Thesis for the Degree of Licentiate of Engineering

Motion Planning for Industrial Robots

Robert Bohlin

Department of Mathematics
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Göteborg, November 1999

Motion Planning for Industrial Robots
ROBERT BOHLIN

ISSN 0347-2809/NO 1999-58
Department of Mathematics
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

This is a thesis of the ECMI (European Consortium for Mathematics in Industry) post-graduate program in Industrial Mathematics at Chalmers University of Technology.

The work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development, project P10499-1.

Matematiskt centrum
Göteborg, November 1999

Abstract

Autonomous motion planning addresses the problem of finding collision-free paths for moving objects – robots – among obstacles. In this report we consider robots operating in workspaces occupied by stationary, completely known obstacles. We describe a new approach to probabilistic roadmap planners (PRMs). The overall theme of the algorithm, called Lazy PRM, is to minimise the number of collision checks performed during planning. Our algorithm builds a roadmap in the configuration space, whose nodes are the user-defined initial and goal configurations and a number of randomly generated nodes. Neighbouring nodes are connected by edges representing the straight line path between the nodes. In contrast with PRMs, our planner initially assumes that all nodes and edges in the roadmap are collision-free, and searches the roadmap at hand for a shortest path between the initial and the goal node. The nodes and edges along the path are then checked for collision. If a collision with the obstacles occurs, the corresponding nodes and edges are removed from the roadmap. Our planner either finds a new shortest path, or first updates the roadmap with new nodes and edges, and then searches for a shortest path. The above process is repeated until a collision-free path is returned.

Lazy PRM is tailored to efficiently answer single planning queries, but can also be used for multiple queries. Experimental results presented in this report show that our lazy method is very efficient in practice.

Keywords: Motion planning, path planning, collision avoidance, probabilistic roadmap.

AMS 1991 subject classification: 05C80, 05C85 (68P10)

Acknowledgements

First, I would like to thank my supervisor Associate Professor Bo Johanson at the Department of Mathematics at Chalmers University of Technology for initiating the project and for his support and enthusiasm throughout the work. I also express my gratitude to the Project Group consisting of Lars Östlund (chairman), Pernilla Johansson and Pär Oscarsson at ABB Digital Plant Technologies AB (DPT) and Bo Johansson. Thanks also to Pernilla Johansson and Anders Ekelund at DPT for providing suitable software and for many interesting discussions.

Parts of this work was performed together with Associate Professor Lydia Kavraki at the Department of Computer Science, Rice University during my visit to the Physical Computing Group at Rice University. Her encouragement, comments on various manuscripts, and experience in the field of robotics have been invaluable to the project and this report.

I would also like to thank the Robotics Group at Oxford University Computing Laboratory, England, with whom I had the opportunity to spend three months. Particularly, I give my gratitude to Stephen Cameron and Joe Pitt-Francis for many fruitful discussions.

Finally, I wish to thank my fiancée Petra for all her support, love and understanding.

Contents

Chapter 1

Introduction and Motivation

In the last few decades, the number of robots has grown dramatically in a wide range of areas. Upon industrial applications these versatile machines are used in surgery, in agriculture, in space, under water, and for transportation. Although most methods and techniques discussed in this report are applicable to many kinds of robots, e.g. mobile robots and free-flying rigid objects, we will focus on industrial manipulators.

In industrial applications, robots are used for different tasks like welding, spray-painting, pick and place operations, and assembly tasks. Most robots are used in order to increase accuracy, reliability, speed, or to prevent humans from working in hazardous environments.

Relying on the experience and knowledge of the operators, these robots are still, to a large extent, programmed *on-line*, by more or less teaching them how to move. By using, for example, a joystick, the operator maneuvres the robot to different positions and stores a sequence of configurations into a program. When the program is executed, the robot moves between successive configurations in the simplest possible way, without considering the environment. The programming generally takes a considerable amount of time, and meanwhile the robot and its work cell are occupied, which may force an entire production line to stop.

To meet demands on flexibility, quality, and efficiency in modern manufacturing systems, *off-line* programming is required. In off-line programming systems, the programmer uses a 3-dimensional computer model of the robot and its work cell, in which the virtual robot easily can be controlled and moved to the desired configurations. When the program is complete, the motions can be simulated, verified, and optimised before the program is transferred to the real robot. Thus, the robot can execute other programs while new programs are created, and only minor adjustments may be necessary to do on-line.

Another important advantage of off-line programming is the improved safety for the operator as well as the robot. The operator avoids the risk of standing close to the robot during programming, and possible programming errors, which may cause collisions and damage the robot, can be corrected off-line.

Although off-line programming improves efficiency in many aspects, the programming work is still performed manually. When a robot is to be moved from an initial configuration to a final configuration, it is unlikely that the straight line path is feasible due to the obstacles in the cell. It is then necessary to add a number of via-points to the path in order to avoid collisions. If the work cell is complicated or cluttered, much time is spent on finding such via-points. Therefore automatic motion planning is one of the most important areas in robotics research.

1.1 Path Planning

Autonomous path planning addresses the problem of finding collision-free paths for moving objects – robots – among obstacles. In this report we consider robots operating in workspaces occupied by stationary, completely known obstacles.

The robot in this context may be any moving object, or collection of objects, associated with a certain configuration space. A *configuration* is a set of independent parameters such that the position of every point of the robot can be determined relative to a fixed frame in the workspace. The *configuration space* $\mathcal{C}$, of dimension equal to the number of degrees of freedom (dof) of the robot, is the set of all configurations. The open subset $\mathcal{F} \subset \mathcal{C}$ is the set of collision-free configurations.

For instance, if $\mathcal{A}$ is a single rigid object in a 3-dimensional workspace, we can specify its position and orientation by six parameters; three coordinates for the translation, and three angles for the orientation. Thus, we can describe the configuration by a point in a 6-dimensional configuration space $\mathcal{C} \subset \mathbf{R}^6$.

A path $\mathcal{P}$ for the robot is simply a continuous curve in $\mathcal{C}$. We will also refer to a path as a sequence of points in $\mathcal{C}$, in which case the path is the piecewise linear curve obtained by linearly interpolating subsequent points.

The basic path planning problem can be formulated as follows: given an initial configuration $\boldsymbol{q}_{init}$ and a goal configuration $\boldsymbol{q}_{goal}$ in $\mathcal{F}$, find a continuous curve in $\mathcal{F}$ connecting these points, or determine that none exists [24]. This formulation of the problem is usually favourable, since it is stated in terms of navigating a point, rather than objects in the workspace.

The complexity of certain versions of the problem is proven to be very high. In the case of a robot consisting of polyhedral bodies among polyhedral obstacles, the problem is PSPACE-hard, see [34]. Hence, there is strong evidence that a solution requires time that grows exponentially with the number of dof of the robot.

An algorithm is called *complete* if it always will find a solution or determine that none exists. Most complete methods, however, are only applicable to problems in low-dimensional configuration spaces, say of dimension three, or less. A complete algorithm, working for arbitrary dimension, was given in [9]. Although the algorithm is exponential in the number of dof, it has the lowest time complexity of all complete algorithms known so far. It is mostly used in theoretical analysis as an upper bound on the complexity of the path planning problem, see [13, 24] and the discussion in Section 2.1.

1.2 Single Query Path Planning

Of particular interest in industrial and real-time applications, are planners that without preprocessing can answer single queries very quickly. Such high performance planners are important in applications where the configuration space obstacles sometimes change. This occurs, for instance, when the robot changes tools, grasps an object, or a new obstacle enters the workspace. Ideally, the time required for planning should relate to the difficulty of the planning task, i.e., a simple path in an uncluttered environment should be found quickly, while a more complicated path may require more time.

In a similar way, the planning time should relate to the desired quality of the returned path. The quality of a path is difficult to quantify (see further discussion in Section 3.2.1), but in general we prefer short paths in $\mathcal{C}$, with respect to some metric. Consider a case where we use a path planner to support the programming of a robot that executes a certain task continually. Then the quality of the path is more important than the planning time, and we need a parameter, tuned by the user, that intuitively adjusts the properties of the planner.

We would also like the planner to learn to some extent, i.e., to use information from previous queries in order to speed up subsequent queries. For example, if the algorithm finds a path through a narrow passage in $\mathcal{F}$, it should be able to use that information when searching for a new path back through the passage.

Due to the complexity of the path planning problem, complete planners are far too slow to be useful in practice. Trading completeness for speed, probabilistic techniques have been successfully applied to many problems in high-dimensional configuration spaces. However, current probabilistic algorithms either heavily rely on fast collision checking, or require long preprocessing. Collision checking is in many real applications with complex work cells very time consuming, making current probabilistic planners yet too slow for single queries.

1.3 Contributions and Outline of this Report

In this report, we present a new probabilistic path planning algorithm, called Lazy PRM. The algorithm is tailored for single queries, but it is also useful for multiple queries. The planner is applicable to general robots and has the properties discussed in Section 1.2. The algorithm is described in detail in Chapter 3. Its performance is theoretically analysed in Chapter 4, and experimentally evaluated in Chapter 5 using a realistic industrial environment.

Chapter 2

Path Planning Techniques

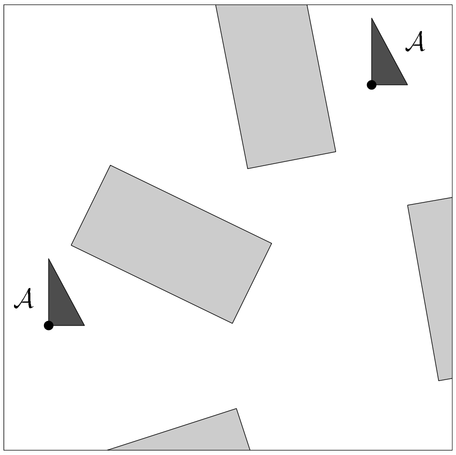
The path planning problem has been extensively studied in the last two decades, and there exist a large number of planners based on a variety of approaches. See [13], [17], and [24] for overviews. We can distinguish between deterministic and probabilistic algorithms. In this chapter, we only give a brief overview of deterministic algorithms, and focus on probabilistic algorithms. Particularly the Probabilistic Roadmap Method is described in detail, since that method forms the base of our solution.

2.1 Deterministic Planners

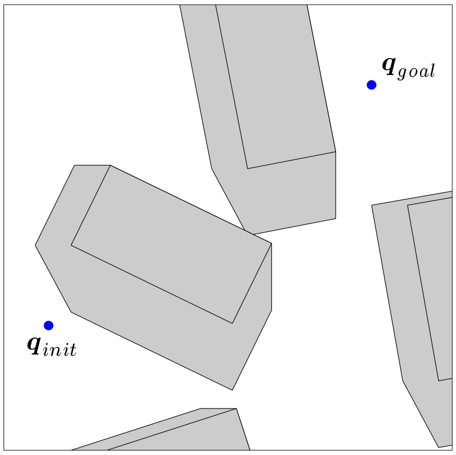
As mentioned earlier, the complexity of the path planning problem increases rapidly with the dimension of the configuration space. In the simplest case we have a 2-dimensional workspace and point robot $\mathcal{A}$ among polygonal obstacles $\mathcal{O}$. Then the workspace and the configuration space coincide. We can also represent the configuration space obstacles by polygons if $\mathcal{A}$ itself is a polygon which is only allowed to translate. If we fix a reference point on $\mathcal{A}$, and specify the position of $\mathcal{A}$ by the coordinates of the reference point, we obtain the configuration space obstacles as the Minkowski set difference $\mathcal{O} \ominus \mathcal{A}$, see Figure 2.1(b).

The 2-dimensional problem with polygonal configuration space obstacles can be solved efficiently by constructing the *visibility graph*, see [24]. It is obtained by interconnecting $\boldsymbol{q}_{init}$, $\boldsymbol{q}_{goal}$, and all vertices of the obstacles by edges which do not intersect the interior of the obstacles, see Figure 2.1(c). Thus, we allow the robot to be in contact with the obstacles, and we consider a path in the closed set $\overline{\mathcal{F}}$ as being collision-free.

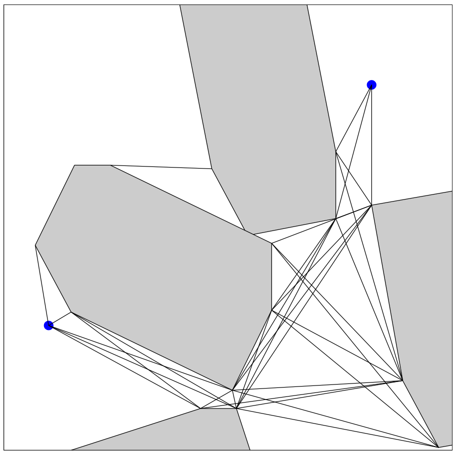
It turns out, however, that this procedure generates too many edges. It is proven that the *reduced visibility graph*, where only tangent edges are retained, always contains a shortest feasible path, see Figure 2.1(d). A line is tangent to a polygon $\mathcal{B}$ at a vertex x if in a neighbourhood around x the interior of $\mathcal{B}$ lies entirely on a single side of the line. An edge between two vertices is a tangent



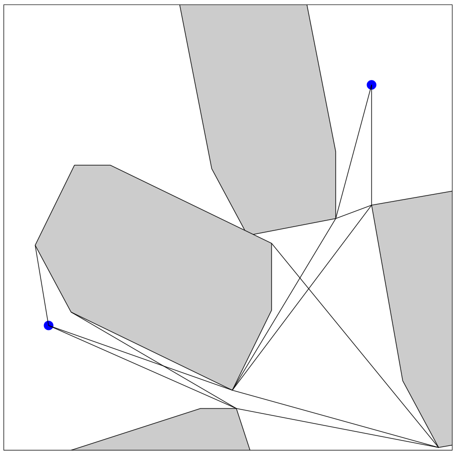
(a): The 2-dimensional workspace with $\mathcal{A}$ in the initial (left) and final configurations. The lower left corner of $\mathcal{A}$ is the reference point.



(b): Corresponding 2-dimensional configuration space. The obstacles are the Minkowski set difference $\mathcal{O} \ominus \mathcal{A}$.



(c): The visibility graph.



(d): The reduced visibility graph.

Figure 2.1: Example of a polygonal robot $\mathcal{A}$ among polygonal obstacles $\mathcal{O}$ (light grey). The robot is only allowed to translate.

edge if the line passing through the vertices is tangent at both vertices, see [24].

If the polygonal robot also is allowed to rotate, the configuration space is 3-dimensional. Since rotations are involved, the configuration space obstacles can no longer be represented by polyhedra. However, efficient planners exist for this kind of problems as well. One way is to discretise the angle of rotation, and for each fixed value construct a slice of polygonal obstacles. The slices can then be merged into polyhedra to approximate the obstacles. Another method is to decompose the configuration space into disjoint cells which are classified either as feasible or infeasible. To find a path, we identify in which cells $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ are located, and then search for a sequence of adjacent feasible cells between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, see [24].

Most techniques for solving problems in low-dimensional configuration spaces are based on some kind of simple and explicit representation of the configuration space obstacles. The difficulties in dimensions higher than three, and particularly if rotations are involved, arise from the complexity of such a representation.

However, by a suitable parametrisation of the configuration space, the obstacles can be represented as a semi-algebraic set, see [9]. This representation handles any kind of rotation, including free-flying rigid objects and linked robot arms with revolute joints (manipulators).

The silhouette method in [9] make use of this representation, and works in any dimension. The idea is briefly to reduce the dimension of the configuration space by recursively sweeping with a hyper plane. The boundary curves of the obstacles – the silhouette – is traced out in the hyper plane. At certain critical points, where the topology of the silhouette changes, the algorithm is called recursively to sweep the current hyper plane. The outcome of the algorithm is a network of paths on the boundary of the obstacles. The algorithm is single exponential in the number of dof and, as mentioned in Section 1.1, it is the best known complete algorithm. However, due to the difficulties of implementing the silhouette method, the algorithm has, as far as the author knows, not yet been implemented for arbitrary dimension.

We will mention two more path planning methods applicable to problems in high-dimensional configuration spaces. The planner in [10] heuristically places subgoals in the configuration space. The subgoals are not points, but disjoint subsets of $\mathcal{C}$. Initially, large subgoals are generated, and a global planner searches for a sequence of adjacent subgoals connecting $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$. A local planner then tries to find paths between adjacent subgoals in the sequence. If the local planner fails in finding a feasible path between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, the global planner searches for a new sequence of subgoals. If no sequence can be found, the subgoals are repeatedly divided into smaller subgoals until a sequence between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ can be found.

The virtual springs method in [29] considers a robot manipulator as a dynamical system in which the links of the robot are somewhat flexible springs. While the end effector is attracted to a hare following a prescribed trajectory in the workspace, the rest of the arm is repelled from the obstacles by a force field. The planner is fast, and the returned paths are smooth and can immediately be used with a real robot. However, due to the difficulty of generating suitable trajectories for the hare, the robot sometimes get stuck in local minima without reaching the target. Within the interactive OxSim framework, described in [8], the trajectories can be specified by the user.

2.2 Probabilistic Planners

Lately, probabilistic planning techniques have gained considerable attention, due to their capability of solving problems in high-dimensional configuration spaces, and in configuration spaces where the obstacles cannot be explicitly represented. The Randomised Path Planner (RPP) in [5] has successfully solved problems for robots with more than 60 degrees of freedom, see [23]. The planner uses a potential field as a guidance towards the goal, and random walks to escape local minima.

Another interesting approach is presented in [28] – the Ariadne's clew algorithm. Considering the initial configuration as a landmark, the algorithm incrementally builds a tree of feasible paths as follows. Genetic optimisation is used to search for a collision-free path from one of the landmarks to a point as far as possible from previous landmarks. A new landmark is then placed at this point, and a path to the goal configuration is searched. New landmarks are placed until the goal configuration can be connected to the tree.

2.3 Probabilistic Roadmap Method

The idea behind the Probabilistic Roadmap Method (PRM), described in [21], [22], and [32], is to represent and capture the connectivity of $\mathcal{F}$ by a random network, a *roadmap*, whose nodes and edges respectively correspond to randomly selected configurations, and path segments. In a preprocessing step, or a *learning phase*, a large number of points are distributed uniformly at random in $\mathcal{C}$, and those found to be in $\mathcal{F}$ are retained as nodes in the roadmap. A local planner is then used to find paths between each pair of nodes that are sufficiently close together. If the planner succeeds in finding a path between two nodes, they are connected by an edge in the roadmap.

In the *query phase*, the start and goal configurations are connected to the roadmap by the local planner. Then the roadmap is searched for a shortest path between the given points.

Even though a powerful local planner will require few nodes to obtain a well connected roadmap, most implemented PRMs show that it is computationally more efficient to distribute nodes densely and use a relatively weak, but fast, local planner, see [22, 32]. The local planner may for instance only check the straight line between two nodes. Other local planners are discussed and evaluated in [1].

A subsequent *node enhancement* step is used to increase the connectivity of the roadmap by adding more nodes in difficult regions of $\mathcal{F}$. Different techniques are used to identify these regions; one way is to distribute new points close to a number of *seeds* randomly selected among the existing nodes. In [21], the probability that a node is selected is proportional to $\frac{1}{1+b}$, where b is the number of edges connected to the node. An alternative selection can be based on a node's ratio of failed attempts by the local planner to find paths to other nodes, see [22]. Other techniques to increase the connectivity of the roadmap are described in [2] and [14].

The PRM has shown to work well in practice in high-dimensional configuration spaces, see [21]. Indeed, it is useful for multiple queries, since once an adequate roadmap has been created, queries can be answered very quickly.

2.4 Variations of PRM

Although the node enhancement step was developed to increase the connectivity of the roadmap, the PRM still has a weakness in finding paths through narrow passages in $\mathcal{F}$. Several recent approaches are intended to improve PRM in this respect by using different sampling strategies. The underlying idea is to distribute nodes close to the boundary of $\mathcal{F}$.

The planner in [15] initially allows the robot to penetrate the obstacles to a certain extent. Small neighbourhoods around the configurations just in collision are then re-sampled in order to place nodes close to the boundary of $\mathcal{F}$. The Obstacle Based PRM (OBPRM) in [2] and [3], repeatedly determines a configuration in collision to be the origin of a number of rays. Binary search is then used along each ray to find points on the boundary of $\mathcal{F}$, where roadmap nodes are placed. In [6], the planner identifies the boundary of $\mathcal{F}$ by distributing points in pairs. Each pair is generated by first picking one point uniformly at random in $\mathcal{C}$, and then picking another point close to the first one. One of the points is added to the roadmap only if it is in $\mathcal{F}$ and the other point is not. Another technique to increase the number of nodes in narrow passages of $\mathcal{F}$ is presented in [36]. Points are picked uniformly at random in $\mathcal{C}$ and then retracted onto the medial axis of $\mathcal{F}$.

A few methods using probabilistic roadmaps do not divide the planning process into a learning phase and a query phase. Given an initial and a goal configuration, the planner in [31] inserts randomly distributed nodes in $\mathcal{F}$, one at a time, and connects them to the different components of the roadmap by a local planner. New nodes are inserted until the initial and goal configurations can be found in the same connected component of the roadmap. See also [11] and [18] for related algorithms. The latter paper gives an adaptive scheme for adjusting the power of the local planner.

Other methods, described in [16] and [25], build two trees rooted at the initial and goal configurations respectively. As soon as the two trees intersect, a feasible path can be extracted. In [16], the trees are expanded by generating new nodes randomly in the vicinity of the two trees, and connecting them to the trees by a local planner. The planner in [25] iteratively generates a configuration, an attractor, uniformly at random in $\mathcal{C}$. Then, for both trees, the node closest to the attractor is selected and a local planner searches for a path of a certain maximum length towards the attractor. A new node is placed at the end of both paths. A new attractor is selected until the two trees intersect.

The general theme for roadmap algorithms is to construct a network of paths verified to be collision-free by a local planner. Unfortunately, it is difficult to find a global strategy that can use these local planners efficiently in order to avoid regions from where the algorithm cannot proceed to the goal. This often means that too much time is spent on planning local paths that will not appear in the final path.

Our solution is to avoid using local planners as much as possible, and instead keep the global view through the entire planning process. In the next chapter, we present Lazy PRM – a path planning algorithm tailored for single queries, but which is also useful for multiple queries. To make the planner fast, the main theme is to minimise the number of collision checks.

Chapter 3

Lazy PRM

This chapter describes a new algorithm for single and multiple query path planning. The algorithm is similar to the original PRM in [21] in the sense that the aim is to find the shortest path in a roadmap generated by randomly distributed configurations. In contrast with existing PRMs, we do not build a roadmap of feasible paths, but rather a roadmap of paths *assumed* to be feasible. The idea is to lazily evaluate the feasibility of the roadmap as planning queries are processed.

In other words, let $\boldsymbol{q}_{init}$, $\boldsymbol{q}_{goal}$, and a number of uniformly distributed points form nodes in a roadmap, and connect by edges each pair of nodes being sufficiently close together. We find a shortest feasible path in the roadmap by repeatedly searching for a shortest path, and then checking whether it is collision-free or not. Each time a collision occurs, we remove the corresponding node or edge from the roadmap, and search for a new shortest path.

This procedure can terminate in either of two ways. If there exist feasible paths in the roadmap between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, we will find a shortest one among them. Otherwise, if there is no feasible path, we will eventually find $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ in two disjoint components of the roadmap. In the latter case, we can either report failure, or, if we still have time, add more nodes to the roadmap (*node enhancement*) and start searching again. A high-level description of the algorithm is given in Figure 3.1.

The rest of this chapter explains the different steps of the algorithm in more detail, Chapter 4 gives a proof of its probabilistic completeness, and Chapter 5 shows some experimental results.

3.1 Building the Initial Roadmap

The first step in the algorithm is to build a roadmap $\mathcal{G}$ in $\mathcal{C}$. There are two parameters that determine the size of $\mathcal{G}$; the number of nodes, N_{init}, and the expected number of neighbours, M_{neighb}, connected to each node.

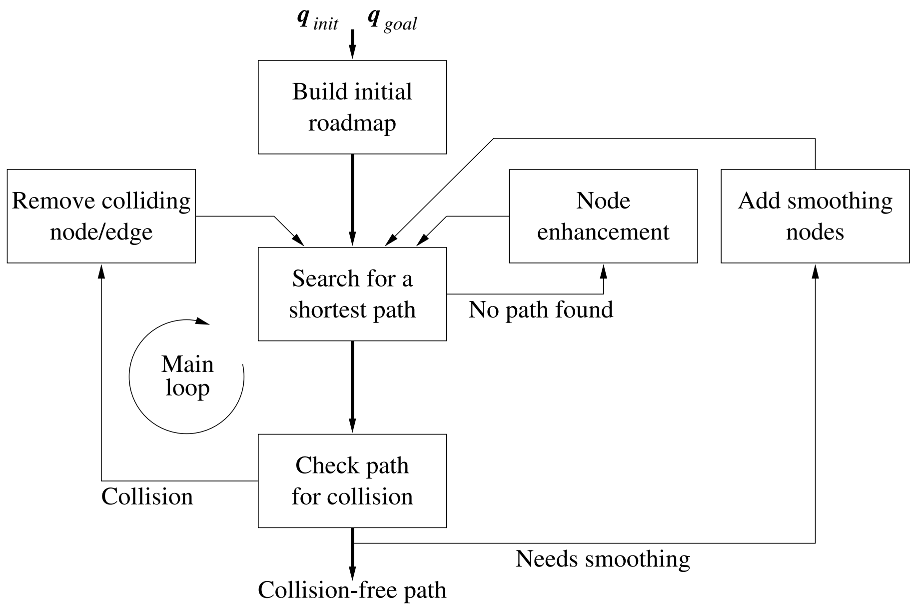


Figure 3.1: High-level description of Lazy PRM.

3.1.1 Initial Distribution of Nodes

Initially, we distribute N_{init} points uniformly at random in $\mathcal{C}$. These points, together with $\boldsymbol{q}_{init} \in \mathcal{F}$ and $\boldsymbol{q}_{goal} \in \mathcal{F}$, form nodes in $\mathcal{G}$. We can, of course, use heuristics to increase the density of nodes in regions we in advance believe are of particular interest, but this is in general difficult.

The initial density of nodes, determined by N_{init}, is strongly correlated to the probability of finding a short path, if one exists. The correlation is hard to quantify, but the following example may give an illustration. Assume there exist only two ways to get to the goal configuration; either a short path through a narrow corridor, or a somewhat longer path through a wide corridor. If $\mathcal{G}$ is sufficiently dense, the algorithm will find a short path through the narrow passage, see Figure 3.2(b). If $\mathcal{G}$ is sparse, the algorithm will find a longer path through the wide passage, see Figure 3.2(a). In the worst case, if the roadmap is too sparse, there will be no feasible path at all in the roadmap, and the algorithm has to go to the enhancement step.

On the other hand, if N_{init} is too large, we will distribute more nodes than necessary. Although we may obtain better paths, this will lead to longer planning times, see Figure 3.2(c). The number of nodes required to find a path is further explored in Chapter 4.

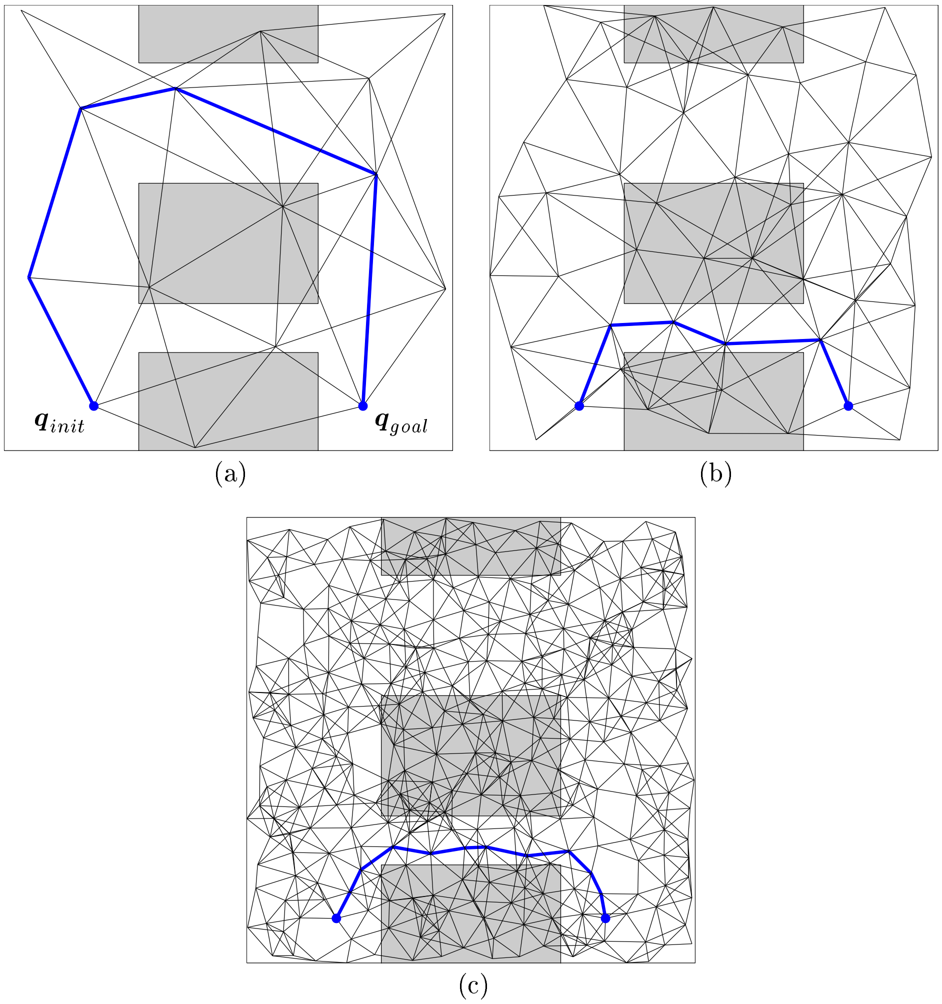


Figure 3.2: Example of a 2-dimensional configuration space with rectangular obstacles (grey). The thick lines show the shortest feasible paths between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ in three different roadmaps. The roadmap in (a) is too sparse and no short feasible path exists. The roadmap in (c) is very dense, and the shortest feasible path will take longer time to find than the shortest feasible path in (b).

3.1.2 Selecting Neighbours

We connect each node in $\mathcal{G}$ by edges to a set of neighbour nodes. An edge represents the straight line path in $\mathcal{C}$ between two nodes. Since it would require far too much memory to connect all pairs of nodes, and it is unlikely that the straight line path between two nodes far apart is feasible, it is natural to only consider nodes which are sufficiently close.

In order to select appropriate neighbours, we need a metric $\rho_{coll} : \mathcal{C} \times \mathcal{C} \to [0, \infty)$ such that the distance between two configurations under this metric reflects the difficulty of connecting them by a collision-free straight line path. Then we connect each pair of nodes $(\boldsymbol{q}, \boldsymbol{q}')$ such that $\rho_{coll}(\boldsymbol{q}, \boldsymbol{q}') \leq R_{neighb}$. For any fixed radius R_{neighb}, the number of neighbours of a node is a random variable, so depending on the initial number of nodes N_{init}, we choose R_{neighb} such that the expected number of neighbours equals the parameter M_{neighb} mentioned in the beginning of Section 3.1.

In many cases it is harder to make feasible connections in certain directions than in others. Consider for instance an articulated robot arm; then it is more likely that a collision occurs when the base joint is moving one unit, than if a joint close to the end-effector is moving one unit. With this in mind, we let ρ_{coll} be a weighted Euclidean metric,

$$\begin{aligned} \rho_{coll}(\boldsymbol{x}, \boldsymbol{y}) &= \Big(\sum_{i=1}^{d} w_i^2 (x_i - y_i)^2 \Big)^{1/2} \\ &= \left((\boldsymbol{x} - \boldsymbol{y})^T W (\boldsymbol{x} - \boldsymbol{y}) \right)^{1/2}, \end{aligned} \tag{3.1}$$

where d is the dimension of $\mathcal{C}$, $\{w_i\}_{i=1}^{d}$ are positive weights, $W = diag(w_1^2, \ldots, w_d^2)$, and $\boldsymbol{x}^T$ is the transpose of $\boldsymbol{x}$. The weights are chosen in proportion to the maximum possible distance (Euclidean distance in the workspace) travelled by any point on the robot, when moving one unit in $\mathcal{C}$ along the corresponding axis. This metric is easy to use and has been shown to work well in our experiments presented in Chapter 5.

3.2 Searching for a Shortest Path

The second step in the algorithm is to find a shortest path in $\mathcal{G}$ between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, or determine that none exists. We use the A^* algorithm [27], and a metric $\rho_{path} : \mathcal{C} \times \mathcal{C} \to [0, \infty)$ to measure the length of a path and the remaining distance to $\boldsymbol{q}_{goal}$.

If the search procedure succeeds in finding a path, we need to check it for collision. Otherwise, if no path exists in the roadmap, we either report failure, or go to the node enhancement step to add more nodes to the roadmap and start searching again. The choice is determined by the overall time allowed to solve the problem.

3.2.1 Choosing an Appropriate Metric for A^*

The tool available to give preference to certain paths and reject others is the metric ρ_{path}. Thus, by defining this metric we decide which paths are of high quality and which paths are of poor quality.

In this report we focus on articulated robots and use the Euclidean configuration space $I_1 \times \cdots \times I_d$, where I_i is the range of joint i and d is the number of dof. Thus, we do not identify angles equal modulo 2π as being equal, although they define the same position in the work space. This is because a real robot in general has supply wires, etc., which otherwise would be entangled. The metric ρ_{path} is a weighted Euclidean metric, similar to (3.1), where the weights are equal to $\frac{1}{v_i}$, $i = 1, ..., d$, where v_i is the maximum angular velocity of joint i. This tends to give preference to paths with short execution time, which in many applications is the most interesting response variable.

In the general case, however, there are a large number of other response variables to consider. Some of them are measurable such as energy consumption, dynamic forces on joints, etc. Others are more subjective; for example, the motion should look natural and smooth from the user's point of view. Under any Euclidean metric, the straight line path in $\mathcal{C}$ between two configurations is the shortest, but considering all of these response variables, the straight line path is not necessarily optimal. Thus, the choice of a configuration space and an appropriate metric is an intricate task itself.

As already pointed out, angles equal modulo 2π define the same position. If any joint of the robot, or the robot itself, is allowed to rotate freely, it is necessary to take this into account. Then the configuration space parameterised by angles is no longer Euclidean, but nothing prevents us from using other metrics more suitable for non-Euclidean configuration spaces. Note that the metric ρ_{coll} in Section 3.1.2 should be changed accordingly, otherwise not all appropriate neighbours will be selected.

3.3 Checking Paths for Collision

When the A^* algorithm has found a shortest path in the roadmap between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, we need to check the nodes and edges along the path for collision. In most applications it is straightforward to perform a collision check for a given configuration, i.e. determine whether a point is in $\mathcal{F}$ or not [35]. It is considerably more complex to obtain more information, for instance to calculate the minimum distance between the robot and the obstacles, or to check whether a path segment is entirely in $\mathcal{F}$ or not [7, 26]. Our algorithm only requires a collision checker for points in $\mathcal{C}$. Path segments, i.e. edges in the roadmap, are discretised and checked with a certain resolution. However, in Section 3.3.3 we describe a variation of the algorithm which makes use of a function giving the minimum distance between the robot and the obstacles.

The overall purpose of the Search, Check, and Remove steps of our algorithm (the Main loop in Figure 3.1), is roughly to identify and remove colliding nodes and edges of the roadmap until the shortest path between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ is feasible. Accordingly, when checking a path for collision, we are not primarily interested in verifying whether an individual node or edge is in $\mathcal{F}$ or not, but rather to remove colliding nodes and edges as efficiently as possible. Since a removal of a node implies all its connected edges to be removed, it seems reasonable to first check the feasibility of the nodes along the path, before checking the edges.

3.3.1 Checking Nodes

Starting respectively with the first and the last node on the examined path and working toward the centre, we alternately check the nodes along the path. As soon as a collision is found, we remove the corresponding node, and search for a new shortest path.

The reason for checking the nodes in this order is that the probability of having the shortest feasible path via a particular node is higher if the node is close to either $\boldsymbol{q}_{init}$ or $\boldsymbol{q}_{goal}$. Consider, for instance, the nodes connected to $\boldsymbol{q}_{init}$; a shortest feasible path (if one exists) must pass through at least one of them. Since we, in a cluttered space, cannot give preference to certain directions, the probability of having the shortest feasible path via a particular neighbour of $\boldsymbol{q}_{init}$ is at least $1/b$, where b is the number of neighbours of $\boldsymbol{q}_{init}$. Nodes connected to $\boldsymbol{q}_{goal}$ have a similar probability, whereas nodes further away from both $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ have a much lower probability of being in the shortest feasible path. Therefore, we check the nodes along a path starting from the end-nodes and working toward the centre.

3.3.2 Checking Edges

If all nodes along the path are in $\mathcal{F}$, we start checking the edges in a similar fashion; working from the outside in. However, to minimise the risk of doing unnecessary collision checks, we first check all edges along the path with a coarse resolution, and then do stepwise refinements until the specified resolution is reached. As with the nodes, if a collision is found, we remove the corresponding edge, and search for a new shortest path. If no collision is found along the path, the algorithm terminates and returns the collision-free path. Figure 3.3 shows an example of a simple planning query.

To make this algorithm efficient, we of course record which nodes have been checked for collision, and to which resolution each edge has been checked, in order to avoid checking any point in $\mathcal{C}$ more than once.

The total number of collision checks depends on the resolution with which the edges along the path are checked. Again, since ρ_{coll} reflects the probability of collision, we determine the resolution with respect to this metric. The resolution is quantified by a step-size δ, but we prefer not to let the user specify the step-size by a certain number, because the resolution should depend on the scale of $\mathcal{C}$ and the weights defining the metric. A more intuitive way is to introduce a parameter M_{coll}, specifying the number of collision checks required to check the longest possible straight line path in $\mathcal{C}$. In other words, assuming that $\mathcal{C}$ is a

d-dimensional rectangle and $\boldsymbol{q}$ and $\boldsymbol{q}'$ are two opposite corners, the step-size is related to the length of the diagonal of $\mathcal{C}$ according to

$$\delta = \frac{\rho_{coll}(\boldsymbol{q}, \boldsymbol{q}')}{M_{coll}}.$$

3.3.3 Collision Checking Using a Distance Function

We have so far assumed that the collision checker is a simple boolean function returning whether a given configuration is in collision or not. The algorithm may of course be used with a more sophisticated *distance function* $D : \mathcal{C} \to \mathbf{R}$ giving the minimum distance in the workspace between the robot and the obstacles, and possibly also a penetration distance in case of collision [7, 26, 30]. We can use such a function to determine a feasible region around a configuration $\boldsymbol{q} \in \mathcal{F}$, and slightly modify our algorithm to efficiently use the extra information given by D.

Let x be an arbitrary point on a robot $\mathcal{A}$. The point x may be given in some local frame attached to the robot, for example a frame attached to a certain link. When the robot is in configuration $\boldsymbol{q}$, the corresponding point in the workspace is denoted $W(x, \boldsymbol{q})$. Let

$$\xi = \sup_{\substack{x \in \mathcal{A} \\ \boldsymbol{q}, \boldsymbol{q}' \in \mathcal{C},\ \boldsymbol{q} \neq \boldsymbol{q}'}} \frac{|W(x, \boldsymbol{q}) - W(x, \boldsymbol{q}')|}{\rho_{coll}(\boldsymbol{q}, \boldsymbol{q}')}.$$

Then

$$|W(x, \boldsymbol{q}) - W(x, \boldsymbol{q}')| \leq \xi \rho_{coll}(\boldsymbol{q}, \boldsymbol{q}') \tag{3.2}$$

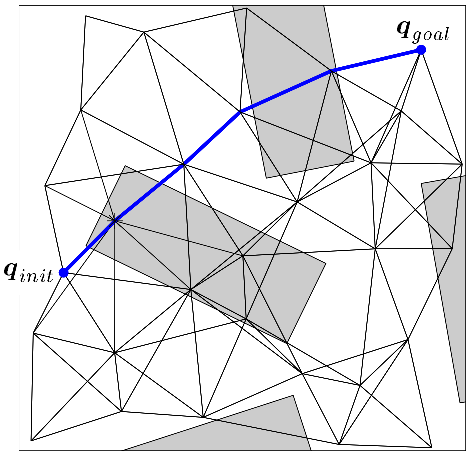
for all $x \in \mathcal{A}$ and $\boldsymbol{q}, \boldsymbol{q}' \in \mathcal{C}$, which particularly means that if $\mathcal{A}$ moves from $\boldsymbol{q}$ to $\boldsymbol{q}'$, then no point on $\mathcal{A}$ moves more than $\xi \rho_{coll}(\boldsymbol{q}, \boldsymbol{q}')$, cf. [33]. Accordingly, if $D(\boldsymbol{q}) > 0$ (i.e., $\boldsymbol{q}$ is collision-free), then all configurations in the ellipsoid

$$\{\boldsymbol{q}' : \rho_{coll}(\boldsymbol{q}, \boldsymbol{q}') < \frac{1}{\xi} D(\boldsymbol{q})\}$$

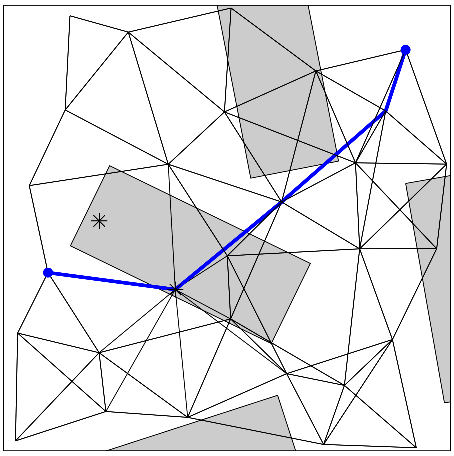
are also collision-free.

Thus, any collision-free path can be covered by a set of ellipsoids which are guaranteed to be in $\mathcal{F}$. Conversely, verifying that a path is collision-free can be considered as a problem of covering the path by ellipsoids.

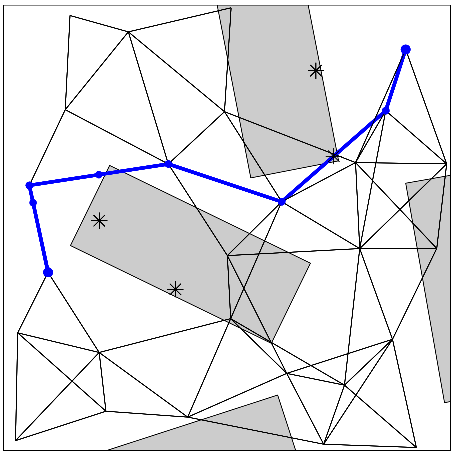
This suggests a somewhat different scheme for checking edges than described in Section 3.3.2. As before, we check the nodes along a path working from outside in, until a collision occurs. Colliding nodes are removed from the roadmap. If all nodes along the path can be covered by feasible ellipsoids, we check the edges along the path by repeatedly checking (and covering by an ellipsoid) the mid-point of the longest path segment not yet covered. The process stops when the path is completely covered, or if a collision occurs. In the latter case the corresponding edge is removed from the roadmap.



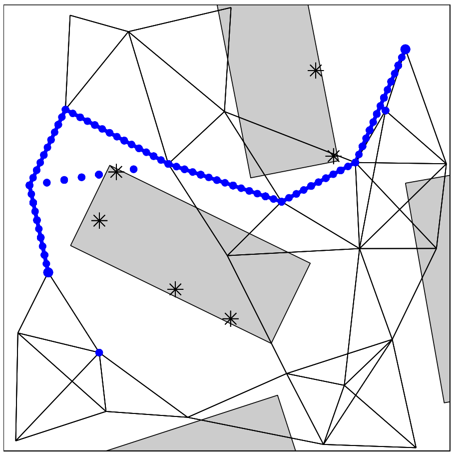
(a): Lazy PRM searches for a shortest path and checks the nodes. A collision is detected ($*$) and corresponding node is removed from the roadmap.



(b): Then Lazy PRM searches for a new shortest path, detects a new collision ($*$) and removes corresponding node.



(c): After a few iterations, a sequence of feasible nodes is found. When checking the edges with a coarse resolution a collision is found ($*$). The edge is removed from the roadmap, and the planner searches for a new shortest path.



(d): Eventually, the planner finds a path whose nodes are collision-free, and whose edges are collision-free to a certain resolution.

Figure 3.3: Example of a planning query in a 2-dimensional configuration space with rectangular obstacles (grey). All collision checks performed are marked with $*$ (collision) or $\bullet$ (collision-free).

3.4 Node Enhancement

If the search procedure fails, no feasible path between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$ exists in the roadmap, and new nodes are necessary in order to find one. In the node enhancement step, we generate N_{enh} new nodes, insert them to $\mathcal{G}$, and select neighbours in the same way as when $\mathcal{G}$ was initially built.

We may not only distribute the new nodes uniformly, but rather use the information available in the roadmap (or what is left of the roadmap), in order to distribute them in difficult regions of $\mathcal{C}$. In a method similar to the node enhancement in [21], we select a number of points in $\mathcal{G}$, called *seeds*, and then distribute a new point close to each of them. Our experience is that it is better to select many seeds and distribute one new node around each of them, instead of selecting few seeds and distribute several nodes around each of them.

Although the seeds may help us identify difficult regions of $\mathcal{C}$, we still want to maintain a smooth distribution all over $\mathcal{C}$, because the knowledge about $\mathcal{C}$ is limited and we do not want to rely too much on the selection of seeds. To ensure probabilistic completeness, we also distribute new nodes uniformly at random in each step. In our algorithm, we let half of the enhancement nodes be uniformly distributed, and the rest distributed around seeds.

3.4.1 Selecting Seeds

The set of edges which have been removed from the roadmap and have at least one end-point in $\mathcal{F}$, will certainly intersect the boundary of $\mathcal{F}$. Using the mid-points of these edges as seeds, may help us distribute points close to the boundary of $\mathcal{F}$.

However, if the enhancement step is executed several times, this may cause problems with clustering of nodes. Assume that we add a new node $\boldsymbol{q}$. This node will give rise to a number of edges which in the next enhancement step may increase the probability of adding even more nodes close to $\boldsymbol{q}$. Thus, the distribution of new enhancement nodes depends on the preceding enhancement steps, and may eventually cause undesired clusters of nodes. To avoid this phenomenon, we only use edges whose end-nodes are generated uniformly at random when selecting seeds.

3.4.2 Distributing New Nodes

The multivariate normal distribution is smooth, easy to use, and allows us to control the distribution of a new point $\boldsymbol{q}$ around a seed $\boldsymbol{\eta}$ in terms of the metric ρ_{coll}. Hence, we can stretch the distribution in directions where the probabilities of making feasible connections are higher.

Introducing two parameters $\alpha \in (0,1)$ and $\lambda > 0$, we can choose the distribution such that

$$\rho_{coll}(\boldsymbol{q},\boldsymbol{\eta}) \leq \lambda R_{neighb} \tag{3.3}$$

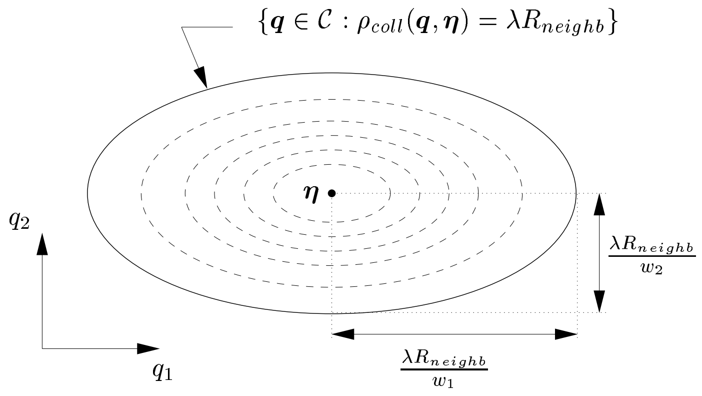


Figure 3.4: Example of a seed $\boldsymbol{\eta}$ in a 2-dimensional configuration space. If a new point $\boldsymbol{q}$ is distributed $N_d(\boldsymbol{\eta}, \Sigma)$, with Σ as in (3.4), then $\boldsymbol{q}$ is distributed within the confidence ellipse (solid line) with probability $1-\alpha$. The dashed ellipses are contours of the distribution function. w_1 and w_2 are the weights defined in (3.1).

is an event with probability $1-\alpha$, see Figure 3.4. R_{neighb} is the maximum length of an edge defined in Section 3.1.2. To achieve this property, we define a covariance matrix Σ as follows:

$$\Sigma = \frac{\lambda^2 R_{neighb}^2}{\chi_d^2(\alpha)} W^{-1}. \tag{3.4}$$

Here W is the same as in (3.1) and $\chi_d^2(\alpha)$ is the upper α percentile of a χ^2-distribution with d dof. Then we let the new point $\boldsymbol{q} \sim N_d(\boldsymbol{\eta}, \Sigma)$, i.e., $\boldsymbol{q}$ is multivariate normally distributed with d dof, mean $\boldsymbol{\eta}$, and covariance matrix Σ. Since Σ is diagonal, this simply means that each component $q_i, i = 1, \ldots, d$, of $\boldsymbol{q}$ is normally distributed with mean η_i and variance $\Sigma_{i,i}$.

To show (3.3), we use that $(\boldsymbol{q}-\boldsymbol{\eta})^T \Sigma^{-1} (\boldsymbol{q}-\boldsymbol{\eta})$ is χ^2-distributed with d dof [19]. Thus, the event

$$(\boldsymbol{q}-\boldsymbol{\eta})^T \Sigma^{-1} (\boldsymbol{q}-\boldsymbol{\eta}) \leq \chi_d^2(\alpha)$$

has probability $1-\alpha$. Using (3.1) and (3.4) gives the confidence ellipsoid in (3.3).

We see in (3.4) that Σ depends on the the ratio $\lambda^2/\chi_d^2(\alpha)$. Since both λ^2, $\lambda > 0$, and $\chi_d^2(\alpha)$, $\alpha \in (0,1)$, are continuous functions whose ranges are $(0,\infty)$, one of the two parameters α and λ is redundant, so we can without loss of generality choose $\alpha = 0.05$. Then, the parameter λ controls the size of the 95% confidence ellipsoid relative to R_{neighb}, see Figure 3.4. In our experiments we found that $\lambda = 1$ is a suitable choice.

Another possibility of distributing the new point $\boldsymbol{q}$, is to let it be uniformly distributed in a rectangular box centred at $\boldsymbol{\eta}$. If we let the sides of the box be of equal length under ρ_{coll}, we stretch the box in a similar way as the ellipsoids above. In our path planning algorithm, however, the normal distribution has a major advantage compared to the uniform distribution; the contours of the distribution function are ellipsoids around $\boldsymbol{\eta}$ (see Figure 3.4). Hence, under the metric ρ_{coll}, which reflects the difficulty of making connections, the distribution is symmetric around $\boldsymbol{\eta}$. In contrast, the uniform distribution favours the directions of the corners of the box, and nodes are more frequently distributed there than in other directions.

3.5 Smoothing

The roadmap consists of randomly selected points, so the shortest feasible path $\mathcal{P}$ returned by the main loop, may be somewhat jerky and unnatural. In this section, we will describe a smoothing procedure that fits perfectly into the algorithm described so far, see Figure 3.1. We simply add new points locally around the collision-free path, and run the main loop again. The main loop either finds a shorter feasible path, which probably also is smoother than $\mathcal{P}$, or, if the new points did not contribute to a shorter path, eventually returns $\mathcal{P}$.

The new nodes around the path can be generated in many ways, for example randomly. In our current implementation however, we add nodes using a simple heuristic rule. To describe the rule, we need some notation. We define the *bounding rectangle* of a set of points $\{\boldsymbol{q}_i\}_{i=1}^{n}$ in a d-dimensional configuration space $\mathcal{C}$, as the intersection of all axis aligned d-dimensional rectangles containing all of the points $\boldsymbol{q}_i$, $i = 1, \ldots, n$. The bounding rectangle may be degenerated, e.g. the bounding rectangle of a single point, but it is still well-defined.

Let $\boldsymbol{p}, \boldsymbol{q}$, and $\boldsymbol{r}$ be three consecutive nodes along $\mathcal{P}$, and $\{\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}\}$ be the subpath between $\boldsymbol{p}$ and $\boldsymbol{r}$. Denote by A the bounding rectangle of $\boldsymbol{p}$ and $\boldsymbol{q}$, and B the bounding rectangle of $\boldsymbol{q}$ and $\boldsymbol{r}$. Let $C = A \cap B$ and $\{\boldsymbol{c}_i\}_{i=1}^{n_C}$ be the corners of C. Then the rule is to insert into the roadmap $\mathcal{G}$ the points $\{\boldsymbol{c}_i\}_{i=1}^{n_C}$ which are not already in $\mathcal{G}$. The points are inserted in a similar way as described in Section 3.1. Figure 3.5 shows which points are added to the roadmap in four cases in a 3-dimensional configuration space.

Since $\boldsymbol{p}$ and $\boldsymbol{q}$ are opposite corners of A, it follows that

$$\rho_{path}(\boldsymbol{p}, \boldsymbol{a}) \leq \rho_{path}(\boldsymbol{p}, \boldsymbol{q}) \quad \text{for all } \boldsymbol{a} \in A$$

Similarly,

$$\rho_{path}(\boldsymbol{b}, \boldsymbol{r}) \leq \rho_{path}(\boldsymbol{q}, \boldsymbol{r}) \quad \text{for all } \boldsymbol{b} \in B$$

Particularly, since $C \subset A$ and $C \subset B$, we have

$$\rho_{path}(\boldsymbol{p}, \boldsymbol{c}_i) + \rho_{path}(\boldsymbol{c}_i, \boldsymbol{r}) \leq \rho_{path}(\boldsymbol{p}, \boldsymbol{q}) + \rho_{path}(\boldsymbol{q}, \boldsymbol{r}), \; i = 1, \ldots, n_C,$$

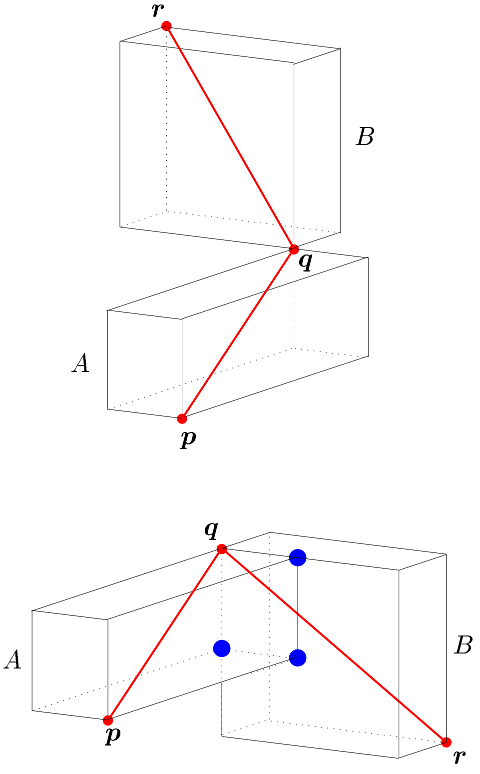


Figure 3.5: The points marked with ● are added to the roadmap in order to find a smoother and shorter path. Which points are added close to a node $\boldsymbol{q}$ depend on the position of the preceding node $\boldsymbol{p}$ and the subsequent node $\boldsymbol{r}$.

i.e., any path $\{\boldsymbol{p}, \boldsymbol{c}_i, \boldsymbol{r}\}$ is shorter than $\{\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}\}$.

This smoothing procedure particularly prevents each joint from undesired over-shooting – which generally looks awkward to the human eye. Over-shooting occurs if a motion of a joint changes direction when moving along a path. By adding the extra nodes to $\mathcal{G}$, we obtain paths whose peaks are cut off, and looks smooth and nice to the user.

3.6 Multiple Queries

When the planner has found a collision-free path, it terminates and returns the path. The information about which nodes and edges have been checked for collision is stored in the roadmap, and as long as the configuration space remains the same, we use the same roadmap when processing subsequent queries, thus benefiting from the information already obtained. The new initial and final configurations are simply added to the roadmap, and the same algorithm, except for the initial generation of nodes, is run again.

As several queries are processed, more and more of the roadmap will be explored, and the planner will eventually find paths via nodes and edges which have already been checked for collision. This makes the planner efficient for multiple queries.

Even in the long run, many nodes and edges may never be explored since they are located in odd regions of $\mathcal{C}$. Thus, given a fixed size of the roadmap, the number of collision checks performed by Lazy PRM will never exceed the number of collision checks performed by the original PRM described in Section 2.3.

Chapter 4

Probabilistic Completeness

In this chapter we give a proof of probabilistic completeness of Lazy PRM. First we need some notation. Let $\gamma : [0, L] \to \mathcal{F}$ be a curve (also called path) parameterised by arc length and with continuous tangent. A *tube* τ of radius r around $\gamma(s)$ is the set of points at distance r from γ measured perpendicular to the tangent $\gamma'(s)$. Similarly, the corresponding *solid tube* is the set of points at distance $\leq r$ from γ. For simplicity, we usually omit the word solid.

A *regular tube* is a tube that does not intersect itself. If γ is enclosed by a regular tube of radius r, this particularly implies that its curvature, $\kappa(s) = |\gamma''(s)|$, is bounded from above by $1/r$. Otherwise the tube would be folded. The following lemma, proved in [12], states a useful property of regular tubes.

Lemma 1. *The volume enclosed by a regular tube around a curve in a d-dimensional Euclidean space is the product of the length of the curve and the $(d-1)$-dimensional area of a cross-section.*

In other words, if B_r^d is the ball of radius r in a d-dimensional space, and μ_d the Lebesgue measure, we can express the volume of a regular tube τ of radius r around γ as

$$\mu_d(\tau) = L\mu_{d-1}(B_r^{d-1}) = Lr^{d-1}\mu_{d-1}(B_1^{d-1}), \tag{4.1}$$

where L is the length of γ.

Assuming there exists a path enclosed by a regular tube in $\mathcal{F}$ between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, the following theorem gives an upper bound on the probability of failure to find a path between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$. Moreover, the theorem says that the probability of failure decreases exponentially in the *total* number of uniformly distributed nodes N. Since N increases in each enhancement step (Figure 3.1 and Section 3.4), the probability of failure vanishes as time tends to infinity. This is equivalent to the definition of *probabilistic completeness*, see [17]. Thus, Lazy PRM is a probabilistically complete path planner.

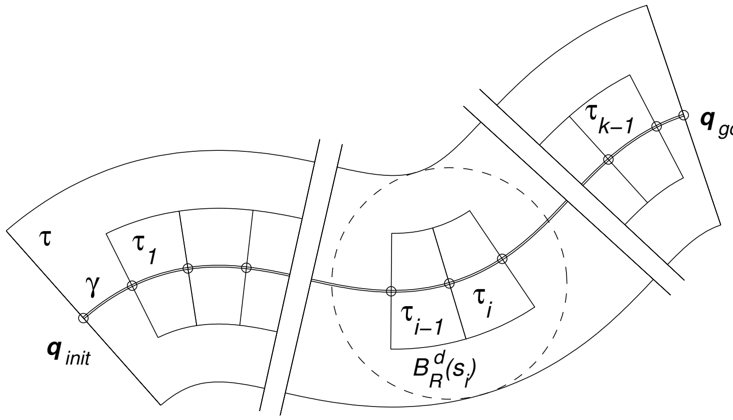


Figure 4.1: Illustration to the proof of Theorem 1.

Theorem 1. *Let N be the total number of nodes generated uniformly at random in $\mathcal{C}$. If there exists a path γ between $\boldsymbol{q}_{init}$ and $\boldsymbol{q}_{goal}$, enclosed by a regular tube τ of radius $R \leq \frac{1}{2}R_{neighb}$ entirely in $\mathcal{F}$, then Lazy PRM will fail to find a path with probability at most*

$$\frac{Ld}{R}e^{-\beta N},$$

where $\beta = \frac{R^d \mu_{d-1}(B_1^{d-1})}{3d\mu_d(\mathcal{C})}$ and L is the length of γ.

Proof. Let $u = R/d$, $r = R(1-1/d)$, and $k = \lfloor L/u \rfloor$. The idea of the proof is to take a tube of radius r, divide it into $k-1$ cells of length u, and calculate the probability of having at least one node in each cell. We will show that any two points in adjacent cells can be connected by a straight line, and that one node in each cell is enough for the planner to succeed.

Let $s_i = iu$, $i = 1, \ldots, k$, and let τ_i be the tube segment around $\gamma(s)$ for $s \in [s_i, s_{i+1})$, $i = 1, \ldots, k-1$. The tube segments $\{\tau_i\}_{i=1}^{k-1}$ are pairwise disjoint and, by (4.1),

$$\frac{\mu_d(\tau_i)}{\mu_d(\mathcal{C})} = ur^{d-1}\frac{\mu_{d-1}(B_1^{d-1})}{\mu_d(\mathcal{C})}.$$

Now, since

$$ur^{d-1} = \frac{R^d}{d}\left(1-\frac{1}{d}\right)^{d-1} \geq \frac{R^d}{d}e^{-1} \geq \frac{R^d}{3d},$$

we get that

$$\frac{\mu_d(\tau_i)}{\mu_d(\mathcal{C})} \geq \frac{R^d \mu_{d-1}(B_1^{d-1})}{3d\mu_d(\mathcal{C})} = \beta. \tag{4.2}$$

The N points generated by the algorithm are uniformly and independently distributed in $\mathcal{C}$. Thus, the probability that τ_i is empty equals $\left(1 - \frac{\mu_d(\tau_i)}{\mu_d(\mathcal{C})}\right)^N$, which, by (4.2), can be estimated:

$$\left(1 - \frac{\mu_d(\tau_i)}{\mu_d(\mathcal{C})}\right)^N \leq (1-\beta)^N. \tag{4.3}$$

Let $B_R^d(s)$ be a ball of radius R centred at $\gamma(s)$, i.e., $B_R^d(s)$ has the same radius as τ. Unless $B_R^d(s)$ is close to the end-points of γ, it will be covered by τ, see Figure 4.1. If it is close to the end-points, however, it might intersect the circular discs at the ends of the tube. Nevertheless, the intersection between $B_R^d(s)$ and τ is still convex, a property we will need later.

By the definition of a tube, for any point $\boldsymbol{q} \in \tau_i$, there exists an $s_q \in [s_i, s_{i+1})$ such that $|\boldsymbol{q} - \gamma(s_q)| \leq r$. Since γ is parameterised by arc length, it follows that $|\gamma(s) - \gamma(t)| \leq |s - t|$, and, by the triangle inequality,

$$\begin{aligned} |\boldsymbol{q} - \gamma(s_i)| &\leq |\boldsymbol{q} - \gamma(s_q)| + |\gamma(s_q) - \gamma(s_i)| \\ &\leq r + u = R. \end{aligned}$$

Hence, the ball $B_R^d(s_i)$ contains τ_i. Similarly, we can show that it also contains τ_{i-1}. Since both cells are covered by τ, they are contained in the convex set $B_R^d(s_i) \cap \tau$, which is entirely in $\mathcal{F}$. Now, $R \leq \frac{1}{2} R_{neighb}$ guarantees that any point in τ_{i-1} is in the neighbourhood of any point in τ_i, thus, any node in τ_{i-1} can be connected to any node in τ_i by a straight line in $\mathcal{F}$, see Figure 4.1. Moreover, since $\boldsymbol{q}_{init} \in B_R^d(s_1)$ and $\boldsymbol{q}_{goal} \in B_R^d(s_k)$, they can be connected to any node in τ_1 and τ_{k-1} respectively. Consequently, it is enough to have at least one node in each of the cells $\tau_1, \ldots, \tau_{k-1}$, in order for Lazy PRM to find a collision-free path between $\boldsymbol{q}_{init}$ or $\boldsymbol{q}_{goal}$.

The probability of failure for our algorithm, $P_{failure}$, can now be estimated:

$$\begin{aligned} P_{failure} &\leq P(\text{some } \tau_i \text{ is empty}) \\ &\leq \sum_{i=1}^{k-1} P(\tau_i \text{ is empty}) \\ &\leq (k-1)(1-\beta)^N, \end{aligned}$$

where we used Boole's inequality and (4.3) in the second and third step respectively. Using that $k - 1 \leq Ld/R$ and $(1-\beta)^N \leq e^{-\beta N}$ gives the desired estimation. □

Note that a related theorem can be found in [4] and [20]. An important difference is that, given a failure probability, Lazy PRM has to reach a certain density of nodes in $\mathcal{C}$, while in the original PRM it is enough to reach approximately the same density in $\mathcal{F}$. This seems like a weakness of our method, but in order to reach the desired density in $\mathcal{F}$, PRM has to distribute nodes uniformly all over $\mathcal{C}$. So whether the density is specified in $\mathcal{F}$ or in $\mathcal{C}$ does not matter. Thus, for both algorithms to reach the same density, the number of nodes checked for collision in the learning phase of PRM has to be the same as the number of uniformly distributed nodes in Lazy PRM.

Chapter 5

Experimental Results

In this chapter we present performance tests of Lazy PRM when applied to a six dof robot in a realistic industrial environment. The planner has been implemented in C++ as a plug-in module to RobotStudio[1] – a simulation and off-line programming software running under Windows NT. The collision checking is handled internally in RobotStudio. The experiments have been run on a PC with a 400 MHz Pentium II processor and 512 MB RAM. In the tests we let $N_{init} = 10000$, $M_{neighb} = 60$, $M_{coll} = 200$, and $N_{enh} = 500$.

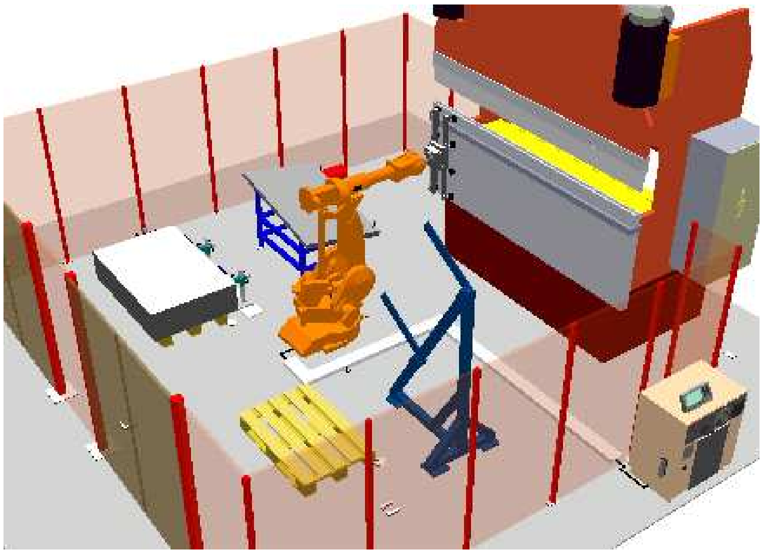


Figure 5.1: The work cell used in the experiments. The robot is in its home configuration denoted by A.

[1]RobotStudio is developed by ABB Digital Plant Technologies AB, Göteborg, Sweden.

5.1 Path Planning Tasks

The test example is a part of a manufacturing process in which an ABB 4400 robot is tending press breaking. Metal sheets are formed by the hydraulic press shown in Figure 5.1. In this particular example, plane sheets of metal are picked at a pallet, bent twice, and then placed at another pallet.

The process is divided into several steps, and our aim is to automatically plan the unconstrained paths of the robot. We let A to J denote ten different configurations shown in Figures 5.1, 5.2, and 5.3. These are used as either initial or goal configurations in eight planning tasks, denoted for example $A \rightarrow B$, where A is the initial configuration and B is the goal configuration.

The scenario is as follows. Starting from the home configuration A, the robot picks a sheet of metal from the pallet at B (task $A \rightarrow B$), adjusts the grip at C (task $B \rightarrow C$), and puts the sheet-metal at the press D (task $C \rightarrow D$). After the breaking, the robot grasps the sheet-metal at E, moves to the re-gripper F (task $E \rightarrow F$), places the sheet-metal, moves to the other side G (task $F \rightarrow G$), grasps the sheet-metal, and moves back to the press H (task $G \rightarrow H$). After the second breaking, the sheet-metal is grasped at configuration I and placed at the pallet J (task $I \rightarrow J$). Then the robot returns to the home configuration A (task $J \rightarrow A$).

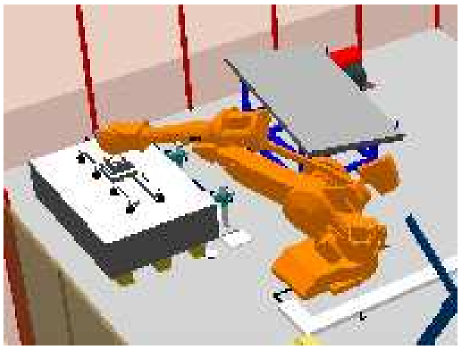
Thus, we have eight paths to plan. Note that during this series of steps, the configuration space changes several times. As soon as we grasp or place a sheet of metal, the collision-free part, $\mathcal{F}$, is changing. Neglecting the small displacement of the sheet-metal caused by the centring operation at C, the tasks $B \rightarrow C$ and $C \rightarrow D$ can be planned in the same configuration space. Accordingly, we have seven different configuration spaces in which to plan, and we have to build one roadmap in each of them.

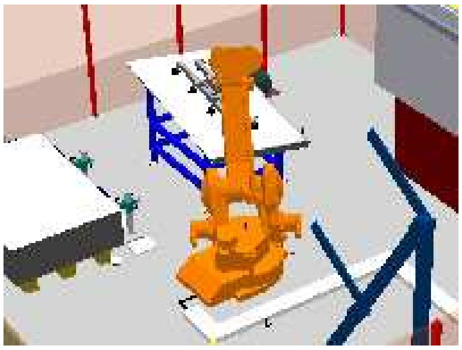
The results include the number of collision checks, the number of enhancement steps, and the planning time. The minimum, average, and maximum values, based on 20 consecutive runs for each task, are shown in Table 5.1(a) - 5.1(c). The average number of collision checks performed on nodes and edges respectively are presented, as well as the average number of collision checks performed on the collision-free paths that the planner returned. Since paths are checked for collision with a certain resolution (see Section 3.3.2), the latter figures correspond to the lengths of the collision-free paths.

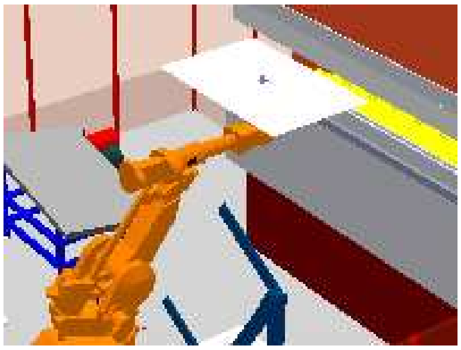
The running times in Table 5.1(c) are divided into three parts. Firstly, graph building, which includes distance calculations and node and edge adding, secondly, graph searching, and finally collision checking.

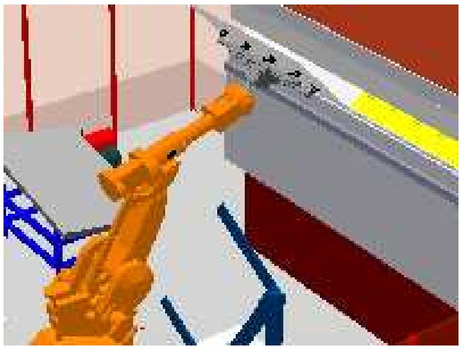
In the last column of Table 5.1, the average values of the recorded data are summed up. Thus, the last column indicates the average number of collision checks and average planning times for the entire press breaking operation.

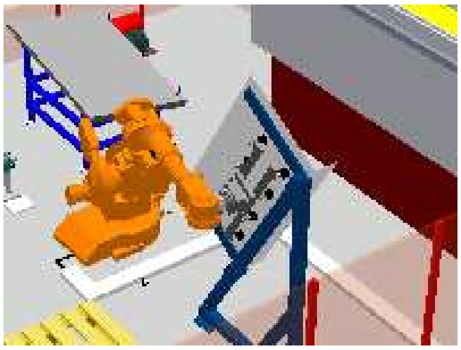
In Table 5.1(d), we have for comparison reasons included some results obtained with a PRM-like algorithm. For each task, we simply checked *all* nodes and edges for collision in one of the roadmaps built by Lazy PRM in the initial step (see Figure 3.1). This corresponds to the learning phase without node enhancement in the original PRM [21]. Due to the long running times, only one full roadmap was


Configuration B


Configuration C


Configuration D


Configuration E


Configuration F

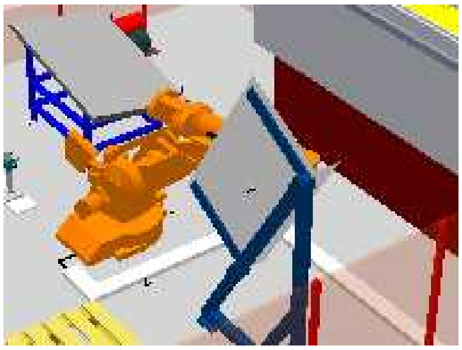
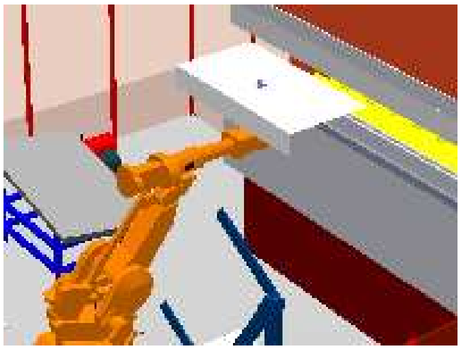
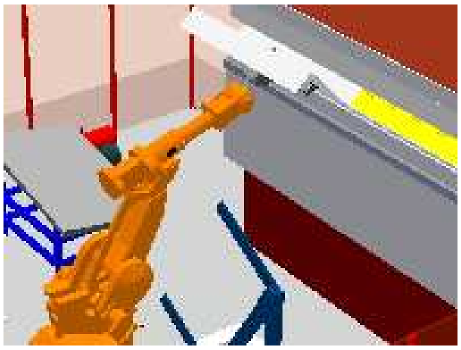

Configuration G

Figure 5.2: Configurations B to G used in the experiments.

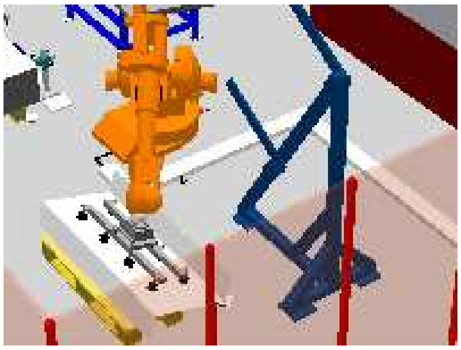
explored for each task. Note that even with this long preprocessing, there is no guarantee that the planner will find a feasible path immediately. We see in 5.1(b) that several enhancement steps are needed with Lazy PRM, thus also needed here, so we can expect even longer running times with the PRM than those shown in



Configuration H



Configuration I



Configuration J

Figure 5.3: Configurations H to J used in the experiments.

Table 5.1(d).

5.2 Interpretation of Results

We clearly see in Table 5.1(c) that the collision checking represents the vast majority of the planning time (80%), but also that the graph building takes a lot of time (18%). Note that for the task $C \to D$, the same roadmap is used as for the task $B \to C$, making the graph building time significantly shorter. Interestingly, the time spent on graph searching is negligible, about 2%. Although we carefully select the points to check for collision, and frequently search the roadmap for the shortest path, the total time spent on that is still very short.

Comparing the average number of collision checks performed by Lazy PRM (92 - 682) in Table 5.1(a) to the number of collision checks required to explore the entire initial roadmap (of order 500 000) in Table 5.1(d), we see that Lazy PRM only explores a small fraction, less than 0.1%, of the roadmap. This is the strength of the algorithm; to either find a collision-free path or to conclude that none exists in the roadmap in a very short amount of time.

We also see in Table 5.1(a) that a large percentage, 26%, of the total number of collision checks are actually performed on the returned collision-free paths, and are therefore inevitable. This large percentage can be explained by two reasons. Firstly, the algorithm finds a sequence of collision-free nodes before edges are being checked. This prevents from planning local paths in regions from where no way out exists. Secondly, we check the edges along the path with increasing resolution and stop as soon as a collision occurs, i.e., we avoid using a local planner and instead keep the global view throughout the planning process. As a consequence, very few edges – often only the edges along the final path – are checked with the finest resolution. This also makes the algorithm relatively insensitive to the resolution with which the paths are checked.

Since all the nodes in the initial roadmap are uniformly distributed, the number of collision-free nodes found by the PRM-like algorithm will give a good estimation of the relative size of $\mathcal{F}$. We see in Table 5.1(d) that for the tasks $A \rightarrow B$, $F \rightarrow G$, and $J \rightarrow A$ approximately 40% of $\mathcal{C}$ is collision-free. For the other tasks approximately 30% of $\mathcal{C}$ is collision-free. As expected, the free part of $\mathcal{C}$ is reduced when the robot grasps a sheet of metal.

Furthermore, from the planner's point of view, the robot's tool includes both the gripper and possibly also a sheet of metal attached to it. If the tool is large and irregularly shaped, then its orientation becomes more important, whereas if the tool is small (e.g. the gripper only), the wrist motions of the robot, which basically determine the orientation, become less important. In this kind of environment, the planning problem is significantly easier if the tool is small. This explains why all of the tasks $A \rightarrow B$, $F \rightarrow G$, and $J \rightarrow A$ are all successfully planned without any node enhancement.

5.3 Discussion

The aim of Lazy PRM is essentially to minimise the number of collision checks while searching the shortest feasible path in a roadmap in the context of a PRM planner. This is done on the expense of frequent graph search. For a complex robot working in a complex workspace, like our six dof example, collision checking is an expensive operation, and careful selection of the points being checked for collision reduces the planning time considerably.

However, if the robot is very simple, like the line segment robots in [21] and [22], then collision checking is a very fast operation, and it is probably not worth re-planning the entire path every time a collision is found. Trading some collision checking for less graph searching may increase performance of Lazy PRM. We can, for instance, in the step described in Section 3.3.1, check *all* nodes along the path before searching for a new path. Then we can remove several nodes from the roadmap in each iteration of the Main loop, see Figure 3.1.

Another modification of Lazy PRM is necessary when the configuration space is very cluttered. This is, for instance, the case with the ten dof robot in [21], where

Table 5.1: Performance data for Lazy PRM based on 20 consecutive runs for each task. Table 5.1(d) shows data for PRM based on one run for each task. The initial number of nodes, N_{init}, is 10000 in all tests.

		Task			
		$A \to B$	$B \to C$	$C \to D$	$E \to F$
Lazy PRM					
Collision checks					
for nodes	ave	9	41	172	263
for edges	ave	83	125	273	235
for returned path	ave	78	60	86	82
	min	74	45	143	154
total	ave	92	166	445	499
	max	131	463	701	1010

Table 5.1(a).

		$A \to B$	$B \to C$	$C \to D$	$E \to F$
No. of enh. steps					
	min	0	0	0	0
	ave	0	0.3	0.8	1.9
	max	0	1	2	5

Table 5.1(b).

		$A \to B$	$B \to C$	$C \to D$	$E \to F$
Running time (sec.)					
graph building	ave	6.6	6.7	0.8	8.3
graph searching	ave	0	0.1	0.5	1.3
coll. checking	ave	6.1	13.3	35.4	42.3
	min	11.2	9.7	10.8	19.7
total	ave	12.7	20.2	36.8	52.0
	max	16.2	45.7	60.9	97.3

Table 5.1(c).

	$A \to B$	$B \to C$	$C \to D$	$E \to F$
PRM				
Collision checks				
for nodes	10000	10000	10000	10000
of which in $\mathcal{F}$	4085	2942	2975	3047
for edges	763063	409561	423443	451254
total	773063	419561	433443	461254
Running time (sec.)				
total	56625	31428	32299	35840

Table 5.1(d).

Table 5.1: continued.

Task				
$F \to G$	$G \to H$	$I \to J$	$J \to A$	Total
129	134	320	18	1088 (40%)
283	158	361	124	1643 (60%)
121	80	114	82	704 (26%)
175	135	139	81	
412	293	682	142	2730
820	442	1290	299	

Table 5.1(a).

0	0	0	0	
0	0.8	1.6	0	
0	2	4	0	

Table 5.1(b).

6.5	7.3	8.2	6.6	51.0 (18%)
0.9	0.4	3.0	0	6.3 (2%)
38.3	24.7	59.8	11.6	231.6 (80%)
22.1	16.8	17.8	13.0	
45.7	32.5	71.0	18.2	289.0
87.3	47.8	129.5	31.2	

Table 5.1(c).

10000	10000	10000	10000	
3976	3038	3090	4121	
728012	447541	447000	787507	
738012	457541	457000	797507	

51774	35097	35200	56234	

Table 5.1(d).

more than 99% of the configuration space is infeasible. If we run our algorithm, we would need a large number of nodes in the initial roadmap, and then remove from the roadmap approximately 99% of the nodes being checked, which would take a lot of time. Fortunately, we can easily modify Lazy PRM to check all nodes *before* we insert them into the roadmap. This would certainly cause unnecessary nodes to be checked for collision, but on the other hand, we would save many inserting and removing operations in the roadmap. After that, we still have the efficient way of exploring the edges along paths.

Consequently, by using either or both of the two modifications of the algorithm suggested above, we can tune the amount of graph search according to the application and the time required to perform a collision check, so that Lazy PRM becomes efficient for a wide range of problems.

Lazy PRM has essentially one parameter that is critical for the performance – N_{init}, the initial number of nodes. As indicated in Theorem 1, N_{init} is strongly correlated to the probability of finding a feasible path without using the node enhancement step. The optimal choice depends on the dimension of $\mathcal{C}$, the workspace, the planning task, and the desired quality of the collision-free path. Our future work includes an investigation of the dependence between N_{init} and the planning time in different environments, as well as different distributions of the nodes.

Probabilistic techniques, like Lazy PRM, often gives very fast planning. However, in Table 5.1(c), we can see that the maximum planning time is approximately twice as long as the average planning time. New improved enhancement techniques, in order to make the algorithms more robust in the sense that the worst case performance is improved, will also be a topic of our future research.

Chapter 6

Summary and Conclusions

We have described a new probabilistically complete path planning algorithm applicable to virtually any kind of robot. The planner is particularly useful in high-dimensional, relatively uncluttered configuration spaces, and when collision checking is an expensive operation. However, variations are proposed that easily handle the cases of cheap collision checking and cluttered configuration spaces.

Single queries are handled very quickly; indeed, no preprocessing is required. Moreover, as subsequent queries are processed the algorithm learns more about the configuration space, since it automatically retains information obtained during previous queries. Thus, the planner works efficiently also for multiple queries.

Bibliography

[1] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1998.

[2] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Robotics: The Algorithmic Perspective*, pages 630–637. AK Peters, 1998.

[3] N.M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 113–120, 1996.

[4] J. Barraquand, L. E. 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):759–775, 1997.

[5] J. Barraquand and J.C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. of Rob. Research*, 10:628–649, 1991.

[6] V. Boor, M.H. Overmars, and F. van der Stappen. The Gaussian sampling strategy for probabilistic roadmap planners. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 1018–1023, 1999.

[7] S. Cameron. Enhancing GJK: Computing minimum distance and penetration distanses between convex polyhedra. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 3112–3117, 1997.

[8] S. Cameron. Motion planning and collision avoidance with complex geometry. In *Proc. Conf. of IEEE Industrial Electronics Society, IECON*, pages 2222 – 2226, 1998.

[9] J.F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[10] P. C. Chen and Y. K. Hwang. SANDROS:a dynamic graph search algorithm for motion planning. *IEEE Tr. on Rob. & Aut.*, 14(3):390–403, 1998.

[11] B. Glavina. Solving findpath by combination of goal-directed and randomized search. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 1718–1723, 1990.

[12] A. Gray. *Tubes*. Addison-Wesley, Redwood City, CA, 1990.

[13] K. Gupta and A. P. del Pobil. *Practical Motion Planning in Robotics*. John Wiley, West Sussex, England, 1998.

[14] T. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom - random reflections at C-space obstacles. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 3318–3323, 1994.

[15] D. Hsu, L.E. Kavraki, J.C. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. In P. Agarwal, L. Kavraki, and M. Mason, editors, *Robotics: The Algorithmic Perspective*, pages 141–154. A K Peters, 1998.

[16] D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 2719–2726, 1997.

[17] Y.K. Hwang and N. Ahuja. Gross motion planning - a survey. *ACM Comp. Surveys*, 24(3):219–291, 1992.

[18] P. Isto. A two-level search algorithm for motion planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 2025–2031, 1997.

[19] R.A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, New Jersey, 1998.

[20] L.E. Kavraki, M.N. Kolountzakis, and J.C. Latombe. Analysis of probabilistic roadmaps for path planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 3020–3025, 1996.

[21] L.E. Kavraki and J.C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 2138–2145, 1994.

[22] L.E. Kavraki, P. Švestka, J.C. Latombe, and M. Overmars. Probabilistic roadmaps for fast path planning in high dimensional configuration spaces. *IEEE Tr. on Rob. & Aut.*, 12:566–580, 1996.

[23] Y. Koga, K. Kondo, J. Kuffner, and J.C. Latombe. Planning motions with intentions. *Computer Graphics (SIGGRAPH'94)*, pages 395–408, 1994.

[24] J.C. Latombe. *Robot Motion Planning*. Kluwer, Boston, MA, 1991.

[25] S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 473–479, 1999.

[26] M.C. Lin and J.F. Canny. A fast algorithm for incremental distance computation. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 1008–1014, 1991.

[27] G.F. Luger and W.A. Stubblefield. *Artificial intelligence and the design of expert systems*. Benjamin/Cummings, Redwood City, CA, 1989.

[28] E. Mazer, J.M. Ahuactzin, and P. Bessière. The Ariadne's clew algorithm. *J. of Art. Intelligence Research*, 9:295–316, 1998.

[29] A. McLean and S. Cameron. Path planning and collision avoidance for redundant manipulators in three dimensions. In *Int. Conf. Intelligent Autonomous Systems*, 1995.

[30] B. Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1998.

[31] M. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Utrecht University, the Netherlands, 1992.

[32] M. Overmars and P. Švestka. A probabilistic learning approach to motion planning. In K.Y. Goldberg, D. Halperin, J.C. Latombe, and R.H. Wilson, editors, *Algorithmic Foundations of Robotics*, pages 19–37. A K Peters, 1995.

[33] B. Paden, A. Mees, and M. Fisher. Path planning using a jacobian-based freespace generation algorithm. In *Proc. IEEE Int. Conf. on Rob. and Autom*, pages 1732–1737, 1989.

[34] J. Reif. Complexity of the mover's problem and generalizations. In *Proc. 20th IEEE Symp. on Found. of Comp. Sci.*, pages 421–427, 1979.

[35] F. Thomas and C. Torras. Interference detection between non-convex polyhedra revisited with a practical aim. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, 1994.

[36] 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. In *Proc. IEEE Int. Conf. on Rob. & Aut.*, pages 1024–1031, 1999.