emerged as a powerful approach for solving complex motion planning problems in robotics. Even beyond robotics, PRMs can be used to predict motions of biological macro-molecules such as proteins and synthesize motions for digital actors. Current PRM-based research focuses on challenges that arise as PRMs are being applied to motion planning problems in various scenarios. In response to some of those challenges, the following contributions are being made in this thesis:

- (1) **Dynamic collision checking.** A PRM planner performs *static* and *dynamic* collision checks to respectively test whether sampled configurations and simple paths between sampled configurations lie in free space. Static checks are usually carried out using an efficient “bounding-volume hierarchy” technique. On the other hand, the classical approach to perform dynamic checks is to sample each path at some fixed resolution and statically check each sampled configuration for collision. This approach is approximate and can miss collisions. A new dynamic checker is presented that exactly determines whether a path lies in free space. This checker chooses the adaptive sampling resolution along the local path by comparing lower bounds on distances between objects in relative motion with upper bounds on lengths of curves traced by points of these moving objects. It is not restricted to PRM planning, and also has potential applications beyond PRM planning, for instance in graphic animation and physical simulation.
- (2) **Dealing with narrow passages.** The efficiency of PRM planners can drop

dramatically if the free space contains narrow passages. A new sampling strategy, called *small-step retraction* (SSR), is presented that allows a PRM planner to efficiently construct roadmaps in free spaces with narrow passages. SSR first slightly fattens the robot’s free space into a new space, then constructs a roadmap in this new space, and finally repairs portions of the roadmap that are not in the original free space by resampling more configurations around them. The intuition behind this method is that, as the fattening operation expands narrow passages relatively more than the rest of the free space, it is easier to construct a roadmap in the fattened space. Experimental tests show that SSR achieves significant speedups (sometimes by two orders of magnitude) over a pre-existing state-of-the-art PRM planner.

(3) **Multi-goal motion planning (MGP).** MGP occurs in manufacturing tasks such as spot welding, car painting, inspection, and measurement, where the end-effector of a robotic arm must reach several input goal configurations in some sequence. But this sequence is not given, and the goal is to compute an optimal or near-optimal tour through the goals. MGP combines two notoriously hard computational problems: the traveling salesman problem (TSP) and the collision-free shortest-path problem. While a typical PRM planner can be used to compute near-optimal paths between pairs of goals, additional reasoning is needed to determine a near-optimal ordering on the goals without invoking the PRM planner a quadratic number of times, which would be too costly in general. A new planner is presented that avoids calls to the PRM planner by delaying them until they are absolutely needed. This planner also handles the case where goals are specified in terms of the robot’s end-effector placements and each end-effector placement can be achieved with several configurations of the robot arm. In this case, the set of goal configurations of the robot is partitioned into groups, and the planner’s objective is to compute a robot tour that visits one configuration in each group and is near optimal over all configurations in every goal group and over all group orderings.

(4) **Manipulation planning for deformable linear objects.** A number of tasks require manipulating deformable objects, in particular deformable linear objects

(DLOs), such as ropes, cables, and surgical sutures. Most previous research in manipulation planning has mainly focused on rigid objects. Deformable objects, which can take many different shapes when submitted to external forces, add challenging issues to the planning process. A new PRM planner is presented that compute the motions and (re-)grasp operations of a two-arm system in order to tie self-knots of DLOs, as well as knots around simple static objects. Here, unlike in traditional motion planning problems, the goal is a topological state of the world (the crossings of the DLO with itself and with static obstacles), rather than a geometric one. The planner constructs a topologically-biased probabilistic roadmap in the DLO's configuration space. The planner has been used in simulation to tie bowline, neck-tie, bow (shoe-lace), and stun-sail knots. Motions computed by the planner have also been successfully tested on a two-PUMA robot hardware platform.

# Acknowledgments

I am deeply indebted to Prof. Jean-Claude Latombe for providing me the “once in a life-time” opportunity to do a Ph.D. under his supervision. He has been very patiently and diligently advising and guiding me over the last five years. His research, advisory, and leadership skills are outstanding and remarkable. My personality has greatly improved by being in his research group.

I am very grateful to my Ph.D. committee members Prof. Fritz. B. Prinz, Prof. Bernard Roth, Prof. Oussama Khatib, and Prof. Steve Rock for donating some of their precious time towards my defense and thesis.

I am deeply indebted to my exceptional research collaborators: Dr. Fabian Schwarzer, Dr. Tim Roghgarden, Dr. Gildardo Sanchez, Dr. Pekka Isto, Yuchi Chang, Phil Fong, and Ankur Dhanik.

I am deeply indebted to Prof. Oussama Khatib, Dr. Jaeheung Park, Irena Pashchenko, and Jinsung Kwon for allowing me to use their Lab facilities to conduct experiments which helped me to demonstrate the viability and effectiveness of my work. I am also very thankful to Prof. Sebastian Thrun for lending me his precious video camera to shoot my experiments.

I am very grateful to industrial giants - ABB Automation and General Motors (GM) - and the National Science Foundation for funding my graduate stint at Stanford.

I am very grateful to my industrial friends: Dr. Jane Shi (GM), Dr. Robert Tilove (GM), Dr. Dev Satyadev (GM), and Shiro Tachibano (Nissan) for providing feedbacks on my Ph.D. research from the industrial perspective and also pointing out possible application domains for my works.

I am very grateful to Dr. Steve Holland (GM) and Dr. Zhangxue Gan (ABB) for providing CAD models and photographs of real robotic workcells which greatly enhanced the industrial appeal of my research results, documents, and presentations.

I am deeply indebted to Hoa Hguyen and Elizabeth Mattson for diligently bearing the burden of all the necessary paperworks throughout my tenure at Stanford. I am also deeply indebted to Miles Davis for always being prompt in responding to my computer support requests.

I am very thankful to the developers of the neat online search engine Google.

I am especially indebted to my angelic parents Kanchan and Mrityunjoy, brother Ankan, and sister Happy for allowing me to leave them and pursue education outside my home country.

For the rest of my life, I will remember all my supporters (parents, supervisors, collaborators, friends, and the Almighty) for all their support, love, and patience towards me in the last five years. This thesis belongs more to them than to me.

# Contents

<b>Abstract</b>	<b>iv</b>
<b>Acknowledgments</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motion Planning . . . . .	1
1.2 Traditional Motion Planning Approaches . . . . .	2
1.3 Probabilistic Roadmaps . . . . .	4
1.4 Thesis Contributions . . . . .	7
<b>2 Adaptive Dynamic Collision Checking</b>	<b>11</b>
2.1 Introduction . . . . .	11
2.1.1 Previous work on dynamic collision checking . . . . .	12
2.1.2 Fixed-resolution dynamic checking . . . . .	14
2.1.3 Adaptive bisection of paths . . . . .	16
2.1.4 Chapter organization . . . . .	17
2.2 Adaptive Dynamic Collision Checking . . . . .	17
2.2.1 Basic result . . . . .	17
2.2.2 Adaptive bisection algorithm . . . . .	19
2.2.3 Ordering of the priority queue . . . . .	20
2.2.4 Checking multi-segment paths . . . . .	21
2.2.5 Covering strategies . . . . .	21
2.2.6 Bounding the running time of ADAPTIVE-BISECTION . . . . .	23
2.3 Bounding Distances Between Objects . . . . .	24

2.3.1	Distance computation using BVHs . . . . .	24
2.3.2	Greedy distance computation algorithm . . . . .	26
2.3.3	Experimental analysis . . . . .	27
2.4	Bounding Motions in Workspace . . . . .	30
2.4.1	Example . . . . .	30
2.4.2	Upper bound in general case . . . . .	32
2.4.3	Computation of factors $R_k^i$ . . . . .	34
2.4.4	Experimental analysis . . . . .	36
2.5	Experimental Results . . . . .	37
2.5.1	Experimental setup . . . . .	37
2.5.2	Random colliding segments . . . . .	41
2.5.3	Random collision-free segments . . . . .	42
2.5.4	PRM planning . . . . .	43
2.5.5	Path Smoothing . . . . .	46
2.5.6	Bad-case scenarios . . . . .	47
2.6	Refined Checker . . . . .	50
2.7	Conclusion . . . . .	53
<b>3</b>	<b>Dealing with Narrow Passages</b>	<b>55</b>
3.1	Introduction . . . . .	55
3.2	Related Work . . . . .	57
3.2.1	Narrow passages . . . . .	57
3.2.2	Sampling and connection strategies . . . . .	58
3.2.3	Filtering strategies . . . . .	59
3.2.4	Retraction strategies . . . . .	60
3.3	Motivation from Experimental Observations . . . . .	61
3.3.1	Observation #1 . . . . .	61
3.3.2	Observation #2 . . . . .	63
3.4	Object Thinning . . . . .	64
3.4.1	Medial-axis computation . . . . .	65
3.4.2	Thinning method . . . . .	67

3.4.3	Object thinning as a pre-computation step . . . . .	69
3.5	Small-Step Retraction Planner . . . . .	71
3.5.1	OPTMIST . . . . .	72
3.5.2	PESSIMIST . . . . .	73
3.5.3	Overall Planner . . . . .	75
3.6	Experimental Results . . . . .	75
3.6.1	Test environments . . . . .	76
3.6.2	Results . . . . .	78
3.6.3	Discussion . . . . .	80
3.7	Conclusion . . . . .	82
<b>4</b>	<b>Multi-goal motion planning</b>	<b>84</b>
4.1	Introduction . . . . .	84
4.2	Related Work . . . . .	88
4.3	Problem Formulation . . . . .	90
4.3.1	Overview . . . . .	90
4.3.2	Function PATH . . . . .	91
4.3.3	Function TOUR . . . . .	92
4.3.4	Objective . . . . .	93
4.4	Lazy Planning Algorithm . . . . .	94
4.4.1	Algorithm . . . . .	94
4.4.2	Analysis . . . . .	95
4.4.3	Improvements . . . . .	97
4.5	Implementation of PATH . . . . .	98
4.6	Experimental Results . . . . .	100
4.6.1	Non-partitioned case . . . . .	105
4.6.2	Partitioned case . . . . .	107
4.6.3	Improved lazy algorithm . . . . .	108
4.6.4	Influence of parameter $\alpha$ . . . . .	110
4.7	Conclusion . . . . .	112

<b>5 Planning for Deformable Linear Objects</b>	<b>114</b>
5.1 Introduction . . . . .	114
5.2 Planning for Robotic Manipulation of DLOs . . . . .	116
5.3 Related Work . . . . .	117
5.3.1 Manipulation planning . . . . .	117
5.3.2 Application of knot theory in robotics . . . . .	118
5.3.3 Vision-based DLO manipulation . . . . .	118
5.4 Modeling a Deformable Linear Object . . . . .	119
5.4.1 Geometric model . . . . .	119
5.4.2 Physical model . . . . .	119
5.4.3 Topological model . . . . .	120
5.5 DLO Manipulation Planning Problem . . . . .	121
5.6 Planning Approach . . . . .	122
5.6.1 Forming sequence . . . . .	122
5.6.2 Loop structure . . . . .	123
5.6.3 Pierced and slip loops . . . . .	124
5.6.4 Motion planning algorithm . . . . .	127
5.7 Tying knots around static rigid objects . . . . .	129
5.8 Experimental Results . . . . .	131
5.8.1 Results in simulation . . . . .	131
5.8.2 Test on hardware platform . . . . .	131
5.8.3 Sensitivity analysis . . . . .	135
5.9 Conclusion . . . . .	136
<b>6 Conclusion</b>	<b>137</b>
<b>A Overview of SBL Planner</b>	<b>141</b>
<b>Bibliography</b>	<b>144</b>

# List of Tables

2.1	Comparison of COLL-CHECKER, GREEDY-DIST, EXACT-DIST, and APPROX-DIST . . . . .	29
2.2	$L_{ref}$ (in units) in different examples. . . . .	40
2.3	Comparison of FIXED-RES-BISECTION with $\varepsilon = 0$ and ADAPTIVE-BISECTION with $\delta = 0$ and $\delta = 0.005$ . . . . .	41
2.4	Comparison of FIXED-RES-BISECTION with $\varepsilon = 0.006$ and ADAPTIVE-BISECTION with $\delta = 0.005$ . . . . .	42
2.5	Average running times of SBL on five distinct path-planning problems	45
2.6	Average running times of A-SBL for successive values of $\delta$ (see main text in Section 2.5.4). . . . .	45
2.7	Results for random walks and path smoothing . . . . .	47
2.8	Number of BV/triangle pairs tested in Figure 2.14a . . . . .	48
2.9	Number of BV/triangle pairs tested in Figure 2.14b . . . . .	48
2.10	Number of BV/triangle pairs tested in Figure 2.14c . . . . .	49
2.11	Comparison of ADAPTIVE-BISECTION and REFINED-ADAPTIVE-BISECTION . . . . .	52
4.1	Comparison of NAIVE-GMGP and LAZY-GMGP, when goal configurations are non-partitioned and $\alpha = 1$ (all times are in seconds) . . . . .	105
4.2	Comparison of NAIVE-GMGP with LAZY-GMGP when each goal group contains 5 configurations and $\alpha = 1$ (all times are in seconds) . . . . .	107
4.3	Results obtained with LAZY-MGP-2 when each goal group contains 5 configurations and $\alpha = 1$ (all times are in seconds) . . . . .	110

5.1 Means and standard deviations of Gaussian distributions from which the three main rope model parameters were chosen for the robustness analysis . . . . .	136
---	-----

# List of Figures

1.1	The basic PRM planning . . . . .	5
1.2	Why PRMs work well? . . . . .	6
2.1	<i>a</i> : Two skinny 20-DOF arms. <i>b</i> : IRB 2400 robot with thin arc welding gun and snapshots along a straight path segment in configuration space	14
2.2	Adaptive bisection algorithm . . . . .	19
2.3	Different covering strategies . . . . .	22
2.4	Greedy distance computation algorithm . . . . .	26
2.5	A bad case for GREEDY-DIST: two distant objects are bounded by disjoint RSSs that are very close . . . . .	28
2.6	Planar linkage with three revolute joints and one prismatic joint . . .	31
2.7	Computation of sphere $S(i, k)$ . . . . .	35
2.8	Construction of $S(i, k)$ at Step 4 of algorithm COMPUTE-SPHERE . .	35
2.9	Histograms showing the distributions of the quality of the bound $\lambda_i(\mathbf{q}, \mathbf{q}')$ for three robots. . . . .	36
2.10	Reference links of IRB 2400 and F 200 robots . . . . .	38
2.11	Examples with thin obstacles. . . . .	38
2.12	<i>a</i> : F 200 robot with arc welding gun, machine tool. <i>b</i> : IRB 2400 robot in workshop. <i>c</i> : Six IRB 2400 robots. <i>d</i> : Six F 200 robot . . . . .	39
2.13	Curve traced by the tip of the end-effector of the robot . . . . .	43
2.14	Three bad-case examples for ADAPTIVE-BISECTION . . . . .	47
2.15	Refined adaptive bisection algorithm . . . . .	51
2.16	Finding elements to insert into the priority queue $Q$ . . . . .	51

3.1	Planning a path through a narrow passage of varying width (Observation #1) . . . . .	62
3.2	Choosing between two narrow passages (Observation #2) . . . . .	65
3.3	Construction of a sub-MA for a box-shaped object . . . . .	66
3.4	Different thinning methods . . . . .	68
3.5	Thinning of sub-MA . . . . .	68
3.6	Thinned robot component . . . . .	69
3.7	Thinning the ABB IRB 2400 robot . . . . .	70
3.8	The algorithm OPTIMIST . . . . .	72
3.9	The algorithm for repairing a configuration that lies in $\partial^*F$ . . . . .	73
3.10	A “pathological” example where OPTIMIST is likely to fail . . . . .	74
3.11	The small-step retraction planner (SSRP) . . . . .	75
3.12	Test environments with narrow passages . . . . .	77
3.13	Test environments without narrow passages . . . . .	78
3.14	Number of polygons in models . . . . .	79
3.15	Performance of OPTIMIST, PESSIMIST, SSRP, and plain SBL in the environments of Figure 3.12 . . . . .	79
3.16	Average number of milestones generated by OPTIMIST, PESSIMIST, and plain SBL in the environments (a) through (d) of Figure 3.12 . . . . .	80
3.17	Performance of OPTIMIST, PESSIMIST, SSRP, and plain SBL in the two environments of Figure 3.13 . . . . .	80
3.18	Standard deviations, means, and maximums of the running times needed to connect start and goal configurations . . . . .	83
4.1	A spot-welding workcell (courtesy of General Motors). . . . .	85
4.2	Two goal configurations in a group . . . . .	86
4.3	Two-dimensional generalized multi-goal motion planning problem . .	86
4.4	TOUR algorithm . . . . .	92
4.5	Lazy multi-goal planning algorithm . . . . .	94
4.6	A worst-case example for LAZY-GMGP . . . . .	98
4.7	Improvement of Step 2.3.3 of LAZY-GMGP . . . . .	99

4.8	Path optimizer . . . . .	101
4.9	(a) Example 1 (10 goal groups); (b) some goal configurations . . . . .	102
4.10	(a) Example 2 (31 goal groups); (b) some goal configurations . . . . .	103
4.11	(a) Example 3 (50 goal groups); (b) some goal configurations . . . . .	104
4.12	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)	106
4.13	Lengths of paths returned by LAZY-GMGP for ten different sets of non-partitioned goals for Examples 1, 2 and 3 . . . . .	108
4.14	Replacing Step 2.3 of LAZY-GMGP by the above steps results in LAZY-GMGP-2 . . . . .	109
4.15	Running time of LAZY-GMGP, length of solution, and number of calls to PATH on Examples 1, 2, and 3 when $\alpha$ grows from 1 to 3 . . . . .	111
5.1	Initial (a) and goal (b)-(c) states in a manipulation problem . . . . .	117
5.2	Representation of a DLO . . . . .	119
5.3	(a): The crossings in the “figure-8” knot. (b): The sign convention for the crossings . . . . .	120
5.4	A knot can be tied crossing-by-crossing in the order implied by its forming sequence . . . . .	123
5.5	The loop structure for the “figure-8” knot . . . . .	123
5.6	(a): Reidemeister move I (b): Reidemeister move II . . . . .	124
5.7	Passive/static sliding supports (tri-needle) is used to maintain the size of the loops to be pierced in future. . . . .	125
5.8	A semi-tight configuration . . . . .	125
5.9	A needle, inspired from real-life, is used to preserve a slip loop. . . . .	126
5.10	In real life, sliding supports (fingers in (a) and scissors in (b)) are commonly used while tying knots. . . . .	126
5.11	The DLO manipulation planning algorithm . . . . .	127
5.12	Searching the configuration space of the DLO . . . . .	128
5.13	Tri-needle placement . . . . .	129

5.14	Taking into account topological interactions of the DLO with rigid objects in the environment. . . . .	130
5.15	Sequences of snapshots along manipulation plans generated by TWO-ARM-KNOTTER for five manipulation problems. . . . .	132
5.16	Two PUMA-560 arms tying a bowline knot . . . . .	133
5.17	(a): Different types of ropes used. (b): Final shapes of bowline knots achieved with different ropes. . . . .	134
6.1	(a): Large number of robots involved in a manufacturing cell. (b): Congested workcell (courtesy of General Motors) . . . . .	140

# Chapter 1

## Introduction

### 1.1 Motion Planning

Motion planning can be broadly defined as the ability for an agent (e.g., a robot, a digital character) to compute its own motions in order to achieve certain goals specified by spatial arrangements of objects.

For example:

- The agent may be a robot arm whose goal is to assemble a product given its individual parts lying on a table. The robot must determine the order in which to assemble the parts, the position of its gripper to grasp each part, and the trajectories for moving the parts.
- The goal of an autonomous lunar multi-legged vehicle may be to bring scientific equipment to a specified location. The robot must select the successive placements of its feet on the ground and the motions of its legs to reach them while maintaining quasi-static or dynamic balance.
- An autonomous two-arm surgical robot may be requested to suture a patient's severed blood vessel at the end of a surgical procedure. The robot must decide how to manipulate a nylon suture through the vessel's severed ends and achieve knots of a certain type.
- A sentry military robot may be introduced in a building to search for potential

enemies. The robot must choose the successive locations where to go in order to sweep the building in such a way that no enemy will remain undetected.

Being a key aspect in intelligent automation, motion planning has attracted considerable research interest for more than three decades. Each specific motion planning problem is usually cast in a parameter space where the solution can be expressed as a curve, called a *path*, e.g., one connecting two given points. The most common parameter space is the *configuration space*, in which each point uniquely determines the placement of the robot in its workspace [84]. Constraints are mapped into that space as forbidden regions that a feasible path must avoid. For instance, workspace obstacles map into configuration space to a subset of configurations where the robot collides with obstacles. A collision-free path is a continuous curve lying in the complement of this subset, often called the *free space*. Computing collision-free paths between given pairs of *query* configurations is often referred to as the *basic* motion planning problem.

Many different types of feasibility constraints can be mapped as forbidden regions in configuration space. For example, if the task of a robot is to track a moving target, then at each instant of time the forbidden region is the subset of configurations from which the target is not visible to the robot (due to visual occlusion); in that case, as the target moves, the forbidden region varies over time and is best represented in configuration  $\times$  time space. If the robot is a multi-legged robot navigating on rough terrain, the quasi-static equilibrium constraint defines subset of configurations where the robot is not in equilibrium. Other constraints apply to the local shape of a path (e.g., kino-dynamic constraints) or to its global shape (e.g., optimality constraints).

## 1.2 Traditional Motion Planning Approaches

Methods developed to solve the basic motion planning problem are often divided into three main approaches [27, 75]:

- (a) **Roadmap approach.** It tries to capture the connectivity of the free space  $F$  by a network of 1D curves, called the roadmap. To represent the connectivity of

$F$  well, a roadmap  $R$  must be such that (1) any configuration in  $F$  can be easily connected to  $R$  and (2) there is a one-to-one correspondence between the connected components of  $R$  and those of  $F$ . Once a roadmap  $R$  has been constructed, each motion-planning query is processed by first connecting the two query configurations to two points of  $R$  and then searching the roadmap for a path connecting these two points. Classical examples of roadmaps include visibility graphs [84], Voronoi diagrams [91], and silhouettes [22].

**(b) Cell decomposition approach.** Here the key idea is to decompose  $F$  into non-overlapping cells such that any two configurations in a cell can be easily connected by a path. A connectivity graph is constructed that represents the adjacency relation between the cells. To find a path between a pair of query configurations, this graph is searched for a sequence of cells connecting the two cells containing the query configurations. A final path is extracted from this sequence. The trapezoidal decomposition [75] (for planar configuration spaces and polygonal obstacles) and the Morse decomposition [1] (for non-planar configuration spaces and non-polygonal obstacles) are well-known cell decomposition methods. In some ways, the cell decomposition approach can be seen as a dual to the roadmap approach.

**(c) Potential field approach.** Here, unlike in the previous two approaches, no connectivity graph is precomputed. Instead, an artificial potential function  $U$  is constructed and used as a heuristics to guide the search of a path. The function  $U$  is often constructed as the sum of an attractive potential (that pulls the robot toward the goal configuration) and a repulsive potential (that repels the robot away from the obstacles) [67]. The negated gradient  $-\nabla U(q)$  at a given robot configuration  $q$  suggests the direction of motion at  $q$ . Because there is no or little precomputation, the potential field approach has been very popular in scenarios requiring on-line real-time navigation [27, 67, 75]. However, potential field methods can easily get trapped into local minima of  $U$  and it is notoriously difficult to construct local-minima-free potential functions [69].

Until recently, these planning approaches have been mainly applied to simple scenarios where geometry has low complexity and robots have few degrees of freedom.

### 1.3 Probabilistic Roadmaps

During the last decade, Probabilistic Roadmap (PRM) methods have emerged as an effective approach to solve complex motion planning problems [3, 4, 27, 49, 53, 66, 104]. They have proven capable of computing the motions of single-robot and multiple-robot systems with many degrees of freedom operating in complex geometric environments (e.g., modeled with millions of triangles) [27]. They can also take into account various types of motion constraints such as collision avoidance, kinodynamic [52, 77], stability [17], visibility [31], and contact [61] constraints. Beyond robotics, they have been used to synthesize the motions for digital actors [116] and to predict motions of biological macro-molecules such as proteins [7].

PRM planning belongs to the roadmap approach presented in the previous section. The main difference, however, is that it does not attempt to construct an exact representation of the shape of the robot's free space  $F$ . Indeed, in most practical scenarios, computing the shape of  $F$  is prohibitively expensive. So, instead, PRM planning works by constructing an extremely simplified, sample-based approximate representation of  $F$ , called a (probabilistic) *roadmap*. This roadmap is a graph whose nodes, called *milestones*, are configurations sampled from the free space  $F$  according to some suitable probability measure, which reflects the uncertainty on the actual shape of  $F$  [54]. An edge, called a *local path*, connects a pair of milestones if a path of simple shape (usually the straight path) joins them in  $F$ . The robot's start and goal configurations,  $s$  and  $g$ , are included among the milestones in the roadmap. Once the roadmap is constructed, a path connecting  $s$  and  $g$  is extracted using standard graph-searching techniques. These steps of the PRM planning are also depicted in Figure 1.1. PRM planning relies on the availability of efficient collision-checking techniques to test whether sampled configurations and local paths lie in  $F$ .

One can think of a probabilistic roadmap as a network of guards (the milestones) watching over  $F$ . Previous theoretical studies have shown that the reason why PRM

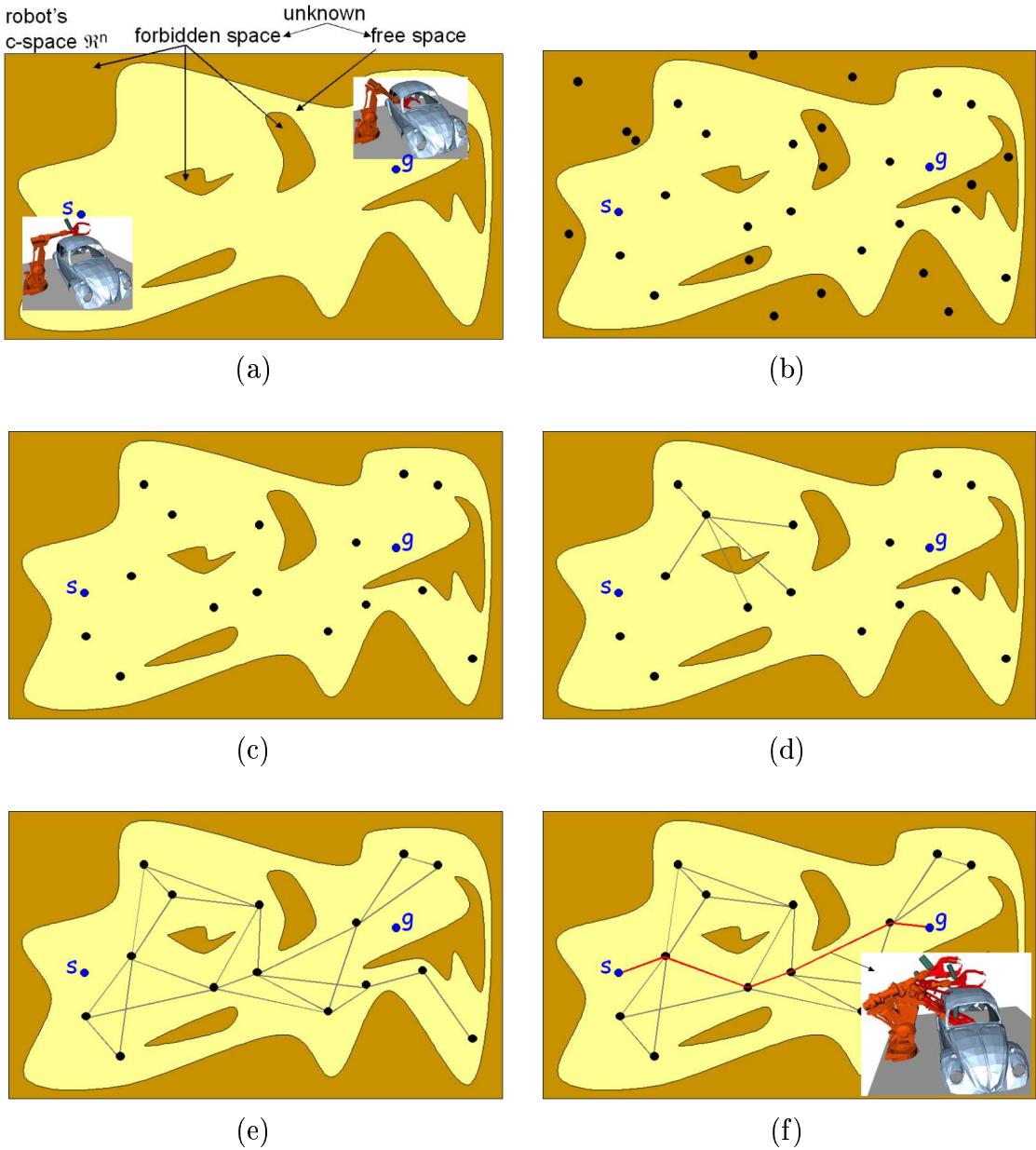


Figure 1.1: The basic PRM planning [66]. (a) Planning is done in the robot's free space, shape of which is unknown. (b) Configurations are sampled using some probability measure. (c) Sampled configurations are tested for collision and the collision-free ones (lying in the free space) are retained as milestones. (d) Each milestone is linked by straight paths to its  $k$ -nearest neighbors. (e) Collision-free links are retained to form the PRM. (f) Finally,  $s$  and  $g$  are connected to the PRM and the PRM is searched for a path (red polyline) from  $s$  to  $g$ . Figure derived from [76].

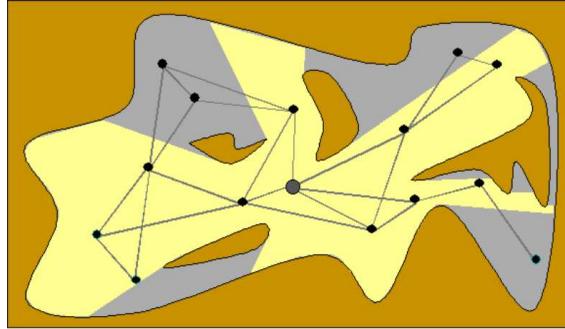


Figure 1.2: The gray region “illuminated” by the milestone (near the center, shown bigger) is a large fraction of the free space. PRMs work well in practice because in most free spaces every configuration illuminates a significant fraction of the feasible space. Figure courtesy of [76].

planning works well in practice is that a free space  $F$  typically encountered in a real-life scenario verifies favorable visibility properties, such that a relatively small number of milestones and local paths are sufficient to represent  $F$  well enough to answer motion-planning queries correctly with high probability [53, 65] (also see Figure 1.2). However, visibility properties are usually not uniformly favorable or defavorable across  $F$ . Hence, a key component of any PRM planner is the algorithm – called the *sampling strategy* – that chooses the probability measure to use in each milestone sampling operation. This measure should allocate a higher density of milestones to regions of  $F$  that are expected to have inferior visibility, e.g., to regions expected to contain narrow passages. Some sampling strategies predict such regions from the geometry of the workspace, e.g., by identifying narrow passages in the workspace and mapping them into configuration space [44, 48, 59]. Other strategies, known as adaptive strategies, use the information gained while constructing the roadmap [57]. In this case, the resulting probability measure varies over time. Yet another family of strategies, known as filtering strategies, first over-sample  $F$ , but then identify local pattern to only retain a small subset of the generated samples as milestones [16, 50].

A PRM planner may pre-compute a roadmap for a given robot and workspace, and then use this roadmap to process many planning queries, each defined by a distinct pair  $(s, g)$  [66]. Such a planner is called a *multi-query* PRM planner. It must construct

a roadmap that “covers” the entire free space well. In contrast, *single-query* planners build a new roadmap for each new query  $(s, g)$ . This roadmap needs only linking  $s$  to  $g$  and therefore does not have to achieve a good coverage of  $F$ . Single-query planners are usually much faster than multi-query planners, even to process several queries in the same setting. RRT [71] and SBL [104] are two examples of single-query planners. Throughout this thesis, we use SBL as our “base” PRM planner in our implemented software. An overview of SBL is given in the appendix of this thesis.

Although PRM planning has been extremely successful in solving complex motion planning problems, a number of difficult issues remain. In particular, the presence of narrow passages in the free space is still considered a major bottleneck, which may dramatically slow down PRM planning. As problems become more complex and require the processing of larger numbers of motion-planning queries, PRM planning must be made more efficient, e.g., by developing faster collision-checking techniques and better sampling strategies. On the application side, PRM planning has mostly been applied so far to “basic” motion planning problems (as defined in Section 1.3). Only recently, has PRM planning made significant inroads in other types of problems. There still exist many important planning problems that have no satisfactory solutions, PRM-based or not.

## 1.4 Thesis Contributions

Our contributions are at the two levels suggested above. On the one hand, we propose two new methods – a dynamic collision checker and a sampling strategy based on fattening the free space  $F$  – to increase both the efficiency and reliability of PRM planning. On the other hand, we extend PRM planning to solve two new important planning problems: multi-goal planning and manipulation planning for linearly deformable objects (DLOs). We briefly introduce these contributions below. They are developed in more detail in the following chapters of this thesis.

**(1) Dynamic collision checking.** A PRM planner performs two types of collision checks: *static* checks to test whether a sampled configuration is in  $F$  and *dynamic*

checks to test whether a local path lies entirely in  $F$ . Static checks are usually carried out using an efficient “bounding-volume hierarchy” technique [42]. On the other hand, the classical approach to perform dynamic checks is to sample each local path at some fixed, prespecified resolution and statically check each sampled configuration for collision.<sup>1</sup> But this approach is approximate and can miss collisions along the path. One may reduce the chances of missing a collision by refining the sampling resolution along the entire local path, but this results into a slower dynamic checker, and, so, into a slower overall planner. To address this important issue, we have devised a new dynamic checker based on adaptive path sampling, which exactly determines whether a local path lies in  $F$ , or not. It chooses the adaptive sampling resolution along the local path by comparing lower bounds on distances between objects in relative motion with upper bounds on lengths of curves traced by points of these moving objects. We have done extensive tests of our new method, showing that its running time compares favorably with that of a fixed-resolution approach at a “suitable” resolution, with the enormous advantage that it never fails to detect a collision. Our new checker is not restricted to PRM planning, and also has potential applications beyond PRM planning, for instance in graphic animation and physical simulation.

**(2) Dealing with narrow passages.** It is well-known that the efficiency of PRM planners can drop dramatically if the free space contains narrow passages, as it is difficult to sample and connect milestones through such passages. For many years this problem has been considered to be the main bottleneck for PRM planning [16, 50, 51]. We have developed a new sampling strategy, which we call *small-step retraction* (SSR), that allows PRM planners to efficiently construct roadmaps in free spaces with narrow passages. SSR slightly fattens the robot’s free space  $F$  into a new space  $F^*$  that completely contains  $F$ . SSR then constructs a roadmap in  $F^*$ , and finally repairs portions of the roadmap that are not in  $F$  by resampling more configurations around them. The intuition behind this method is

---

<sup>1</sup>This sampling of local paths to perform collision checks should not be confused with the sampling of milestones to construct a roadmap.

that, as the fattening operation expands narrow passages relatively more than the rest of  $F$ , visibility properties are more favorable in  $F^*$  than in  $F$ . So, it is easier to construct a roadmap in  $F^*$  than in  $F$ . However, no part of  $F^*$  is far away from  $F$ , so repairing the roadmap can be done efficiently. Of course, the exact shape of fattened free space  $F^*$  is not explicitly computed. Instead, the geometric models of workspace objects (robot links and/or obstacles) are thinned around their medial axis in a precomputation stage. To construct a roadmap in  $F^*$ , collision checks (static and dynamic) are performed using the thinned models. Comparative tests show that SSR achieves significant speedups (sometimes by two orders of magnitude) over a pre-existing PRM planner SBL [104].

**(3) Multi-goal motion planning (MGP).** MGP occurs in industrial manufacturing tasks such as spot welding, car painting, inspection, and measurement, where the end-effector of a robotic arm must reach several input goal configurations in some sequence. This sequence is not given, and the goal is to compute an optimal or near-optimal tour through the goals. MGP combines two notoriously hard computational problems: the traveling salesman problem (TSP) and the collision-free shortest-path problem. While a typical PRM planner can be used to compute near-optimal paths between pairs of goals, additional reasoning is needed to determine a near-optimal ordering on the goals without invoking the PRM planner a quadratic number of times (once for each pair of goals), as that would often be too costly. To solve this problem, we have designed a “lazy” multi-goal planner that avoids calls to the PRM planner by delaying them until they are needed. It does so by initializing the distance between every two goals to an easily computed lower bound. Then it runs the PRM planner on pairs of goals until a tour has been found to be within a given factor of the optimal tour. Our planner also handles the case where goals are specified in terms of the robot’s end-effector placements and each end-effector placement can be achieved with several configurations of the robot arm (distinct solutions of the arm’s inverse kinematics). In this case, the set of goal configurations of the robot is partitioned into groups, and the planner’s objective is to compute a robot tour that visits one configuration in each group and

is near optimal over all configurations in every goal group and over all group orderings.

**(4) Manipulation planning for deformable linear objects.** A number of tasks require manipulating deformable objects, in particular deformable linear objects (DLOs), such as ropes, cables, and surgical sutures. However, research in manipulation planning (including PRM-based manipulation planning) has mainly focused on rigid objects [106]. A noticeable exception is the work described in [73]. Indeed, deformable objects, which can take many different shapes when submitted to external forces, add a number of challenging issues to the planning process. We present a new PRM planner that computes the motions and (re-)grasp operations of a two-arm system in order to tie self-knots of DLOs, as well as knots around simple static objects. Here, unlike in traditional motion planning problems, the goal is a topological state of the world (the crossings of the DLO with itself and with static obstacles), rather than a geometric one. The planner constructs a topologically-biased probabilistic roadmap in the DLO’s configuration space. We have experimented with it in simulation with several goals representing knots like the bowline, neck-tie, bow (shoe-lace), and stunsail knots. We have also successfully tested the motions computed by the planner on a two-PUMA robot hardware platform with various household ropes.

The following four chapters describe these contributions in detail.

# Chapter 2

## Adaptive Dynamic Collision Checking\*

### 2.1 Introduction

In virtually all real-life motion-planning scenarios, a PRM planner must perform many collision checks – typically tens of thousands to hundreds of thousands – in order to build a roadmap. In fact, PRM planners spend most of their running times (often 99% or more) performing such checks. So, collision checks must be very fast, even when the workspace has complex geometry, but they must also be accurate, i.e., they must not miss collisions or incorrectly detect collisions.

There are two types of collision checks, *static* and *dynamic*. In a static check a single configuration of objects is tested for overlaps, while in a dynamic check an entire path (hence, a continuous set of configurations) is tested for overlaps among objects. Static checks are used to test the milestones of a roadmap and dynamic checks to test its local paths. Static checks are usually carried out with efficient Bounding Volume Hierarchy (BVH) techniques [43, 58, 68, 108]. There is little space for further improvement here, except perhaps by developing and exploiting specialized

---

\*This chapter is based on the journal article: “Adaptive Dynamic Collision Checking for Single and Multiple Articulated Robots in Complex Environments”, *IEEE Transactions on Robotics*, 21(3):338-353, June 2005. Fabian Schwarzer and Jean-Claude Latombe are the co-authors of the article.

computing hardware (e.g., GPUs). On the other hand, dynamic collision checking techniques are far from being entirely satisfactory. Our work reported in this chapter has focused on improving them.

Traditionally, a dynamic check is done by sampling a given local path at a pre-specified resolution and statically checking each of configuration sampled along the path. Hence, a PRM planner usually spends much more time testing local paths than milestones. A coarse resolution is often chosen to make dynamic checks faster, but this choice also increases the risk of missing collisions. Conversely, a fine resolution increases reliability, but also slows down the overall planning process. Any tradeoff is delicate and depends on object geometry. In this chapter, a new dynamic collision checker is described which works with an adaptive resolution, in order to achieve both reliability and speed. This checker is guaranteed to never miss a collision. Our experiments show that a PRM planner using this checker is both reliable and efficient timewise. The applicability of our new checker is not limited to PRM planning or motion planning in general. For example, it can also be used to test continuous paths in graphic animation and physical simulation [19, 62, 83].

### 2.1.1 Previous work on dynamic collision checking

Four major families of methods have been previously proposed for dynamic collision checking:

- *Feature-tracking* methods track pertinent features (vertices, edges, faces) of two objects – usually, the pair of closest features – to determine if the objects remain separated along a path. They rely on the following *coherence assumption*: the pertinent features change relatively rarely and, when they do change, the new ones can be computed efficiently from the old ones [11, 28, 81, 82, 86]. This assumption requires each object to be made of few convex components and to have relatively small geometric complexity. It is poorly verified by kinematic chains (e.g., robot arms) in practical environments. Then, paths must be tested by tiny increments, to avoid missing collisions, especially collisions involving links at the end of the chains.

- *Bounding-volume hierarchy* (BVH) methods pre-compute, for each object (robot link, obstacle), a hierarchy of BVs (e.g., spheres, boxes) that approximates the geometry of the object at successive levels of detail [43, 58, 68, 74, 95, 108]. To check two objects for collision, their BVHs are searched from the top down, making it possible to quickly discard large subsets of the objects contained in disjoint BVs. Such methods have been applied to complex objects with surfaces described by several 100,000 triangles, and more [43, 74]. But they are fundamentally static methods (and are actually used by most PRM planners to test the roadmap milestones). As described above, to test a path, the common approach is to check intermediate configurations spaced along the path at a prespecified resolution. If all these configurations are found collision-free, then the path is declared collision-free, but this answer may not be correct.
- *Swept-volume intersection* methods compute the volumes swept out by the objects and test these volumes for overlap [20, 37]. However, exact computation of swept volumes is expensive, especially when objects undergo rotations and have complex geometry. Moreover, the overlap test can no longer be speeded up by using pre-computed data structures, such as BVHs. Another difficulty is that swept volumes for pairs of moving objects may overlap even when the objects do not collide. Hence, when multiple objects move relative to each other, one must either consider the relative motions for all pairs, or compute and test volumes swept out in 4D space-time. Both ways yield costly computations.
- *Trajectory parameterization* methods express the geometry of the objects along the tested path by algebraic polynomials in terms of a single variable  $t$  [21, 105]. These polynomials are then used to construct a collision condition on  $t$ . In principle, finding the values of  $t$  verifying this condition gives the exact intervals of collision along a given path. But, in general, the polynomials have high degrees, so that solving the condition for  $t$  can be prohibitively expensive and furthermore pose numerical problems. Simplifications yielding polynomial equations of degrees no greater than 3 are proposed in [97].



Figure 2.1: *a*: Two skinny 20-DOF arms (320 triangles each) in environment with three fixed thin tori (6,000 triangles each), and detail showing arms in top ring. *b*: IRB 2400 robot with thin arc welding gun (3,500 triangles) and snapshots along a straight path segment in configuration space

Combinations of methods have also been proposed, where a BVH method is used to filter out irrelevant parts of objects. For instance, in [34] the bounding surface of each object is decomposed into convex surface patches, out of which a hierarchy of convex hulls is created. During a collision check, the precomputed BVHs are used to quickly prune pairs of surface patches that cannot intersect and a feature-tracking method is applied to test the remaining pairs. Similarly, in [98], a BVH method (augmented with interval-arithmetic techniques) is combined with a trajectory-parameterization method [97]. The methods in [34, 98] can find the time of first contact between two objects along a short trajectory segment, hence are well suited for haptic interaction and dynamic simulation, where penetration could lead to an inappropriate collision response. In contrast, in applications like motion planning, it is more crucial to determine as quickly as possible if a given trajectory of arbitrary length collides.

### 2.1.2 Fixed-resolution dynamic checking

Hence, while there exist effective methods for static collision checking (e.g., BVH methods), dynamic checking remains a practical bottleneck in many applications. In particular, probabilistic roadmap (PRM) planners rely on the availability of efficient checkers to test simple path segments (called local paths, usually straight segments)

between randomly sampled configurations (called milestones) [4, 10, 14, 53, 63, 102]. Most PRM planners use a static BVH method to test intermediate configurations spaced along each local path at some given resolution  $\varepsilon$  (in the configuration space metric). These configurations are usually obtained by recursively bisecting the local path, until either a collision is found, or any two successive configurations are closer apart than  $\varepsilon$  [102, 30].

Choosing  $\varepsilon$  requires a delicate compromise between efficiency and reliability. This is especially true in scenarios with articulated arms and/or multiple robots. Rather large values of  $\varepsilon$ , which reduce the number of static checks, may be acceptable when robot links and obstacles are fat. But when these objects are thin or have sharp edges, the checker will then easily miss collisions. Trying to increase reliability by reducing  $\varepsilon$  results in slow checking of path segments. In the example of Figure 2.1a, which contains two long skinny serial linkages and three thin obstacles, tiny changes in joint angles can make the linkages jump over obstacles and/or each other. In Figure 2.1b, a small rotation of the robot’s base may cause the welding gun to pass through an obstacle. In both examples,  $\varepsilon$  must be set very small for collisions to be reliably detected, yielding a slow dynamic checker.

One way to address this difficulty is to “grow” object models [8], by pre-computing the Minkowski sums of the original models and a sphere of radius  $\varrho$ . The value of  $\varrho$  is chosen such that, if the grown models do not overlap at the intermediate configurations, then the original models are guaranteed to be collision-free between these configurations. However, this further complicates the choice of  $\varepsilon$ . A large  $\varrho$  allows a large  $\varepsilon$ , but it also increases the chances that a path segment is incorrectly found to collide. To reduce the number of false collisions, while rarely missing true collisions, one must choose both  $\varrho$  and  $\varepsilon$  small, which yields again an inefficient checker. The checker proposed in [8] addresses this difficulty by pre-computing several grown copies of each model, each with a different value of  $\varrho$ . Then, at each step of a segment test, the checker switches to the largest collision-free grown model and adjusts the next step size accordingly. However, this approach is expensive (in both memory and time) and requires difficult tuning.

### 2.1.3 Adaptive bisection of paths

This chapter introduces a new dynamic collision checking method, based on BVH, that avoids these difficulties by locally adjusting the value of  $\varepsilon$ . By comparing lower bounds on distances between objects in relative motions with upper bounds on lengths of curves traced by points of these moving objects, this method automatically decides whether a path segment between two collision-free configurations needs to be bisected further. It not only frees the user from choosing  $\varepsilon$ ; it also guarantees that no collision will ever be missed. Our checker is applicable to path segments of any given shape (e.g., straight, circular) in configuration space or collections of such segments (e.g., multi-segment paths). It is particularly suited for scenarios with manipulator arms and/or multiple robots, where it can detect collision between robot links and obstacles, as well as between links of the same or different robots.

The basic idea of relating distances to path lengths has been suggested before (e.g., [19, 10]), though rarely implemented. We exploit this idea further by separately adjusting the bisection resolution for different pairs of objects. Indeed, very few pairs of objects (link-obstacle or link-link) require bisecting a path down to the same resolution. Exploiting this fact leads to testing a rapidly decreasing number of object pairs at each new level of bisection. At the end of this chapter, we push this idea one step further by separately adjusting the resolution for different pairs of BVs.

We also give a number of new techniques and heuristics that make the approach more efficient, especially when it is used in PRM planners and/or applied to geometrically complex environments. One is a BVH algorithm that computes non-trivial lower bounds on distances between pairs of objects almost as efficiently as if it was only testing the objects for collision. Another technique bounds link motions: given a path segment of a robot in configuration space and a link of this robot, it quickly computes an upper bound on the lengths of the curves traced in workspace by all points in this link.

We have extensively tested our dynamic checker on randomly generated path segments and multi-segment paths produced by randomized planners. These tests show that its running time compares favorably to that of a fixed-resolution checker at “suitable” resolution, with the enormous advantage that our checker never fails to

detect collision.

### 2.1.4 Chapter organization

Section 2.2 describes our dynamic collision checking method based on adaptive bisection. Sections 2.3 and 2.4 respectively present the techniques used to compute lower bounds on distances between objects and upper bounds on lengths of curves traced out by points of moving objects. Section 2.5 discusses experimental results on various examples that were obtained with the implemented new adaptive checker and a fixed-resolution checker, for comparison. Section 2.6 refines the method by separately adjusting the bisection resolution for different pairs of BVs (instead of objects). Section 2.7 summarizes our work, describes its limitations, and points to possible future work.

Throughout this chapter, we model the workspace as the Euclidean space  $\mathbf{R}^3$ . Distances between objects and lengths of curves in workspace are all measured using the Euclidean metric.

## 2.2 Adaptive Dynamic Collision Checking

We begin by establishing a basic result on which our algorithm is based.

### 2.2.1 Basic result

We consider the robot(s) and all obstacles as a collection of rigid objects  $\mathcal{A}_1, \dots, \mathcal{A}_n$  whose placements in workspace are uniquely determined by a tuple  $\mathbf{q} = (q_1, \dots, q_d)$ , the configuration of the system. All objects are allowed to move.

Let  $\mathcal{A}_i(\mathbf{q})$  denote object  $\mathcal{A}_i$  at configuration  $\mathbf{q}$ . Let  $\eta_{ij}(\mathbf{q})$  be any non-trivial lower bound on the Euclidean distance between  $\mathcal{A}_i(\mathbf{q})$  and  $\mathcal{A}_j(\mathbf{q})$ , *i.e.*,  $\eta_{ij}(\mathbf{q}) = 0$  if and only if  $\mathcal{A}_i(\mathbf{q})$  and  $\mathcal{A}_j(\mathbf{q})$  overlap.

Each  $\mathcal{A}_i$  is a set of points. Each point traces a distinct curve segment in workspace when the configuration of the system is interpolated between configurations  $\mathbf{q}_a$  and  $\mathbf{q}_b$  along a path segment  $\pi$ . For a given  $\pi$ , we define  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  to be an upper bound on

the lengths of the curves traced by all points in  $\mathcal{A}_i$ , such that  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) = 0$  whenever  $\mathcal{A}_i$  stays fixed during the interpolation (for example, if  $A_i$  is a fixed obstacle).

We now state a sufficient condition for two objects  $\mathcal{A}_i$  and  $\mathcal{A}_j$  not to collide along  $\pi$  in configuration space:

**Theorem 1** *Two objects  $\mathcal{A}_i$  and  $\mathcal{A}_j$  do not collide at any configuration  $\mathbf{q}$  on the path segment  $\pi$  joining  $\mathbf{q}_a$  and  $\mathbf{q}_b$  in configuration space, if:*

$$\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b) < \eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b). \quad (2.1)$$

*Proof:* Assume that Inequality (2.1) is verified. If  $\mathcal{A}_i$  and  $\mathcal{A}_j$  collide, then a point  $\mathbf{p}_i$  of  $\mathcal{A}_i$  must coincide with a point  $\mathbf{p}_j$  of  $\mathcal{A}_j$  at some intermediate configuration  $\mathbf{q}_c$  along  $\pi$ . Let  $\ell_i(\mathbf{q}, \mathbf{q}')$  be the length of the curve traced by  $\mathbf{p}_i$  between any two configurations  $\mathbf{q}$  and  $\mathbf{q}'$  in  $\pi$ . Define  $\ell_j(\mathbf{q}, \mathbf{q}')$  in the same way for point  $\mathbf{p}_j$ . For  $\mathbf{p}_i$  and  $\mathbf{p}_j$  to coincide at  $\mathbf{q}_c$ , we must have:

$$\begin{aligned} \ell_i(\mathbf{q}_a, \mathbf{q}_c) + \ell_j(\mathbf{q}_a, \mathbf{q}_c) &\geq \eta_{ij}(\mathbf{q}_a), \\ \ell_i(\mathbf{q}_c, \mathbf{q}_b) + \ell_j(\mathbf{q}_c, \mathbf{q}_b) &\geq \eta_{ij}(\mathbf{q}_b). \end{aligned}$$

Since  $\ell_i(\mathbf{q}_a, \mathbf{q}_c) + \ell_i(\mathbf{q}_c, \mathbf{q}_b) = \ell_i(\mathbf{q}_a, \mathbf{q}_b)$  and  $\ell_j(\mathbf{q}_a, \mathbf{q}_c) + \ell_j(\mathbf{q}_c, \mathbf{q}_b) = \ell_j(\mathbf{q}_a, \mathbf{q}_b)$ , summing the previous two relations yields:

$$\ell_i(\mathbf{q}_a, \mathbf{q}_b) + \ell_j(\mathbf{q}_a, \mathbf{q}_b) \geq \eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b).$$

Using  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) \geq \ell_i(\mathbf{q}_a, \mathbf{q}_b)$  and  $\lambda_j(\mathbf{q}_a, \mathbf{q}_b) \geq \ell_j(\mathbf{q}_a, \mathbf{q}_b)$ , we get:

$$\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b) \geq \eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b)$$

which contradicts our initial hypothesis that Inequality (2.1) is verified. So,  $\mathcal{A}_i$  and  $\mathcal{A}_j$  do not collide. ■

The reverse of Theorem 1 is not true:  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b)$  may exceed  $\eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b)$  without introducing a collision.

**Algorithm** ADAPTIVE-BISECTION( $\mathbf{q}, \mathbf{q}'$ )

1. Initialize priority queue  $Q$  with  $[\mathbf{q}, \mathbf{q}']_{ij}$  for all pairs of objects ( $\mathcal{A}_i, \mathcal{A}_j$ ) that need to be tested.
2. While  $Q$  is not empty do
  - 2.1  $[\mathbf{q}_a, \mathbf{q}_b]_{ij} \leftarrow \text{remove-first}(Q)$
  - 2.2 If  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b) \geq \eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b)$  then
    - 2.2.1  $\mathbf{q}_{mid} \leftarrow \text{mid-configuration along path segment between } \mathbf{q}_a \text{ and } \mathbf{q}_b$
    - 2.2.2 If  $\eta_{ij}(\mathbf{q}_{mid}) = 0$  then return *collision*
    - 2.2.3 Else insert  $[\mathbf{q}_a, \mathbf{q}_{mid}]_{ij}$  and  $[\mathbf{q}_{mid}, \mathbf{q}_b]_{ij}$  into  $Q$
3. Return *no collision*

Figure 2.2: Adaptive bisection algorithm

### 2.2.2 Adaptive bisection algorithm

Our algorithm uses Theorem 1 to decide whether a given path segment between two collision-free configurations  $\mathbf{q}_a$  and  $\mathbf{q}_b$  must be bisected: if Inequality (2.1) is verified for all pairs of objects  $A_i$  and  $A_j$ , then the segment is collision-free; otherwise, it must be bisected. Theorem 1 guarantees that no collision can be missed.

However, considering all pairs of objects simultaneously may yield a large amount of unnecessary work. Indeed, pairs of objects that are well-separated and undergo small displacements require fewer bisections than pairs that are closer and/or undergo greater displacements. So, taking advantage of the fact that Inequality (2.1) applies to an individual pair of objects, we check this condition for each pair, independent of the other pairs, and discard those pairs which verify the inequality. In this way, as a path segment gets bisected at a finer resolution, the number of remaining object pairs tends to drop quickly. Moreover, we can perform the tests in an order that speeds up the discovery of a collision, when there is one.

Figure 2.2 shows our algorithm, ADAPTIVE-BISECTION, to check a path segment between two collision-free configurations  $\mathbf{q}$  and  $\mathbf{q}'$ . It maintains a priority queue  $Q$  of elements of the form  $[\mathbf{q}_a, \mathbf{q}_b]_{ij}$ . The presence of  $[\mathbf{q}_a, \mathbf{q}_b]_{ij}$  in  $Q$  indicates that the objects  $\mathcal{A}_i$  and  $\mathcal{A}_j$  still need to be tested for collision between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . At Step 1,  $Q$

is filled with the elements  $[\mathbf{q}, \mathbf{q}']_{ij}$ , for all object pairs that need to be tested. Then, at each loop of Step 2, the first element, say  $[\mathbf{q}_a, \mathbf{q}_b]_{ij}$ , is removed from  $Q$ . If this element satisfies Inequality (2.1), then  $\mathcal{A}_i$  and  $\mathcal{A}_j$  cannot possibly collide between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ , and the algorithm continues with the next element in the queue. Else it computes  $\eta_{ij}(\mathbf{q}_{mid})$ , where  $\mathbf{q}_{mid}$  is the mid-configuration along the path segment between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . If this computation reveals a collision — that is, if  $\eta_{ij}(\mathbf{q}_{mid}) = 0$  — then the algorithm reports the collision and halts. Otherwise, two new elements,  $[\mathbf{q}_a, \mathbf{q}_{mid}]_{ij}$  and  $[\mathbf{q}_{mid}, \mathbf{q}_b]_{ij}$ , are inserted into  $Q$ . When  $Q$  is empty, the path segment between  $\mathbf{q}$  and  $\mathbf{q}'$  is reported free of collision.

In Sections 2.3 and 2.4, we will describe the techniques used in our implementation of ADAPTIVE-BISECTION to compute bounds on distances and curve lengths.

### 2.2.3 Ordering of the priority queue

If the path segment tested by ADAPTIVE-BISECTION is collision-free, then the ordering of  $Q$  has no impact on the running time of the algorithm, since all elements in  $Q$  will eventually have to be processed. But, for a colliding segment, an appropriate ordering can lead to finding a collision quicker. This is important in applications like PRM planning, where a large fraction of candidate paths are colliding.

In [102, 30], it was shown that in practice the prior probability of a path segment to be colliding increases sharply with its length in configuration space. This result justifies bisecting a segment into two sub-segments of equal lengths. The planners in [102, 89] exploit this result further to test multi-segment paths, by maintaining a priority queue of (sub-)segments sorted by decreasing lengths and treating the longest (sub-)segment first.

ADAPTIVE-BISECTION takes also advantage of the computed bounds on both distances between objects and lengths of traced curves. Intuitively, two objects are more likely to collide when they are closer to each other at one or both segment endpoints and/or the points in these objects trace longer curves. This intuition is directly related to Inequality (2.1) and leads us to sort the entries  $[\mathbf{q}_a, \mathbf{q}_b]_{ij}$  in the priority queue by decreasing values of the difference:  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b) - \eta_{ij}(\mathbf{q}_a) - \eta_{ij}(\mathbf{q}_b)$ . Our

tests show that, on average, this heuristic ordering leads to finding a collision faster than sorting  $Q$  by decreasing lengths of segments.

### 2.2.4 Checking multi-segment paths

ADAPTIVE-BISECTION can be extended to concurrently test multiple segments forming a continuous, multi-segment path. This is simply done by filling  $Q$ , at Step 1, with the elements  $[\mathbf{q}, \mathbf{q}']_{ij}$ , for all segments  $[\mathbf{q}, \mathbf{q}']$  and all object pairs that need to be tested. The same heuristic ordering of  $Q$  as above can be used. In general, it will lead to discovering a colliding segment before having spent much time testing collision-free segments.

However, a slightly more useful implementation in practice is to maintain several priority queues: one for the path and one for each segment. The priority queues for the individual segments are maintained as above. The queue for the path contains one entry per segment that has not yet been shown to be collision-free. The entries of the path queue are sorted by decreasing values of the differences  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b) - \eta_{ij}(\mathbf{q}_a) - \eta_{ij}(\mathbf{q}_b)$ , each computed for the first element of the corresponding segment's queue. Intuitively, this ordering corresponds to placing the segment that is the most likely to collide on top of the path queue. The advantage of using several queues is the following: if a segment in the path is eventually found to collide, we then cache the priority queue associated with each of the other segments that has not yet been found to collide or not to collide. If any of these segments must later be tested for collision, the cached queue of this segment is re-used, hence saving the collision-checking work previously done. This improvement is particularly important when the collision checker is used in a PRM planner that delays collision tests [102].

The same extension actually holds for an arbitrary collection of segments.

### 2.2.5 Covering strategies

Inequality (2.1) can be illustrated by the following diagram: draw a line segment of length  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j(\mathbf{q}_a, \mathbf{q}_b)$  and two circles of radii  $\eta_{ij}(\mathbf{q}_a)$  and  $\eta_{ij}(\mathbf{q}_b)$  centered at the endpoints of this segment. See Figure 2.3a. (The segment in this figure is *not*

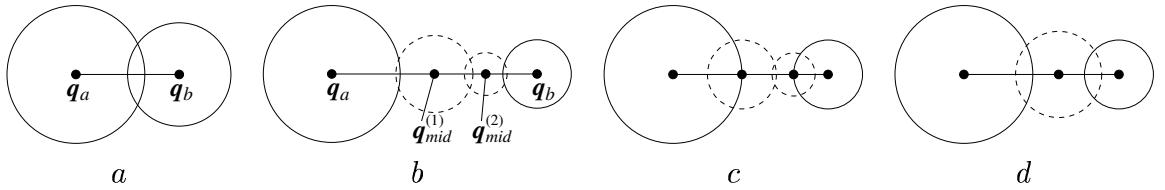


Figure 2.3: Different covering strategies (see text)

the path segment in configuration space.) We call the circles the *covering circles* of  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . If they cover the entire segment, as is the case in Figure 2.3a, then  $\mathcal{A}_i$  and  $\mathcal{A}_j$  do not collide along the straight path joining  $\mathbf{q}_a$  and  $\mathbf{q}_b$  in configuration space. Otherwise, and if there is no collision along the path, ADAPTIVE-BISECTION will produce intermediate configurations  $\mathbf{q}_{mid}$  whose covering circles will eventually complete the coverage of the entire line segment, as depicted in Figure 2.3b.

The above diagram, though only illustrative (as it is embedded neither in configuration space, nor in workspace), suggests decomposition/covering strategies of the path between  $\mathbf{q}_a$  and  $\mathbf{q}_b$  other than bisecting at the midpoint. For example, we could compute the two configurations where the covering circles of  $\mathbf{q}_a$  and  $\mathbf{q}_b$  intersect the segment (Figure 2.3c), and check whether their covering circles complete the covering of the segment. This strategy would require inserting at most one new entry into the priority queue at each iteration, instead of two. But half the coverage by the new circles would be wasted to cover parts of the segment that were covered by previous circles. This could nevertheless be a useful strategy if one wanted to check a path by small increments from one end to the other, as is the case, for example, in haptic interaction with virtual worlds and physical simulation in order to find the time of first contact [34], [98].

Another strategy would be to place a configuration at the middle point of the uncovered section (assuming it can be easily computed), as illustrated in Figure 2.3d. In term of coverage, this would be slightly better than the bisection strategy of our algorithm. However, our experimental tests indicate that the potential gains are small. Moreover, this strategy produces different intermediate configurations for different pairs of objects. This is a drawback with articulated linkages, since it then

requires computing the forward kinematics of these linkages at many more configurations. Instead, with the strategy of Figure 2.3b, the results of the forward kinematics computation done to determine the placement of a link at some configuration can be cached and later re-used to retrieve or compute the placement of another link in the same linkage at the same configuration.

### 2.2.6 Bounding the running time of ADAPTIVE-BISECTION

Ignoring floating-point arithmetics issues, ADAPTIVE-BISECTION always gives a correct answer in a finite amount of time. However, this time is not bounded in the worst case, as one can easily create examples where the algorithm would bisect an arbitrary number of times. Very bad cases are unlikely in practice, but they may eventually occur when several thousands of path segments are tested, as is often the case in PRM planning.

There are several ways to deal with this issue. One is to switch to another dynamic collision-checking method when a (sub-)segment shorter than some threshold requires being bisected further. Then, the potential collision is already well localized, so that only restricted subsets of the objects (hence, small number of triangles) need to be considered. A similar idea has been previously exploited in [34, 98].

Another way is to modify slightly the definition of the lower bound  $\eta_{ij}(\mathbf{q})$  on the distance between  $\mathcal{A}_i$  and  $\mathcal{A}_j$  at configuration  $\mathbf{q}$ . In the new definition,  $\eta_{ij}(\mathbf{q}) = 0$  whenever the actual distance between the objects is less than some small predefined  $\delta$ , and  $\eta_{ij}(\mathbf{q}) \geq \delta$  otherwise. Any  $\delta > 0$  results in bounding the running time of ADAPTIVE-BISECTION. Then, the algorithm is slightly conservative: while it still cannot miss a collision, it may incorrectly return that a path segment is colliding when two objects come closer than  $\delta$  apart. Note that choosing  $\delta$  is very different from setting the resolution  $\varepsilon$  of a fixed-resolution checker. A fixed-resolution checker always breaks a segment into sub-segments of length  $\varepsilon$  to determine that the segment is collision-free. In contrast, in most cases, our checker stops bisecting before any  $\eta_{ij}(\mathbf{q})$  gets smaller than  $\delta$ . Setting  $\delta$  is also different from growing the objects by some  $\varrho$  as suggested at the end of Section 2.1.2. Indeed,  $\varrho$  must be chosen large

enough to prevent any collision from happening along sub-segments of length  $\varepsilon$ . So,  $\varrho$  is directly related to the length of the maximal displacement of points in the moving objects, and can be quite large.

In our implementation, we bound the running time of ADAPTIVE-BISECTION by setting a threshold  $\delta$  as described above. Our experiments show that  $\delta$  can be set very small without affecting significantly the average running time of ADAPTIVE-BISECTION, meaning that the threshold is rarely needed in practice.

## 2.3 Bounding Distances Between Objects

ADAPTIVE-BISECTION requires that non-trivial lower bounds on distances between objects be efficiently computed. Tighter bounds may yield fewer bisections, but are usually more expensive to compute. Here, we describe a greedy algorithm, based on bounding volume hierarchies (BVHs), that computes lower bounds on distances at about the same cost as if it was testing the objects for collision.

### 2.3.1 Distance computation using BVHs

Bounding volume hierarchies (BVHs) are widely used to check collision and/or compute distances between objects of high geometric complexity. An object’s BVH is an approximately balanced binary tree whose leaves are triangles of the object surface and intermediate nodes are BVs, each bounding the triangles below it. It is pre-computed using techniques such as those described in [43, 74, 95]. Various types of BVs have been proposed, including spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB), and rectangle swept spheres (RSS) [83].

To determine if two objects collide at a given configuration, their hierarchies are searched from the top down. Each step of a search path consists of testing whether two BVs or two triangles, one from each hierarchy, are separated. A collision is reported whenever two triangles are found to intersect. A search path is terminated as soon as two tested BVs are separated, because disjoint BVs cannot contain intersecting triangles. In the worst case, if each object has  $O(n)$  triangles, the algorithm takes

$O(n^2)$  time. But, in practice, the average running time is often sub-linear, as many search paths are terminated early.

The same algorithm can be used to compute the *exact* distance between two objects, with few changes to maintain the closest distance  $d$  found so far. Initially,  $d$  is set to a very large number. A search path is terminated whenever the distance between two tested BVs is greater than  $d$ . If a search path reaches a pair of triangles and the distance between these triangles is less than  $d$ , then  $d$  is re-set to this new distance. The algorithm terminates as soon as  $d = 0$  (the objects are colliding) or when all search paths have been terminated. In both cases, the final value of  $d$  is the distance between the two objects.

BVH methods are widely used, because they have been found more efficient in practice than other techniques, especially for complex objects. Nevertheless, they take much more time to compute exact distances than to check collision. As any two tested BVs are more likely to be disjoint than separated by at least  $d > 0$ , search paths to compute distances are longer.

Two approaches have been proposed to speed up distance computation:

- One approach is to prune search paths more effectively, either by re-using data computed at a previous configuration, or by better selecting which search path to explore next [74, 78]. For example, *triangle caching* initializes  $d$  to the distance between the two closest triangles at the previous configuration. But it is effective only if the coherence assumption is verified. *Priority directed search* schedules the pending tests of BVs and triangles into a priority queue sorted by distances. It does not rely on any coherence assumption.
- The other approach computes an *approximate* distance with a *guaranteed* bound on the relative error [74, 95]. The algorithm in [95] computes a lower bound  $d'$  on the distance  $d$  between two objects, such that  $(d - d')/d \leq \gamma$ , where  $0 < \gamma < 1$  is an input constant. It initializes  $d'$  to a large value (as in the exact case). When it finds that two triangles are closer than  $d'$  apart, it resets  $d'$  to be  $1 - \gamma$  times the distance between them. The final value of  $d'$  verifies  $(1 - \gamma)d \leq d' \leq d$ . The algorithm becomes faster as  $\gamma$  is increased, but it is still

```

Algorithm GREEDY-DIST( $B_i, B_j$ )
1.  $d \leftarrow \text{distance}(B_i, B_j)$ 
2. If  $B_i$  and  $B_j$  are both triangles then return  $d$ 
3. If  $d > 0$  then return  $d$ 
4. If  $B_i$  is bigger than  $B_j$  then switch  $B_i$  and  $B_j$ 
5. Set  $B_{j1}$  and  $B_{j2}$  to the two children of  $B_j$  in the BVH
6.  $\alpha \leftarrow \text{GREEDY-DIST}(B_i, B_{j1})$ 
7. If  $\alpha > 0$  then
    7.1  $\beta \leftarrow \text{GREEDY-DIST}(B_i, B_{j2})$ 
    7.2 If  $\beta > 0$  then return  $\min\{\alpha, \beta\}$ 
8. Return 0

```

Figure 2.4: Greedy distance computation algorithm

slower than pure collision checking. The algorithm in [74] computes a upper bound on the distance.

In fact, the speed of distance computation turns out to be crucial for the overall efficiency of the new adaptive dynamic collision checking approach. That is, while ADAPTIVE-BISECTION may perform fewer bisections to test a path than a fixed-resolution checker with small  $\varepsilon$ , it could nevertheless take longer to run because standard methods for distance computation are much slower than pure collision checking. The next section therefore proposes a new algorithm, called GREEDY-DIST, that computes lower distance bounds almost as fast as a typical BVH algorithm checks for collision.

### 2.3.2 Greedy distance computation algorithm

To compute a lower bound on the distance between two objects, our implementation of ADAPTIVE-BISECTION calls  $\text{GREEDY-DIST}(B_\alpha, B_\beta)$ , where GREEDY-DIST is the algorithm shown in Figure 2.4 and  $B_\alpha$  and  $B_\beta$  identify the root BVs of the hierarchies representing these objects. The call returns a positive lower bound on the distance, if the objects are separated, and 0 otherwise.

`GREEDY-DIST` works like a classical BVH collision checker [74]. It follows the same search paths, tests the same pairs of BVs and triangles, and terminates each search path when a pair of tested BVs has null intersection. But, instead of testing if two BVs or triangles intersect, it computes the distance between them and eventually returns the smallest distance found. It is faster than an approximate distance computation algorithm because it skips the additional search needed to verify that the relative error is smaller than a given  $\gamma$ . Though `GREEDY-DIST` offers no guarantee on the relative error, our tests show that it returns a good approximation on average.

`GREEDY-DIST` is independent of the choice of BV, as long as the distance between pairs of BVs can be computed efficiently. In our implementation, we use RSSs. An RSS is defined as the Minkowski sum of a rectangle and a sphere [74]. The RSS of an object is created by first estimating the two principal directions spanned by the object [43]. A rectangle  $R$  is then constructed along these directions to enclose the projection of the object onto the plane defined by these directions. The RSS is the Minkowski sum of  $R$  and the sphere whose radius is half the length of the interval spanned by the object along the direction perpendicular to  $R$ . In comparison, the OBB of the object is the cross-product of  $R$  by this interval. It is often possible to improve the quality of the RSS fit by shrinking  $R$  (each side can be reduced by at most twice the radius). Like OBBs, RSSs provide reasonably tight fits for objects of various shapes. But distance computation between RSSs is faster than between OBBs [74].

To bound the running time of `ADAPTIVE-BISECTION` using the parameter  $\delta$  (see Section 2.2.6), we modify the algorithm of Figure 2.4 by replacing the zeros in step 3, 7, and 7.2 by  $\delta$ . Then, `GREEDY-DIST` returns a positive lower bound whenever the distance is greater than  $\delta$ , and 0 otherwise.

### 2.3.3 Experimental analysis

As Figure 2.5 illustrates, in a bad case, the relative error on a bound computed by `GREEDY-DIST` can be arbitrarily close to 1 (when  $\delta = 0$ ). Our experiments, however, show that the average bounds returned by `GREEDY-DIST` are quite good

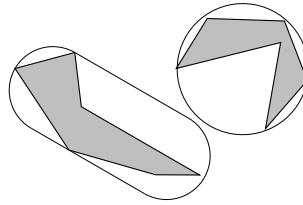


Figure 2.5: A bad case for GREEDY-DIST: two distant objects are bounded by disjoint RSSs that are very close

and compare favorably to those computed (at a greater cost) by an approximate distance computation algorithm.

We tested GREEDY-DIST (with  $\delta = 0$ ) and three similar BVH algorithms: COLL-CHECKER for pure collision checking, EXACT-DIST for exact distance computation, and APPROX-DIST for approximate distance computation with relative error  $\gamma = 0.5$ . APPROX-DIST uses the approach of [95], hence returns a value between  $0.5d$  and  $d$ , where  $d$  is the exact distance. We use RSSs in EXACT-DIST and APPROX-DIST, as in GREEDY-DIST, but OBBs in COLL-CHECKER. Indeed, while being closely related to RSSs, OBBs are slightly faster to test for overlap. Finally, we “tuned” EXACT-DIST using both priority directed search and triangle caching (see Section 2.3.1). To make the best use of triangle caching, we initialize the distance  $d$  (used for pruning the search) with the exact distance (computed separately). This leads EXACT-DIST to terminate a search path as soon as a BV pair is found to be further apart than the exact distance. In practice, such perfect initialization is impossible, and could only be approached in cases where the coherence assumption is verified extremely well.

We compared the performance of the four algorithms as follows. For each of the seven environments shown in Figures 2.1a, 2.11a-b and 2.12a-d, we generated 1,000 random configurations of the robot(s) and, at each configuration, we tested all robot links against all fixed obstacles. The results are summarized in Table 2.1. Column 1 identifies the environment and column 2 indicates the total number of object pairs examined by each of the four algorithms. Columns 3-6 give the average numbers of pairs of BVs/triangles tested per query by each of the four algorithms. As expected, the numbers for COLL-CHECK and GREEDY-DIST are about the same. The small differences result from the fact that one uses OBBs and the other RSSs.

Fig.	# queries	COLL-CHECKER	GREEDY-DIST	EXACT-DIST	APPROX-DIST	$\mu$	$\mu'$
2.1a	40,000	1.7 / 0.1	2.1 / 0.1	859 / 141	3.7 / 0.1	0.99	0.53
2.11a	54,000	47 / 0.6	54 / 0.8	946 / 78	105 / 2.0	0.82	0.64
2.11b	42,000	26 / 1.0	25 / 0.8	827 / 67	58 / 1.9	0.81	0.61
2.12a	14,000	14 / 0.4	14 / 0.3	1,444 / 268	70 / 2.2	0.64	0.60
2.12b	22,000	42 / 0.9	46 / 1.0	819 / 125	129 / 5.4	0.51	0.58
2.12c	18,000	25 / 1.3	26 / 1.4	1,248 / 128	111 / 3.2	0.57	0.62
2.12d	21,000	2.6 / 0.1	3.9 / 0.2	703 / 170	74 / 16	0.58	0.58

Table 2.1: Comparison of COLL-CHECKER, GREEDY-DIST, EXACT-DIST, and APPROX-DIST

Since computing the distance between two RSSs is only a small factor slower than testing two OBBs for overlap, GREEDY-DIST is almost as fast as COLL-CHECK. In contrast, EXACT-DIST examines many more pairs of BVs and triangles, despite the perfect initialization of triangle caching. APPROX-DIST also examines significantly more BVs and triangles than GREEDY-DIST; its running time is greater in the same ratio.

The last two columns of Table 2.1 measure the quality of the bounds. Column 7 gives the average ratio  $\mu$  of the bound returned by GREEDY-DIST by the exact distance, in each environment, computed only for the collision-free pairs of objects. The values of  $\mu$  are excellent for the first three environments in which obstacles are tightly bounded by RSSs, and they remain greater than 0.5 in the other four environments, where objects have diverse shapes. The last column gives the average ratio  $\mu'$  computed for the bounds returned by APPROX-DIST. In the first three environments,  $\mu'$  is smaller than  $\mu$ ; in the last four, the two factors are similar, meaning that GREEDY-DIST returns on average similar bounds, at smaller computational cost.

For ADAPTIVE-BISECTION, the average performance of the algorithm computing distance bounds matters more than its worst-case performance. Indeed, if GREEDY-DIST returns a bad lower bound, then ADAPTIVE-BISECTION may have to bisect the

segment once more and call GREEDY-DIST again at a new configuration. The probability of encountering several bad cases in a row is small. We have tested ADAPTIVE-BISECTION with GREEDY-DIST, EXACT-DIST, and APPROX-DIST (with different values of  $\gamma$ ). The best results were obtained with GREEDY-DIST.

## 2.4 Bounding Motions in Workspace

ADAPTIVE-BISECTION also requires computing an upper bound  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  on the lengths of the curve segments traced by all points of a moving object  $\mathcal{A}_i$ , when the system is interpolated between  $\mathbf{q}_a$  and  $\mathbf{q}_b$  along some path segment  $\pi$ . Tight bounds could be computed by numeric integration, but this may be quite slow in practice. Moreover, the cost of the computation would increase with the distance between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ .

In this section, we derive the expression of an upper bound  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  that is fast to evaluate, and is valid for kinematic chains with revolute and prismatic joints. We first establish this expression on a simple example, then in the general case. Next, we propose a technique to compute constant factors appearing in this expression. Finally, we experimentally evaluate the quality of the bound.

Throughout this section, we adopt simplifying conventions that do not restrict the generality of our results. We assume a one-to-one correspondence between the joint parameters of the robot(s) and moving obstacles and the configuration parameters. We define these parameters such that: (1) a rotation of any revolute joint by some angle  $\omega$  (in radians) produces a variation of  $\omega$  of the corresponding configuration parameter, and (2) a translation of any prismatic joint by some vector  $\mathbf{v}$  results in a variation of  $\|\mathbf{v}\|$  of the corresponding parameter.

### 2.4.1 Example

We first establish an expression of  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  for the planar linkage of Figure 2.6 moving along a *linear* path segment (i.e., a straight line in configuration space). This linkage has three revolute joints corresponding to joint parameters  $q_1$ ,  $q_2$ , and  $q_4$ , and

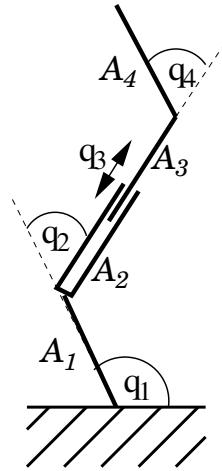


Figure 2.6: Planar linkage with three revolute joints and one prismatic joint

one prismatic joint corresponding to  $q_3$ . Each of the links  $\mathcal{A}_1, \dots, \mathcal{A}_4$  has length  $L$  and zero width. Each revolute joint can perform a single full rotation, hence we define  $q_1$ ,  $q_2$ , and  $q_3$  to vary, each, between 0 and  $2\pi$ . When prismatic joint 3 is completely retracted, the distance between revolute joints 2 and 4 (equivalently, between the base of  $\mathcal{A}_2$  and the tip of  $\mathcal{A}_3$ ) is  $L$ . When joint 3 is maximally extended, this distance is  $L + D$ . So, we define  $q_3$  to vary between 0 and  $D$ .

For this linkage, we can write the bounds  $\lambda_i$  ( $i = 1, \dots, 4$ ) as follows:

$$\begin{aligned}\lambda_1(\mathbf{q}_a, \mathbf{q}_b) &= L |q_{a,1} - q_{b,1}| \\ \lambda_2(\mathbf{q}_a, \mathbf{q}_b) &= 2L |q_{a,1} - q_{b,1}| + L |q_{a,2} - q_{b,2}| \\ \lambda_3(\mathbf{q}_a, \mathbf{q}_b) &= (2L + D) |q_{a,1} - q_{b,1}| \\ &\quad + (L + D) |q_{a,2} - q_{b,2}| + |q_{a,3} - q_{b,3}| \\ \lambda_4(\mathbf{q}_a, \mathbf{q}_b) &= (3L + D) |q_{a,1} - q_{b,1}| \\ &\quad + (2L + D) |q_{a,2} - q_{b,2}| + |q_{a,3} - q_{b,3}| \\ &\quad + L |q_{a,4} - q_{b,4}|\end{aligned}$$

The bound  $\lambda_1(\mathbf{q}_a, \mathbf{q}_b)$  is established by considering the point in  $\mathcal{A}_1$  that is the furthest away from the center of rotation of joint 1. While the second term of  $\lambda_2(\mathbf{q}_a, \mathbf{q}_b)$  is

established in a similar way, its first term is derived by considering the maximal distance between a point of  $\mathcal{A}_2$  and the center of rotation of joint 1. This distance is achieved by the tip of  $\mathcal{A}_2$  when  $\mathcal{A}_1$  and  $\mathcal{A}_2$  are aligned. Clearly, no point of  $\mathcal{A}_2$  can move by a larger amount than the sum of the two terms defining  $\lambda_2(\mathbf{q}_a, \mathbf{q}_b)$ . The other two bounds are generated in a similar way.

We can write each of the bounds above in the form:

$$\lambda_i(\mathbf{q}_a, \mathbf{q}_b) = \sum_{k=1}^i R_k^i |q_{b,k} - q_{a,k}|$$

where  $i = 1, \dots, 4$ . If  $k$  is a prismatic joint, then  $R_k^i = 1$ ; otherwise,  $R_k^i$  is an upper bound on the distances between the points of  $\mathcal{A}_i$  and the center of rotation of joint  $k$ . So, in the above expression, we have:

$$\begin{aligned} R_1^1 &= L \\ R_1^2 &= 2L, \quad R_2^2 = L \\ R_1^3 &= 2L + D, \quad R_2^3 = L + D, \quad R_3^3 = 1 \\ R_1^4 &= 3L + D, \quad R_2^4 = 2L + D, \quad R_3^4 = 1, \quad R_4^4 = L \end{aligned}$$

#### 2.4.2 Upper bound in general case

In the following, we focus our attention on the moving object  $\mathcal{A}_i$ . Without loss of generality, we let  $q_1, \dots, q_i$  denote the configuration parameters that influence  $\mathcal{A}_i$ 's placement. We first assume that the system configuration is linearly interpolated between  $q_a$  and  $q_b$  (Theorem 2). Next, we show how the result extends to non-linear path segments between  $q_a$  and  $q_b$ .

**Theorem 2** *When the system configuration is linearly interpolated between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ , an upper bound on the lengths of the curves traced by the points of object  $\mathcal{A}_i$  is:*

$$\lambda_i(\mathbf{q}_a, \mathbf{q}_b) = \sum_{k=1}^i R_k^i |q_{b,k} - q_{a,k}| \tag{2.2}$$

where  $q_1, \dots, q_i$  are the configuration parameters that influence the placement of  $\mathcal{A}_i$ , and  $R_k^i > 0$  is a constant factor defined as follows:

- if  $k$  is a prismatic joint, then  $R_k^i = 1$ ,
- otherwise,  $R_k^i$  is an upper bound on the distances between the points of  $\mathcal{A}_i$  and the axis of rotation of joint  $k$ .

*Proof:* Let  $\mathbf{p}$  be an arbitrary point of  $\mathcal{A}_i$ . The straight path segment in configuration space between  $\mathbf{q}_a$  and  $\mathbf{q}_b$  can be parameterized by  $\mathbf{q}(t) = (1-t)\mathbf{q}_a + t\mathbf{q}_b$ , with  $t \in [0, 1]$ . Let  $\mathcal{A}_i(t)$  and  $\mathbf{p}(t)$  denote the placement of  $\mathcal{A}_i$  and  $\mathbf{p}$  at configuration  $\mathbf{q}(t)$ .

The length  $L_{\mathbf{p}}$  of the curve traced by  $\mathbf{p}(t)$  when  $t$  varies from 0 to 1 is:

$$L_{\mathbf{p}} = \int_0^1 \|\dot{\mathbf{p}}(t)\| dt \quad (2.3)$$

where:

$$\dot{\mathbf{p}}(t) = \sum_{k=1}^i \frac{\partial \mathbf{p}}{\partial q_k} \dot{q}_k(t).$$

We can bound the modulus of  $\dot{\mathbf{p}}(t)$  by applying the triangle inequality:

$$0 \leq \|\dot{\mathbf{p}}(t)\| \leq \sum_{k=1}^i \left\| \frac{\partial \mathbf{p}}{\partial q_k} \right\| |\dot{q}_k(t)|. \quad (2.4)$$

Along the straight path between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ , we have:

$$\dot{q}_k(t) = q_{b,k} - q_{a,k}. \quad (2.5)$$

Moreover, by definition of the configuration parameters and the constants  $R_k^i$ , we have:

$$\left\| \frac{\partial \mathbf{p}}{\partial q_k} \right\| \leq R_k^i. \quad (2.6)$$

By first plugging (2.4) into (2.3) and then using relations (2.6) and (2.5), we get:

$$\begin{aligned} L_{\mathbf{p}} &\leq \int_0^1 \sum_{k=1}^i \left\| \frac{\partial \mathbf{p}}{\partial q_k} \right\| |\dot{q}_k(t)| dt \\ &\leq \sum_{k=1}^i R_k^i \int_0^1 |\dot{q}_k(t)| dt \end{aligned}$$

$$= \sum_{k=1}^i R_k^i |q_{b,k} - q_{a,k}|. \quad (2.7)$$

Since we made no assumption on the location of  $\mathbf{p}$  in  $\mathcal{A}_i$ , this bound holds for all points of  $\mathcal{A}_i$ . ■

Note that the bound  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  defined by Equation (2.2) is null if and only if  $\mathbf{q}_a = \mathbf{q}_b$ . In general, we use Equation (2.2) to compute both  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  and  $\lambda_j(\mathbf{q}_a, \mathbf{q}_b)$  in (2.1). However, for two links  $i$  and  $j$  ( $j < i$ ) on the same kinematic chain, we can derive tighter bounds by considering motions in the local frame of link  $j$ . Thus,  $\lambda_j(\mathbf{q}_a, \mathbf{q}_b) = 0$  and for  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$ , we simply sum over  $k = j + 1, \dots, i$  instead of  $k = 1, \dots, i$  in (2.2).

The result in Theorem 2 can be extended to parameters  $q_k(t)$  that are non-linear functions in the “pseudo-time”  $t$ , as long as  $|\dot{q}_k(t)|$  can be bounded. An upper bound on  $|\dot{q}_k(t)|$  then replaces  $|q_{b,k} - q_{a,k}|$  in (2.7), hence in (2.2). This modification covers any linkage made of revolute and/or prismatic joints, including those with closed loops. (Note that parallelogram mechanisms, which involve only linear parametric changes, can be handled directly without this extension.) It also makes it possible to bound curve lengths when the path segment  $\pi$  in configuration space is not straight.

### 2.4.3 Computation of factors $R_k^i$

We now describe a general technique to compute the constant factors  $R_k^i$  in Equation (2.2) for robot arms in three-dimensional workspace, when  $k$  is a revolute joint. Recall that  $R_k^i$  must be an upper bound on the distances between the points of  $\mathcal{A}_i$  and the axis of rotation of joint  $k$ .

Our technique computes a sphere bounding the volume swept out by each link when the configuration parameters vary over their full ranges (which we take to be at least  $2\pi$  for revolute joints). Thus, the computed factors  $R_k^i$  are independent of any considered path segment in configuration space, and are computed only once during pre-processing.

If  $i = k$ , then we compute the distances of all link vertices to the axis of rotation of joint  $i$  and return  $R_i^i$  to be the maximum of these distances.

**Algorithm** COMPUTE-SPHERE( $i, k$ )

1. If  $i = k$  then  $S(i, k + 1) \leftarrow \text{ENCLOSING-SPHERE}(\mathcal{A}_i)$
2. Else  $S(i, k + 1) \leftarrow \text{COMPUTE-SPHERE}(i, k + 1)$
3. If joint  $k$  is prismatic then  
Sweep  $S(i, k + 1)$  along the full translational range of joint  $k$  and  
construct the sphere  $S(i, k)$  that tightly encloses the swept volume.
4. Else if joint  $k$  is revolute then  
Sweep  $S(i, k + 1)$  around the axis of joint  $k$  by performing a full  $2\pi$   
rotation and construct the sphere  $S(i, k)$  that tightly encloses  
the swept volume.
5. Return  $S(i, k)$

Figure 2.7: Computation of sphere  $S(i, k)$

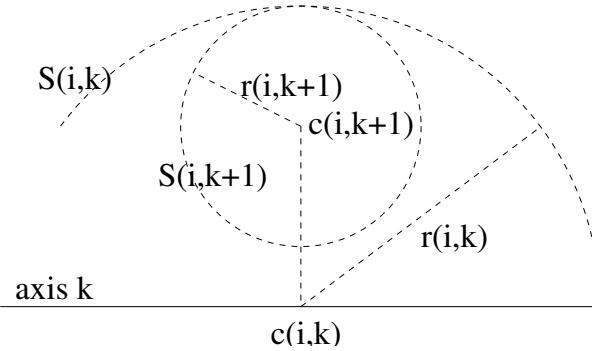


Figure 2.8: Construction of  $S(i, k)$  at Step 4 of algorithm COMPUTE-SPHERE

For every  $k < i$ , we compute a sphere  $S(i, k)$  of radius  $R_k^i = r(i, k)$  centered at a point  $\mathbf{c}(i, k)$  located in the axis of rotation of joint  $k$ , that is guaranteed to enclose link  $i$  for any values of the configuration parameters  $q_k, \dots, q_i$ .

The spheres  $S(i, k)$ ,  $k \leq i$ , are computed recursively by the algorithm of Figure 2.7. At Step 1,  $\text{ENCLOSING-SPHERE}(\mathcal{A}_i)$  computes a tight enclosing sphere of  $\mathcal{A}_i$  using a standard algorithm [112]. At Step 3, computing the sphere that tightly encloses the volume swept by moving  $S(i, k + 1)$  along the entire range of the prismatic joint  $k$  is straightforward. At Step 4, we proceed as follows: we compute the center  $\mathbf{c}(i, k)$  of  $S(i, k)$  as the projection of  $\mathbf{c}(i, k + 1)$  on the axis of rotation of joint  $k$  and the radius  $r(i, k)$  of  $S(i, k)$  as the sum of  $r(i, k + 1)$  and the distance between  $\mathbf{c}(i, k + 1)$  and

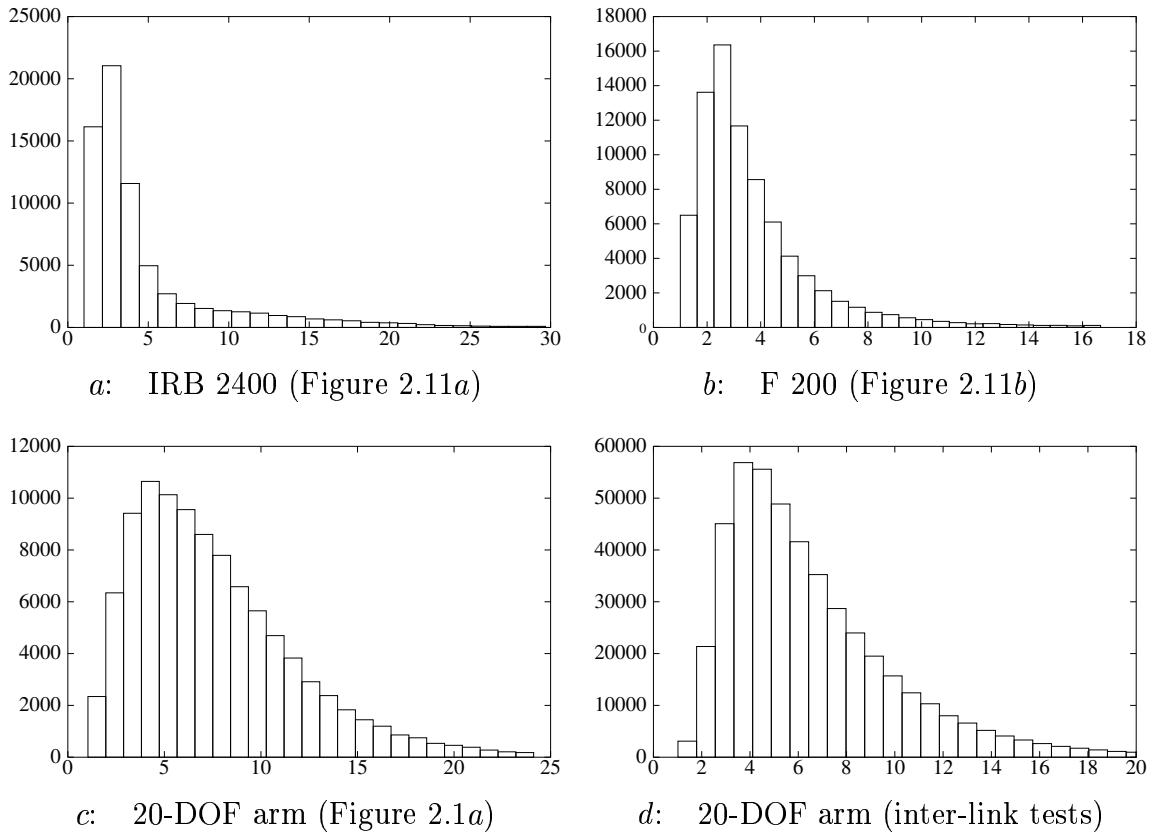


Figure 2.9: Histograms showing the distributions of the quality of the bound  $\lambda_i(\mathbf{q}, \mathbf{q}')$  for three robots. The horizontal axis shows an upper bound on the ratio of  $\lambda_i(\mathbf{q}, \mathbf{q}')$  by the length of the curve. The closer to 1 this ratio, the better the quality. Histogram *d* additionally includes slightly different ratios that account for self-collision testing (see text)

$\mathbf{c}(i, k)$ , as illustrated in Figure 2.8.

#### 2.4.4 Experimental analysis

The bounds defined as above are clearly conservative, but they are also very fast to compute. To evaluate their average quality, we made the following experiments with the robots of Figures 2.1a, 2.11a, and 2.11b.

For each robot, we generated 10,000 segments  $[\mathbf{q}, \mathbf{q}']$  by uniformly sampling two independent endpoints  $\mathbf{q}$  and  $\mathbf{q}'$ . For each segment, we computed the bounds  $\lambda_i(\mathbf{q}, \mathbf{q}')$

for all links, as well as lower bounds  $\ell_i(\mathbf{q}, \mathbf{q}')$  obtained by integrating Equation (2.3) at a low resolution for some randomly chosen point on  $\mathcal{A}_i$ . We computed the ratio  $\lambda_i(\mathbf{q}, \mathbf{q}')/\ell_i(\mathbf{q}, \mathbf{q}')$  for each link over all 10,000 segments. Note that this ratio is not bounded in the worst case, even if  $\ell_i(\mathbf{q}, \mathbf{q}')$  was the exact length of the longest curve traced by a point of  $\mathcal{A}_i$ .

The histograms  $a$ ,  $b$ , and  $c$  in Figure 2.9 present 99% of the results in order to crop outliers. They show that, most of the time, the bounds  $\lambda_i(\mathbf{q}_a, \mathbf{q}_b)$  are within a rather small factor from optimal. For example, for the IRB 2400 and F 200 robots, more than 80% of them are within factor 5 from the corresponding lower bounds  $\ell_i(\mathbf{q}, \mathbf{q}')$ . Even for the hyper-redundant arm (histogram  $c$ ), the ratio is smaller than 15 most of the time.

The histogram  $d$  shows the distribution of the ratios  $\lambda_i(\mathbf{q}, \mathbf{q}')/\ell_i(\mathbf{q}, \mathbf{q}')$  as above plus additional, slightly different, ratios  $\lambda_{ij}(\mathbf{q}, \mathbf{q}')/\ell_{ij}(\mathbf{q}, \mathbf{q}')$  that account for self-collision testing, as follows. For two links  $\mathcal{A}_i$  and  $\mathcal{A}_j$  ( $j < i$ ) of the same robot, we compute  $\lambda_{ij}(\mathbf{q}, \mathbf{q}')$  to bound the lengths of curves traced by all points on  $\mathcal{A}_i$  in  $\mathcal{A}_j$ 's local reference frame, as described in Section 2.4.2. Similarly,  $\ell_{ij}(\mathbf{q}, \mathbf{q}')$  is a non-trivial lower bound on the length of curves traced by all points of  $\mathcal{A}_i$  in  $\mathcal{A}_j$ 's local frame, obtained by integrating Equation (2.3), in  $\mathcal{A}_j$ 's frame, for some randomly chosen point on  $\mathcal{A}_i$ .

## 2.5 Experimental Results

We have experimented with ADAPTIVE-BISECTION on numerous problems ranging from testing randomly generated segments, to testing local paths in multi- and single-query roadmaps, to optimizing jerky paths. In this section, we present a subset of our results.

### 2.5.1 Experimental setup

All the results below were obtained on a 1GHz Pentium III PC with 1GB RAM. Our implementation of ADAPTIVE-BISECTION computes lower bounds on distances

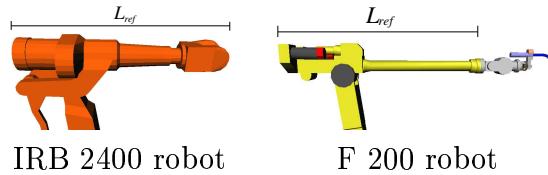
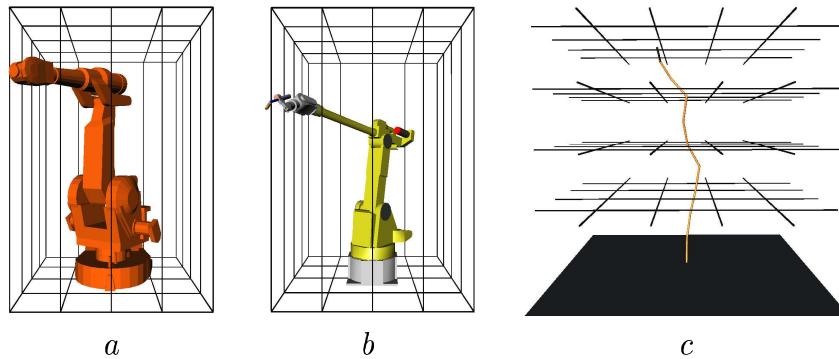


Figure 2.10: Reference links of IRB 2400 and F 200 robots

Figure 2.11: Examples with thin obstacles. *a*: IRB 2400 robot (2,991 triangles), cage (432 triangles). *b*: F 200 robot with arc welding gun (2,502 triangles), cage (432 triangles). *c*: 20-DOF arm (320 triangles), obstacle lattice (384 triangles)

between objects using the algorithm GREEDY-DIST of Section 2.3.2 and upper bounds on lengths of traced curves using Equation (2.2), with the factors  $R_k^i$  computed as described in Section 2.4.3. Our implementation of GREEDY-DIST uses functions from the PQP library<sup>1</sup> to construct RSS hierarchies and compute distances between pairs of RSSs. We tried several algorithms other than GREEDY-DIST to compute exact and approximate distances, but GREEDY-DIST gave the best results overall.

Since ADAPTIVE-BISECTION bisects (sub-)segments in the same way for all pairs of objects, we share results of forward kinematics computations across pairs, by caching the rigid-body transforms defining the positions and orientations of robot links at every configuration considered by the checker. In environments with manipulator arms, such caching avoids many redundant computations.

Pairs of objects that cannot possibly collide (due to constraints on their motion) are identified in a pre-processing phase by bounding the swept volumes of all objects

---

<sup>1</sup><http://www.cs.unc.edu/~geom/SSV/>

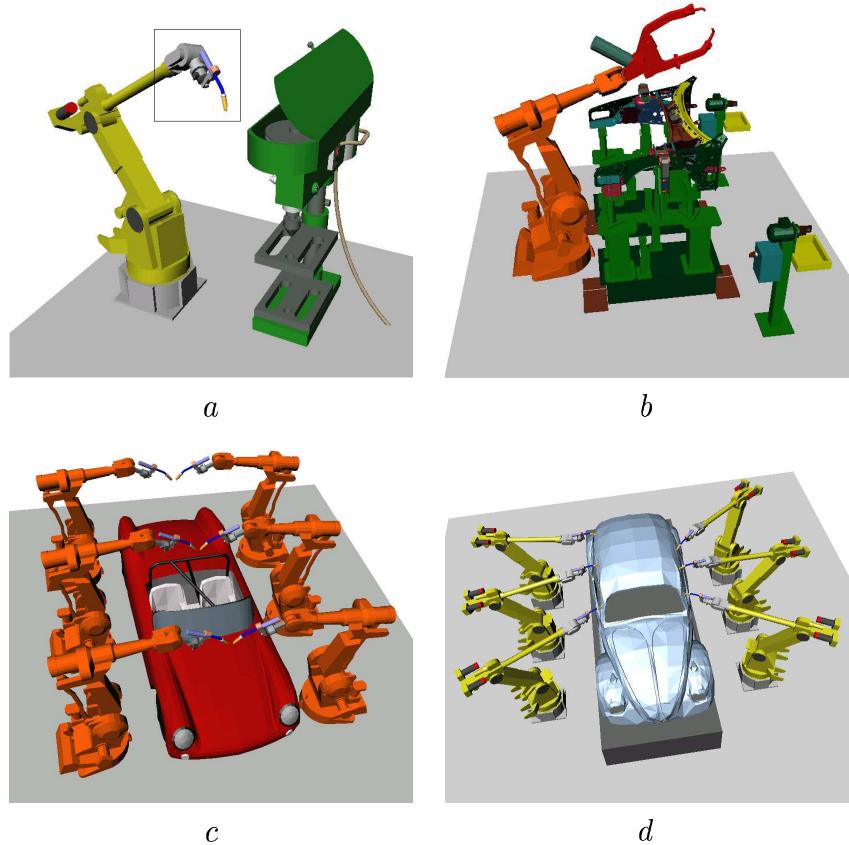


Figure 2.12: *a*: F 200 robot with arc welding gun (2,502 triangles), machine tool (34,171 triangles). *b*: IRB 2400 robot (5,450 triangles) in workshop (74,681 triangles). *c*: Six IRB 2400 robots ( $6 \times 3,594$  triangles), car body (19,668 triangles). *d*: Six F 200 robots ( $6 \times 2,502$  triangles), car body (4,016 triangles)

Fig.	2.1a	2.11a	2.11b	2.11c	2.12a	2.12b	2.12c	2.12d
$L_{ref}$	0.5	1.77	1.65	0.5	1.16	0.64	1.59	1.32

Table 2.2:  $L_{ref}$  (in units) in different examples.

by spheres (as in Section 2.4.3) and then testing these spheres for intersection. If two swept spheres are disjoint, then the underlying object pair is never inserted in the priority queue  $Q$  processed by ADAPTIVE-BISECTION. Likewise, pairs of static objects (e.g., robot bases or obstacles) are not tested for intersection.

We tested ADAPTIVE-BISECTION with various values of  $\delta$ . To relate these values to the size of the robots and obstacles in each environment, we give the lengths  $L_{ref}$  of key robot links in Table 2.2. For the linkages in Figures 2.1a and 2.11c,  $L_{ref}$  is the length of any individual link. For the IRB 2400 and F 200 robots, it is the length of the robot’s forearm as shown in Figure 2.10 (Different values for the same reference link across examples are due to different robot scaling factors.)

We compared ADAPTIVE-BISECTION to a classical fixed-resolution checker, which we call FIXED-RES-BISECTION. This checker follows exactly the bisection algorithm given in [102]: Given a straight path segment with collision-free end-points, it bisects this segment and its sub-segments until either the mid-point of a (sub-)segment is found to collide, or all sub-segments are shorter than some given  $\varepsilon$  (in the Euclidean metric of configuration space). Longer sub-segments are bisected before shorter ones, hence in a breadth-first ordering, in order to find collisions quicker (as longer sub-segments are more likely to collide). For each tested mid-point, FIXED-RES-BISECTION checks all pairs of objects for collision, except those identified as never colliding in the pre-processing phase. Like ADAPTIVE-BISECTION, FIXED-RES-BISECTION can be used to test individual segments or multi-segment paths.

FIXED-RES-BISECTION uses hierarchies of OBBs to test configurations. The construction of the OBB hierarchies and the collision test of two OBB hierarchies are carried out by functions of the PQP library.

Fig.	FIXED-RES-BISECTION			ADAPTIVE-BISECTION					
	$\varepsilon = 0$			$\delta = 0$			$\delta = 0.005$		
	$t$ (ms)	BV	Tri	$t$ (ms)	BV	Tri	$t$ (ms)	BV	Tri
2.11a	0.44	375	3.1	0.33	197	3.0	0.29	169	2.6
2.11b	0.46	380	3.5	0.43	324	3.9	0.38	255	3.2
2.12a	0.67	522	18.7	0.40	501	18.0	0.37	420	21.8
2.12b	1.21	1,035	10.1	0.57	571	10.8	0.40	440	13.4
2.12c	3.11	2,866	13.1	2.44	686	12.1	2.23	610	12.7
2.12d	1.14	999	16.3	1.93	609	19.5	1.71	528	21.6

Table 2.3: Comparison of FIXED-RES-BISECTION with  $\varepsilon = 0$  and ADAPTIVE-BISECTION with  $\delta = 0$  and  $\delta = 0.005$

### 2.5.2 Random colliding segments

In this series of experiments, we compared the performance of the adaptive and fixed-resolution checkers on *colliding* path segments. In each of the six environments of Figures 2.11a-*b* and 2.12a-*d*, we generated 1,000 colliding segments by sampling pairs of collision-free endpoints uniformly at random, and retaining those segments which ADAPTIVE-BISECTION (with  $\delta = 0$ ) found to be colliding. We then tested each segment using: (1) FIXED-RES-BISECTION with  $\varepsilon = 0$ , (2) ADAPTIVE-BISECTION with  $\delta = 0$ , and (3) ADAPTIVE-BISECTION with  $\delta = 0.005$ . When a segment is known to be colliding, FIXED-RES-BISECTION with  $\varepsilon = 0$  is guaranteed to detect a collision in finite time.

Table 2.3 shows data gathered during these tests. For each checker, we indicate the average running time and the average number of BV and triangle pairs tested per segment. ADAPTIVE-BISECTION (with  $\delta = 0$  and 0.005) tests significantly fewer pairs of BVs than FIXED-RES-BISECTION. This gain derives from both the use of Inequality (2.1), which results in bisecting path segments at a coarser average resolution, and the better ordering of the priority queue.

The most significant savings were achieved in the environment of Figure 2.12c where collisions between thin end-effectors are quite difficult to detect. The environment of Figure 2.12d is similar, but collisions between a robot and the car body are easier to detect in this case, which results in smaller differences between the checkers.

Fig.	FIXED-RES-BISECTION			ADAPTIVE-BISECTION		
	$t$ (ms)	BV	Tri	$t$ (ms)	BV	Tri
2.11b	14.6	15,060	19.8	11.3	9,239	200.8
2.12a	5.1	4,032	17.9	3.9	3,834	158.6

Table 2.4: Comparison of FIXED-RES-BISECTION with  $\varepsilon = 0.006$  and ADAPTIVE-BISECTION with  $\delta = 0.005$

The times shown in the table indicate that, except for the environment of Figure 2.12d, ADAPTIVE-BISECTION with  $\delta = 0$  is faster on average than FIXED-RES-BISECTION, despite the fact that computing distances between RSSs is not as fast as testing OBBs for overlap.

The results with  $\delta = 0.005$  show a small speed-up over those obtained with  $\delta = 0$  because with  $\delta > 0$ , some segments are rejected as soon as a pair of objects is found to come closer than  $\delta$ .

### 2.5.3 Random collision-free segments

The above experiment favors the fixed-resolution checker, since we could set  $\varepsilon = 0$ . When segments may not be colliding, this is not possible. To illustrate the respective effects of  $\varepsilon$  and  $\delta$ , we used FIXED-RES-BISECTION and ADAPTIVE-BISECTION to test *collision-free* path segments. For the results presented below, we set  $\varepsilon = 0.006$  and  $\delta = 0.005$ .

Like in the previous experiment, in each considered environment we generated 1,000 random collision-free segments, by picking end-points at random and using the adaptive checker (with  $\delta = 0.005$ ) to test them. Table 2.4 shows the same quantities as Table 2.3 measured in the environments of Figures 2.11b and 2.12a. The value of  $\varepsilon$  was chosen so that the fixed-resolution checker took only slightly longer than the adaptive checker.

Figure 2.13 shows, as a thin line, the curve traced by the tip of the end-effector when the robot of Figure 2.11b moves along a typical straight path segment in configuration space. The dots along this curve correspond to the configurations tested by FIXED-RES-BISECTION with  $\varepsilon = 0.006$ . In reality, each dot is a cube whose side

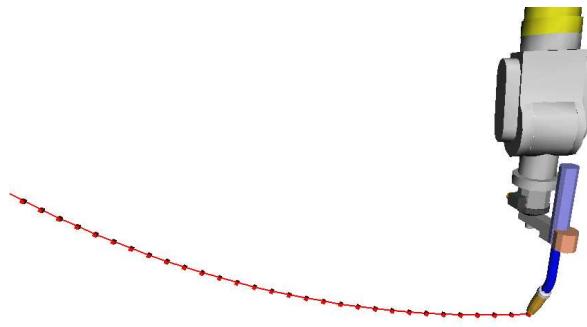


Figure 2.13: Curve traced by the tip of the end-effector of the robot in Figure 2.11b for a typical path segment. The black dots along the curve correspond to the configurations tested by FIXED-RES-BISECTION, with  $\varepsilon = 0.006$ . Each dot is a cube whose side length is equal to 0.01 unit

length is 0.01 units (in the workspace metric), hence twice the value selected for  $\delta$ . To make FIXED-RES-BISECTION sample the curve at resolution  $\delta$ , we would have to set  $\varepsilon$  so small that the running time of FIXED-RES-BISECTION would be considerably longer.

#### 2.5.4 PRM planning

In this section we compare the dynamic and fixed-resolution checkers when they are integrated in the PRM planner described in [102], called SBL. We briefly recall how this planner works.

SBL constructs a roadmap made of two trees of milestones (collision-free configurations sampled at random) rooted at the two query configurations input by the user. At each iteration, the planner performs the following operations:

- (1) It picks a new milestone  $m'$  at random in a neighborhood of an existing milestone  $m$  and installs  $m'$  as a child of  $m$  *without* testing the straight path segment between them for collision.
- (2) It selects two milestones, one from each tree, and connects them by a candidate straight path segment. This connection creates a multi-segment path between the two query configurations, but only the vertices of this path (milestones) have so far

been tested for collision.

(3) It checks this entire path for collision. If no collision is detected, it returns the path. Otherwise it removes the segment found to be colliding from the roadmap, caches the collision-checking work done along other segments (to avoid repeating it if any of these segments is part of a candidate path at a later iteration), and proceeds to the next iteration.

Hence, SBL tests segments between milestones only when this is absolutely needed. This so-called “lazy collision checking” strategy has been shown to significantly reduce the amount of work spent on collision-free segments that are not on the final solution path.

The implementation of SBL described in [102] uses **FIXED-RES-BISECTION** to test multi-segment paths. Hence, it requires setting the value of  $\varepsilon$  for each new environment, and is never guaranteed to return a collision-free path. We have created a new version of SBL, called A-SBL, by replacing the fixed-resolution checker by **ADAPTIVE-BISECTION**. Like SBL, A-SBL also records the collision-checking work done along path segments that have not been fully tested (by maintaining a distinct priority queue for each segment, as described in Section 2.2.4).

We ran SBL and A-SBL on several path-planning problems, each defined by a pair of query configurations in a given environment. For each problem, we performed several series of runs of SBL with decreasing values of  $\varepsilon$ . During each series,  $\varepsilon$  kept the same value, but a different seed of the random number generator was used at each run to construct a different roadmap, hence a different path. We tested each path returned by SBL using **ADAPTIVE-BISECTION** with  $\delta = 0$ . Whenever a collision was detected in a path, we terminated the series, cut the value of  $\varepsilon$  by 1.2, and started another series. Starting with a rather large value of  $\varepsilon$ , we retained the first series of 50 (respectively 100) consecutive runs of SBL that did not return a single colliding path.

Tables 2.5 and 2.6 list results obtained for six planning problems in the environments of Figures 2.11a-*b* and 2.12a-*d*. For each problem, Table 2.5 gives the first value of  $\varepsilon$  that produced  $N$  consecutive correct runs, for  $N = 50$  and 100, and the average running time of SBL over these  $N$  runs. Note that  $\varepsilon$  is much smaller for  $N = 100$

Fig.	$N = 50$		$N = 100$	
	$\varepsilon$	$t$ (s)	$\varepsilon$	$t$ (s)
2.11a	0.00402	4.13	0.00232	5.15
2.11b	0.00078	12.26	0.00078	13.87
2.12a	0.00833	0.17	0.00833	0.17
2.12b	0.00162	7.52	0.00038	16.45
2.12c	0.00694	2.72	0.00279	3.82
2.12d	0.01200	0.67	0.00833	0.92

Table 2.5: Average running times of SBL on five distinct path-planning problems for  $N = 50$  and  $N = 100$  consecutive correct runs (see main text in Section 2.5.4).

$\delta$	.01	.005	.001	.0005	.0001	.00005	.00001	.0
2.11a	5.59	5.04	5.60	7.95	11.17	13.38	17.07	24.85
2.11b	11.78	12.37	11.47	13.17	16.03	16.99	20.59	27.54
2.12a	0.16	0.22	0.42	0.49	1.32	1.87	3.48	21.48
2.12b	6.71	7.33	9.55	12.63	27.53	31.91	67.52	251.56
2.12c	2.00	1.59	1.80	1.76	2.95	2.89	3.83	5.54
2.12d	0.43	0.65	0.71	0.86	2.05	2.42	3.23	10.90

Table 2.6: Average running times of A-SBL for successive values of  $\delta$  (see main text in Section 2.5.4).

in all cases, except two (2.11b and 2.12a), meaning that the values of  $\varepsilon$  obtained for  $N = 50$  are usually not sufficient to guarantee the reliability of SBL.

Table 2.6 lists the average running times (over 100 independent runs) of A-SBL for decreasing values of  $\delta$ . As expected, the average running time of A-SBL increases when  $\delta$  goes to 0. However, the increase is moderate until  $\delta$  reaches very small values, and even then remains relatively small for most problems. Note that in two examples (2.11a and 2.12c), for relatively large  $\delta$ , the running time even decreases slightly when the value of  $\delta$  goes down. In these cases, planning may be slightly harder for larger values of  $\delta$  as more collision-free paths are rejected by ADAPTIVE-BISECTION.

A-SBL has the considerable advantage over SBL that it is guaranteed to never return a colliding path. This is not the case of SBL, even for the values of  $\varepsilon$  allowing 100 consecutive correct runs. In most examples, selecting a value of  $N$  greater than 100 would have led to smaller values of  $\varepsilon$ .

### 2.5.5 Path Smoothing

We conducted the following experiment in the environment of Figure 2.1a. First, starting at a configuration where both linkages stand straight up through the three rings, we performed a random walk of collision-free steps until both linkages lie entirely below the rings, hence were retracted from the rings. ADAPTIVE-BISECTION was used to test each attempted step (straight segment) of the random walk. Next, we smoothed the obtained path by performing  $K$  times the following operations: Pick a point  $\mathbf{q}$  at random in the path; let  $\mathbf{q}_{le}$  ( $\mathbf{q}_{ri}$ ) be the midpoint along the path between the first (last) configuration of the path and  $\mathbf{q}$ . If the straight line segment between  $\mathbf{q}_{le}$  and  $\mathbf{q}_{ri}$  tests collision-free, shortcut the path section between them by this segment, otherwise reset  $\mathbf{q}_{le}$  ( $\mathbf{q}_{ri}$ ) to the midpoint along the path between itself and  $\mathbf{q}$ , and recurse. Again, ADAPTIVE-BISECTION was used to test each attempted shortcut. We did the same experiment in the environment of Figure 2.11c, first with a single linkage, then with 9 identical linkages, each threaded through a different set of grid holes in their initial configuration.

The goal of these experiments was to demonstrate that ADAPTIVE-BISECTION

Fig.	Random walk		Smoothing		
	#segments	t (s)	K	t (s)	$\sigma$
2.1a	40,439	82	1,000	152	0.06
2.11c [1 linkage]	4,208	5.9	500	16	0.11
2.11c [9 linkages]	47,744	823	6,000	5,952	0.25

Table 2.7: Results for random walks and path smoothing

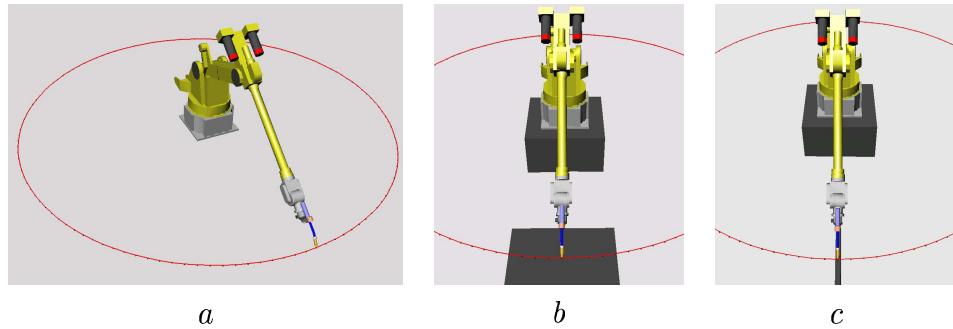


Figure 2.14: Three bad-case examples for ADAPTIVE-BISECTION

enables the implementation of a brute-force planning scheme requiring many collision tests between very thin objects. These experiments would be quasi-impossible using FIXED-RES-BISECTION, as we would then have to chose  $\varepsilon$  extremely small.

Table 2.7 shows data obtained for a typical run of each experiment. For the random walk, it gives the number of segments tested and the computation time of the walk. For path smoothing, it lists the value of  $K$ , the running time, and the ratio  $\sigma$  of the final length of the path by its initial length. Note that each of the  $K$  iterations usually results in many segment tests, due to the recursive attempts to shortcut path sections.

### 2.5.6 Bad-case scenarios

A very unfavorable case for ADAPTIVE-BISECTION is when, along a path segment in configuration space, two or more objects in relative motion stay at very close distance from each other, without coming closer than the threshold  $\delta$ . To study the behavior of ADAPTIVE-BISECTION in such cases, we constructed three examples

		All links checked		Only end-effector checked	
$\zeta$	$\delta$	BV	Tri	BV	Tri
0.1	0.1	11,232	0	8,789	0
0.01	0.01	62,866	0	62,727	0
0.001	0.001	990,766	0	990,595	0
0.0001	0.0001	7.91e+06	0	7.91e+06	0
0.1	0.0001	258	0	127	0
0.01	0.0001	753,939	0	753,800	0
0.001	0.0001	1.12e+06	0	1.12e+06	0
0.0001	0.0001	7.91e+06	0	7.91e+06	0

Table 2.8: Number of BV/triangle pairs tested in Figure 2.14a

		All links checked		Only end-effector checked	
$\zeta$	$\delta$	BV	Tri	BV	Tri
0.1	0.1	1,421	105	1,397	105
0.01	0.01	5,664	101	5,646	101
0.001	0.001	73,018	122	72,998	122
0.0001	0.0001	578,220	3,358	578,200	3,358
0.1	0.0001	33	0	17	0
0.01	0.0001	23,801	0	23,783	0
0.001	0.0001	67,165	0	67,145	0
0.0001	0.0001	578,220	3,358	578,200	3,358

Table 2.9: Number of BV/triangle pairs tested in Figure 2.14b

		All links checked		Only end-effector checked	
$\zeta$	$\delta$	BV	Tri	BV	Tri
0.1	0.1	653	5	634	5
0.01	0.01	3,059	51	3,045	51
0.001	0.001	27,685	87	27,671	87
0.0001	0.0001	202,870	2,153	202,856	2,153
<hr/>					
0.1	0.0001	27	0	15	0
0.01	0.0001	315	0	301	0
0.001	0.0001	15,426	0	15,412	0
0.0001	0.0001	202,870	2,153	202,856	2,153

Table 2.10: Number of BV/triangle pairs tested in Figure 2.14c

shown in Figure 2.14. In all three examples, the robot (2,502 triangles) moves a full  $2\pi$  rotation around its first joint, with all other joints remaining fixed. This motion defines a long straight segment in configuration space. In Figure 2.14a, the robot is placed directly on the floor, a flat surface tessellated by 8,000 triangles. During the motion, the tip of the end-effector (1,080 triangles) traces a circle at a constant clearance above the floor that is slightly bigger than a given  $\zeta > 0$ . The example in Figure 2.14b is similar, but the robot is now placed on top of a box, and during the rotation the end-effector grazes the top face of another box, again with a constant clearance slightly bigger than  $\zeta > 0$ . Hence, the end-effector now comes close to an obstacle only during a fraction of the rotation. Each face of the box is modeled with 8,000 triangles. The example in Figure 2.14c is almost identical: the box is narrower, but its faces still contain 8,000 triangles, each.

Tables 2.8–2.10 show the numbers of pairs of BVs and triangles tested in each of the above three examples, for different combinations of values of  $\delta$  and  $\zeta$ . In each table the numbers are reported for all robot links and for the end-effector only.

The numbers of BV/triangle pairs tested by the checker depend on both  $\zeta$  and  $\delta$ . However, in each example, the tables show that links other than the end-effector barely influence these numbers. This is due to the fact that the checker tests each pair of objects independently of the others, starting with the object pairs that are the

most likely to collide. Links other than the end-effector, which are rather far away from any obstacle, are quickly eliminated (as not colliding), if ever tested.

Another observation is that the checker focuses on the critical portion of the path segment. Indeed, the numbers of BV/triangle tests decrease significantly from the first example to the third, although the three path segments have equal length and the obstacle's top surface has the same number of triangles in each case. This comes from the fact that the use of Inequality (2.1) allows the checker to sample the path segment more coarsely when obstacles are further away from the end-effector.

As we expected, this experiment shows that the performance of ADAPTIVE-BISECTION degrades when two objects in relative motion remain very close and both  $\delta$  and  $\zeta$  approach zero. Nevertheless, even for very small values of  $\delta$  and  $\zeta$ , the checker remains reasonably efficient as long as the small clearance occurs over short path sections. In PRM path planning, where segment endpoints are selected at random, this condition is likely to be verified. But in some applications, such as machining and assembly, small clearances over long segments may occur more frequently. Then, methods based on swept-volume computation or trajectory parameterization might be more appropriate, possibly in combination with adaptive bisection.

## 2.6 Refined Checker

In the above checker, the priority queue  $Q$  is filled with elements  $[\mathbf{q}_a, \mathbf{q}_b]_{ij}$  indicating that a pair of objects —  $\mathcal{A}_i$  and  $\mathcal{A}_j$  — must be tested for collision along a given path segment connecting configurations  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . However, whenever the same two objects are tested, the traversal of their BVHs starts at their respective roots. Instead, Inequality (1) could be used during a test to detect that pairs of BVs need not be tested again later. This yields a refined version of the checker in which the priority queue is filled with elements  $[\mathbf{q}_a, \mathbf{q}_b]_{BB'}$  indicating that a pair of BVs —  $B$  and  $B'$  — from two different hierarchies must be tested for collision along the path segment connecting  $\mathbf{q}_a$  and  $\mathbf{q}_b$ .

Figure 2.15 describes the refined checker, REFINED-ADAPTIVE-BISECTION. At Step 2.2, the new checker calls a function, COLLISION-TEST, to determine the elements

**Algorithm** REFINED-ADAPTIVE-BISECTION( $\mathbf{q}, \mathbf{q}'$ )

1. For every pair of objects  $(\mathcal{A}_i, \mathcal{A}_j)$  that needs to be tested, insert  $[\mathbf{q}, \mathbf{q}']_{B_i B_j}$  into  $Q$ , where  $B_i$  and  $B_j$  denote the roots of the BVHs of  $\mathcal{A}_i$  and  $\mathcal{A}_j$ , respectively
2. While  $Q$  is not empty, do
  - 2.1  $[\mathbf{q}_a, \mathbf{q}_b]_{BB'} \leftarrow \text{remove-first}(Q)$
  - 2.2 COLLISION-TEST( $B, B', \mathbf{q}_a, \mathbf{q}_b$ )

Figure 2.15: Refined adaptive bisection algorithm

**Algorithm** COLLISON-TEST( $B, B', \mathbf{q}_a, \mathbf{q}_b$ )

1.  $T \leftarrow$  intersection of the two trees of BV pairs computed by GREEDY-DIST2( $B, B', \mathbf{q}_a$ ) and GREEDY-DIST2( $B, B', \mathbf{q}_b$ )
2. Perform a depth-first traversal of  $T$ . For every node  $(b, b')$  of  $T$ , if  $\lambda_b(\mathbf{q}_a, \mathbf{q}_b) + \lambda'_b(\mathbf{q}_a, \mathbf{q}_b) < \eta_{bb'}(\mathbf{q}_a) + \eta_{bb'}(\mathbf{q}_b)$ , then backtrack.  
Else if  $(b, b')$  is a leaf of  $T$  then
  - 2.1  $\mathbf{q}_{mid} \leftarrow$  mid-configuration along path segment between  $\mathbf{q}_a$  and  $\mathbf{q}_b$
  - 2.2 If  $\eta_{bb'}(\mathbf{q}_{mid}) = 0$  then return *collision*
  - 2.3 Else insert  $[\mathbf{q}_a, \mathbf{q}_{mid}]_{bb'}$  and  $[\mathbf{q}_{mid}, \mathbf{q}_b]_{bb'}$  into  $Q$

Figure 2.16: Finding elements to insert into the priority queue  $Q$ 

that must be inserted into  $Q$ . This function, shown in Figure 2.16, identifies the pairs of BVs compared by the greedy distance algorithm at both  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . We describe how this is done below.

For any node  $B$  of a BVH, we let  $|B|$  denote the set of triangles associated with the leaves of the sub-hierarchy rooted at  $B$ . Let the function GREEDY-DIST2( $B, B', \mathbf{q}$ ) compute a lower-bound  $\eta_{BB'}(\mathbf{q})$  on the distance between  $|B|$  and  $|B'|$  at  $\mathbf{q}$ . It is the same algorithm as GREEDY-DIST (described in Section 2.3.2), with one difference: it either returns 0 (meaning that a collision occurs at  $\mathbf{q}$  between the triangles in  $|B|$  and those in  $|B'|$ ), or it provides the tree of BV pairs  $(b, b')$  that were compared by the recursive calls. Each node  $(b, b')$  of this tree contains the lower bound  $\eta_{bb'}(\mathbf{q})$

Fig.	ADAPTIVE-BISECTION			REFINED-ADAPTIVE-BISECTION		
	$t$ (ms)	BV	Tri	$t$ (ms)	BV	Tri
2.11a	0.066	17491	45	0.037	1510	10
2.11b	0.060	13182	201	0.040	1342	14
2.12a	0.020	5038	222	0.010	546	15
2.12b	0.032	7299	66	0.023	1144	14
2.12c	0.155	15978	367	0.142	2914	30
2.12d	0.507	73579	4623	0.373	8523	169

Table 2.11: Comparison of ADAPTIVE-BISECTION and REFINED-ADAPTIVE-BISECTION

computed for this pair. In addition, we let  $\lambda_B(\mathbf{q}_a, \mathbf{q}_b)$  stand for an upper bound on the lengths of the curves traced out by the vertices of the triangles in  $|B|$  when the configuration varies from  $\mathbf{q}_a$  to  $\mathbf{q}_b$  along the considered path segment. The function  $\lambda_B$  is computed using the same techniques as presented in Section 2.4.

**COLLISION-TEST**( $B, B', \mathbf{q}_a, \mathbf{q}_b$ ) first computes the intersection of the two trees computed by **GREEDY-DIST2**( $B, B', \mathbf{q}_a$ ) and **GREEDY-DIST2**( $B, B', \mathbf{q}_b$ ). It then performs a depth-first traversal of the tree formed by this intersection. At each traversed node it performs a test similar to that of Inequality (1). If the node passes this test, then the triangles in  $B$  and  $B'$  cannot collide between  $\mathbf{q}_a$  and  $\mathbf{q}_b$ . Otherwise, it yields Steps 2.1 through 2.3, which are the same as Steps 2.2.1 through 2.2.3 in the original checker of Figure 2.

Table 2.11 compares REFINED-ADAPTIVE-BISECTION with ADAPTIVE-BISECTION. The results are averages over 1,000 randomly generated segments in the same environments used in Section 2.5 and under the same conditions. The running times of the refined checker range between 0.5 and 0.9 times those of the original checker. The gains are modest, but still significant.

## 2.7 Conclusion

This chapter describes a new dynamic collision checker to test path segments in configuration space or collections of such segments, including continuous multi-segment paths. This checker is general, but particularly suited for PRM planners applied to manipulator arms and/or multi-robot systems. Its main advantage over the commonly used fixed-resolution approach is to never miss a collision. In addition, it eliminates the need for experimentally determining a suitable value of the configuration space resolution parameter  $\varepsilon$  and its running time compares favorably to that of a fixed-resolution checker.

Reliability and efficiency are obtained by dynamically adjusting the local resolution at which configurations along a path are tested by relating the distances between objects in the workspace to the maximum lengths of the paths traced out by points on these objects. This idea is combined with other techniques, including:

- a greedy distance computation algorithm that is nearly as efficient as a pure collision checker,
- a fast technique for bounding lengths of paths traced out in workspace,
- a heuristics for ordering collision tests to reveal collisions as quickly as possible.

Relatively simple extensions are possible. For example, one could guarantee that no two objects come closer than some minimal distance  $\varrho > 0$  along a tested path segment, by using the following variant of Inequality (2.1):

$$\lambda_i^r(\mathbf{q}_a, \mathbf{q}_b) + \lambda_j^r(\mathbf{q}_a, \mathbf{q}_b) < \eta_{ij}(\mathbf{q}_a) + \eta_{ij}(\mathbf{q}_b) - 2\varrho$$

where  $\lambda_i^r(\mathbf{q}_a, \mathbf{q}_b)$  (similarly  $\lambda_j^r(\mathbf{q}_a, \mathbf{q}_b)$ ) bounds the motions of  $\mathcal{A}_i^r$  which is defined as the Minkowski sum of  $\mathcal{A}_i$  and a sphere of radius  $r = \varrho/2$ .

We believe that our checker could still gain in efficiency by computing tighter bounds on lengths of curves traced out by points of moving objects. For example, instead of computing factors  $R_k^i$  that are valid across the entire configuration space, we could partition the configuration space of each linkage into a coarse grid of cells, and compute smaller factors over each cell.

The new checker has certain limitations, however. In some applications, one may want to determine the first collision configuration when moving from one end of a path segment to the other. Then, our bisection algorithm, which is tuned to determine as quickly as possible whether a path segment is colliding, or not, is not the best approach. However, even in this case, a variant of Inequality (2.1) could be used to decompose the segment into a series of safe segments. Another limitation is that the checker loses efficiency when two moving objects come arbitrarily close and/or stay very close along a long section of a path segment. To bound the running time we have introduced a clearance parameter  $\delta > 0$ . Though  $\delta$  can be set quite small in practice, it may be undesirable for some applications (e.g., mechanical assembly). An alternative discussed in Section 2.2.6 is to locally switch to another dynamic collision-checking method, for instance, a swept-volume or a trajectory parameterization method. Such a method is then made practical by the fact that the potential collision can only occur on restricted sections of the segment and between restricted portions of the objects. Finally, the new checker leads PRM planners to consume more memory. Unlike a fixed-resolution checker, it requires maintaining a priority queue of pairs of objects to be tested for collision.

# Chapter 3

## Dealing with Narrow Passages\*

### 3.1 Introduction

The efficiency of PRM planners often decreases dramatically when the free space contains narrow passages. Design of effective sampling strategies that enable PRM planners to deal with narrow passages has been a “grand challenge” ever since PRMs were introduced. The most notable strategies proposed so far are the *filtering* and the *retraction* strategies. Filtering strategies include Gaussian sampling [16] and the bridge test [50]. They over-sample configuration space, but then use heuristics to retain relatively few promising milestones. Retraction strategies [5, 51, 79, 114] use configurations sampled in collision space as helpers to generate promising milestones. Although these strategies have led to significant improvements, they often remain too slow. They also waste time in spaces with no or simple narrow passages.

In this chapter we present a new retraction-like sampling strategy that enable PRMs to efficiently deal with narrow passages and, unlike previous methods, is efficient even when there are no narrow passages in free space. We call our new strategy *small-step retraction* because it only tries to retract barely colliding configurations into free space. It consists of pre-computing “thinned” geometric models of the robot

---

\*This chapter is based on the journal article: “Finding Narrow Passages with Probabilistic Roadmaps: The Small-Step Retraction Method”, *Autonomous Robots*, 19(3):301-319, December 2005. Yu-Chi Chang, Jean-Claude Latombe, and Fritz B. Prinz are the co-authors of the article.

and/or the obstacles, constructing a roadmap using the thinned models, and finally repairing portions of this roadmap that are not in free space by resampling more milestones around them. Thinning an object is done by reducing the volume it occupies around its medial axis, as described in Section 3.4. This operation guarantees that the free space associated with the thinned models – we call it *fattened* free space  $F^*$  – is a superset of actual free space  $F$ . Narrow passages in  $F$  become wider in  $F^*$ , making it easier to connect a roadmap through them (Section 3.3). But there is also a risk that some passages, called “false” passages, exist in  $F^*$  that were not present in  $F$ . Paths through such passages cannot be repaired. For this reason, we propose two repair strategies: OPTIMIST and PESSIMIST(Section 3.5). OPTIMIST assumes that false passages are not an issue and that, consequently, repairs will always be possible. So, it postpones repairs until a complete path has been found in  $F^*$ ; then, it repairs only the portions of this path that are not in  $F$ . It is often very fast, but returns failure in cases (rare in practice) where the generated path traverses a false passage. Instead, PESSIMIST immediately repairs all parts of the roadmap that are not in  $F$ . It is slower, but more reliable. A simple combination of the two strategies yields an integrated planner that is both fast and reliable.

This small-step retraction method has several advantages over previous retraction methods:

- Most previous methods try to repair all colliding configurations, most of which are deeply buried into obstacle regions. This is computationally expensive. Only repairing those configurations where thinned objects do not overlap is much faster.
- The method in [51] also performs a form of small-step retraction. To test whether a colliding configuration should be repaired, it computes the penetration depth of the robot in obstacles. Such computation is notoriously expensive for non-convex objects [80].
- Our method works well even when there are no narrow passages in free space. In contrast, previous filtering and retraction methods incur a significant overhead in running time.

- While most previous narrow-passage methods were designed for multi-query roadmaps, ours works with single-query planners. In practice, such planners are often much faster per query than multi-query ones, hence more useful.

We have implemented the small-step retraction method as a new planner – called SSRP – which is an extension of the single-query PRM planner SBL. We tested SSRP on many examples, some presented in Section 3.6. On problems with narrow passages, it is consistently faster than SBL, usually by orders of magnitude. On problems without narrow passages, it is as fast as SBL.

## 3.2 Related Work

In the following subsections we review previous work studying the impact of narrow passages on PRM planning and describing methods created to deal with them.

### 3.2.1 Narrow passages

Although the narrow-passage issue was recognized when PRM planning was first introduced, neither the notion of a narrow passage, nor its impact on PRM planning is straightforward.

In [64], the clearance of a path is defined as the minimum distance between a configuration on this path and free space boundary. It is used to establish an upper bound on planning time. However, path clearance incompletely characterizes narrow passages. Indeed, it is independent of the length of a passage and the number of dimensions along which it is narrow. It does not capture either the fact that the passage might be jaggy, which may greatly affect the difficulty of finding a path through it. Furthermore, while finding a path through a single isolated passage may be difficult, many parallel narrower passages may turn out easier to deal with.

A more careful analysis of narrow passages yields the notion of the *expansiveness* of free space  $F$  [56]. Given any  $S \subset F$ , the lookout of  $S$  is the set of all configurations in  $S$  that can be connected by a local path to a configuration in  $F \setminus S$ .  $F$  is expansive if all its subsets have a relatively large lookout. Expansiveness was used to relate the

number of milestones needed to connect two configurations, separated by a narrow passage, with a given probability.

### 3.2.2 Sampling and connection strategies

A sampling strategy selects the probabilistic distribution measure used to sample milestones. The simplest measure is the uniform distribution. More complex strategies vary the probabilistic distribution during roadmap construction. A connection strategy chooses which pairs of milestones to connect.

Many strategies have been proposed for PRM planners. A survey can be found in [104]. Their goal is to process path-planning queries at minimal computational cost. For multi-query planners, this means building a roadmap with two properties [51]:

1. *Coverage*: the roadmap must be distributed over free space, so that it is easy to later connect any query configuration to it.
2. *Connectedness*: there must be a one-to-one correspondence between the path-connected components of the free space and those of the roadmap.

In single-query planners, strategies aim at constructing the smallest possible roadmap connecting a given pair of query configurations.

All successful strategies alleviate the narrow passage problem to some extent. For instance, the sampling strategy described in [66] pre-computes a roadmap in two stages: first a roadmap is computed by sampling configurations at random, with a uniform distribution over configuration space; next, additional configurations are sampled uniformly in neighborhoods of milestones that have no or few connections to other milestones. The rationale is that poorly connected milestones lie in “difficult” regions of free space, such as narrow passages. Another general sampling strategy that has given good results on problems with narrow passages is PRT (Probabilistic Roadmap of Trees), which constructs a roadmap as collections of trees [3]. Connection strategies are also relevant to finding paths through narrow passages. For instance, in [60] the use of search techniques to generate local paths yields multi-query roadmaps with fewer connected components.

However, the above strategies do not attack the narrow-passage issue by specifically trying to sample milestones within narrow passages. It is usually considered that a strategy dedicated to finding narrow passages should exploit configurations sampled in collision space to increase the planner’s chances of eventually placing milestones in these passages. Two such families of sampling strategies are the *filtering* and the *retraction* strategies.

### 3.2.3 Filtering strategies

Filtering strategies consist of over-sampling configuration space and retaining each sampled collision-free configuration as a milestone with a probability estimating its likelihood of being in a narrow passage. As many configurations sampled in free space do not end up being retained as milestones, more time is spent on average per milestone. But the resulting set of milestones is better distributed and smaller, so that fewer connections are eventually attempted between them. Testing that a connection is collision-free (dynamic checking) takes much more time than testing a single configuration (in practice, 10 to 100 more time). Hence, one may save a significant amount of time in total. *Gaussian sampling* [16] and the *bridge test* [50] are two particular filtering strategies.

Gaussian sampling generates each milestone as follows. It samples a configuration  $\mathbf{q}$  uniformly at random from the configuration space and then a second configuration  $\mathbf{q}'$  using a Gaussian distribution centered at  $\mathbf{q}$ . If only one of the two configurations is collision-free, it retains it as a milestone with probability 1. Instead, if both  $\mathbf{q}$  and  $\mathbf{q}'$  are collision-free, it retains either one as a milestone, but with low probability. The resulting distribution of milestones is sparse in wide-open region of free space and denser near free space boundary. Since configurations in narrow passages are necessarily near this boundary, Gaussian sampling increases the likelihood of milestones in narrow passages. However, especially in high-dimensional spaces, most milestones are still “wasted” in regions close to free space boundary, but not in narrow passages.

For this reason, the bridge test extends the idea underlying Gaussian sampling. It samples two configurations  $\mathbf{q}$  and  $\mathbf{q}'$  in the same way as Gaussian sampling. If

both are in collision and the mid-configuration  $(c + c')/2$  is collision-free, then it retains the mid-configuration as a milestone with probability 1, since it very likely lies in a narrow passage. Otherwise, if only one of the two configurations  $\mathbf{q}$  and  $\mathbf{q}'$  is collision-free, it retains it as a milestone with lower probability. If both  $\mathbf{q}$  and  $\mathbf{q}'$  are collision-free, it retains either one as a milestone with very low probability. At first glance, the bridge test looks like an expensive method, but results show that it is a very promising approach when there exist narrow passages.

### 3.2.4 Retraction strategies

Retraction strategies exploit configurations sampled in collision space to guide the search for promising milestones. They consist of moving colliding configurations out of collision.

One technique casts rays along directions selected at random from a colliding configuration  $\mathbf{q}$  and performs discrete walks along these rays until a configuration tests collision-free [5]. Fixed-step and bisection (binary search) walks have been used. But in high-dimensional spaces, rays may easily miss narrow passages, especially if  $\mathbf{q}$  is deeply buried in collision space.

Other techniques use the medial axis (MA) of the workspace [44, 59, 48], which is defined as the subset of points for which at least two points in the obstacle boundary are equally close. In [48] each configuration  $\mathbf{q}$  sampled by the planner is adjusted by minimizing distances between points (called “handles”) pre-selected by the user on the robot and the MA.

A different type of MA technique is introduced in [114] for a rigid free-flying robot. Here, the PRM planner retracts each configuration it samples at random onto the free space’s MA. To avoid the prohibitive cost of pre-computing this MA, an iterative technique translates the robot until its configuration lies in the MA. However, sections of narrow passages cannot be sampled this way because they would require rotating the robot as well. An extension of this technique to arbitrary robots is discussed in [79].

In general, the above retraction techniques are expensive because they retract all

configurations sampled by the planner, including those which lie deep in collision space. Observation #1 reported in Section 3.3.1 suggests that it is sufficient to retract configurations near the boundary of free space, which is what our small-step retraction approach does. This same idea has been previously exploited in [51], but using the penetration depth of the robot into workspace obstacles to decide whether a configuration is worth retracting. The high cost of computing penetration depth of non-convex objects was a serious limitation of this technique, which has never been tested in geometrically complex environments. Instead, in our approach, we pre-compute a thinned version of the robot and obstacle geometry. We retract a configuration  $\mathbf{q}$  only if the thinned robot placed at  $\mathbf{q}$  does not collide with thinned obstacles.

The idea of shrinking robot geometry was introduced in [9], where a colliding straight path in configuration space is progressively transformed into a collision-free path by growing back the robot links to their original geometry. However, this iterative transform may get stuck into a dead-end and the planner has no systematic way of dealing with such failures. In addition, the shrinking technique in [9] only applies to simple robot links modeled as rectangular boxes.

### 3.3 Motivation from Experimental Observations

Several components of our small-step retraction method were suggested by observations made during simple preliminary experiments in two-dimensional configuration spaces. We report on two key observations below. These observations have been later confirmed by our experiments with the SSRP planner (Section 3.6).

#### 3.3.1 Observation #1

Consider the example in Figure 3.1(a). The robot  $\mathbf{R}$  is a small disc of radius  $r$ . Its configuration is defined by the coordinates of its center. The start and goal configurations are  $\mathbf{s}$  and  $\mathbf{g}$ , respectively. The two obstacles create a long and narrow jaggy passage of width  $w$ , which the robot must traverse to reach  $\mathbf{g}$  from  $\mathbf{s}$ . Although

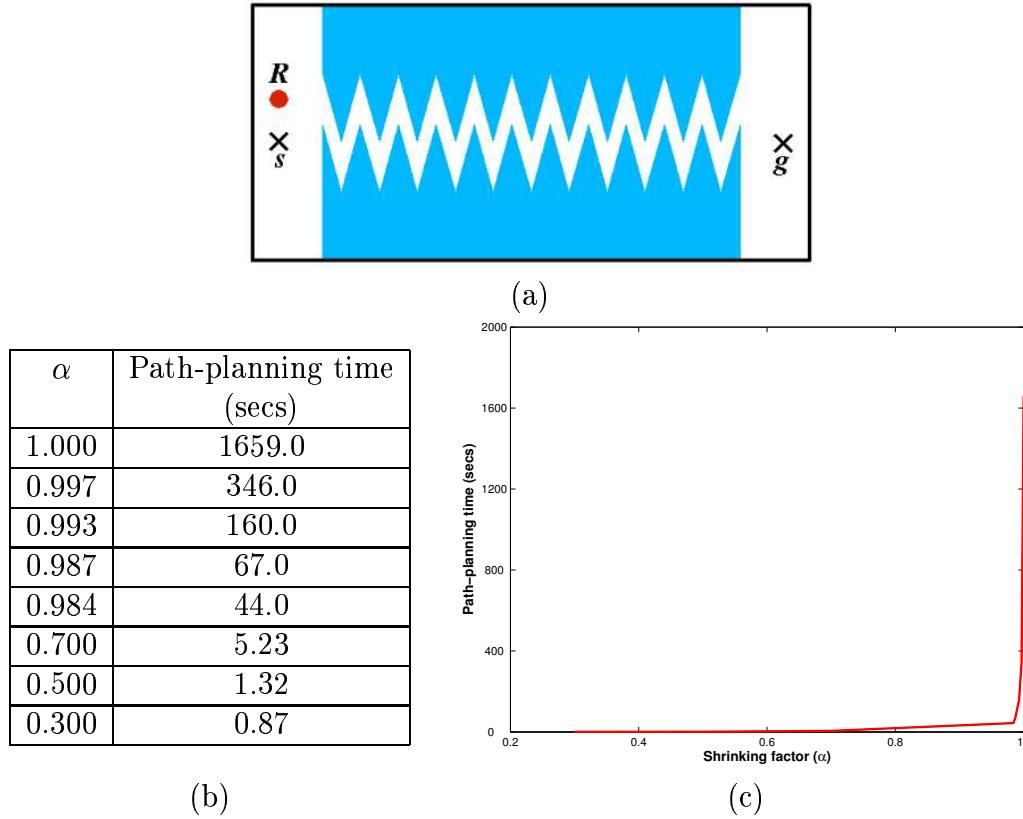


Figure 3.1: Planning a path through a narrow passage of varying width (Observation #1)

the figure shows the robot's workspace, the configuration space is also two-dimensional with similar geometry. However, the passage's width in configuration space is  $w - 2r$ , since the obstacles must be grown by the robot's radius  $r$ .

We used the PRM planner SBL [104] on this example, with decreasing values of  $r$ . Smaller values of  $r$  yield wider passages in configuration space. For each value of  $r$ , SBL was run 100 times with different seeds of the random number generator, each time until a path was found. We measured the average planning time over the 100 runs. For all values of  $r$ , in all runs,  $s$  and  $g$  were fixed as shown in Figure 3.1(a).

The table in Figure 3.1(b) lists the average planning times, for decreasing values of the ratio  $\alpha = 2r/((1 - \epsilon)w)$ , where  $\epsilon$  was set to 0.005. The value  $\alpha = 1$  corresponds to the largest radius  $(1 - \epsilon)w/2$ , hence the narrowest passage in configuration space

(with a width of  $0.005 \times w$ ). The value  $\alpha = 0.3$  corresponds to the widest passage with a width of roughly  $0.7 \times w$ . The plot of these times in Figure 3.1(c) clearly shows that planning time decreases dramatically when  $\alpha$  gets only slightly smaller than 1. The curve flattens out quickly (around  $\alpha = 0.95$ ), but planning time is already a small fraction of what it was for  $\alpha = 1$ . Planning time for  $\alpha = 0.987$  is almost 38 times smaller than for  $\alpha = 1$ .

These results are easy to explain: widening the narrow passage yields a large relative increase of its volume, so that the probability that each configuration sampled at random falls into the widened passages also increases. This simple observation is at the core of the small-step retraction strategy. Fattening free space by only a small amount - which is achieved by thinning the robot and/or the obstacles - greatly simplifies the task of a PRM planner. Of course, a path in fattened free space may not fully lie in actual free space. The purpose of small-step retraction is to “repair” the portions that are not in free space.

### 3.3.2 Observation #2

Consider now the environment of Figure 2(a) in which there are two narrow passages: one at the top of the workspace has width  $w_1$  and one at the bottom has width  $w_2$ . The robot  $\mathbf{R}$  is a disc of radius  $r$ .

In this example, we ran two series of tests. In the first, we set  $w_1 = w_2$  and we considered three pairs  $(\mathbf{s}_i, \mathbf{g}_i)$ ,  $i = 1$  to 3, of query configurations, as shown in the figure. For each pair, the configurations lie on both sides of the passages, with  $(\mathbf{s}_1, \mathbf{g}_1)$  closest to the top passage and  $(\mathbf{s}_3, \mathbf{g}_3)$  at mid distance between the two passages. For each pair  $(\mathbf{s}_i, \mathbf{g}_i)$ , we ran SBL 100 times and we counted the numbers of paths through each passage. The results in Figure 3.2(b) indicate that with high frequency SBL generates paths through the closest narrow passage. For instance, for  $(\mathbf{s}_1, \mathbf{g}_1)$ , 88% of the paths traverse the top passage. As one would expect, for  $(\mathbf{s}_3, \mathbf{g}_3)$ , the paths are equally divided between the two passages.

In the second series of tests, we used only the pair  $(\mathbf{s}_3, \mathbf{g}_3)$  and we set  $w_2 = \beta \times w_1$ , with  $\beta > 1$ . We ran SBL for increasing values of  $\beta$  (100 times for each value) and

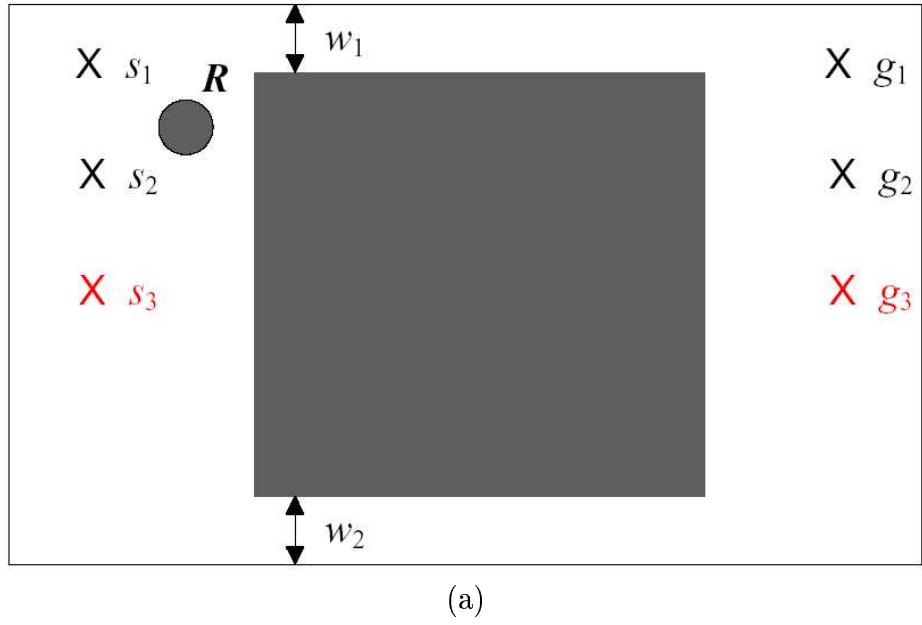
we counted the number of paths through each passage. The results are given in Figure 3.2(c). With high frequency SBL generates paths through the wider passage. The fraction of paths through this passage increases quickly when  $\beta$  increases slightly and reaches 100

This experiment shows that, in the presence of multiple narrow passages, SBL tends to generate paths through the “easiest” one. We used this observation to design the overall strategy of SSRP. As mentioned above, fattened free space may contain false narrow passages (that do not exist in actual free space), but often these false passages are “more difficult” than the widened true passages. Hence, it is reasonable to assume that in most cases a path in fattened free space will be repairable into a path in actual free space. OPTIMIST of SSRP makes this assumption, but the pessimist strategy does not. Since the former is usually orders of magnitude faster than the latter, SSRP tries it first, and uses the pessimist strategy only as back-up.

## 3.4 Object Thinning

The purpose of object thinning is to construct a version of the robot and obstacle geometry that the PRM planner’s collision checker can use to test whether a configuration is barely colliding - in which case it is worth trying to retract it into free space - or not. The thinning method presented below applies to both robots and obstacles, but our experiments with SSRP show that thinning robots (and not obstacles) is often sufficient to achieve major planning speed-up. This is useful, since obstacles usually have more complex geometry than robots and change more often between tasks. So, their thinning is more computationally demanding.

The only constraint on thinning is that the space occupied by a thinned object be contained in the space occupied by the original object. So, in particular, if  $\mathbf{R}(\mathbf{q})$  and  $\mathbf{R}^*(\mathbf{q})$  respectively stand for the space occupied by the original robot and the thinned robot at configuration  $\mathbf{q}$ , then we must have  $\mathbf{R}^*(\mathbf{q}) \subset \mathbf{R}(\mathbf{q})$ . Most scaling methods do not guarantee such result for non-convex objects. For a given object (robot link, obstacle), our method thins its geometry around the object’s medial axis MA. To thin a robot, we thin each of its links, while leaving the robot’s kinematics (relative



(a)

(start, goal)	% Top	% Bottom
$(s_1, g_1)$	88	12
$(s_2, g_2)$	80	20
$(s_3, g_3)$	52	48

(a)

$\beta$	% Top	% Bottom
1.000	52	48
1.003	9	91
1.006	5	95
1.012	2	98
1.500	0	100

(b)

Figure 3.2: Choosing between two narrow passages (Observation #2)

positions of joints) unchanged.

### 3.4.1 Medial-axis computation

The MA of a 3D object  $X$  is the 2D locus of all centers of locally maximal balls contained in  $X$  [12]. The points on the MA together with the radii of the associated locally maximal balls define the Medial-Axis Transform (MAT) of  $X$ . The locally maximal balls are called the MA balls. The entire geometry of  $X$  can be reconstructed from the MAT. A number of algorithms have been proposed to compute the exact

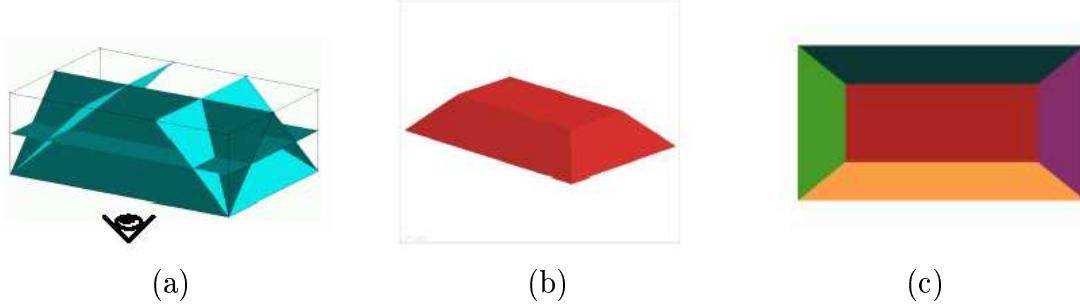


Figure 3.3: Construction of a sub-MA for a box-shaped object [25]: (a) bisecting planes associated with bottom face; (b) visible portions of planes along normal direction rendered with different colors; (c) sub-MA.

or approximate MAT of an object (e.g., [6, 32, 46, 59, 47, 38]). Here, we use a new algorithm described in [25]. Since the thinning method presented in Section 3.4.2 depends partially on this algorithm, we describe it briefly below.

We assume that each object is modeled as a polyhedron. The MA of an object  $X$  is constructed by stitching together several *sub-MA*'s, each contributed by a *governing* boundary element (face, edge, or vertex) of  $X$ . Each sub-MA is computed independently from the others using graphics hardware. Since our thinning method only uses the subset of the MA contributed by faces, we describe the construction and assembly of the sub-MA's governed by faces of  $X$ .

By definition, each point  $p$  on a sub-MA corresponds to the closest point  $q$  on the governing element  $e$ , where the MA ball centered at  $p$  touches  $e$ . The open line segment  $pq$  does not intersect the MA or the object boundary. In addition, every point on the MA belongs to at least two sub-MA's and is therefore equidistant from their governing elements.

The construction of the sub-MA governed by face  $e$  of  $X$  is illustrated in Figure 3. We first consider every other face  $e'$  of  $X$  that fully or partially lies on the inward side of the supporting plane of  $e$ . Each  $e'$  yields the bisecting plane of all points equidistant from  $e$  and  $e'$ . See Figure 3(a), where  $e$  is the bottom face. In this example, each of the other 5 faces of the block contributes a bisecting plane. The bisecting planes are

then viewed along the normal direction of  $e$ . The visible portions (i.e., the portions not occluded by other bisecting planes) form the sub-MAT governed by  $e$ . They are obtained by rendering the bisecting planes with distinct colors in a graphic buffer, as illustrated in Figure 3.3(b). The content of the buffer is then converted into the sub-MAT, as depicted in Figure 3.3(c). A different technique using graphic hardware was previously presented in [46].

Once the sub-MA's governed by all faces of  $X$  have been computed, they are stitched together to form the MA. Consider two sub-MA's having  $e$  and  $e'$  for their respective governing elements, such that each sub-MA contains a face contributed by the bisector plane of  $e$  and  $e'$ . The two sub-MA's can be stitched together by aligning them relative to this plane. The entire MA is obtained by repeating this operation for all sub-MA's.

Note that to construct a complete MA, one would also have to consider concave edges and vertices as governing elements [25]. Such elements may contribute curved portions of the MA, but as indicated before, these portions of the MA are not used by our thinning method.

### 3.4.2 Thinning method

To thin object  $X$ , we reduce the radii of the MA balls and we construct the geometry defined by the reduced balls. Different ways of reducing radii produce distinct outcomes. For example, we may multiply the radii of the MA balls by a constant factor  $\beta < 1$ , or instead subtract a constant offset  $\delta$  from all radii and remove all MA balls whose radii are less than  $\delta$  (see Figure 3.4). In our implementation, we chose the latter technique mainly because the outcome is much easier to compute. We set  $\delta$  to  $\alpha \times r_{max}$ , where  $r_{max}$  is the radius of the largest MA ball and  $\alpha$  is a constant less than 1 called the *thinning factor*; in our experiments, we used  $\alpha \approx 0.2$ .

For each sub-MA governed by face  $e$ , let  $P$  denote the plane obtained by translating the supporting plane of  $e$  inward by the offset distance  $\delta$ . We project the portions of the sub-MA faces that lie on the inward side of  $P$  into  $P$ . The boundary of the thinned geometry of  $X$  consists of all the projected sub-MAT faces for all governing

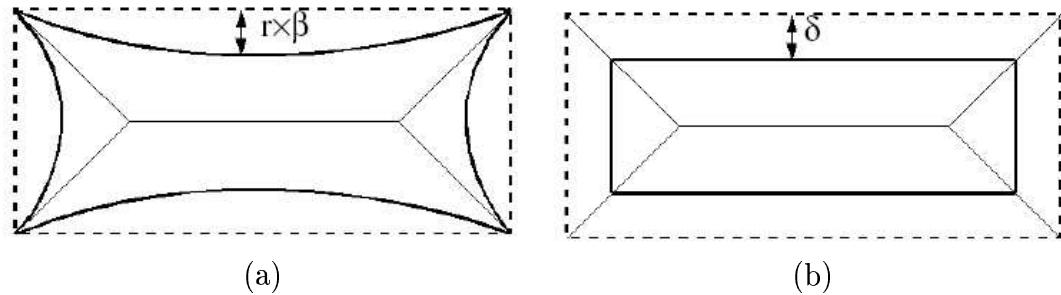


Figure 3.4: Different thinning methods: (a) the radii of MA balls are multiplied by a constant factor  $\beta$  smaller than 1; (b) they are reduced by a constant amount  $\delta$ .

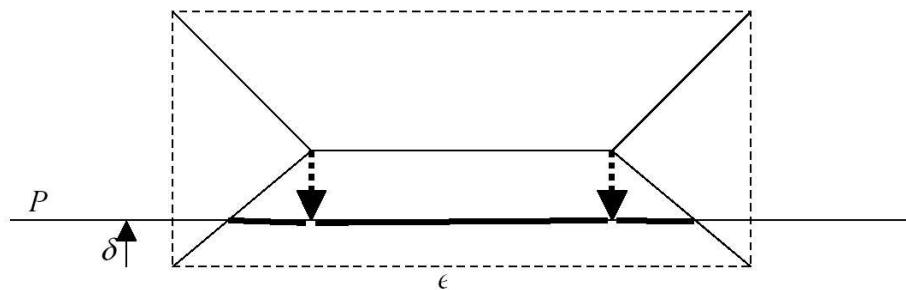


Figure 3.5: Thinning of sub-MA

faces of  $X$ . This technique is illustrated in 2D in Figure 3.5, where  $X$  is a rectangle (in dashed line) and  $e$  is its bottom edge; the sub-MAT governed by  $e$  consists of three edges, which are partially or fully projected into the line  $P$ . The offset distance  $\delta$  can be set to different values for different objects.

This “projection” technique often leads to transforming a single face  $e$  of  $X$  into several co-planar faces of the thinned objects. So, most objects thinned by this technique have a greater number of polygonal faces than the original objects. However, this has no discernable impact on the running time of our planner’s collision checker, which uses a bounding-volume hierarchy approach. An alternative thinning technique would be to compute the intersection of plane  $P$  with the volume bounded by  $e$  and the sub-MAT it governs. We chose the “projection” technique for its robustness, as it is insensitive to floating-point approximations.

The incompleteness of the MA may cause the boundary of the thinned object

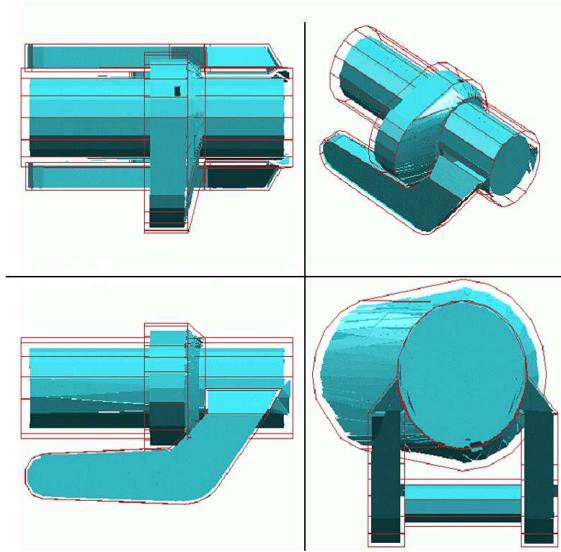


Figure 3.6: Thinned robot component

to contain gaps. There are simple ways to fill these gaps, and we have actually implemented one of them. But, in practice, gaps are small enough to be ignored. At worst, they cause the collision checker to incorrectly classify a few configurations sampled by the PRM planner as lying in fattened free space. The impact on the overall performance of the planner is negligible.

Figure 3.6 shows the result of thinning a component of an ABB IRB 2400 robot. The outer wire frame represents the original component and the inner shaded model depicts its thinned geometry. Similarly, Figure 3.7(a) displays the entire robot (wire frame) and its thinned geometry (shaded model). We decomposed this 6-degree-of-freedom robot into 10 components. The table in Figure 3.7(b) lists the numbers of polygons modeling each component (original and thinned geometry) and the running times (in seconds) to compute its MA and thin its geometry. Note that the computation the sub-MA's could easily be parallelized.

### 3.4.3 Object thinning as a pre-computation step

In this work, we consider object thinning as a pre-computation step for our planner. This means that we do not include its running time into the planner's total running



(a)

components	#polygons (original model)	#polygons (thinned model)	MA computation time (sec.)	thinning time (sec)
1	321	1531	313	0.5
2	1455	10293	1916	2.5
3	205	517	206	0.5
4	137	847	129	0.5
5	135	1086	128	0.5
6	447	2239	461	0.9
7	698	2516	797	1.0
8	119	980	118	0.5
9	376	2713	452	0.9
10	160	863	151	0.5

(b)

Figure 3.7: Thinning the ABB IRB 2400 robot

time. Since the algorithms for computing an object's MA often take more time than the planner itself, this point deserves some discussion.

The main justification for seeing thinning as a pre-computation is that it needs to be done only once for each object, independent of this object's position and orientation. So, it is not repeated for each new planning problem. Moreover, object models are often available long before they are used for motion planning, leaving plenty of time for the pre-computation. Finally, the generation of an object's geometric model, e.g., by using a CAD modeler or by sensing real objects, often takes more time than computing its MA.

For similar reasons, several other computations are classically regarded as pre-computations in motion planning. For instance, like ours, most planners assume that geometric models describe object surfaces by polygonal meshes. When models are not input in this form, extracting polygonal meshes from the given representation is treated as a pre-computation step. Similarly, if a planner uses a BVH collision checker (most PRM planners do), computing bounding-volume hierarchies is considered a pre-computation step, since it is done only once for each object.

Nevertheless, increasing pre-computation has drawbacks. In some robotics applications, object models are not available in advance and the sequence sensing-modeling-planning-execution must be executed as quickly as possible. Any additional computation reduces the robot's reactivity. When time is less critical, one must still store pre-computed results and update them if geometric models are later modified. Fortunately, in most examples, we have observed that, for our planner, it is sufficient to thin the robot and leave the obstacles unchanged.

### 3.5 Small-Step Retraction Planner

Our small-step retraction planner SSRP is built upon the pre-existing SBL planner [104], but we could have used almost any other single-query PRM planners as well. It combines two algorithms (each calling SBL) - OPTMIST and PESSIMIST. OPTMIST, which is very fast, but not fully reliable, is called first. If it fails to find a path, then SSRP calls PESSIMIST, a slower but more reliable algorithm.

```
Algorithm OPTIMIST( $s, g, M, M^*$ )
1.  $\tau^* \leftarrow \text{SBL}(s, g, M^*)$ 
2. Return REPAIR( $\tau^*, M$ )
```

Figure 3.8: The algorithm OPTIMIST

Throughout this section,  $F$  denotes robot's free space and  $F^*$  fattened free space.  $F$  (resp.  $F^*$ ) is the set of all configurations where the robot (resp., the thinned robot) does not collide with workspace obstacles (resp., thinned obstacles). To test if a sampled configuration or a local path is in  $F$  (resp.,  $F^*$ ), the planner uses a BVH (Bounding Volume Hierarchy) collision checker [42, 80, 96] that determines whether the boundaries of the robot and the obstacles (resp., the thinned robot and the thinned obstacles) intersect. By construction of the thinned objects, we have  $F \subset F^*$ . We call the set  $\partial^*F = F^* \setminus F$  the *fattened boundary* of  $F$ .

### 3.5.1 OPTMIST

Finding a path in  $F^*$  is easier for a PRM planner than finding one in  $F$  because narrow passages have been widened. In addition, as true passages in  $F^*$  are the results of widening passages existing in  $F$ , they are usually wider than false passages (if any), which did not exist in  $F$ . So, as suggested by Observation #2 (Section 3.3.2), a planner like SBL will more frequently generate paths through true passages than false ones.

OPTMIST constructs a roadmap in  $F^*$  using SBL. It assumes that paths generated by SBL traverse true passages, hence can be easily repaired. So, it postpones repairs as much as possible. Only after having found a complete path in  $F^*$  between the query configurations  $s$  and  $g$  does it try to repair this path by retracting the colliding portions into  $F$ . If it does not quickly repair the path, it simply returns failure. The algorithm is shown in Figure 3.8, with  $M$  referring to the geometric models of the robot and the obstacles and  $M^*$  to the thinned models.

At Step 1, SBL searches for a path  $\tau^*$  in  $F^*$ . Step 2 calls REPAIR to move the portions of  $\tau^*$  that are in  $\partial^*F$  into  $F$ . REPAIR returns either a path entirely in  $F$ ,

```

Algorithm REPAIR-CONF( $c, M$ )
1.  $\rho \leftarrow \rho_{min}$ 
2. For  $k = 1, \dots, K$  do
   2.1. Sample a configuration  $\mathbf{q}'$  uniformly
        at random from  $B(c, \rho)$ 
   2.2. If  $\mathbf{q}' \in F$  then return  $\mathbf{q}'$  else  $\rho \leftarrow \rho \times \eta$ 
3. Return failure

```

Figure 3.9: The algorithm for repairing a configuration that lies in  $\partial^*F$

or failure. It first retracts the configurations on  $\tau^*$  out of collision, then its edges (straight segments). The algorithm REPAIR-CONF, shown in Figure 3.9, is used to repair every configuration  $\mathbf{q}$  on  $\tau^*$  that lies in  $\partial^*F$  by sampling configurations inside balls  $B(c, \rho)$  of increasing radii  $\rho$  centered at  $\mathbf{q}$ , starting with some small given radius  $\rho_{min}$  (the factor  $\eta > 1$  used to increase  $\rho$  is an input constant):

If REPAIR-CONF fails after  $K$  iterations, the configuration  $\mathbf{q}$  and, so, the path  $\tau^*$  are considered non-repairable. Otherwise, the configuration  $\mathbf{q}$  on  $\tau^*$  is replaced by the returned configuration  $\mathbf{q}'$ . Once all configurations on  $\tau^*$  have been repaired, every path edge  $e$  that intersects  $\partial^*F$  is repaired in turn. This is done recursively by calling REPAIR-CONF( $c, M$ ) with  $\mathbf{q}$  set to the mid-point of  $e$ . Again, any failure causes OPTIMIST to fail. Because repairable paths in  $F^*$  usually lie close to the boundary of  $F$ , the parameter  $K$  is set relatively small, which avoids wasting time on non-repairable paths. In our implementation, we use  $K = 100$ .

### 3.5.2 PESSIMIST

However, SBL( $s, g, M^*$ ) sometimes constructs paths through false passages. To illustrate, Figure 3.10 shows a hand-made pathological example in a 2D configuration space. The white region in this figure is free space  $F$ , the light and dark coloured region the collision space, and the light coloured region the fattened boundary  $\partial^*F$ . The query configurations  $s$  and  $g$  are close to each other, but separated by collision space, so that a path in  $F$  must make a long detour through the passage at the top

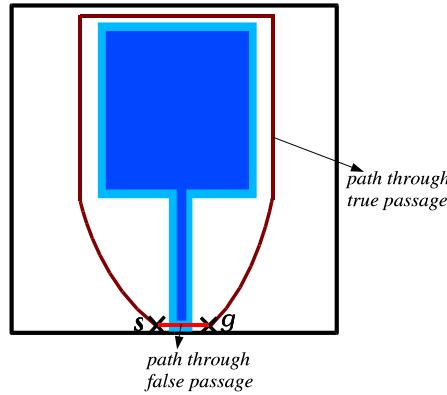


Figure 3.10: A “pathological” example where OPTIMIST is likely to fail

of the configuration space. In this example,  $F^*$  contains a false passage (bottom) through which paths between  $s$  and  $g$  are much shorter. Such paths are more likely to be generated by  $\text{SBL}(s, g, M^*)$  than paths through the wider, but further away top passage.

Unlike OPTIMIST, PESSIMIST immediately repairs every configuration sampled in  $\partial^*F$ , instead of waiting until a path in  $F^*$  has been found. This is done by modifying SBL slightly. Within PESSIMIST, each time SBL samples a new configuration  $q$ , it proceeds as follows:

1. If  $q \in F$ , then add  $q$  to the set of milestones
2. Else if  $q \in^* F$  and  $\text{REPAIR-CONF}(c, M)$  returns a configuration  $q'$   
then add  $q'$  to the set of milestones
3. Else discard  $q$

SBL uses a lazy collision-checking strategy that postpones testing edges between milestones until it has found a path of milestones between the two query configurations (See Appendix A). Only then, it checks that the edges contained in the path are collision-free. If an edge is found to collide, it is removed from the roadmap. SBL does not give up, but continues sampling new milestones until it finds another path. We chose to keep this feature of SBL un-changed in PESSIMIST. So, PESSIMIST does not repair colliding edges, nor paths through false passages.

```

Algorithm SSRP( $s, g, M, M^*$ )
  1. Repeat  $N$  times
    If OPTIMIST( $s, g, M, M^*$ ) returns a path,
      then return this path
  2. Return PESSIMIST( $s, g, M, M^*$ )

```

Figure 3.11: The small-step retraction planner (SSRP)

For every configuration  $c$  sampled outside  $F$ , PESSIMIST calls the collision checker twice and repairs  $q$  if it lies in  $\partial^*F$ , even though many repaired configurations will not end up being on the final path. Hence, on average, it spends more time per milestone than either OPTIMIST or plain SBL. But, by repairing milestones immediately, it avoids the fatal mistakes of OPTIMIST. Because its sampling strategy yields better distributions of milestones, it usually needs smaller roadmaps to generate paths than the plain SBL planner.

### 3.5.3 Overall Planner

As the experiments reported in Section 3.6 will show, when OPTIMIST succeeds, it is much faster than PESSIMIST (often by very large factors). But PESSIMIST is more reliable and in general still significantly faster than SBL. These results led us to design our small-step retraction planner, SSRP, by combining OPTIMIST and PESSIMIST as shown in Figure 3.11.

Our experiments show that if OPTIMIST fails a few times in a row, then it continues failing consistently. So, we set  $N$  small (more precisely, equal to 5 in our implementation). Since OPTIMIST is much faster than PESSIMIST, the impact of successive failures of OPTIMIST on the total running time of SSRP is small.

## 3.6 Experimental Results

OPTIMIST and PESSIMIST are written in C++ using the PRM planner SBL from the Motion Planning Kit (MPK) available at:

*http://robotics.stanford.edu/~mitul/mpk.*

All experiments reported below were performed on a 1GHz Pentium III PC with 1GB RAM.

### 3.6.1 Test environments

Figure 3.12 shows several environments with difficult narrow passages used to evaluate our algorithms and overall strategy:

- In (a) and (b) an IRB 2400 robot must move among thin obstacles. In (b), the robot is in a bar cage and must find a way for its endpoint in and out of the cage, between bars.
- In (c) the robot must move from an initial configuration where the end-effector (a welding gun) is outside the car to a goal configuration where it is deep inside the car. The only way in is through the door window.
- In (d), the robot's end-effector forms a close loop that must be taken off one of the tall vertical poles and placed around the other. The robot must first move up the end-effector along the first pole and then down along the second pole. In this case, free space contains two narrow passages separated by a large open area where the end-effector does not encircle any of the two poles.
- (e) is inspired by typical spot welding tasks. The welding gun must be inserted on both sides of a flat object.
- (f) is a toy example designed to create a short false passage in fattened free space.
- Finally, the so-called alpha puzzle in (g) is a classical test for PRM planners [4]. One of the two identical objects is arbitrarily chosen to be a 6-degree-of-freedom free-flying robot and the other to be a fixed obstacle. The geometry of the objects and the small clearance between them create a very narrow passage in free space that can only be traversed by a tight coordination of rotation and

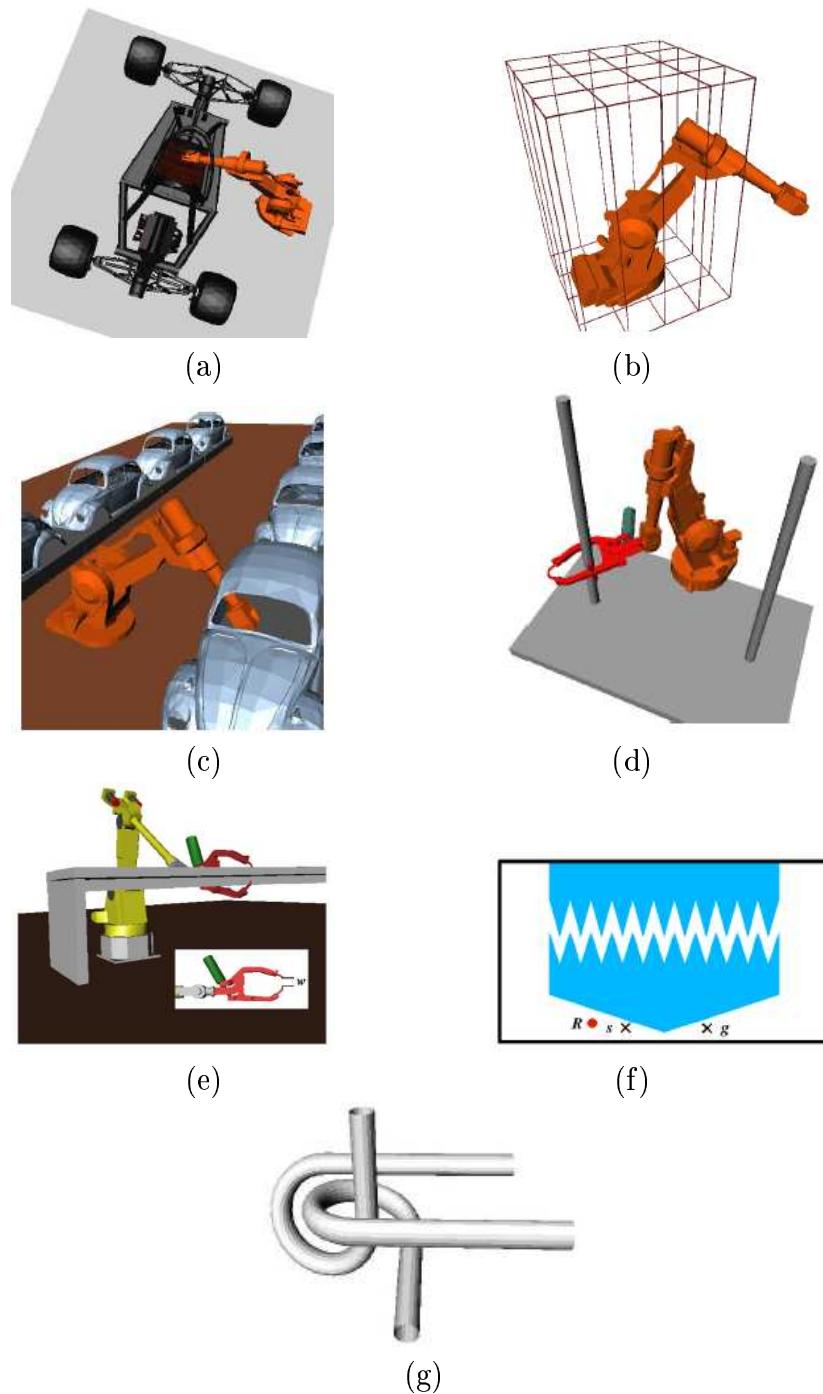


Figure 3.12: Test environments with narrow passages

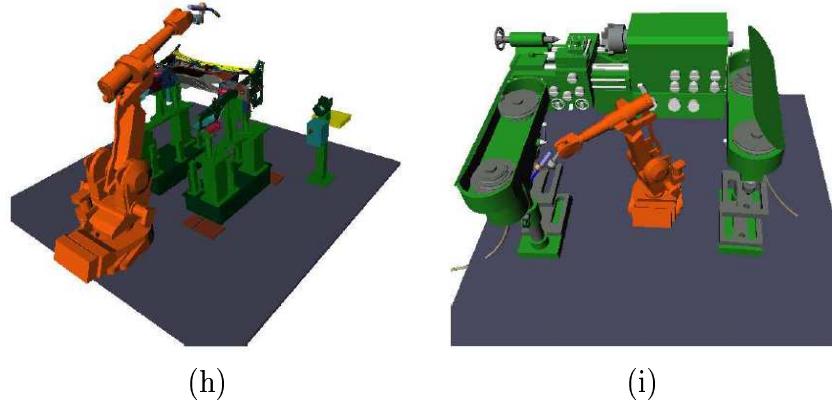


Figure 3.13: Test environments without narrow passages

translation of the robot. The alpha puzzle example exists in several versions named 1.0 to 1.5. Alpha puzzle 1.0 has the narrowest passage and hence is the hardest to solve. In our tests, we used the versions 1.0 and 1.1.

In examples (a), (b), (c), (f) and (g), we only thinned the robot. Instead, in examples (d) and (e), we only thinned the obstacles, because they have much simpler geometry. (Experiments not reported here show that in all these examples thinning both the robot and the obstacles results in no significant additional reduction of planning time.) All objects were thinned with the MA-based method described in Section 3.3 using the thinning factor  $\alpha \approx 0.2$ . Thinning is considered a pre-computation step, whose running time is not included in the planning times below (see Section 3.4.3).

Figure 3.13 shows two additional test environments (h) and (i) for which there are no narrow passages in free space. They were also used to evaluate our planner, since many practical planning problems do not involve difficult narrow passages.

The table of Figure 3.14 gives the number of polygons modeling the objects in all test environments, except (f).

### 3.6.2 Results

The table in Figure 3.15 lists the running times of OPTIMIST, PESSIMIST, SSRP, and plain SBL on planning problems in the environments of Figure 3.12. Each time is

	(a)	(b)	(c)	(d)	(e)	(g)	(h)	(i)
robot	4053	4053	4053	7596	4505	1008	4853	4853
thinned robot	23585	23585	23585	7596	4505	8976	24385	24385
obstacles	43530	432	4028	140	48	1008	74681	83934
thinned obstacles	43530	432	4028	286	80	1008	74681	83934

Figure 3.14: Number of polygons in models

	% success OPTIMIST	$T_{optimist}$	$T_{pessimist}$	$T_{SSRP}$	$T_{SBL}$
(a)	100	9.4	1284	9.4	12295
(b)	100	32	1040	32	5955
(c)	100	2.1	15	2.1	41
(d)	100	492	634	492	863
(e)	100	65	351	65	631
(f)	0	12	325	386	572
(g)-1.1	0	111	2810	3365	>100000
(g)-1.0	100	13588	>100000	14069	>100000

Figure 3.15: Performance of OPTIMIST, PESSIMIST, SSRP, and plain SBL in the environments of Figure 3.12

expressed in seconds and is an average over 100 runs on the same pair of query configurations, except for SBL in example (g) and PESSIMIST in example (g)-1.0, which found no path in several runs, some taking more than 30 hours of computation (which we indicate with “>100000”). The first column of the table identifies the environment. The second column gives the success rate of OPTIMIST. On problems (a)-(e) OPTIMIST consistently succeeded to find a path, while on the last two it consistently failed. The columns  $T_{optimist}$  and  $T_{pessimist}$  give the average running times of OPTIMIST and PESSIMIST, respectively (for better comparison, we ran PESSIMIST even when OPTIMIST succeeded). Except for (g)-1.0, PESSIMIST never failed. The column TSSRP indicates the total planning time of SSRP. It is equal to  $T_{optimist}$  in examples (a)-(e) and (g)-1.0 and to  $5 \times T_{optimist} + T_{pessimist}$  in examples (f) and (g)-1.1, since OPTIMIST is run 5 times before PESSIMIST is invoked. Finally, the last column ( $T_{SBL}$ ) gives to the average time taken by the original SBL planner, which succeeded consistently on

	OPTIMIST	PESSIMIST	SBL
(a)	3202	95559	250515
(b)	20581	76356	132451
(c)	3023	16025	30154
(d)	81015	118829	149359

Figure 3.16: Average number of milestones generated by OPTIMIST, PESSIMIST, and plain SBL in the environments (a) through (d) of Figure 3.12

	$T_{optimist}$	$T_{pessimist}$	$T_{SSRP}$	$T_{SBL}$
(h)	1.68	1.59	1.68	1.60
(i)	2.59	2.16	2.59	2.40

Figure 3.17: Performance of OPTIMIST, PESSIMIST, SSRP, and plain SBL in the two environments of Figure 3.13

all problems but the last problem.

The table in Figure 3.16 lists the average numbers of milestones in the roadmaps generated by OPTIMIST, PESSIMIST and plain SBL over 100 runs, for the examples (a) through (d) of Figure 3.12. Recall that milestones generated by OPTIMIST lie in  $F^*$ , while milestones generated by PESSIMIST and plain SBL lie in  $F$ .

The table in Figure 3.17 lists the same results as in Figure 3.15, but for the two examples of Figure 3.13. In both cases, OPTIMIST, PESSIMIST, and plain SBL succeeded in all runs.

### 3.6.3 Discussion

The running times of SBL in examples (h)-(i) are much smaller than in examples (a)-(g). This gives a good indication of the difficulty of finding paths through narrow passages in examples (a)-(g).

In all examples (a)-(g), OPTIMIST is considerably faster than PESSIMIST, so that it is worth running it a few times before invoking PESSIMIST. Even when OPTIMIST fails in examples (f) and (g)-1.1, the effect on the total running time of SSRP is still relatively small. In examples without narrow passages (h) and (i), OPTIMIST is only

as fast as PESSIMIST, but succeeds consistently, so that in practice SSRP does not invoke PESSIMIST.

In every example, OPTIMIST succeeds or fails consistently. This observation strongly suggests that the maximum number ( $N$ ) of calls of OPTIMIST in SSRP be set low. As indicated before, we set it to 5 in our implementation.

PESSIMIST alone is significantly faster than SBL in examples with narrow passages, because repairing configurations sampled in  $\partial^*F$  leads to placing roadmap nodes in narrow passages with higher probability. Hence, PESSIMIST finds paths with smaller roadmaps as shown in Figure 3.16.

On average across various problems with narrow passages, SSRP is faster than PESSIMIST alone, because OPTIMIST often succeeds and costs little when it fails. SSRP is much faster – often by more than one order of magnitude – than SBL alone and more reliable.

The results in Figure 3.17 show that in examples without narrow passages (h)-(i) SSRP is as fast as plain SBL. In fact, in those examples, OPTIMIST, PESSIMIST, SSRP, and plain SBL take about the same amount of time on average.

Example (g) deserves specific comments. Recall that in (g)-1.1, the clearance between the two parts is greater than in (g)-1.0. In both cases, we used the same thinning factor. In (g)-1.1, thinning deforms the narrow passage greatly, to the extent that rotation is almost no longer needed. Thus, OPTIMIST fails to retract many configurations sampled in the deformed passage into the true passage of  $F$ . The deformation of the narrow passage in (g)-1.0 is less dramatic and OPTIMIST can repair configurations sampled in  $\partial^*F$ . But, in this example, paths in  $F$  consists of many tiny segments, so that PESSIMIST, which only repairs configurations sampled in  $\partial^*F$  (and not local paths), fails to find a solution path in reasonable time (as this would require sampling an unrealistic number of configurations). Making PESSIMIST repair local paths would be possible, but would yield a much slower algorithm.

SSRP does not recover the partial roadmaps built by OPTIMIST to speed-up PESSIMIST. As PESSIMIST builds usually much larger roadmaps than OPTIMIST (see Figure 3.16), it is not worth recovering and repairing the roadmaps built by OPTIMIST. Moreover, we observed that, when OPTIMIST fails, the roadmap it has built is

often heavily biased toward false passages; hence, it would be an undesirable input for PESSIMIST.

Figure 3.18 shows plots for standard deviations, means, and maximums of the running times of the SBL planner in the environments of Figure 3.1(a) and Figure 3.12(a)-(c), as the free space is fattened. It shows that fattening of the free space not only quickly reduces the mean running time of a PRM planner, it also quickly reduces the standard deviation and maximum of the running time.

## 3.7 Conclusion

In this chapter we presented a new retraction-based sampling and connection strategy – called small-step retraction – designed to help PRM planners sample paths efficiently through narrow passages. The core idea underlying this strategy is that retracting sampled configurations that are barely colliding is sufficient to speed up the generation of roadmap milestones within narrow passages. Barely colliding configurations are relatively easy to retract out of collision. In addition, they can be efficiently identified by running a classical collision checker on thinned geometric models pre-computed using an MA-based technique. A slight drawback of the method is that it increases pre-computation time.

Another important idea of our work is the combination of two algorithms, OPTIMIST and PESSIMIST, which complement each other well. The entire method was implemented as a new PRM planner - SSRP - that extends the pre-existing SBL planner. SSRP was tested on many examples. Results show that on examples with narrow passages it is significantly faster, and more reliable, than SBL. On examples without narrow passages, it is as fast as SBL. Although our tests of SSRP have given excellent results, example (g) in Figure 3.12 with its two versions 1.0 and 1.1 indicates that there likely exist problems for which neither OPTIMIST nor PESSIMIST would work well, OPTIMIST failing when the shape of a fattened narrow passage is too different from that of the original passage and PESSIMIST when passages are so narrow and curved that it would have to repair local paths, in addition to sampled configurations. However, such a combination is likely to be very rare in practice.

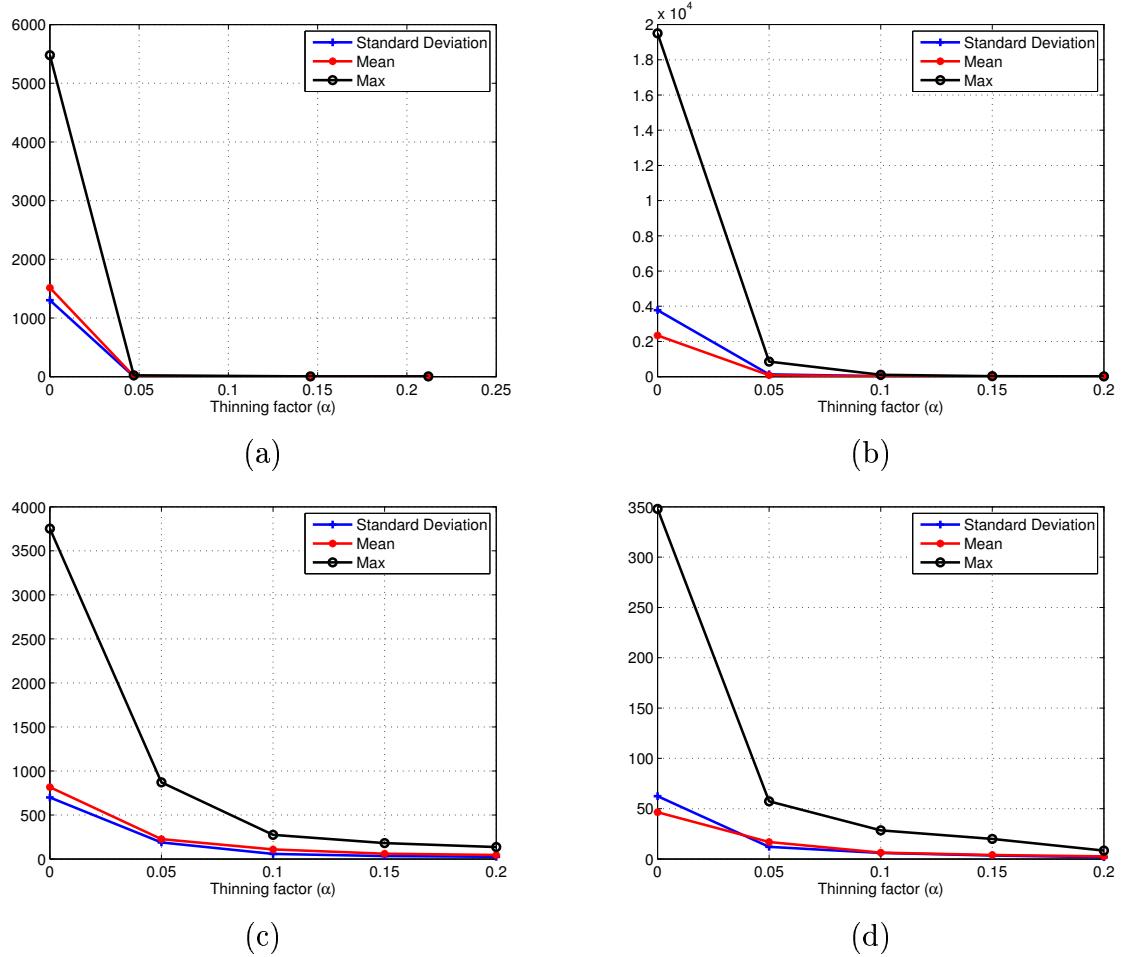


Figure 3.18: Standard deviations, means, and maximums of the running times needed to connect start and goal configurations in the environments of Figure 3.1(a) and Figure 3.12(a)-(c). The  $x$ -axis corresponds to the amount of thinning ( $\alpha=0$  corresponds to no thinning).

# Chapter 4

## Planning Multi-Goal Tours\*

### 4.1 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 chapter 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 4.1 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 4.2 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,

---

\*This chapter is based on the journal article: "Planning Tours of Robotic Arms among Partitioned Goals", *International Journal of Robotics Research*, 25(3):207-223, March 2006. Tim Roughgarden and Jean-Claude Latombe are the co-authors of the article.



Figure 4.1: A spot-welding workcell (courtesy of General Motors).

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 4.3. 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 chapter 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.

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 [23]. 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

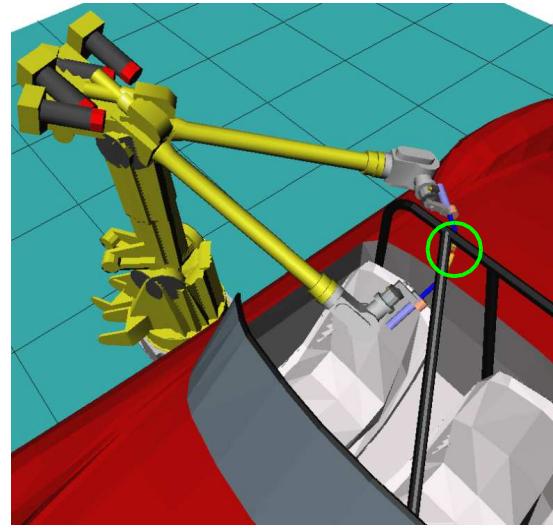


Figure 4.2: Two goal configurations in a group

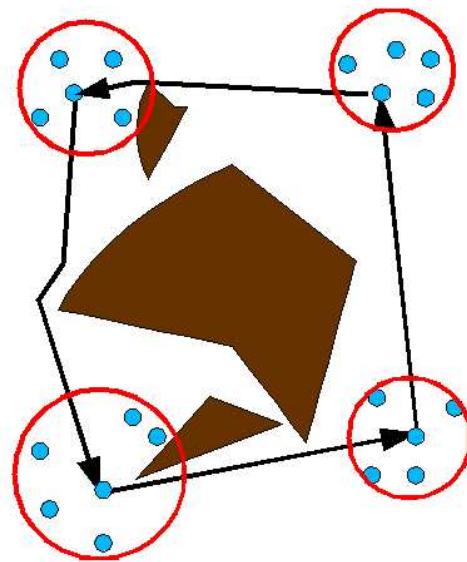


Figure 4.3: Two-dimensional generalized multi-goal motion planning problem

costs. Here, the vertices are partitioned into groups and the tour must only contain one member from each group. This subproblem is at least as hard as finding the shortest tour through points in the Euclidean plane, which is NP-complete [93]. In this chapter, we use two functions to compute sub-optimal, but still satisfactory solutions of these problems. They are:

- 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. PATH makes use of a pre-existing PRM planner (SBL).
- 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. TOUR makes use of a recent approximation algorithm for the group Steiner tree problem [26].

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  $2\beta$  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 [29].

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  $2\beta$  of the length of the optimal tour, where  $\beta$  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 4.6) 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 chapter is organized as follows. Section 4.2 relates our work to previous work on multi-goal planning. Section 4.3 gives a precise statement of the generalized multi-goal planning problem. Section 4.4 describes and analyzes our planning algorithm. Section 4.5 presents in detail the implemented function PATH. Section 4.6 analyzes the experimental performance of the implemented planner on several examples.

## 4.2 Related Work

The multi-goal motion planning problem was previously introduced in [31, 107, 115]. We review these works below and relate them to ours.

*a)* In [107] 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 [66]. 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 [29, 87].

This planner is based on a set of coherent choices. The CMM problems considered in [107] 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 [107], 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 [55] and refined in [103] (more details are given in Section 4.5). 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 4.6), 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 [115] 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 [115] 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 [31] 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 [31], they are computed using the approximate randomized “art-gallery” algorithm given in [90]. 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 [87]. 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 [31].
- 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 [24] and genetic algorithms [15].

## 4.3 Problem Formulation

### 4.3.1 Overview

Let  $C$  denote the configuration space of a robot,  $F \subseteq C$  its collision-free subset, and  $\ell$  the measure that maps any given path  $\tau$  in  $C$  to its length  $\ell(\tau)$ . Let  $\tilde{g}_0, \tilde{g}_1, \dots, \tilde{g}_r$  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  $k^{\text{th}}$  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  $\tau$  is the concatenation of  $r + 1$  goal-to-goal paths. The length of  $\tau$  is the sum of the lengths (measured by  $\ell$ ) 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.

### 4.3.2 Function PATH

We assume that the function PATH defined as follows is given. For any two goals  $g_{ik}$  and  $g_{jl}$ ,  $\text{PATH}(g_{ik}, g_{jl})$  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 [103], 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 4.5 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.

<b>Algorithm</b> TOUR( $G_c$ )
1. $T \leftarrow \text{GSTREE}(G_c)$
2. Return PREORDER-WALK( $T$ )

Figure 4.4: TOUR algorithm

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  $\tilde{g}_0, \tilde{g}_1, \dots, \tilde{g}_r$  when every goal-to-goal path is computed by PATH.

### 4.3.3 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  $i = 0, 1, \dots, r$ ,  $G$  is said to be *non-partitioned*. A *tour* of  $G$  is any list  $\pi = (g_{0k_0}, g_{i_1 k_1}, \dots, g_{i_r k_r}, g_{0k_0})$  such that  $\langle i_1, \dots, i_r \rangle$  is a permutation of  $\{1, \dots, r\}$ .

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  $\pi$  of  $G$  is the sum of the costs of the edges traversed by  $\pi$ , hence:

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

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

In particular, when the cost of every edge  $\{g_{ik}, g_{jl}\}$  in  $E$  is equal to  $\ell(\text{PATH}(\{g_{ik}, g_{jl}\}))$ , we denote the weighted goal graph by  $G_\ell$ . So, the cost of the optimal tour of  $G_\ell$  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 4.4 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  $T$  [29].

More precisely, when  $G$  is non-partitioned,  $T$  is simply a spanning tree of  $G$  and  $\text{GSTREE}(G_c)$  computes the minimum-cost spanning tree of  $G$  in time  $O(r^2)$  using the Prim's algorithm [29]. In this case,  $\text{TOUR}$  is a classical TSP algorithm, which has also been used in the multi-goal planners described in [31, 107, 101]. Then  $\text{TOUR}(G_\ell)$  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  $G_\ell$  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 [26, 99]. This problem is provably hard, even to approximate closely [45], but the polynomial-time algorithm given in [26] 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<sup>1</sup>  $\epsilon > 0$ . (A slightly better worst-case ratio is possible using linear programming [41], but this approach would be too computationally expensive for our purposes.) When  $G$  is partitioned, our function  $\text{GSTREE}$  uses this algorithm. Then  $\text{TOUR}(G_\ell)$  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.

#### 4.3.4 Objective

The naive multi-goal planning algorithm **NAIVE-GMGP** first computes  $G_\ell$  by weighting every edge  $\{g_{ik}, g_{jl}\}$  of  $G$  by  $\ell(\text{PATH}(\{g_{ik}, g_{jl}\}))$ , then evaluates  $\text{TOUR}(G_\ell)$  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  $\text{TOUR}(G_\ell)$ . 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

---

<sup>1</sup>Note that  $\epsilon$  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
  - 2.1.  $T \leftarrow \text{GSTREE}(G_c)$
  - 2.2.  $\kappa \leftarrow c(T)$
  - 2.3. Repeat while  $c(T) \leq \alpha \times \kappa$ 
    - 2.3.1. If all edges in  $T$  are exact then return multi-goal path defined by  $\text{PREORDER-WALK}(T)$
    - 2.3.2. Pick a non-exact edge  $e = \{g_{ik}, g_{jl}\}$  in  $T$
    - 2.3.3. Modify  $G_c$  by resetting the cost of  $e$  to  $\ell(\text{PATH}(g_{ik}, g_{jl}))$

Figure 4.5: Lazy multi-goal planning algorithm

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  $\text{PATH}$  is high relative to that of  $\text{TOUR}$  and we wish to use  $\text{PATH}$  sparingly, even if this requires evaluating  $\text{TOUR}$  (or just  $\text{GSTREE}$ ) 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.

## 4.4 Lazy Planning Algorithm

### 4.4.1 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  $\ell$  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` 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  $\ell(\text{PATH}(g_{ik}, g_{jl}))$ , we say that this edge is *exact*. This yields the lazy algorithm `LAZY-GMGP` shown in Figure 4.5 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 2.3 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  $\alpha$  of the initial cost recorded in  $\kappa$ , 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 2.3.3, 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 2.1, 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 4.5.

#### 4.4.2 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 4.3.3 that  $\mathcal{L}_N \leq 2\beta \times \mathcal{L}_{Opt}$ , where  $\mathcal{L}_{Opt}$  is the length of the optimal tour in  $G_\ell$ . 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:*

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

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

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

and:

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

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

Therefore, at Step 2.2, we have  $\kappa \leq \beta \times c(T_\ell^*)$ . Step 2.3 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  $c(T) \leq \alpha \times \kappa \leq \alpha\beta \times c(T_\ell^*)$ . Since the length  $\mathcal{L}_G$  of the multi-goal path determined by  $\text{PREORDER-WALK}(T)$  at Step 2.3.1 is within factor 2 of  $c(T)$ , we have  $\mathcal{L}_G \leq 2\alpha\beta \times c(T_\ell^*)$ . We also have  $c(T_\ell^*) \leq \mathcal{L}_{Opt}$ , hence  $\mathcal{L}_G \leq 2\alpha\beta \times \mathcal{L}_{Opt}$ . ■

Note that when  $\alpha$  is set to 1, the lengths of the paths respectively computed by NAIVE-GMGP and LAZY-GMGP are within the same factor  $2\beta$  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 [101].

The tour returned by  $\text{PREORDER-WALK}(T)$  at Step 2.3.1 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  $\text{PATH}$  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  $T$  and its preorder walk, the tour returned by  $\text{PREORDER-WALK}(T)$  has cost within a factor 2 of that of  $T$ .

The experimental results of Section 4.6 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  $\alpha$  gets larger, while the lengths of the paths only increase slightly.

However, for any given value of  $\alpha$ , 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  $g_0, g_1, \dots, g_r$  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  $\varepsilon$ . 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 4.6). Then, each execution of Step 2.3 (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 2.1 necessarily includes at least one non-exact edge. Therefore, LAZY-GMGP will terminate only when all goal-to-goal paths have been computed.

#### 4.4.3 Improvements

Below we briefly discuss three possible improvements of the LAZY-GMGP algorithm shown in Figure 4.5. 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 2.3.3 of LAZY-GMGP by the steps given in Figure 4.7, where  $\gamma \geq 0$  is a real constant aimed at skipping the update when  $\text{PATH}(g_{ik}, g_{jl})$  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 2.3 of LAZY-GMGP, we can try

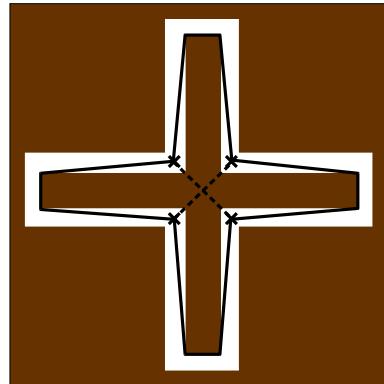


Figure 4.6: A worst-case example for LAZY-GMGP in a two-dimensional configuration space  $C$ . The free space  $F$  is shown in white and each of the 4 cross-marks denotes a goal configuration. The dashed and plain lines depict the shortest paths between the goals in  $C$  and in  $F$ , respectively. 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.

to pick, at Step 2.3.2, 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 [103]).

c) Step 2.1 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-partitioned 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  $O(k\sqrt{r})$  [35, 39]. We are not aware of a similar algorithm for updating group-spanning trees.

## 4.5 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

- $\lambda \leftarrow \text{cost of } e \text{ in } G_c$
- 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 4.7: Improvement of Step 2.3.3 of LAZY-GMGP

PATH uses a heuristic two-phase approach: first, compute a collision-free path; next, optimize this path. This approach was introduced in [13].

We use a bi-directional tree-expansion PRM planner (more precisely, the SBL planner presented in [103]) 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 [55, 103]. 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 guarantee 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 [3], except that in PATH all trees are rooted at goal configurations.

Several optimizers can be used to improve a path generated by the PRM planner. For instance, the variational optimization technique used in [13] 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 4.8 [49]. This optimizer assumes that the shortest path in configuration space between two arbitrary configurations is the straight segment. The input path,  $\tau$ , is a polygonal line in configuration space. The optimizer repeatedly replaces sub-paths of  $\tau$  by collision-free straight segments. Though the outcome is only locally optimal at best, it is usually quite satisfactory.

## 4.6 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 4.5).

For PATH we used the same implementation of SBL as used in Chapter 3. When

**Algorithm** OPTIMIZE( $\tau$ )

Repeat  $N$  times

1. Pick two configurations  $q$  and  $q'$  on  $\tau$  at random
2.  $\tau \leftarrow \text{SHORTEN}(q, q', \tau)$

**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.  $\tau \leftarrow \text{SHORTEN}(q, q'', \tau)$
  - 2.3. Return  $\text{SHORTEN}(q'', q', \tau)$

Figure 4.8: Path optimizer

goals are not partitioned, a minimum-cost spanning tree is computed using the Prim's algorithm [29]. Otherwise, a near-optimal group-spanning tree is computed using an implementation of the algorithm given in [26] (since the input to this algorithm must be a tree, as suggested in [26], we use the algorithm proposed in [36] 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 4.4.3 *a*) and *b*).

Throughout this section, we consider the three examples depicted in Figures 4.9, 4.10, and 4.11, 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

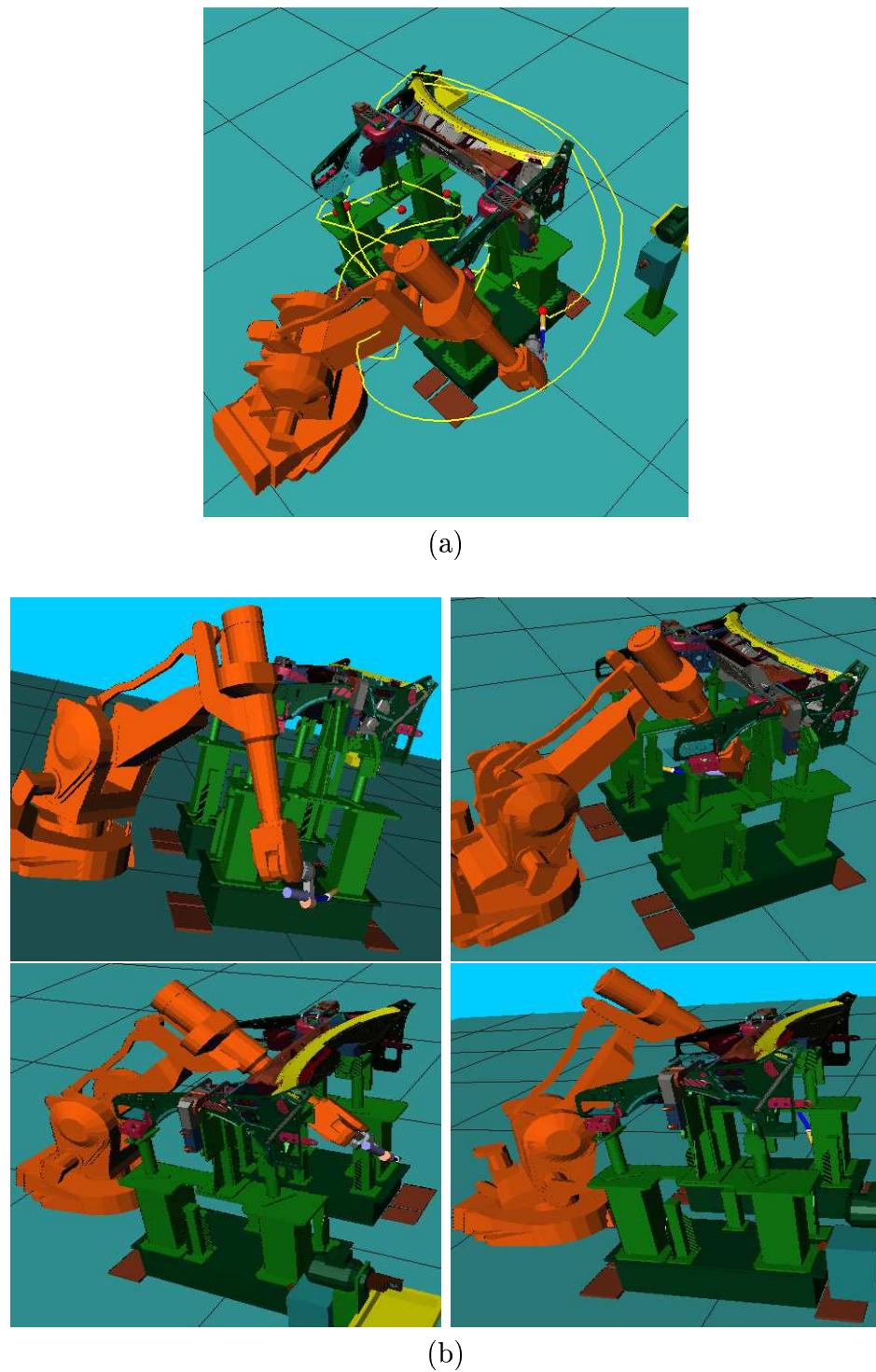


Figure 4.9: (a) Example 1 (10 goal groups); (b) some goal configurations

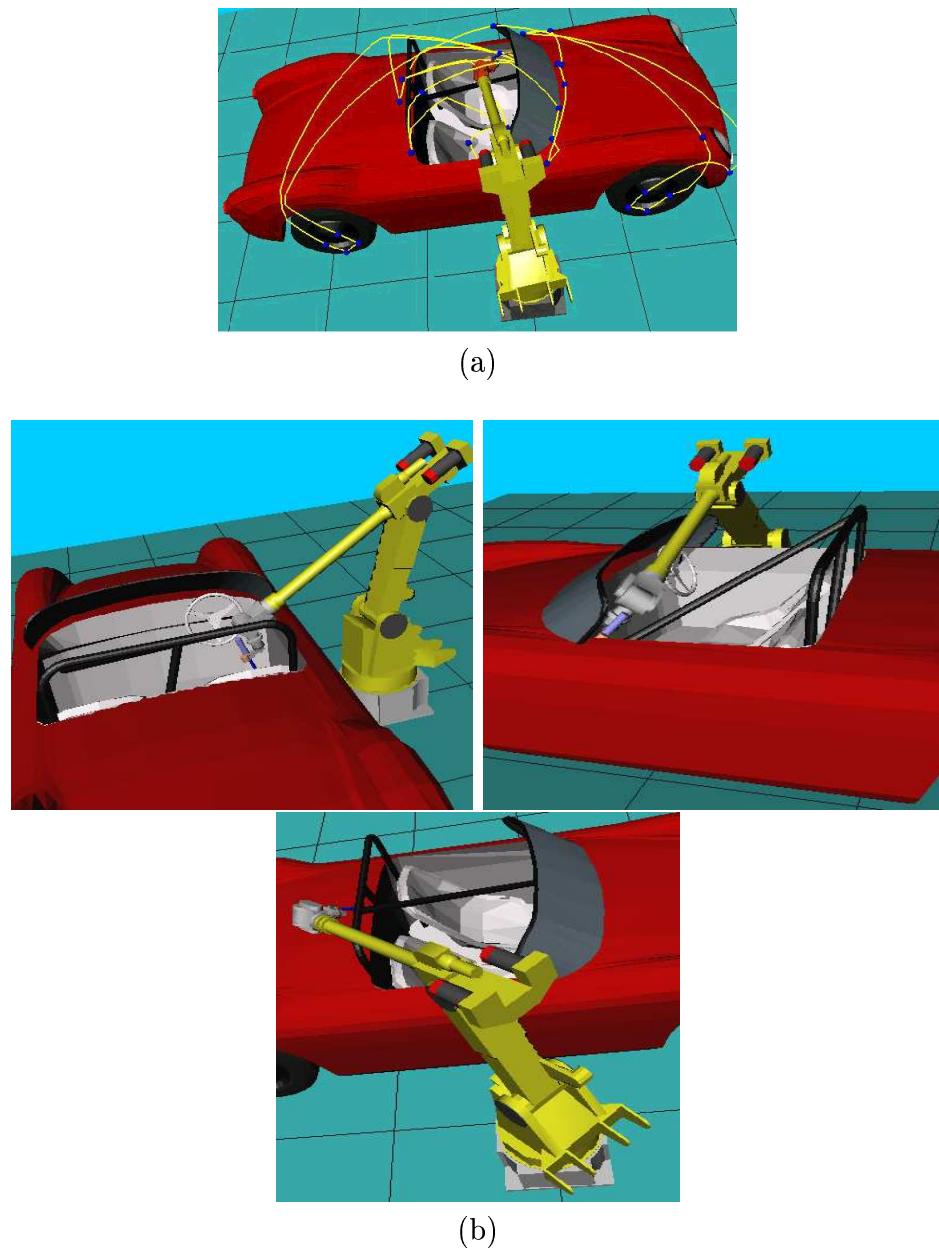


Figure 4.10: (a) Example 2 (31 goal groups); (b) some goal configurations

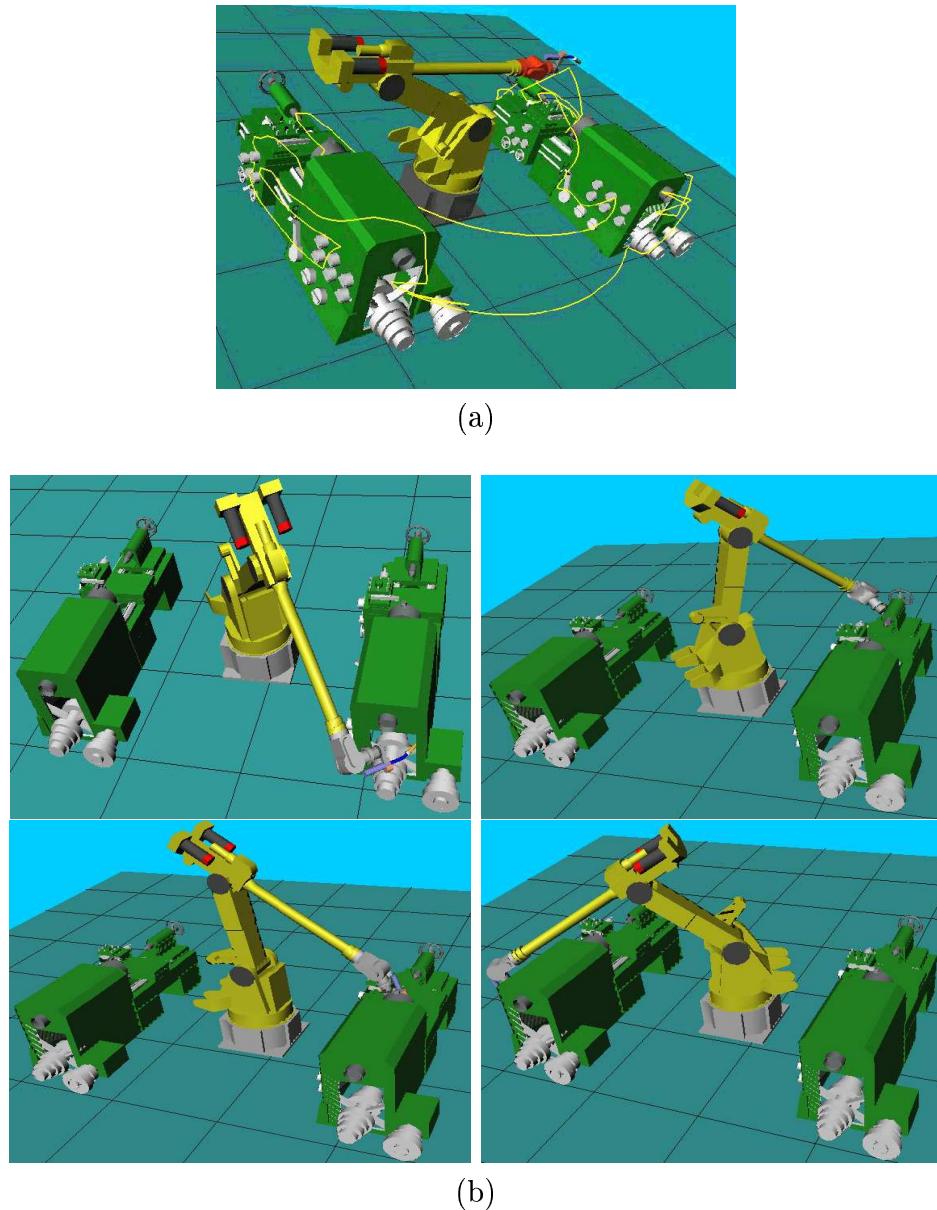


Figure 4.11: (a) Example 3 (50 goal groups); (b) some goal configurations

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.

#### 4.6.1 Non-partitioned case

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

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

Table 4.1 lists performance data with the parameter  $\alpha$  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 4.12, which plots the running times of LAZY-GMGP and NAIVE-GMGP when 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 4.4.3 a) and b) are not included.

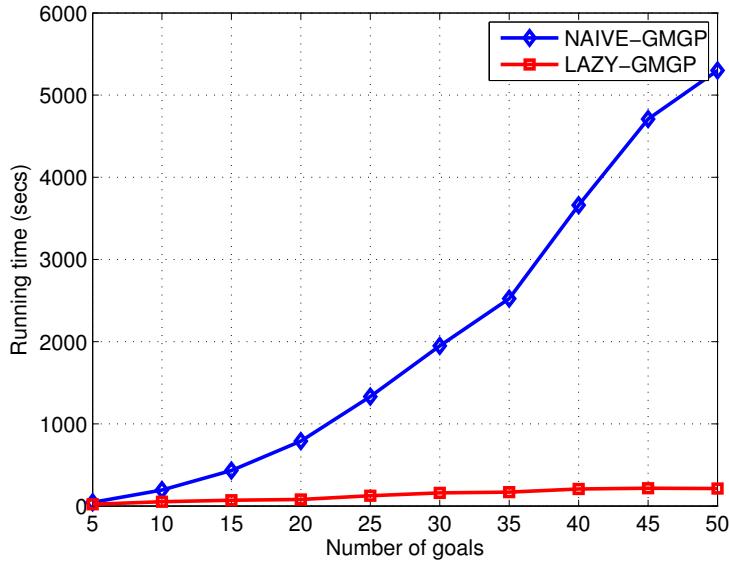


Figure 4.12: Running times of NAIVE-GMGP and LAZY-GMGP as the number of goal configurations grows from 5 to 50 in Example 3 (non-partitioned case)

In each example, the two planners generate the same multi-goal path. This is not surprising. Indeed, when  $G$  is non-partitioned and  $\alpha$  is set to 1, LAZY-GMGP terminates at Step 2.3.1 with a spanning tree  $T$  that is optimal in both  $G_c$  and  $G_\ell$ . So, if  $G_\ell$  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,  $G_\ell$  has a unique optimal spanning tree. This is the case in our three examples.

Table 4.1 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 [107] (see Section 4.2 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 avoid computing un-necessary goal-to-goal paths.

#### 4.6.2 Partitioned case

Table 4.2 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 4.6.3.

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

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  $G_\ell$  are greater than or equal to those in  $G_c$ , GSTREE( $G_c$ ) may be luckier than GSTREE( $G_\ell$ ) and compute a tree  $T_c$  whose cost is less than that of the tree  $T_\ell$  computed by GSTREE( $G_\ell$ ). Secondly, even if  $c(T_c) > c(T_\ell)$ , it is still possible that  $T_c$  leads to a shorter pre-order walk than  $T_\ell$ .

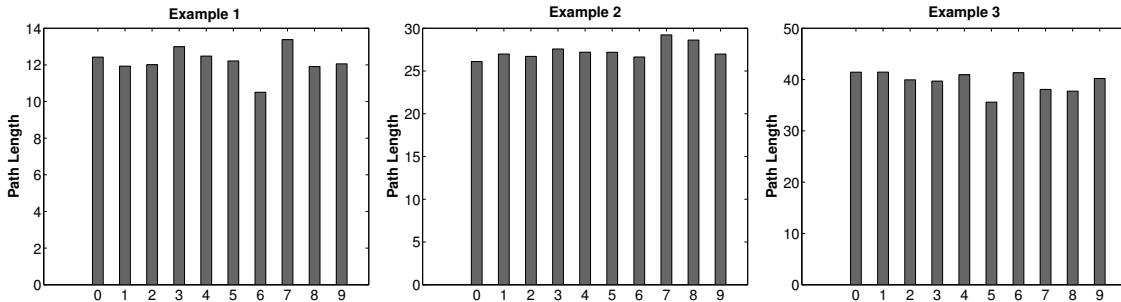


Figure 4.13: Lengths of paths returned by LAZY-GMGP for ten different sets of non-partitioned goals for Examples 1, 2 and 3

Since GSTREE is exact in the non-partitioned case, but only approximate in the partitioned case, it is interesting to compare the lengths of the 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 4.1 were generated in this way, and all three lengths reported in Table 4.2 by solving the partitioned problem are significantly better. We conducted more experiments summarized in the three bar graphs of Figure 4.13. 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.

#### 4.6.3 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 4.2 reveals that, in Example 3, evaluating GSTREE takes 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

```

2.1.  $\delta_{gstree} \leftarrow \text{TIME}(\text{GSTREE})$ 
2.2.  $\delta_{path} \leftarrow 0$ 
2.3. Repeat while  $c(T) \leq \alpha \times \kappa$  or  $\delta_{path} \leq \delta_{gstree}$ 
    2.3.1. If all edges in  $T$  are exact
        2.3.2. then if  $c(T) \leq \alpha \times \kappa$ 
            2.3.3. then return multi-goal path defined by PREORDER-WALK( $T$ )
            2.3.4. else exit loop 2.e
        2.3.5. else
            2.3.6. Pick a non-exact edge  $e = \{g_{ik}, g_{jl}\}$  in  $T$ 
            2.3.7. Modify  $G_c$  by resetting the cost of  $e$  to  $\ell(\text{PATH}(g_{ik}, g_{jl}))$ 
        2.3.8.  $\delta_{path} \leftarrow \delta_{path} + \text{TIME}(\text{PATH})$ 

```

Figure 4.14: Replacing Step 2.3 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

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 2.3 of LAZY-GMGP with the steps shown in Figure 4.14. The function  $\text{TIME}(x)$ , where  $x = \text{PATH}$  or  $\text{GSTREE}$ , returns the time spent in the last evaluation of  $x$ . Thanks to the test at Step 2.5.2, 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 4.3 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		
	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 4.3: Results obtained with LAZY-MGP-2 when each goal group contains 5 configurations and  $\alpha = 1$  (all times are in seconds)

LAZY-GMGP-2 has an interesting additional property over LAZY-GMGP. In a worst-case example, like the one shown in Figure 4.6, 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.

#### 4.6.4 Influence of parameter $\alpha$

Here, we measure the impact of the parameter  $\alpha$  on the performance of LAZY-GMGP. The three plots in Figure 4.15 correspond to the three examples of Figures 4.9, 4.10, and 4.11, with non-partitioned goals. Each plot shows three curves that respectively represent the total running time of LAZY-GMGP, the length of the computed multi-goal path, and the number of evaluations of PATH when  $\alpha$  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  $\alpha$  increases, but then level out. In

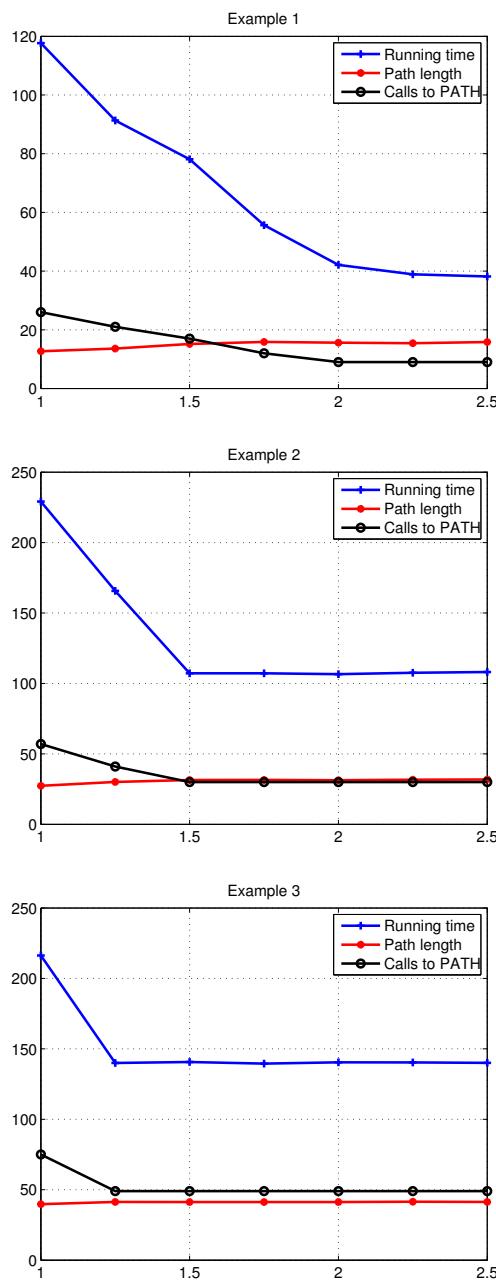


Figure 4.15: Running time of LAZY-GMGP, length of solution, and number of calls to PATH on Examples 1, 2, and 3 when  $\alpha$  grows from 1 to 3

fact, when the value of  $\alpha$  gets greater than a certain threshold  $\alpha_0$ , Step 2 of LAZY-GMGP returns a solution at the first cycle. Then, the average running time of LAZY-GMGP stops improving. The value  $\alpha_0$  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 examples,  $\alpha_0$  ranges between 1.2 and 2.3.

## 4.7 Conclusion

The main contribution of this chapter is a new PRM-based planner LAZY-GMGP for solving multi-goal motion planning problems (MGP). The objective in MGP is to compute an optimal or near-optimal collision-free path for a robot arm through several goal configurations given as input. The order in which the robot should visit the goal configurations is not given and must be automatically determined. MGP can be seen as a generalization of the classical goal-to-goal shortest path problem. In fact, MGP combines two notoriously hard sub-problems in computer science – the collision-free shortest-path and the traveling-salesman problems (TSP). MGP occurs in many basic industrial operations, such as in spot-welding, car-painting, inspection, and measurement tasks.

NAIVE-GMGP, a naive algorithm to solve the MGP, uses a single-query PRM planner to find a collision-free path between every pair of goal configurations – hence quadratic number of times – to determine a near-optimal multi-goal tour. Since invoking a PRM planner quadratic number of times could be prohibitively expensive, the main challenge for LAZY-GMGP has been to avoid computing paths between every pair of goal configurations and still be able to determine a multi-goal tour as good as the one that NAIVE-GMGP would determine.

LAZY-GMGP generates a near-optimal tour of the goal 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 [26].

As mentioned before, LAZY-GMGP 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 LAZY-GMGP invokes a PRM planner to compute a goal-to-goal path much less frequently than NAIVE-GMGP and hence is much faster, while producing paths of similar lengths.

# Chapter 5

## Manipulation Planning for Deformable Linear Objects\*

### 5.1 Introduction

There are many important tasks, such as surgical suturing, assembling and manipulating cable harnesses, laying out household carpets, and handling fabric, where it is necessary to manipulate deformable objects. However, so far research in manipulation planning (including PRM-based) has mainly focused on manipulating rigid objects. In this last chapter we consider the problem of planning for manipulation of a deformable linear objects(DLO), such as a rope, a cable, or a suture, with robot arms. The two main challenges arise when trying to apply the PRM approach to this class of planning problems:

- DLO exhibits a much greater diversity of behaviors than a rigid object, by taking many different shapes when submitted to external forces. It may interact with other objects (sliding and knotting) and also with itself (to form self-knots). While planning the complete sequence of robot motions which achieves the desired goal, the planner must keep track of how the shape of the deformable

---

\*This chapter is based on the conference publication: “Motion Planning for Robotic Manipulation of Deformable Linear Objects”, *International Symposium on Experimental Robotics*, July 2006. Pekka Isto and Jean-Claude Latombe are the co-authors in the publication.

object is altered. The physical behaviors are different for different DLOs. The planner should not rely on any particular type of DLO and should be able to generate manipulation plans for various DLOs, ranging from ropes to thin nylon surgical sutures.

- Unlike in more traditional motion planning problems, the goal is a topological state of the world, rather than a geometric one. For instance, if the task is to wind a cable around an object, it is more important to achieve the desired number of wounds than any specific shape. Similarly, when tying a knot with a rope, the exact geometry of the knot is not important.

The contribution of this chapter is a new PRM motion planner for manipulating DLOs and tying knots (self-knots and knots around simple static objects) using co-operating robot arms. We blend new ideas with pre-existing concepts and techniques from knot theory, robot motion planning, and computational physical modeling of DLOs. Our planner does not depend on any particular physical model of the DLO. Instead, it takes a model as input, in the form of a state-transition function. The goals to be achieved are specified as topological states of the world. The planner constructs a probabilistic roadmap that is biased toward achieving the goal topology of the goal state of the DLO. During roadmap construction, the planner tests that the grasp points on the rope are accessible by the arms without collision. The planner assumes that simple static sliding supports (independent of the robot arms), which we call needles (by analogy to the needles used in knitting) are available and can be used when needed, to maintain the integrity of certain portions of the DLO during manipulation. A novel method is used to account for the interaction of the DLO with simple rigid objects. Curve representations of the objects, obtained from their skeletonization, are “chained” with the DLO to produce a *composite* semi-deformable linear object (sDLO). Thereafter, the problem reduces to that of tying self-knots with the composite sDLO. We implemented the planner and tested it in simulation and with real robots.

The implemented planner was tested in simulation to achieve various knots like bowline, neck-tie, bow (shoe-lace), and stun-sail. We also used it to tie bowline knots

with various household ropes on a hardware platform with two PUMA 560 robots.

The rest of the chapter is organized as follows. Section 5.2 introduces the problem of planning for robotic manipulation planning of deformable linear objects. Section 5.3 relates our work to previous work. Section 5.4 describes our modeling strategy for DLOs. Section 5.5 precisely formulates the class of manipulation planning problem that our new planner addresses. Section 5.6 provides a technical description of our new planner. Section 5.8 presents experimental results.

## 5.2 Planning for Robotic Manipulation of DLOs

Robotic manipulation of rigid objects is a rather well-studied problem. Here, we are interested in manipulating deformable linear objects (DLOs), such as ropes, cables, and sutures. Progress in robotic manipulation of DLOs can benefit many application domains, like manufacturing, medical surgery, and agriculture, where DLOs are ubiquitous. It can also benefit humanoid robots, since tying knots is a common activity in daily life. However, DLOs add a number of difficulties to the manipulation task. They exhibit a much greater diversity of behaviors than rigid objects, by taking many different shapes when submitted to external forces. In particular, self-collisions are possible and must be considered. Furthermore, the manipulation of DLOs almost inevitably requires two, or more, arms performing well-coordinated motions and re-grasp operations. Finally, the topology of the goal state of a DLO is usually far more important than its exact shape.

Figure 5.1 shows two typical problems encountered while manipulating DLOs. In Figure 5.1(a), a segment of rope is initially unwound. Figures 5.1(b) & (c) depict two types of goal states in which the rope forms a self-knot (bowline) and winds around some static objects, respectively. Our planner does not depend on any particular physical model of the DLO. Instead, it takes a model as input, in the form of a state-transition function. Using this function and the model of the robot arms, the planner constructs a probabilistic roadmap in the configuration space of the DLO. The sampling of this roadmap is biased toward achieving the topology of the goal state of the DLO. During roadmap construction, the planner tests that the grasp points on the

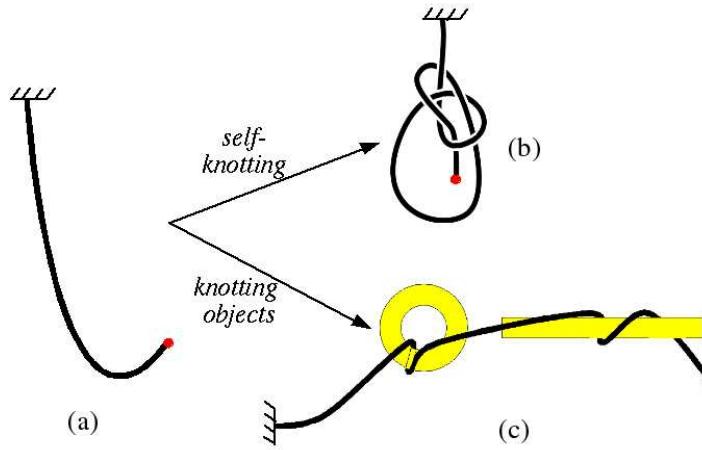


Figure 5.1: Initial (a) and goal (b)-(c) states in a manipulation problem

rope are accessible by the arms without collision. The planner assumes that simple static sliding supports (independent of the robot arms), which we call needles (by analogy to the needles used in knitting) are available and can be used when needed, to maintain the integrity of certain portions of the DLO during manipulation. A novel method is used to account for the interaction of the DLO with simple rigid objects. Curve representations of the objects, obtained from their skeletonization, are “chained” with the DLO to produce a *composite* semi-deformable linear object (sDLO). Thereafter, the problem reduces to that of tying self-knots with the composite sDLO. We implemented the planner and tested it in simulation. We demonstrated its effectiveness by tying commonly used knots like bowline, neck-tie, bow (shoe-lace), and stun-sail.

## 5.3 Related Work

### 5.3.1 Manipulation planning

Robot manipulation planning with rigid objects was first addressed in [113]. The randomized algorithm proposed in [70] generates motion paths for multiple cooperating manipulators to manipulate a movable rigid object. The algorithm assumes that a

set of discrete grasps is given as input. The algorithm proposed in [106] works with continuous sets of grasps of rigid objects but plans for a single manipulator. The regrasp is done by releasing the object. In the case of a DLO, releasing the grasp makes the DLO very unstable due to its deforming nature. An algorithm for planning paths for elastic objects, especially for flexible plates and cables, is presented in [77]. This algorithm plans motions only for the object, not for the manipulator. The problem of path planning for DLOs in presence of obstacles is addressed as a variational problem in [109]. Their formulation does not consider the manipulator either. All these works focus on geometric planning while we focus on topological planning, because while manipulating DLOs, especially to tie knots, it is more important to achieve an acceptable topology than a specific geometry.

### 5.3.2 Application of knot theory in robotics

Knot theory provides means to capture and analyse the topological states and state transitions of a DLO [2]. Its applications in robotics include work presented in [88]. Like us, they present a data structure for describing the state of a DLO as a sequence of signed crossings. State transitions are caused by Reidemeister moves and a crossing operation that moves the end of the DLO over another part to make a new crossing. Similar ideas are being used in [110] to build a vision guided robot system for one-handed manipulation of a DLO with the aid of the floor. However, collision constraints and the physical behavior of the DLO are not considered during the planning phase. In [72], motion planning techniques from robotics are used to untangle mathematical knots.

### 5.3.3 Vision-based DLO manipulation

The difficulty of accurately modeling deformable objects has motivated vision-based approaches to DLO manipulation. Among other examples, in [85], methods for DLO modeling, recognition, and parameter identification are presented, which have been embedded in a system capable of tying a rope around a cylinder with two manipulators. In [100], a sensing-based method is proposed for picking up hanging DLOs.

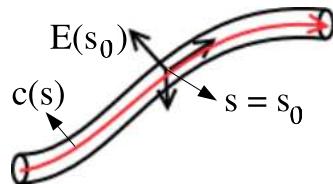


Figure 5.2: Representation of a DLO

The implementation of our proposed manipulation planning algorithm could be used on the sensing and manipulation hardware, developed in these works (and in [88]), to achieve real-life manipulation of DLOs.

## 5.4 Modeling a Deformable Linear Object

### 5.4.1 Geometric model

We describe the geometry of a DLO by a curved cylinder of length  $L$  and circular cross-section of constant non-zero radius (see Figure 5.2). The *axis* of this cylinder is a smooth curve  $c$  parameterized by the Euclidean length  $s$  along the DLO, i.e.:

$$c : s \in [0, L] \rightarrow c(s) \in \mathbf{R}^3,$$

where  $c(0)$  is the *tail* of the DLO and  $c(L)$  its *head*. Whenever the physical model of the DLO includes torsion or twist, we also attach a Cartesian coordinate frame  $E(s)$  with each point  $c(s)$  as shown in Figure 5.2. We call  $q = (c, E)$  a *configuration* of the DLO.

### 5.4.2 Physical model

Designated points located at  $s_1, \dots, s_k$  on the DLO are *grasp* points. These are the only points whose positions  $c(s_i)$  and orientations  $E(s_i)$ ,  $i = 1, \dots, k$ , can be directly controlled by the robotic manipulation system.

The physical model of a DLO is given in the form of a *state transition* function  $f$  that maps both a configuration  $q_{old}$  of the DLO and a  $k$ -vector  $u$  of small displacements

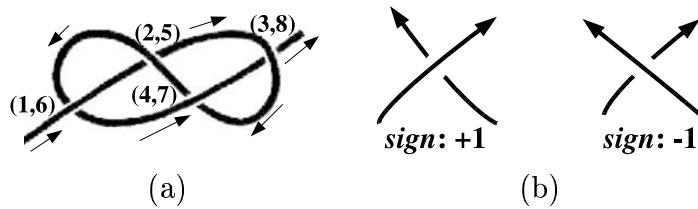


Figure 5.3: (a): The crossings in the “figure-8” knot. (b): The sign convention for the crossings

of the grasp points (the control input) to a new configuration  $q_{new}$ . We assume that both  $q_{old}$  and  $q_{new}$  are stable configurations, although the DLO may exhibit dynamic behavior (e.g., snapping) during the transition from  $q_{old}$  to  $q_{new}$ . We do not consider elapsed time between  $q_{old}$  and  $q_{new}$ , although it could be easily added to the model.

We assume that the model in  $f$  handles collisions with obstacles, as well as self-collisions, so that the DLO does not “jump” over itself or obstacles. In addition, if  $u$  would cause violations of physical constraints associated with the DLO, e.g., over-stretching, then the function  $f$  indicates that the execution of  $u$  is impossible. As we do not allow the robot arms to touch the DLO at points other than the grasp points,  $f$  reports failure if it detects a collision between the DLO and an arm.

Several physical models of DLOs proposed in the literature (e.g., [18, 33, 92, 94, 111]) are relevant to the construction of  $f$ .

### 5.4.3 Topological model

We characterize the topology of the DLO at some configuration  $q = (c, E)$  by means of its crossing configuration [2]. A crossing configuration is defined with respect to a reference plane  $P$ . Let  $c'$  be the projection of  $c$  into  $P$ . The crossing configuration of  $c$  with respect to  $P$  is defined only when no more than two points in  $c$  project to the same point in  $P$ . Moreover, for any two points  $a$  and  $b$  in  $c$  that projects to the same point  $a' = b'$  in  $P$ ,  $c'$  must admit two distinct tangents at  $a' = b'$ , which are the respective projections of the tangents to  $c$  at  $a$  and  $b$ . Then a *crossing*  $X$  in  $P$  is any point on  $c'$  that is the projection of two distinct points of  $c$ .

Let  $c(s_1)$  and  $c(s_2)$ , with  $s_1 < s_2$ , be the two points on  $c$  that projects to the

crossing  $X$  in  $P$ . Let  $\tau_1$  and  $\tau_2$  be the projections of the tangent to  $c$  at  $c(s_1)$  and  $c(s_2)$ , respectively. The status of  $X$  is said to be *over* if  $c(s_1)$  lies above  $c(s_2)$  with respect to  $P$ , otherwise it is *under*. Moreover,  $X$  is assigned a sign. This sign is + if (1)  $X$  is over and the counter-clockwise angle between  $\tau_1$  and  $\tau_2$  is less than  $\pi$ , or (2)  $X$  is under and the clockwise angle between  $\tau_1$  and  $\tau_2$  is less than  $\pi$ . The sign of  $X$  is - in all other cases. The sign convention is also depicted in Figure 5.3(b).

Suppose that  $c'$  has  $n$  crossings  $X_1, \dots, X_n$ . Let us “walk” along  $c'$  from  $s = 0$  to  $s = L$  and assign the integral labels  $1, \dots, 2n$  in increasing order to the crossings as they are encountered. Since each crossing  $X_j$  is encountered exactly twice, it receives two distinct labels  $X_j^1$  and  $X_j^2$ . Figure 5.3(a) illustrates the labeling of crossings in the “figure-8” knot. The *crossing configuration* of  $c$  with respect to  $P$  is defined by the set of triplets:

$$\{(X_j^1, X_j^2, \epsilon_j)\}_{j=1, \dots, n}.$$

where  $\epsilon_j$  denotes the sign of the crossing  $X_j$ . Moreover, the over/under status associated with  $X_j$  can be encoded by assigning opposite signs to  $X_j^1$  and  $X_j^2$ . In Figure 5.3(a) and in the rest of the chapter we will ignore the signs, when listing the crossings, for the sake of compactness.

Note that many configurations  $q$  of a DLO can achieve the same crossing configuration with respect to  $P$ . We denote by  $C(x)$  the set of all configurations  $q$  of the DLO that achieve a crossing configuration  $x$ .

## 5.5 DLO Manipulation Planning Problem

A DLO manipulation planning problem is defined by the following inputs: the radius and length  $L$  of the DLO, the coordinates  $s_1, \dots, s_k$  of the grasp points, the state transition function  $f$ , an initial configuration  $q_{init}$  of the DLO, a reference projection plane  $P$ , a goal crossing configuration  $x_{goal}$  of the DLO, a model of the robot arms forming the manipulation system, and a set of fixed obstacles.

The solution to this problem is a sequence of collision-free paths of the robots that achieve the goal crossing configuration  $x_{goal}$  with respect to  $P$ , that is, a configuration

$q \in C(x_{goal})$ . Any two consecutive paths are separated by (re-)grasp operations. During the manipulation, the robots are not allowed to touch the DLO, except at the grasp points. The DLO is allowed to touch obstacles. The transition function  $f$  models the interaction between the DLO and the obstacles, and rejects attempted moves that cause the DLO to touch an arm.

In the rest of this chapter, we make the following assumptions:

- The DLO admits only two point grasp, located at  $s = 0$  (tail) and  $s = L$  (head). The tail of the DLO is fixed at some given position and orientation.
- The robotic system consists of two arms, which can both grasp the DLO’s head. At any point of time, a single arm moves the head, but both arms simultaneously grasp the head to eventually switch grasp.
- In its initial configuration  $q_{init}$ , the DLO has no crossing in  $P$  (we say that it is *unwound*).
- Simple passive/static sliding supports, which we call needles (see Section 5.6.3), are available and can be used to maintain the integrity of certain portions of the DLO during manipulation.

In Section 5.7 we will extend the definition of a crossing configuration of a DLO to take obstacles into account, in order to tie knots around obstacles or, instead, to avoid undesired loops of the DLO around obstacles.

## 5.6 Planning Approach

### 5.6.1 Forming sequence

The first step of our planning approach is to ignore the manipulating arms and derive a “qualitative” plan, which we call the *forming sequence* of the crossings in the goal topology  $x_{goal}$  of the DLO. It will be used later to bias the sampling of a probabilistic roadmap in the DLO’s configuration space.

Suppose we walk along the DLO, in a given configuration, from its tail to its head. We say that a crossing is *formed* when it is encountered for the second time.

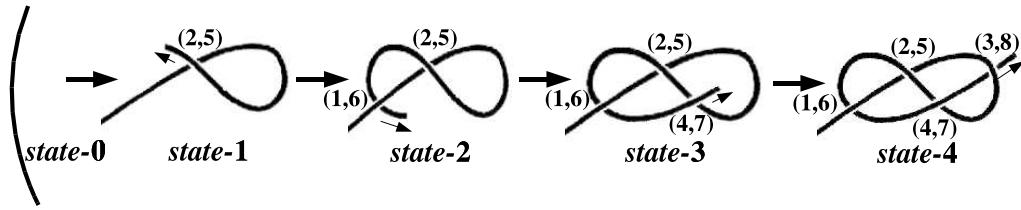


Figure 5.4: A knot can be tied crossing-by-crossing in the order implied by its forming sequence. The above figure illustrates this fact for the “figure-8” knot, whose crossing configuration is  $\{(1, 6), (2, 5), (3, 8), (4, 7)\}$  and the associated forming sequence is  $((2, 5), (1, 6), (4, 7), (3, 8))$ .

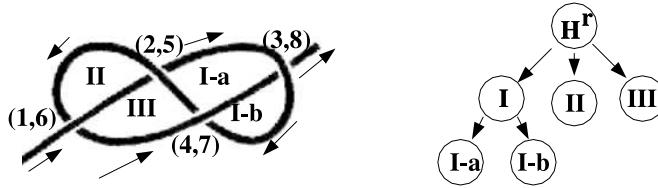


Figure 5.5: The loop structure for the “figure-8” knot

*Forming sequence* is the sequence in which crossings are formed during the walk. Alternatively, if the goal crossing configuration is  $x_{goal} = \{(X_j^1, X_j^2)\}_{j=1, \dots, n}$ , then its forming sequence is  $((X_j^1, X_j^2))_{j=j_1, \dots, j_n}$ , where  $\{j_1, j_2, \dots, j_n\}$  is a permutation of  $\{1, 2, \dots, n\}$  such that  $X_{j_k}^2 < X_{j_l}^2$  if  $k < l$ . Qualitatively, knots can be tied crossing-by-crossing in the order implied by the forming sequence of the goal crossing configuration (see Figure 5.4).

### 5.6.2 Loop structure

The loop structure is built to later identify portions of the DLO whose integrity must be maintained during manipulation by means of needles (see Section 5.6.3). Let  $c'$  be any curve in the reference projection plane  $P$  that forms the crossings defined in  $x_{goal}$ . Let us draw  $c'$  from the tail to the head. Each time a crossing is formed, either a new *loop* is created, or an existing loop is split into two loops. For example, in Figure 5.5, the crossing (2,5) is first formed, which creates the loop denoted I; then crossing (1,6) creates loop II and crossing (4,7) creates loop III; finally, crossing (3,8) splits loop I

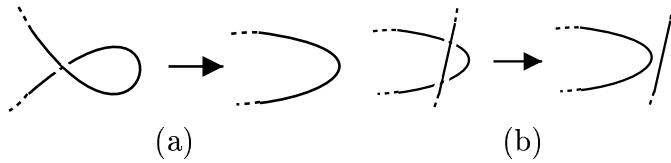


Figure 5.6: (a): Reidmeister move I (b): Reidmeister move II

into two loops denoted by I-a and I-b. The *loop structure* is the hierarchy of all the loops thus formed. The root of this structure points to the newly created loops (I, II, and III in Figure 5.5). Each loop in the structure that has been split (only loop I in Figure 5.5) points toward the two loops resulting from that split. The structure may have arbitrary many levels.

During manipulation, it is critical to maintain some loops sufficiently wide open, so that they can later be split. In addition, some loops could be undone by pulling the head of the DLO. The planner guarantees the integrity of all such loops by introducing needles through them, as described below.

### 5.6.3 Pierced and slip loops

Reidmeister moves are a classical technique used in knot theory to simplify crossing configurations without changing the topology of a knot [2]. Figure 5.6 shows the Reidmeister moves I and II. We do not use such moves to simplify the input goal crossing configuration  $x_{goal}$ . Instead, we assume that the planner's user has appropriately described  $x_{goal}$  so that all the loops it implies are desired loops. However, since common knots do not contain arbitrary loops, we make some additional assumptions about the loops that  $x_{goal}$  may imply.

Let us say that a goal crossing configuration of a DLO in  $P$  is *tight* if it cannot be simplified (i.e., no crossing can be removed) by any Reidmeister move and its forming sequence is *alternating*, i.e., the successive crossings in the sequence are alternately over and under. The crossing configuration depicted in Figure 5.3(a) is tight.

In a tight goal crossing configuration, each split loop  $O$  (loop I in Figure 5.5) is eventually pierced, meaning that split occurs just after two consecutive crossings of

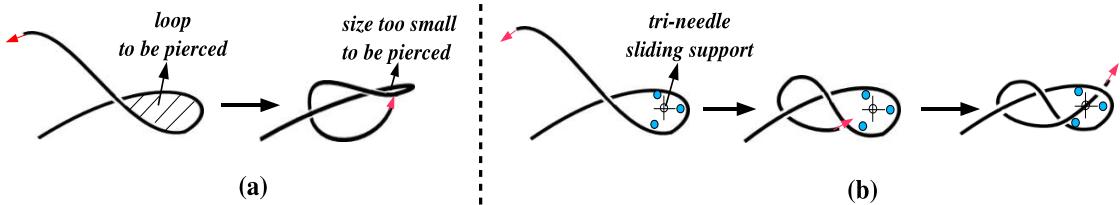


Figure 5.7: (a): After a loop has been created, it is not guaranteed that its size will be maintained during further manipulation (b): Passive/static sliding supports (tri-needle) is used to maintain the size of the loops to be pierced in future.



Figure 5.8: A semi-tight configuration

different over/under status are formed.  $O$  must be wide enough to make it possible for the robot arms to move the head of the DLO through it. The planner achieves this condition by using a tri-needle. The role of the tri-needle is illustrated in Figure 5.7. Its size depends on whether any of the two loops resulting from the split of  $O$  will be split in turn. The tri-needle could be defined in many ways. Here, it consists of three thin straight bars inserted through the loop perpendicular to  $P$ .

We also allow  $x_{goal}$  to be semi-tight. A *semi-tight* crossing configuration is one in which (1) at most two consecutive crossings in the forming sequence have the same over/under status and (2) any two consecutive crossings  $(X_j^1, X_j^2)$  and  $(X_k^1, X_k^2)$  in the forming sequence that have the same over/under status are such that  $|X_j^1 - X_k^1| = 1$  and  $|X_j^2 - X_k^2| = 1$ . Figure 5.8 shows a semi-tight crossing configuration. Most practical knots that rely on friction along the DLO for their integrity (e.g., shoe-lace knot) yield semi-tight crossing configurations. In such a configuration, a loop bounded by a curve segment joining two consecutive crossings with the same over/under status is called a *slip loop*. In Figure 5.8 there are two slip loops shown with striped interiors. To prevent a slip loop from being undone during manipulation, a mono-needle perpendicular to  $P$  is used, as shown in Figure 5.9. Pierced loops in semi-tight crossing configurations are handled with tri-needles as previously described.

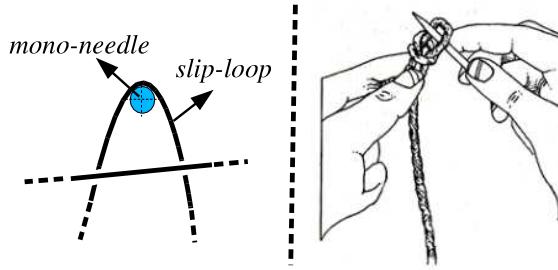


Figure 5.9: A needle, inspired from real-life (right figure), is used to preserve a slip loop.

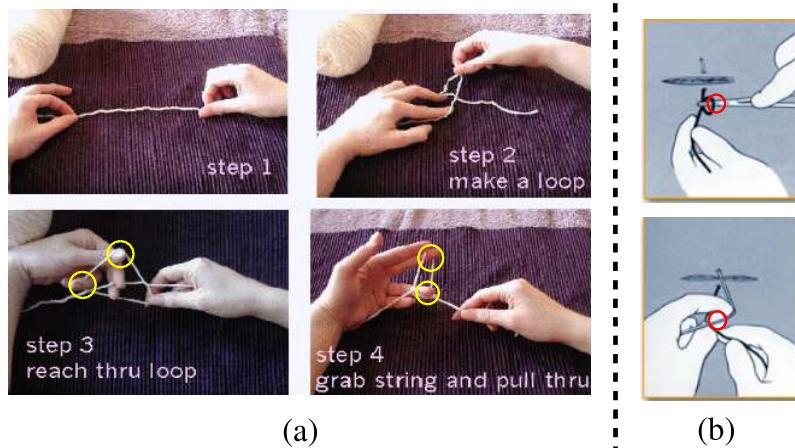


Figure 5.10: In real life, sliding supports (fingers in (a) and scissors in (b)) are commonly used while tying knots.

We assume that the goal crossing configuration  $x_{goal}$  is either tight or semi-tight. Once  $x_{goal}$  is achieved, all needles can be removed by translating them perpendicular to  $P$ .

The needles are structural supports along which the DLO can slide during manipulation. They are inspired from the way people use their extra fingers and tools during manipulation. While two fingers in one hand (usually, the thumb and the index) are used to grasp a DLO, other fingers are often used to maintain the integrity of loops (see Figure 5.10(a)). Tools such as scissors and needles may also be used as sliding supports (Figure 5.10(b)).

```

Algorithm TWO-ARM-KNOTTER ( $q_{init}, x_{goal}, P$ )
1.  $FS_g \leftarrow \text{Forming-Seq}(x_{goal}), H_g \leftarrow \text{Loop-Struct}(FS_g)$ 
2.  $T.\text{Insert}(\text{NULL}, q_{init})$ 
3. Loop  $Max$  times
   3.1. Pick  $q$  from  $T$  with a probability measure biased towards  $x_{goal}$ 
   3.2. Pick control vector  $u$  at random
   3.3.  $q_{new} \leftarrow f(q, u)$ 
   3.4. Return to Step 3 if:
        -  $f$  returned that the move is impossible, or
        - the forming sequence of  $q_{new}$  is not a sub-sequence of  $FS_g$ 
   3.5. Inspect  $H_g$  and add needles if needed
   3.6. Return to Step 3 if the arms cannot perform  $u$ 
   3.7.  $T.\text{Insert}(q, q_{new})$ 
   3.8. If  $q_{new} \in C(x_{goal})$ , then exit with manipulation path
4. Exit with failure

```

Figure 5.11: The DLO manipulation planning algorithm

#### 5.6.4 Motion planning algorithm

The algorithm is shown in Figure 5.11 (also see Figure 5.12). At Step 1 it computes the forming sequence  $FS_g$  and the loop structure  $H_g$  of  $x_{goal}$ . Then it constructs a tree-shaped single-query probabilistic roadmap  $T$  in the DLO’s configuration space. Our approach is similar to those proposed in [52] and [73] to plan trajectories under kinodynamic constraints. The roadmap  $T$  is a tree rooted at the initial configuration  $q_{init}$ . Each node of  $T$  is a sampled configuration of the DLO and each edge is a transition computed by the state-transition function  $f$  (see Section 5.4.2).  $T$  is built iteratively. At each iteration (Step 3), the algorithm selects a node  $q$  from  $T$  (Step 3.1) and a small move of the DLO’s head  $u$  (Step 3.2), and evaluates  $f(q, u)$  to obtain a new configuration  $q_{new}$ , which is then inserted in  $T$  as a successor of  $q$  (Step 3.7).

However, before  $q_{new}$  is added to  $T$ , the algorithm checks that a number of conditions are satisfied. Any violation of these conditions leads to start a new iteration at

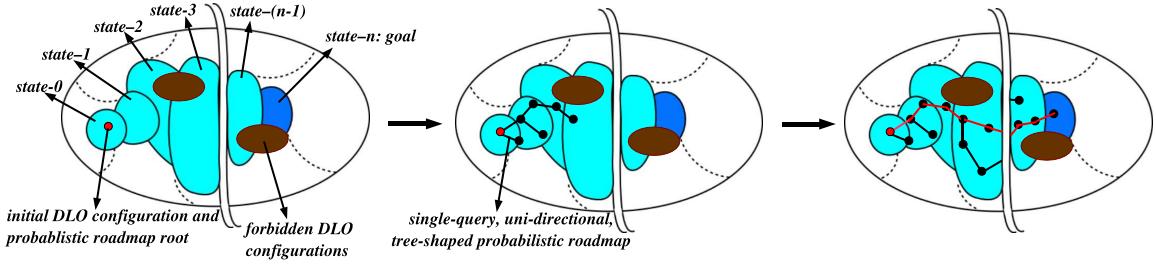


Figure 5.12: Our manipulation planning algorithm works by growing a tree-shaped probabilistic roadmap in the topologically-partitioned configuration space of the DLO. The tree is grown until it discovers the goal subspace where all configurations have the same topology characterized by the crossing configuration  $x_{goal}$ . The initial DLO configuration and the goal subspace is separated by a sequence of subspaces ( $state_1, \dots, state_n$ ). The subspace  $state_i$  consists of all DLO configurations whose crossing configuration is a subsequence of  $FS_g$  (forming seqence of  $x_{goal}$ ) formed by the first  $i$  crossings of  $FS_g$ .

Step 3. First,  $f(q, u)$  must have indicated that the control vector  $u$  is possible (see Section 5.4.2). If this is the case, the planning algorithm verifies that the forming sequence at  $q_{new}$  is a subsequence of  $FS_g$  beginning at the same crossing as  $FS_g$  (this causes the topological biasing of  $T$ ). If yes, it checks whether new needles are needed and adds them (they then become obstacles). Needles are placed when a slip or split loop is about to be formed, as indicated by  $H_g$ . They are placed along the DLO where the loop is expected to be formed (see Figure 5.13). However, the planner does not plan for the manipulation of the needles. It can be done with the help of additional manipulators and some movable fixtures for the needles. Throughout the planning, the crossing configuration of  $x_{new}$  is determined with respect to  $P$ . Next, the planner checks that the robot arm currently grasping the head of the DLO can track the motion of the DLO's head without colliding (using the arm's IK). If not, it checks whether the other arm can perform the motion instead, after a grasp switch between the two arms at  $q$ . The collision free motion of the arms for performing a grasp switch can be computed using any single-query probabilistic-roadmap planner [27]. In our implementation, we use the SBL planner [104]. The motion of the arms and the needle placements are stored along with  $q_{new}$  in  $T$ .

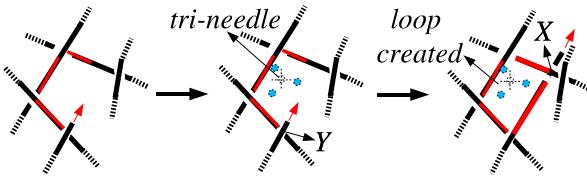


Figure 5.13: Suppose a split loop is created when the crossings  $Y$  followed by  $X$  are formed. A tri-needle is placed, just after  $Y$  is formed, along the DLO in the region inferred by matching the crossings associated with the current configuration of the DLO and the crossings in  $x_{goal}$  which define the loop.

The planner succeeds when it achieves a configuration  $q_{new} \in C(x_{goal})$ , which occurs when the number of crossings in  $q_{new}$  is equal to the number of crossings in  $x_{goal}$ . It returns a manipulation path, retrieved by backtracking from  $q_{new}$  to  $q_{init}$  in  $T$ , which consists of sequence of collision-free paths of the robots, separated by (re-)grasp operations, and the description of needle placements. The planner fails if it has not achieved a desired configuration after a specified number of iterations at Step 3.

At Step 3.1, the configuration  $q$  is selected at random among the nodes currently in  $T$ , with a probability measure that favors the nodes with more crossings, since they are topologically closer to  $C(x_{goal})$ . At Step 3.2, the control vector  $u$  is a small move of the DLO's head selected uniformly at random. Here, one can also bias this choice to favor the creation of the next crossing in  $FS_g$ .

## 5.7 Tying knots around static rigid objects

So far, we have focussed on achieving topological states of a DLO defined with respect to itself, i.e., tying self-knots. But in many applications, DLOs also knot around static objects. Here, we provide a simple technique to account for the topological interaction of a DLO with static objects for which curve representations exist. We say that for a given object  $B$ , its curve representation exists if there exists a curve  $r(B)$  such that any point in  $B$  is within  $\delta$  distance from  $r(B)$ , where  $\delta$  is small compared to the length of the curve. Examples of such objects are torus, cylinders, and long bars. In

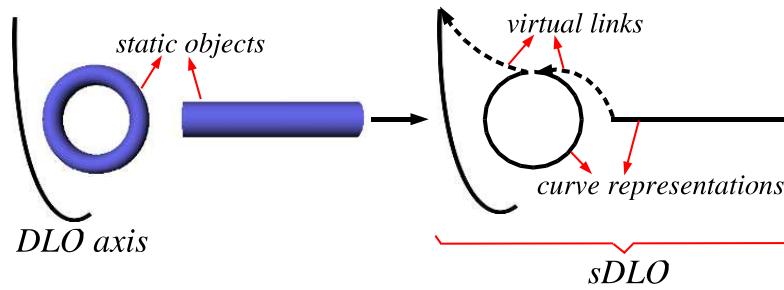


Figure 5.14: A composite semi-deformable linear object sDLO is constructed by chaining the curve representations of the rigid objects with the DLO axis. Thereafter, sDLO is used to account for the topological interactions of the DLO with rigid objects in the environment.

the case of a torus, the curve swept by the center of its generator conic and in the case of cylinder its central axis could be chosen as their curve representations. In general, curve representations can be obtained by skeletonizing the objects by the methods presented in [40] and pruning less prominent branches of the skeletal tree to obtain a curve.

We will account for the interaction of the DLO with rigid objects by defining a *composite* semi-deformable linear object (sDLO), and work with the crossing configuration of the sDLO thereafter. Let  $S$  be the set of curve representations of the concerned static objects. sDLO is created by “chaining” the curves in  $S$  and  $c$  sequentially,  $c$  being the last (see Figure 5.14). There will be virtual links connecting consecutive curves in the sequence. We will ignore the crossings between the curves in  $S$ , and also any associated with the virtual links.

Here we choose a reference projection plane such that the curves in  $S$  are minimally distorted after projection. This reduces the possibilities of crossings, between the projections of a curve in  $S$  and the DLO axis  $c$ , lumping into a small region. Any plane parallel to the one that minimizes the sum of squares of distances between the points of the curves in  $S$  and the plane could be used. However, it is difficult to choose a good plane in situations when a plane suited for one object is bad for another object, or when an object is not quasi-planar.

## 5.8 Experimental Results

We implemented the DLO manipulation planner TWO-ARM-KNOTTER in C++ and ran knot-tying experiments on a 1.5GHz Intel Xeon PC with 1GB RAM. In this implementation, we use the physical model of a DLO described in [111], which takes into account the essential mechanical properties of a typical DLO such as stretching, compressing, bending and twisting, as well as the effect of gravity. It manages self-collisions efficiently and also accounts for the interaction of the DLO with other static objects in the environment. We tuned the parameters of the rope model such that the simulated rope “visually” behaved like a typical rope, as one of its ends was being manipulated and the other end was kept fixed. Two robot arms, each with 6 degrees of freedom, are used to perform the manipulation. The videos of the results are available at: <http://ai.stanford.edu/~mitul/dlo>.

### 5.8.1 Results in simulation

Figure 5.15 shows sequences of snapshots from the manipulation motions generated by the planner for five manipulation problems. In the first four sequences, the goal is to tie common knots, respectively bowline, neck-tie, bow (shoe-lace), and stun-sail. In the fifth sequence the goal requires winding the DLO around static objects. Bowline and bow require one tri-needle each, while neck-tie and stun-sail require two tri-needles each. Bow also requires a mono-needle to maintain its slip loop. The fifth problem does not require any needle. The planner took between 10 to 15 minutes of CPU time to generate each of these motions.

### 5.8.2 Test on hardware platform

We tested manipulation plans generated by TWO-ARM-KNOTTER on a hardware platform made up of two PUMA-560 arms. Figure 5.16 shows several snapshots of the two arms tying a bowline knot (see the entire video at: <http://ai.stanford.edu/~mitul/dlo>). For this plan, a single tri-needle is needed as

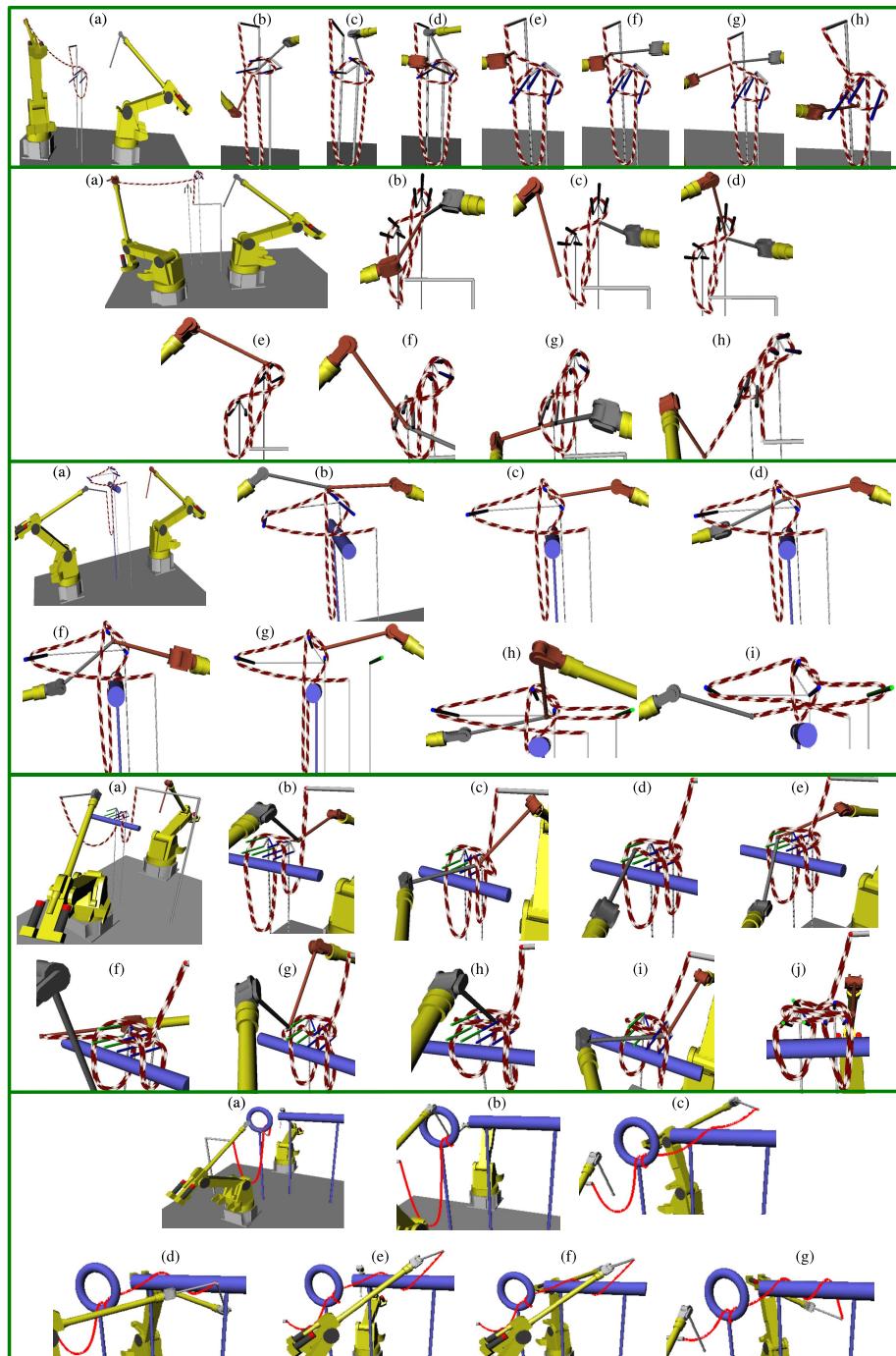


Figure 5.15: Sequences of snapshots along manipulation plans generated by TWO-ARM-KNOTTER for five manipulation problems.

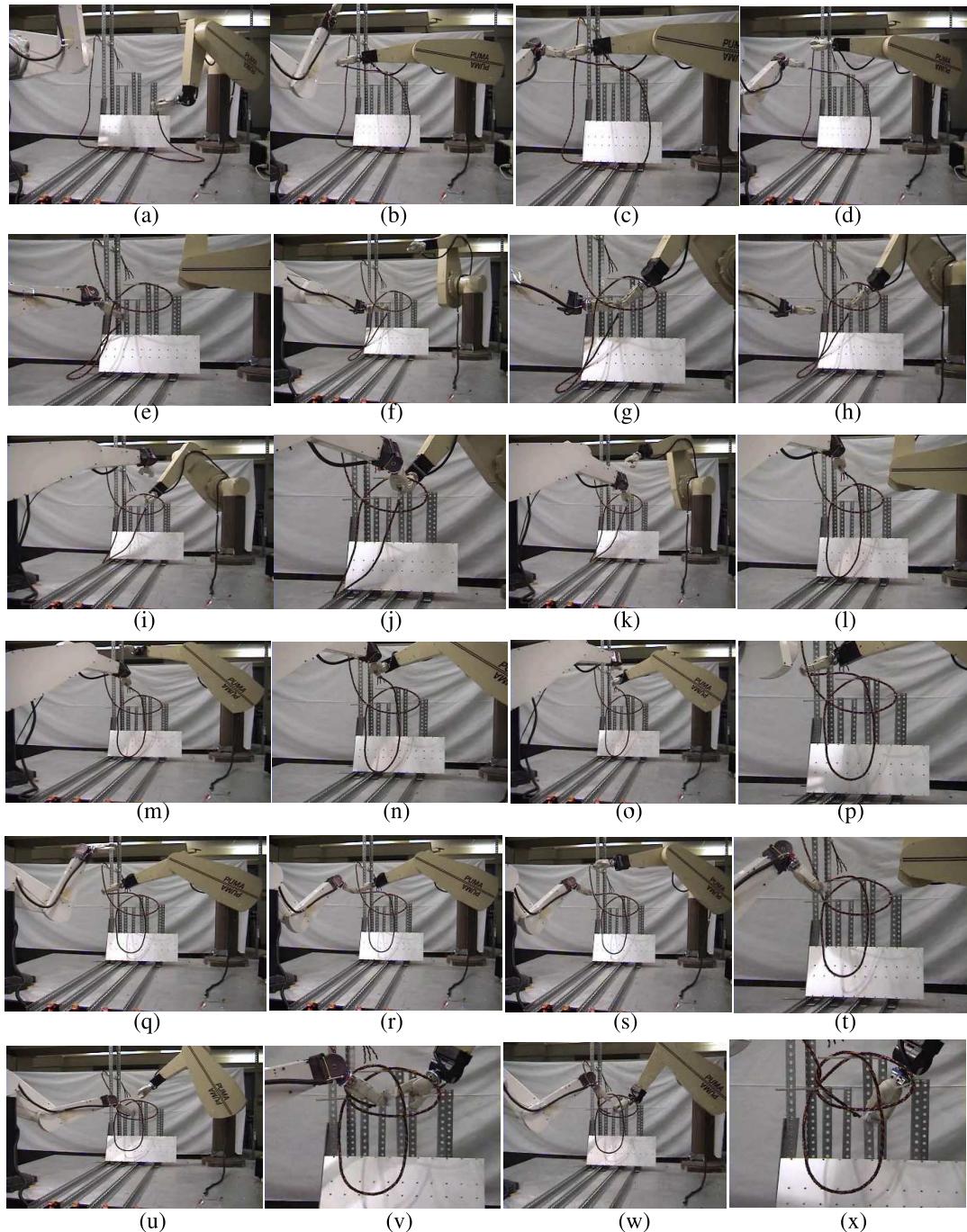
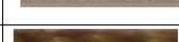
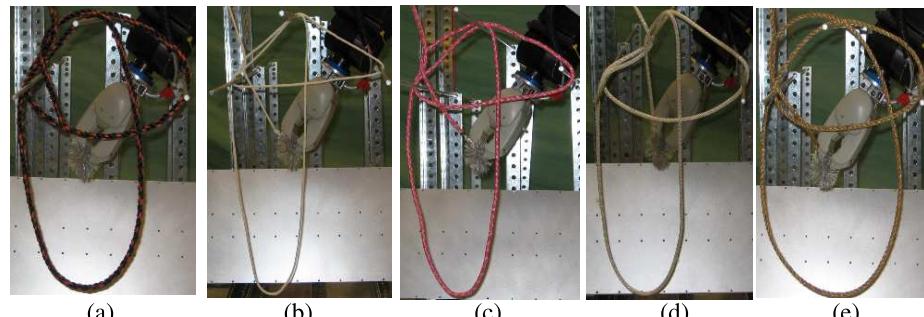


Figure 5.16: Two PUMA-560 arms tying a bowline knot

	Rope	Material	Thickness(mm)
(a)		polypropylene	9.66
(b)		cotton	3.44
(c)		polyester	6.48
(d)		nylon	5.66
(e)		jute	5.94
(f)		plastic	4.55

(a)



(a)

(b)

(c)

(d)

(e)

(b)

Figure 5.17: (a): Different types of ropes used. (b): Final shapes of bowline knots achieved with different ropes.

soon as the execution of the plan starts (see snapshot (a) of the first example in Figure 5.15). This needle, which is made of three thin aluminium rods, was hand-placed at the position determined by the planner. We executed the same plan computed by TWO-ARM-KNOTTER using six types of ropes with different materials and diameters (Figure 5.17(a)). The rope model used by the planner did not exactly represent the physics of any of these ropes. However, the execution of the plan succeeded consistently on the first five ropes, but failed with the plastic rope, which turned out to be too stiff for the motion plan. Figure 5.17(b) shows a final bowline shape obtained with each of the first five ropes.

The ability of our planner to generate robust plans relies critically on the needles. We consider them as key tools for manipulating deformable objects. However, both sensory feedback and re-planning could still be used to further improve robustness, at least to detect when the physical model used for planning is too far off for a plan to succeed.

### 5.8.3 Sensitivity analysis

In the above experiment, we were actually surprised that the same manipulation plan worked well with quite different ropes. This led us to quantify the sensitivity of plans generated by TWO-ARM-KNOTTER to inaccuracies in the rope model. Since rope manufacturers do not usually provide numerical values for the mechanical properties of ropes, we proceeded in the following manner. After generating a manipulation plan for tying a bowline using a particular rope model, we tested in simulation if the same manipulation plan still achieves a bowline after corrupting the rope model parameters with Gaussian noises. Table 5.1 lists the means and standard deviations (estimated numerically) for the Gaussian distributions from which three main parameters  $\alpha$ ,  $\beta$ , and  $\gamma$  of the model<sup>1</sup> were independently chosen, such that a bowline was achieved more than 90% of the time. The mean values correspond to the original model parameter values, used to generate the manipulation plan. The high values of the standard deviations indicate high degree of robustness.

---

<sup>1</sup>In [111],  $\alpha$ ,  $\beta$ , and  $\gamma$  correspond to stretching, bending, and twisting constants of the spring primitives, respectively.

model parameter	mean	standard deviation
$\alpha$	13500	3950
$\beta$	1250	455
$\gamma$	25	15

Table 5.1: Means and standard deviations of Gaussian distributions from which the three main rope model parameters were chosen for the robustness analysis

## 5.9 Conclusion

The main contribution of this chapter is a new PRM motion planner for manipulating deformable linear objects (DLOs) using cooperating dual robot arms to tie self-knots and knots around simple static objects. We have blended new ideas with pre-existing concepts and techniques from knot theory, robot motion planning, and computational physical modeling of DLOs. The ability of robots to autonomously manipulate DLOs is desirable in application domains such as medical surgery (suturing), manufacturing (loading cable harnesses and robot dresses), and service-based robotics (laying out household cables).

Unlike a typical PRM planner, our new DLO manipulation planner achieves topological goals. This is important in DLO manipulation, because it is usually more important to achieve an acceptable topological state of the DLO rather than an exact geometry, especially when tying knots or winding loops around objects. Our planner does not assume a specific physical model of the DLO. To our knowledge, it is the first planner of its kind, i.e., we are not aware of any other planner that can generate collision-free motions for robot arms for manipulating a DLO in an environment with obstacles.

We have demonstrated the effectiveness of our planner by tying the popular Bowline knot with various household ropes on a hardware platform with two PUMA robots, using a manipulation plan generated by the planner. Additionally in simulation, the planner was tested to achieve popular knots like bowline, neck-tie, bow (shoe-lace), and stun-sail.

# Chapter 6

## Conclusion

Probabilistic roadmaps have emerged as a powerful approach for solving complex motion planning problems with many degrees of freedoms, involving complex geometry and various types of motion constraints such as collision avoidance, non-holonomic, kinodynamic, stability, visibility, and contact. Current PRM-based research focuses on challenges that arise as PRMs are being applied to motion planning problems in various scenarios. These challenges are of two types: basic (*i.e.*, independent of any specific problem or application – for instance the narrow passage issue) and application-specific (for instance, dealing with deformable objects). In response to some of these challenges, our thesis makes the following contributions:

- **Dynamic Collision Checking:** We proposed a new dynamic collision checker to test path segments in configuration space or collections of such segments, including continuous multi-segment paths, in response to the basic need of efficient collision checking in PRM planning. The main advantage of our new checker over the commonly used fixed-resolution approach is to never miss a collision. In addition, it eliminates the need for experimentally determining a suitable value of the configuration space resolution parameter and its running time compares favorably to that of a fixed-resolution checker.
- **Dealing with Narrow Passages:** We proposed a new retraction method – called small-step retraction – designed to help PRM planners sample paths

through narrow passages. The core idea underlying this method is that retracting sampled configurations that are barely colliding is sufficient to speed up the generation of roadmap milestones within narrow passages. Barely colliding configurations are also relatively easy to retract out of collision. In addition, they can be efficiently identified by running a classical collision checker on thinned geometric models pre-computed using an MA-based technique.

- **Multi-Goal Planning:** We proposed a 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.
- **Manipulation Planning for Deformable Linear Objects:** We proposed a new PRM-based topological motion planner for manipulating deformable linear objects (DLOs) using cooperating robot arms to tie self-knots and knots around simple static objects. Some examples of DLOs are ropes, cables, and surgical sutures. Unlike previous PRM planners, this planner takes a topological state of the world (the crossings of the DLO with itself and with static obstacles) as the goal to achieve. During planning it communicates with the input physical model of the DLO to construct a probabilistic roadmap. To our knowledge, our planner is a first of its kind, i.e., we are not aware of any other planner that can generate collision-free motions for robot arms which lead to DLO manipulation in environments with obstacles. We have demonstrated the effectiveness of our planner by tying some commonly used knots both in simulation and in real-life on a two-PUMA robot hardware platform.

In the future, additional research can still be done to improve on these contributions. For example:

- (a) Our dynamic collision checker (Chapter 2) samples a path in configuration space at an adaptive resolution determined by relating motion in configuration space to motion in workspace. A similar idea could be used to determine how much to thin objects in the workspace in order to achieve a desired amount of widening of narrow passages in configuration space. This could lead to improving our small-step retraction method (Chapter 3).
- (b) The multi-goal planning problem addressed in Chapter 4 could be generalized even further. For instance, in some applications, a partial ordering is imposed on the input goals. In those cases, how to efficiently incorporate this ordering into the multi-goal planner? When the robot kinematics is redundant, there may be infinitely many inverse kinematics solutions. In our experiments, we pre-sampled some of those solutions. 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. It would also be interesting to investigate the multi-goal problem for multi-robot systems, when each goal may be reached by several robots.
- (c) Our DLO manipulation planner (Chapter 5) assumes that the DLO is being manipulated through a single grasp point located at one of its end. But in practice many knots, for instance the shoelace knot, are tied by grasping a DLO at locations other than the end points. In the future, we would like to generalize the DLO manipulation planner so that it can also accept grasp points at locations other than

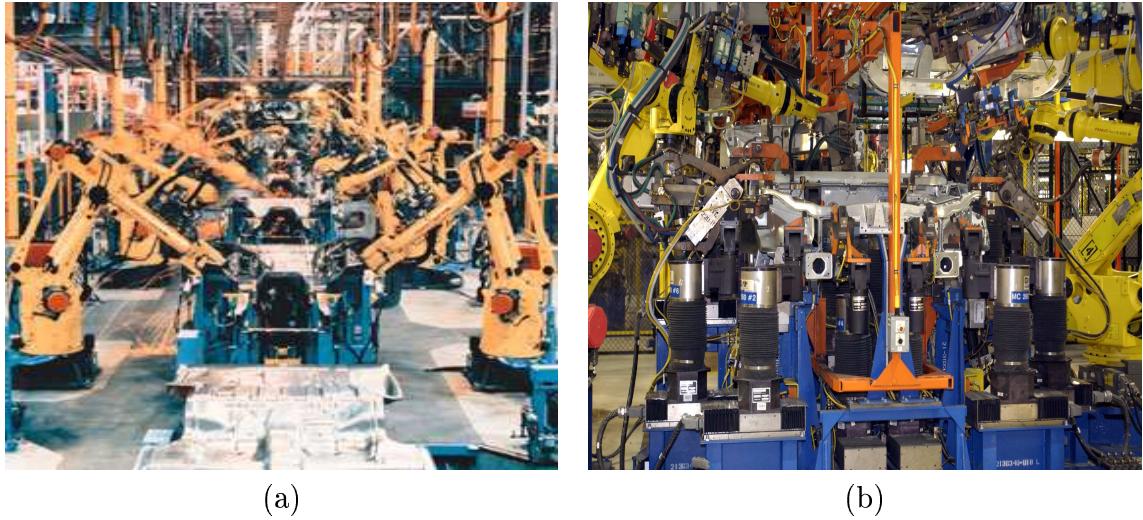


Figure 6.1: (a): Large number of robots involved in a manufacturing cell. (b): Congested workcell (courtesy of General Motors)

the end points. Such a generalization would also make the overall manipulation strategy and execution more complicated.

We believe that today we are at a point where one can seriously think of integrating design and motion planning. Figure 6.1 shows photographs of robotized automotive manufacturing cells. Designing such workcells is a tremendous challenge: (1) How many robots are needed? (2) Where should they be placed? (3) How to maximize cooperation between robots? (4) Is there any waste of space? The designer simulates the workcell using computational and graphic visualization tools. Usually he/she has to hand-design collision-free robot trajectories. An integrated motion planner would automatically compute the trajectories. But its most appealing task would be to prompt the designer when the targets are no longer reachable, after some robots have been moved, added, or removed by the designer in his simulation. This use of a planner would be similar in concept to that of an integrated spell-checker that detects spelling errors on the fly while a text is being edited. Along the same line, the integrated motion planner would continuously notify the designer of consequences as he/she is changing the design parameters of the workcell.

## Appendix A

### Overview of SBL Planner [102]

SBL (which stands for Single-query Bi-directional Lazy-collision-checking) builds a probabilistic roadmap made of two trees ( $T_s$  and  $T_g$ ) of milestones, each rooted at one of the two query configurations ( $s$  and  $g$ ), until a collision-free local path connects a milestone in one tree and a milestone in the other tree. Its efficient lazy collision-checking strategy postpones collision tests along the edges of the roadmaps. The overall algorithm is the following:

**Algorithm**  $\text{SBL}(s, g)$

1. Install  $s$  and  $g$  as the roots of  $T_s$  and  $T_g$ , respectively
2. Repeat  $K$  times
  - 2.1. EXPAND
  - 2.2.  $\tau \leftarrow \text{CONNECT}$
  - 2.3. If  $\tau \neq \text{nil}$  then return  $\tau$
3. Return *failure*

Each loop of Step 2 performs two operations: EXPAND adds a milestone to one of the two trees, while CONNECT tries to connect the two trees. SBL returns failure if it has not found a solution path after  $K$  iterations at Step 2.

Each extension of the roadmap is done as follows:

**Algorithm EXPAND**

1. Pick  $T$  to be either  $T_s$ , or  $T_g$ , each with probability  $1/2$
2. Repeat
  - 2.1. Pick a milestone  $m$  from  $T$  at random,  
with probability  $\pi \sim 1/\eta(m)$
  - 2.2. For  $i = 1, 2, \dots$ 
    - 2.2.1. Pick a new configuration  $\mathbf{q}$  uniformly at random  
from  $B(m, \rho/i)$ .
    - 2.2.2. If  $\mathbf{q}$  is collision-free then install it as a child milestone  
of  $m$  in  $T$  and return

The algorithm first selects the tree  $T$  to expand. The alternation between the two trees prevents one tree from eventually growing much bigger than the other. The number  $\eta(m)$  associated with each milestone  $m$  in  $T$  measures the current density of milestones of  $T$  around  $m$ . A milestone  $m$  is picked from  $T$  with probability proportional to  $1/\eta(m)$  and a collision-free configuration  $\mathbf{q}$  is picked at close distance (less than some pre-specified  $\rho$ ) from  $m$ .  $\mathbf{q}$  is stored in  $T$  as a new milestone. The probability  $\pi(m) \sim 1/\eta(m)$  at Step 2.1 guarantees that the distribution of milestones eventually diffuses through the component(s) of free space reachable from  $s$  and  $g$ .

Step 2.2 selects a series of milestone candidates, at random, from increasingly smaller neighborhoods of  $m$ , starting with a neighborhood of radius  $\rho$ . When a candidate  $\mathbf{q}$  tests collision-free, it is retained as the new milestone  $m'$ . The segment from  $m$  to  $m'$  is not checked here for collision. On the average, the distance from  $m$  to  $m'$  is greater in wide-open regions of free space than in narrow regions.

The connection between the two trees is performed by algorithm CONNECT:

**Algorithm** CONNECT

1.  $m \leftarrow$  most recently created milestone
2.  $m' \leftarrow$  closest milestone to  $m$  in the tree  
not containing  $m$
3. If  $d(m, m') < \rho$  then
  - 3.1. Connect  $m$  and  $m'$  by a bridge  $w$
  - 3.2.  $\tau \leftarrow$  path connecting  $s$  and  $g$
  - 3.3. Return  $\text{TEST-PATH}(\tau)$
4. Return  $nil$

Let  $m$  now denote the milestone that was just added by EXPAND and  $m'$  be the closest milestone to  $m$  in the other tree. If  $m$  and  $m'$  are less than  $\rho$  apart, then CONNECT joins them by a segment, called a bridge. The bridge creates a path  $\tau$  joining  $s$  and  $g$  in the roadmap. The segments along  $\tau$ , including the bridge, are now tested for collision. TEST-PATH returns  $\tau$  if it does not detect any collision and  $nil$  otherwise. In the latter case, the segment found to collide is removed, which separates the roadmap into two trees (in this process, some edges may shift from one tree to the other).

# Bibliography

- [1] E. U. Acar, H. Choset, A. A. Rizzi, P. Atkar, and D. Hull. Morse decompositions for coverage tasks. *Int. J. of Robotics Research*, 21:331–344, 2002.
- [2] C. C. Adams. *The Knot Book*. W.H. Freeman and Company, New York, NY, 1994.
- [3] 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 *Proc. 11th Int. Symp. on Robotics Research*, pages 80–89. Springer, STAR 15, 2005.
- [4] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. Obprm: An obstacle-based prm for 3d workspace. In P.K. Agarwal and et al., editors, *Robotics: The Algorithmic Perspective*, pages 155–168. A K Peters, Natick, MA, 1998.
- [5] N.M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 113–120, Minneapolis, MN, 1996.
- [6] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. 6th ACM Symp. on Solid Modeling and Applications*, pages 249–260, 2001.
- [7] M. Apaydin, D. Brutlag, C. Guesttin, D. Hsu, J. Latombe, and C. Varma. Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion. *J. Computational Biology*, 10(3-4):257–281, 2003.

- [8] B. Baginski. Efficient dynamic collision detection using expanded geometry models. In *Proc. IEEE/RSJ/GI Int. Conf. on Intelligent Robots and Systems*, pages 1714–1719, 1997.
- [9] B. Baginski. Local motion planning for manipulators based on shrinking and growing geometry models. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3303–3308, Minneapolis, MN, 1997.
- [10] J. Barraquand, L. Kavraki, J. C. Latombe, T.Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for path planning. *Int. J. of Robotics Research*, 16(6):759–774, 1996.
- [11] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 1997.
- [12] H. Blum. A transformation for extracting new descriptors of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 326–380. MIT Press, Cambridge, MA, 1967.
- [13] J.E. Bobrow. Optimal robot path planning using the minimum-time criterion. *IEEE J. Robot. Automat.*, 4(4):443–450, August 1988.
- [14] R. Bohlin and L. Kavraki. Path planning using lazy PRM. In *Proc. Int. Conf. on Robotics and Automation*, pages 521–528, 2000.
- [15] M. Bonert, L.H. Shu, and B. Benhabib. Motion planning for multi-robot assembly systems. In *Proc. ASME Design Engineering Technical Conferences*, Las Vegas, NV, Sept. 1999.
- [16] V. Boor, M.H. Overmars, and A.F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1018–1023, Detroit, MI, 1999.
- [17] T. Bretl. Motion planning of multi-limbed robots subject to equilibrium constraints: The free-climbing robot problem. *Int. J. of Robotics Research*, 25(4):317–342, April 2006.

- [18] J. Brown, J.C. Latombe, and K. Montgomery. Real-time knot tying simulation. *The Visual Computer J.*, 20(2-3):165–179, 2004.
- [19] S. Cameron. A study of the clash detection problem in robotics. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 488–493, March 1985.
- [20] S. A. Cameron. Collision detection by four-dimensional intersection testing. *IEEE J. Robot. Automat.*, 6:291–302, June 1990.
- [21] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. Pattern Anal. Machine Intell.*, 8(2):200–209, March 1986.
- [22] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.
- [23] J.F. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proc. IEEE Conf. on Foundations of Computer Science*, pages 39–48, 1987.
- [24] 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.
- [25] Y.C. Chang. Near parallel computation of mat of 3d polyhedra. Ph.D. thesis, Mechanical Engineering Dept., Stanford University, Stanford, CA, 2006.
- [26] C. Chekuri, G. Even, and G. Kortsarz. A lazy approximation algorithm for the group steiner problem. *To appear in Discrete Applied Mathematics*, 2005.
- [27] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion*. MIT Press, Cambridge, MA, 2005.
- [28] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proc. Sym. on Interactive 3D Graphics*, pages 189–196, 1995.

- [29] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
- [30] L. Dale, G. Song, and N. Amato. Faster, more effective connection for probabilistic roadmaps. Technical Report TR00-005, Department of Computer Science, Texas A&M University, 20, 2000.
- [31] T. Danner and L.E. Kavraki. Randomized planning for short inspection paths. In *Proc. IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, April 2000.
- [32] T. K. Dey and W. Zhao. Approximate medial axis as a voronoi subcomplex. In *Proc. 7th ACM Symp. on Solid Modeling and Applications*, pages 356–366, 2002.
- [33] A. Dhanik. Development of a palpable virtual nylon thread and handling of bifurcations. Masters thesis, NUS, Singapore, 2005.
- [34] S. A. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Proc. Eurographics*, volume 20(3), pages 500–510. 2001.
- [35] 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.
- [36] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proc. ACM Symp. on Theory of Computing*, pages 448–455, 2003.
- [37] A. Foisy and V. Hayward. A safe swept volume method for collision detection. In *Proc. The Sixth Int. Symp. of Robotics Research*, pages 61–68, Pittsburgh (PE), Oct. 1993.

- [38] M. Foskey, M. Lin, and D. Manocha. Efficient computation of a simplified medial axis. In *Proc. ACM Symposium on Solid Modeling and Applications*, 2003.
- [39] 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.
- [40] N. Gagvani and D. Silver. Parametric controlled volume thinning”. *Graphics Models and Image Processing*, 61(3):149–164, May 1999.
- [41] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *J. of Algorithms*, 37:66–84, 2000.
- [42] S. Gottschalk, M. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. In *Proc. ACM SIGGRAPH*, pages 171–180, 1996.
- [43] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30:171–180, 1996.
- [44] L. Guibas, C. Holleman, and L. Kavraki. A probabilistic roadmap planner for flexible objects with a workspace medial-axis based sampling approach. In *Proc. IEEE Int. Conf. on Intelligent Robots and Systems*, 1999.
- [45] E. Halperin and R. Krauthgamer. Polylogarithmic inapproximability. In *Proc. ACM Symp. on Theory of Computing*, pages 585–594, 2003.
- [46] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proc. SIGGRAPH*, pages 277–286, 1999.
- [47] C. Hoffman. How to construct the skeleton of csg objects. In A. Bowyer, editor, *Computer-Aided Surface Geometry and Design*, pages 421–437. Oxford University Press, 1994.
- [48] C. Holleman and L. Kavraki. A framework for using the workspace medial axis in prm planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1408–1413, San Francisco, CA, April 2000.

- [49] D. Hsu. Randomized single-query motion planning in expansive space. Ph.D. thesis, Computer Science Dept., Stanford University, Stanford, CA, May 2000.
- [50] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 4420–4426, 2003.
- [51] D. Hsu, L. Kavraki, J.C. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. In P.K. Agarwal and et al., editors, *Robotics: The Algorithmic Perspective*, pages 151–153. A K Peters, Natick, MA, 1998.
- [52] D. Hsu, R. Kindel, J.C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *Int. J. of Robotics Research*, 21(3):233–255, March 2002.
- [53] D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. of Computational Geometry and Applications*, 9(4-5):495–512, 1999.
- [54] D. Hsu, J.C. Latombe, and H. Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. In *Proc. Int. Symp. on Robotics Research*, San Francisco, CA, October 2005.
- [55] D. Hsu, J.C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2719–2726, 1997.
- [56] D. Hsu, J.C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. of Computational Geometry and Applications*, 9(4-5):495–512, 1999.
- [57] D. Hsu, G. Sanchez-Ante, and Z. Sun. Hybrid prm sampling with a cost-sensitive adaptive strategy. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3885–3891, 2005.

- [58] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. on Graphics*, 15(3):179–210, 1996.
- [59] K.E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proc. IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, April 2000.
- [60] P. Isto. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2323–2328, 2002.
- [61] X. Ji and J. Xiao. On random sampling in contact configuration space. In *Proc. Workshop on Algorithmic Foundations of Robotics*, March 2000.
- [62] P. Jiménez, F. Thomas, and C. Torras. 3D collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001.
- [63] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Automat. Contr.*, 12(4):566–580, 1996.
- [64] L.E. Kavraki, M. Kolountzakis, and J.C. Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE J. Robot. Automat.*, 14(1):166–171, February 1998.
- [65] L.E. Kavraki, J.C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot motion planning. *Journal of Computer and System Sciences*, 57(1):50–60, August 1998.
- [66] L.E. Kavraki, P.Svestka, J.C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE J. Robot. Automat.*, 12(4):566–580, 1996.
- [67] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. of Robotics Research*, 5(1):90–98, 1986.

- [68] J. T. Klosowski, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Visual. Comput. Graphics*, 4(1):21–36, 1998.
- [69] D.E. Koditscheck and E. Rimon. *Advances Appl. Math.*, 11:412–442, 1990.
- [70] Y. Koga and J. C. Latombe. On multi-arm manipulation planning. In *Proc. IEEE Int. Conf. Robotics and Automation*, San Diego, CA, 1994.
- [71] J.J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 995–1001, 2000.
- [72] A. M. Ladd and L. E. Kavraki. Using motion planning for knot untangling. *Int. J. Robotics Research*, 23(7-8):797–808, 2004.
- [73] F. Lamiraux and L. E. Kavraki. Planning paths for elastic objects under manipulation constraints. *Int. J. Robotics Research*, 20(3):188–208, 2001.
- [74] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast distance queries with rectangular swept sphere volumes. In *Proc. IEEE Conf. on Robotics and Automation*, 2000.
- [75] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Press, 1991.
- [76] J.C. Latombe. Probabilistic roadmaps. In *lecture notes, CS26N ("Motion Planning for Robots, Digital Actors, and Other Moving Objects")*, Stanford University, Stanford, CA, Spring 2005-06.
- [77] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. *Int. J. of Robotics Research*, 20(5):278–300, May 2001.
- [78] T.Y. Li and J.S. Chen. Incremental 3D collision detection with hierarchical data structures. In *Proc. ACM Symp. on Virtual Reality Software and Technology*, pages 139–144, Taipei, Taiwan, 1998.

- [79] J. M. Lien, S. L. Thomas, and N. M. Amato. A general framework for sampling on the medial axis of the free space. 2003.
- [80] M. Lin and D. Manocha. Collision and proximity queries. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, 2nd edition*, chapter 35, pages 787–807. Chapman & Hall/CRC, 2004.
- [81] M. Lin, D. Manocha, J. Cohen, and S. Gottschalk. Collision detection: Algorithms and applications. In J. P. Laumond and M. Overmars, editors, *Proc. Algorithms for Robotic Motion and Manipulation*, pages 129–142. A. K. Peters, 1996.
- [82] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *IEEE Int. Conf. on Robotics and Automation*, pages 1008–1014, 1991.
- [83] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In *Proc. IMA Conf. on Mathematics of Surfaces*, volume 1, pages 602–608, San Diego (CA), 1998.
- [84] T. Lozano-Pérez and M.A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, 1979.
- [85] T. Matsuno, T. Fukuda, and F. Arai. Flexible rope manipulation by dual manipulator system using vision sensor. In *Proc. IEEE/ASME Int. Conf. Advanced Intelligent Mechatronics*, Como, Italy, 2001.
- [86] B. Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Trans. on Graphics*, 17(3):177–208, July 1998.
- [87] J.S.B. Mitchell. Shortest paths and networks. *Discrete and Computational Geometry, J.E. Goodman and J. O'Rourke (eds.)*, CRC Press, Boca Raton, FL, pages 445–466, 1997.
- [88] T. Morita, J. Takamatsu, K. Ogawara, H. Kimura, and K. Ikeuchi. Knot planning from observation. In *Proc. IEEE Int. Conf. Robotics and Automation*, Taipei, Taiwan, 2003.

- [89] Ch. Nielsen and L. E. Kavraki. A two-level fuzzy PRM for manipulation planning. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Japan, 2000.
- [90] H.H. Gonzalez-Banos and J.C. Latombe. A randomized art-gallery algorithm for sensor placement. In *Proc. ACM Symp. on Computational Geometry*, pages 232–240, 2000.
- [91] C. O'Dúnlaing and C.K. Yap. A retraction method for planning the motion of a disc. *J. of Algorithms*, 6:104–111, 1982.
- [92] D. K. Pai. Strands: interactive simulation of thin solids using cosserat models. In *Proc. Eurographics*, Saarbruken, Germany, 2002.
- [93] C. Papadimitriou. The euclidean traveling salesman problem is np-complete. *Theoretical Computer Science*, 4:237–244, 1977.
- [94] J. Phillips, A. Ladd, and L. E. Kavraki. Simulated knot tying. In *Proc. IEEE Int. Conf. Robotics and Automation*, Washington, DC, 2002.
- [95] S. Quinlan. Efficient distance computation between non-convex objects. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3324–3329, 1994.
- [96] S. Quinlan. Proc. ieee int. conf. on robotics and automation. In *Efficient Distance Computation Between Non-Convex Objects*, pages 3324–3329, 1994.
- [97] S. Redon, A. Kheddar, and S. Coquillart. An algebraic solution to the problem of collision detection for rigid polyhedral objects. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3733–3738, San Francisco (CA), 2000.
- [98] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum*, 21(3), 2002.
- [99] G. Reich and P. Widmayer. Beyond steiner's problem: A vlsi oriented generalization. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS*, volume 411, pages 196–210, 1990.

- [100] A. Remde, D. Henrich, and H. Worn. Pick-up deformable linear objects with industrial robots. In *Proc. Int. Symp. on Robotics*, Japan, 1999.
- [101] M. Saha, G. Sánchez, and J.C. Latombe. Planning multi-goal tours for robot arms. In *Proc. IEEE Int. Conf. on Robotics and Automation*, Taipei, Taiwan, 2003.
- [102] G. Sánchez and J.C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proc. Int. Symp. on Robotics Research*, Lorne, Victoria, Australia, 2001.
- [103] 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.
- [104] G. Sanchez-Ante and J. C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. *Int. J. of Robotics Research*, 2002.
- [105] A. Schweikard. Polynomial time collision detection for manipulator paths specified by joint motions. *IEEE Trans. Automat. Contr.*, 7(6):865–870, 1991.
- [106] T. Simeon, J. P. Laumond, J. Cortes, and A. Sahbani. Manipulation planning with probabilistic roadmaps. *Int. J. Robotics Research*, 23(7-8):729–746, 2004.
- [107] S.N. Spitz and A.A.G. Requicha. Multiple-goals path planning for coordinate measuring machines. In *Proc. IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, 2000.
- [108] G. Van der Bergen. Efficient collision detection of complex deformable models using AABB trees. *J. of Graphic Tools*, 2(4):1–13, 1997.
- [109] H. Wakamatsu and S. Hirai. Static modeling of linear object deformable based on differential geometry. *Int. J. Robotics Research*, 23(3):293–311, 2004.
- [110] H. Wakamatsu, A. Tsumaya, Eiji Arai, and Shinichi Hirai. Planning of one-handed knotting/raveling manipulation of linear objects. In *Proc. IEEE Int. Conf. Robotics and Automation*, LA, 2004.

- [111] F. Wang, E. Burdet, V. Ronald, and H. Bleuler. Knot-tying with visual and force feedback for vr laparoscopic training. In *Proc. 27th IEEE EMBS Annual Int. Conf.*, Shanghai, China, 2005.
- [112] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, LNCS 555, pages 359–370. Springer Verlag, 1991.
- [113] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proc. ACM Sym. on Computational Geometry*, Urbana-Champaign, IL, 1988.
- [114] S.A. Wilmarth, N.M. Amato, and P.F. Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. proc. In *IEEE Int. Conf. on Robotics and Automation*, pages 1024–1031, Detroit, MI, 1999.
- [115] C. Wurll, D. Henrich, and H. Wörn. Multi-goal path planning for industrial robots. In *Proc. Int. Conf. on Robotics and Applications*, Santa Barbara, CA, 1999.
- [116] K. Yamane, J.J. Kuffner, and J.K. Hodgins. Synthesizing animations of human manipulation tasks. *ACM Trans. on Graphics*, 23(3):532–539, 2004.