Path Planning in Changeable Environments

Printed in The Netherlands by Ponsen & Looijen B.V., Wageningen

ISBN 978-90-393-4508-5

Path Planning in Changeable Environments

Padplanning in Veranderlijke Omgevingen
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad
van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof.dr. W.H. Gispen,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen
op woensdag 23 mei 2007 des middags te 2.30 uur

door

Dennis Nieuwenhuisen

geboren op 23 augustus 1975
te Gouda

Promotor:	prof. dr. M. H. Overmars
Co-promotor:	dr. ir. A. F. van der Stappen

This work was financially supported by the BSIK-BRICKS program in the context of the Modelling, Simulation and Visualization project (MSV2: Interactive Virtual Environments).

Contents

CHAPTER 1

Introduction

Ever since people were able to use tools, they realized that technology could make their lives easier. The first widespread signs of technology were seen in the stone age (Figure 1.1). People started building the first houses, created pottery to store food and drinks and used arrowheads to hunt for animals with higher success rates than before. Later, in the bronze and iron ages, people started creating more advanced tools which allowed them to advance their agricultural practices and also provided for more sophisticated weapons, both for hunting and warfare.

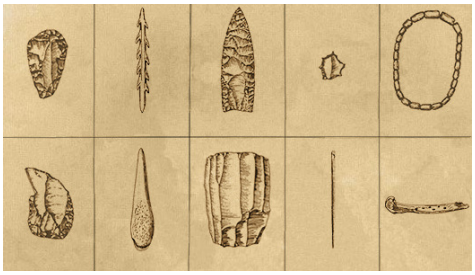


Figure 1.1: A stone age toolkit.

Up to current times, this cycle of development has continued. People invent better tools, which make life easier and allow for advancements. Using these advancements and the time that was saved led to the development of even better tools. Nowadays this cycle has resulted in a broad attention to creating sophisticated machines that are capable of performing complicated tasks. If a machine is able to perform complicated tasks without a human interfering it is usually referred to by the term *robot*. Robotics research com-

prises a broad range of techniques ranging from developing automatic lawn mowers and artificial pets to replacing soldiers by machines and exploring remote worlds by swarms of autonomous mechanical creatures.

In the last decade the usage of the term robot has extended to the domain of virtual environments. In this domain the task of a robot is often to imitate human behavior. A relative simple example is the problem of trying to mimic human conversation by means of a textual interface. In this domain a robot is often referred to as *chatterbot* or just *bot*. An ultimate test of this problem is the *Turing test*. The test is successful if a human is not able to determine whether it is communicating with another human or a chatterbot.

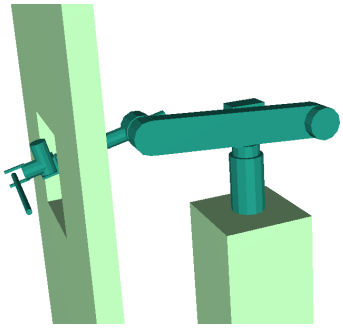
With the development of graphical systems also the need for human-like behavior in the non-verbal sense has increased. Imagine a driving simulation in which people can practice for their drivers license. One would expect the other characters in that simulation, e.g. other drivers, cyclists or pedestrians, to behave as they would do in real life. For example if a pedestrian wants to cross a road, he or she would probably first stand still on the sidewalk, check for approaching traffic and then cross the road in a direction perpendicular to the road. Another example is a game in which a virtual character navigates through a building. The routes it takes, the obstacles it clears or avoids, its behavior around other characters, all should be as human-like as possible for a convincing experience.

The first two parts of this thesis deal with one important aspect of this problem: path planning in a virtual environment. How can a virtual character or robot navigate through an environment? We look at the interesting case where the environment is not static but rather contains obstacles with which the robot can interact. Such environments are useful in games and training simulations. The first part looks at problems in which obstacles can be moved by external factors (e.g. other robots or a human). Because many path planning algorithms assume the world to be static, such changes in the environment are difficult to cope with. The second part considers problems in which the robot itself needs to move obstacles out of the way in order to reach its goal. Because of the many choices involved (which obstacles are candidates to move, where to grasp those obstacles etc.) it is a challenging problem. The third part of this thesis also deals with path planning and looks at the problem of a robot that has the task to push an obstacle to a certain goal position. In this part we explore a new combination of techniques that provides for a novel way of navigating through an environment by pushing. Since the technique is very robust against small errors, it can serve as a basis for new navigation planners in real-world environments.

1.1 The Path Planning Problem

Since a robot R operates in and maybe interacts with an environment its state should be described with respect to this environment. To uniquely describe the state of R within an environment, we need a description of all parameters related to its state: its *degrees of freedom*. These degrees of freedom do not only consist of the robot's position and orientation within the environment, but also of other physical parameters such as the orientations of the links of an artificial arm. A description of all degrees of freedom form a robots *configuration*.

(a) The enemy in a game needs to behave naturally, good path planning is an important part of that (courtesy of id Software).



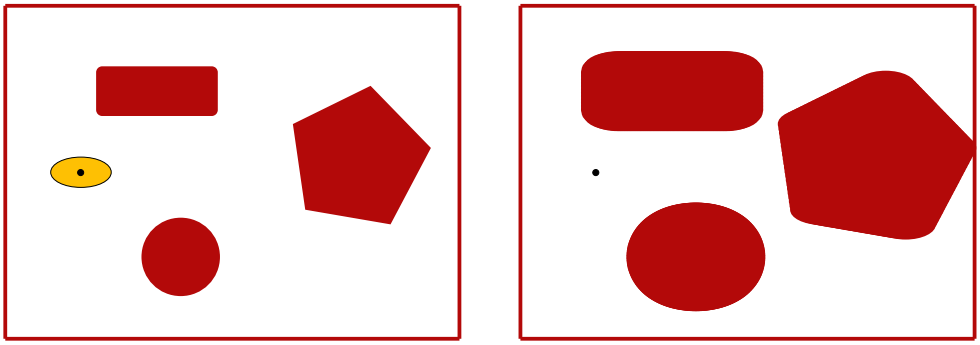
(b) A robot arm needs to maneuver a part through a hole.

Figure 1.2: Examples of path planning problems.

The path planning problem is defined as finding a motion for R from one configuration to another. Stated more formally, it is a *transition* from one configuration to another. Whether R is a robot arm grasping an object, a car-like robot driving to some location or a computer controlled character operating in a virtual environment, it is all essentially the same problem. During a transition the robot is not allowed to collide with any of the obstacles in the environment; the path has to be *collision-free*. Examples of path planning problems are shown in Figure 1.2.

The physical space robot R operates in, is called the *workspace* W. A path planning problem in which R has to maneuver from a start configuration to a goal configuration is usually defined in the *configuration space* C (Udupa, 1977). In C every degree of freedom of the path planning problem corresponds to a dimension, thus a point in C corresponds to a complete configuration of R in W. Every obstacle in W corresponds to an obstacle in C. The union of all obstacles in C form the forbidden space, C_{forb}. Besides obstacles, C_{forb} also consists of configurations that correspond to self-collisions of R. The union of the valid, collision-free configurations is called C_{free}. The path planning problem of a robot among a set of obstacles in W is now translated to planning a (collision-free) path for a point in C. A path is free if it is entirely contained in C_{free}. Because of the many degrees of freedom involved, C is often high dimensional. An example of an environment with its corresponding (two dimensional) configuration space is shown in Figure 1.3.

The path planning problem has been an active topic of research in the last decades. Many planners have been developed, some specific on the type of problems they can handle, some very general. In the book of Latombe (1991) the following distinction is made based on the general approaches of the planners: *roadmap* methods, *cell decomposition* methods and *potential field* methods. The first two can be further classified in *exact* and *approximate* methods. Exact methods are guaranteed to find and report a path if one exists. Also if no path exists, they are capable of reporting so. Since often exact methods are computationally expensive, approximate algorithms were developed. These methods



(a) Environment consisting of 3 obstacles and an oval robot. The center of the robot is used as a reference point.

(b) The corresponding configuration space. The robot is now a point object.

Figure 1.3: Example of a configuration space. The robot is not allowed to rotate.

may fail to find a path even though one exists, but in many situations they outperform exact methods.

Roadmap methods aim at capturing the connectivity of C_{free} in a network of one-dimensional curves. After the construction of the roadmap, it can be used to quickly answer path planning queries. For this, the start and goal configurations are connected to the roadmap which is then searched for a path. Thus, the problem of creating a path between two configurations is reduced to a graph searching problem. Early roadmap algorithms solved the path planning problem exact by creating roadmaps based on different criteria. One of the earliest path planning algorithms is the *visibility graph* method developed by Lee (1978). The idea is to create a graph in a two dimensional polygonal environment by connecting the vertices of the polygons by straight line segments if these segments do not intersect any of the obstacles. In addition the start and goal configurations are added to the roadmap. Finally, the roadmap can be searched for the shortest path.

Another example of an exact roadmap method is the *Voronoi diagram*. In a two-dimensional configuration space, the Voronoi diagram consists of cells such that the points in each cell are closer to a specific feature of the environment than to any other feature. The boundaries of the cells form a roadmap used for planning. An interesting property of this roadmap is that its path maximize the clearance to the obstacles. Voronoi based methods are part of the *retraction* methods in which C_{free} is retracted to a one-dimensional subset of itself. Ó'Dúnlaing and Yap (1982) described an algorithm that is capable of retracting C_{free} to the Voronoi diagram in $O(n^2)$ where n is the number of vertices. Their method is restricted to two-dimensional environments where C_{free} is the interior of a bounded polygonal region. More efficient algorithms exist, e.g. by Yap (1985), that are capable of creating the Voronoi diagram in optimal $O(n \log n)$ time.

The most successful general roadmap based planner is the one created by Canny (1988). It is capable of creating a roadmap in C_{free} in time that is single-exponential in

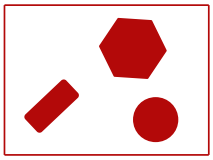
the dimension of C. Unfortunately no implementations exist that are fast enough to be practically usable.

Exact cell decomposition methods subdivide C_{free} in non overlapping cells whose union is exactly C_{free}. A connectivity graph on the adjacency relations between the cells is then created. A search of this graph results in a sequence of cells that connects the start and goal configurations. The general idea is that the cells should be created such that planning a path within a cell is "easier" than planning in the original environment. In two-dimensional, polygonal environments a trapezoidal decomposition provides a straightforward solution. Schwartz and Sharir (1983a) use an exact cell decomposition to solve problems for a straight segment robot in a polygonal environment. Using a Collins decomposition of the free space, Schwartz and Sharir (1983b) described a method that is capable of planning a path for a point robot in any smooth semi-algebraic manifold. Unfortunately their algorithm is too inefficient to be of practical use. To plan the path for a polygon capable of both translation and rotation operating in a polygonal environment, Avnaim et al. (1989) created an exact cell decomposition by decomposing only the boundaries of C_{free} (and some additional subsets of C) into two-dimensional cells. Their method is relatively easy to implement and provides reasonable results in practice.

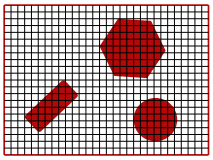
The above methods are especially useful if C can be constructed exactly. If the number of degrees of freedom increases, exact solutions become computationally infeasible because of the high complexity of the problem. In a two dimensional workspace in which R is a point mass, $C = W$. If R has volume and is only allowed to translate, then (given a reference point in R), the obstacles in C are the Minkowski sum of the obstacles in W and R. If R is also allowed to rotate, the shape of C_{forb} becomes complicated. In higher dimensions (e.g. a free flying robot in $\mathbb{R}^3$ or a multiple joint robot arm) creating C_{forb} exactly becomes infeasible because the math to exactly determine intersections of higher dimensional surfaces is non-existent.

Because of the limitations of complete methods, interest shifted to approximate methods. Cell decomposition approaches are also used for planners that only guarantee *resolution completeness* (Brooks and Lozano-Pérez, 1985, Faverjon, 1986, Kambhampati and Davis, 1986). These planners are guaranteed to find a solution to a path planning problem if the resolution of the discretization they choose is high enough (provided that a solution exists). Theoretically these methods apply to the full spectrum of motion planning problems, but in higher dimensions the memory consumption grows quickly. The general idea is to approximate C_{free} by cells, usually boxes. Because of the simple shapes of these cells, the boundaries of more complex obstacles (e.g. with curved boundaries) need to be approximated. Therefore the union of the cells does not exactly represent C_{free} but rather is a subset of C_{free}. The higher the resolution of the subdivision, the better the union of the cells represents C_{free} but the higher the memory consumption. An example is shown as Figure 1.4.

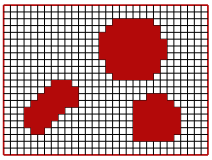
The **potential field method** is another type of approximate planner (Khatib, 1986). If



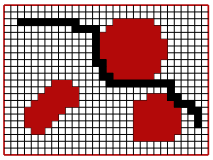
(a) A scene consisting of 3 obstacles.



(b) A regular grid is used to subdivide the environment in cells.



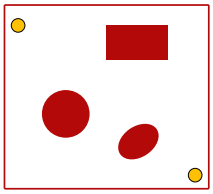
(c) Using the cells, an approximation of C_{free} is created.



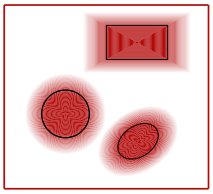
(d) A path is extracted using a shortest path algorithm on the free cells.

Figure 1.4: Example of an approximate cell decomposition.

C is discretized in a fine regular grid, the potential field method guides the search process for a free path. While most implementations are *incomplete* (i.e. they are not guaranteed to find a solution even if one exists) the potential field method has been applied to a broad range of problems. It works with attracting and repulsing forces. The idea is that the robot R is attracted by the goal configuration and repulsed by the obstacles. The distance to the obstacles defines the strength of the repulsing forces; the closer R comes to an obstacle, the larger the repulsing force is. At every iteration of the algorithm the force on R is calculated. The direction of this force is considered the most promising direction of motion. An example of the potential field method is shown as Figure 1.5.

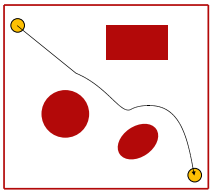


(a) A two dimensional environment with three obstacles. The left top shows the initial position of the robot, bottom right is the goal.



(b) The repulsive potential field generated by the obstacles.

(c) The attractive potential field generated by the goal.

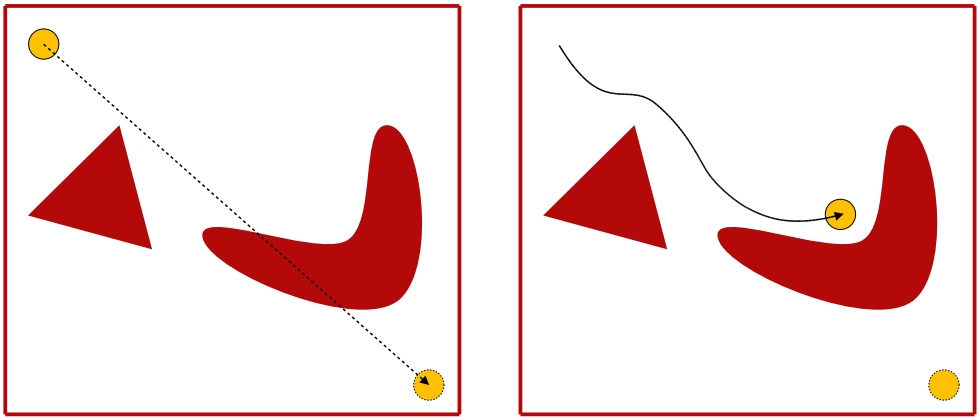


(d) The resulting path generated by the potential field method.

Figure 1.5: Example of the potential field method.

While successful in certain types of environments the potential field method has one major drawback. The robot moves in the direction of a minimum. If this minimum is also the global minimum (i.e. the goal configuration) then the method is successful. If this is not the case, then the robot will get trapped inside this local minimum (Figure 1.6). Much research has been conducted to overcome this problem. Two types of heuristics that aim at solving the problem can be distinguished: reducing the number of local minima by more advanced potential functions (Khosla and Volpe, 1988, Koditschek, 1987, Krogh, 1984) and creating methods to escape local minima (Barraquand et al., 1992, Barraquand and Latombe, 1990). An extensive overview of the field of path planning up to 1990 can

be found in the book of Latombe (1991). More recent work can be found in the books by Choset et al. (2005) and LaValle (2006).



(a) The robot needs to move from the top left to the bottom right.

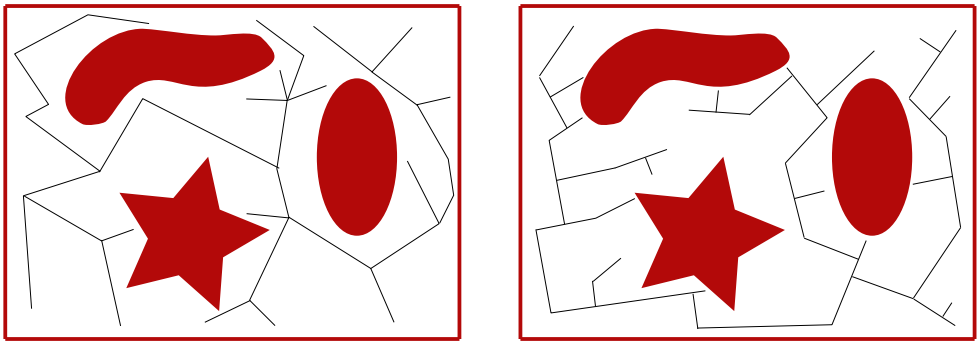
(b) The robot encounters a local minimum and fails.

Figure 1.6: Example in which the potential field method fails.

More recently a new class of roadmap planners has been developed: **sampling-based methods**. These planners are usually *probabilistically complete*. This property guarantees that a solution to a path planning problem will be found if a method is applied sufficiently long (provided that a solution exists). One of the earliest and most successful planners is the Probabilistic Roadmap Method (PRM) developed by Kavraki et al. (1996), Overmars and Švestka (1994). The method constructs a graph (the *roadmap*) of the free space by randomly generating collision-free configurations in C_{free}, this process is called *sampling*. These configurations are connected to their neighboring configurations if a collision-free path exists between them. This process is repeated until the roadmap is an adequate representation of C_{free}. Given a start and a goal configuration, the roadmap can be used to answer path planning queries. First the start and goal are connected to the roadmap. Next a shortest path algorithm (for example that of Dijkstra (1959)) can be used to extract a path (*query*) very efficiently. An example roadmap created by the PRM method is shown as Figure 1.7(a). The PRM method has been successfully applied to a broad range of problems, ranging from lower dimensional car-like problems to planning coordinated paths for multiple robot arms and very high dimensional protein folding problems. In Section 2.1 a more thorough description and an extensive overview of the PRM method and its applications is given.

Another example of a sampling-based path planning technique is the Rapidly-exploring Random Trees (RRT) algorithm developed by LaValle and Kuffner (2001). Here a tree is grown in C_{free} from the start configuration. Initially the tree only consists of the start configuration. A random configuration is generated and the configuration in the tree closest to this random configuration is found. Next, a path is tested for collision starting at the

configuration in the tree. If an obstacle is encountered on this path, the last collision-free configuration on the path is added to the tree. This ensures that the tree stays consistent (i.e. does not fall apart in multiple trees). This process is repeated until the goal configuration can be added to the tree. An example of a roadmap created by the RRT method is shown as Figure 1.7(b).



(a) Example roadmap created by of the PRM method. **(b)** Example roadmap created by of the RRT method.

Figure 1.7: Sampling-based methods.

The fundamental difference between the PRM and the RRT methods is that the first is a *multiple shot* approach, while the latter is a *single shot* approach. A single shot approach aims at solving one path planning query. A start and a goal configuration are given and the algorithm tries to efficiently find a valid path between them. A multiple shot approach aims at capturing C_{free} as good as possible in a *preprocessing* phase, usually by creating a graph. This graph can then be used to answer multiple queries quickly by connecting the start and goal configurations to the graph, the *query phase*. Because a lot of work has already been done in the preprocessing phase, queries can be answered quickly.

In simple, low dimensional configuration spaces where C_{free} can be constructed exactly, basic geometry can be used to check if a configuration or a line segment is collision-free. If the robot and the environment get more complicated, creating C_{free} exactly becomes infeasible and therefore a *collision checker* is used. The collision checker operates in W and is, given a configuration for R in W, able to check whether this configuration is collision-free. A lot of research has been conducted to create efficient collision checkers. Throughout this thesis, whenever a collision checker was necessary, we used Solid. A complete description of this collision checker can be found in the book of Van den Bergen (2003).

While very successful in many applications, sampling-based path planning methods suffer from some drawbacks. The most important one is what is commonly known as the *narrow passage* problem. A narrow passage is a part of C_{free} in which the probability that a random configuration will be generated is very small. Therefore it may take a long time before the two parts of the roadmap on opposite sides of the narrow passage get

connected. An example of a narrow passage is shown in Figure 1.8. Here an L-shaped robot needs to maneuver through a small corridor. To be able to pass this corridor, the robot needs to make a very precise rotation. The robot operates in a two-dimensional environment (two translational degrees of freedom) and is able to rotate (one rotational degree of freedom). The range of the rotational degree of freedom θ is $[0,2\pi)$. Therefore the configuration space $C = \mathbb{R}^2 \times [0,2\pi)$. Since the orientation of the robot is equal if $\theta = 0$ and $\theta = 2\pi$, the corresponding dimension of C is circular, this is usually denoted by S, thus $C = \mathbb{R}^2 \times S^1$. In C the problem corresponds to a point having to move through a long narrow corridor. If a sampling based approach is used, a precise sequence of configurations is required. It may take a long time before this sequence is generated by a sampling based algorithm. A lot of research has been aimed at providing solutions for this narrow passage problem. Most of them try to focus the sampling process toward the narrow passages. More details of these methods can be found in Section 2.1.

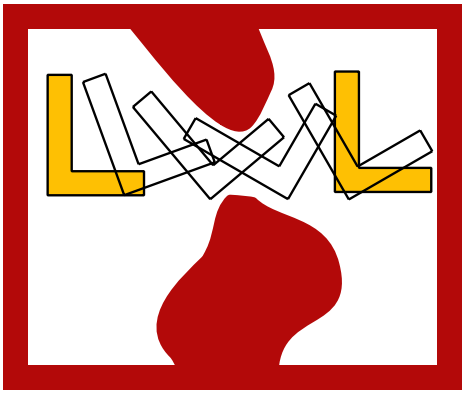


Figure 1.8: Example of a narrow passage. The L-shaped object has to move from left to right. It needs a precise rotation to pass the small corridor.

All methods described above assume that the environment is static. This means that between the time the roadmap, cell decomposition or potential field was created and the moment of the query, the environment is not supposed to change. If it does change, for example because an obstacle is moved, then usually the method has to be restarted. Although for the PRM method, variants have been proposed that aim at making the roadmap more robust against changes in the environment, planning in true dynamic environments remains an active topic of research.

This thesis addresses three scenarios in which existing sampling-based path planning techniques are inefficient or even fail. All three parts of the thesis concern techniques to efficiently do path planning in *changeable* environments. In changeable environments obstacles and objects are allowed to change their configuration after the preprocessing phase. These changes can have external causes (e.g. someone moves a chair or closes a door), but they can also be initiated by the robot itself. Part I of this thesis is aimed at path

planning in environments in which objects can be moved by external causes. Instead of repairing the roadmap after obstacles have changed their positions, which involves expensive collision checks, we propose a solution that makes the roadmap *robust* against these changes. The query phase is therefore still fast, allowing for interactive performance. Part II deals with environments in which the robot itself is able to manipulate the environment. The task of the robot is to navigate through an environment consisting of both static and *movable* obstacles. If such a movable obstacle blocks the path of the robot, it is allowed to move the obstacle out of the way by using pushing and/or pulling actions. Finally, in Part III we consider the problem in which the robot needs to push a passive object to a goal configuration. We explore a novel technique in which *compliance* is used as an aid for the robot to push the object. A compliant motion is defined as a motion in which the object is allowed to slide along the boundaries of the environment. This way of navigating provides for more robustness against uncertainty, allows for paths that would not be possible otherwise and also solves the narrow passage problem in many cases.

1.2 Overview of this Thesis

In this thesis we deal with changeable environments. In the first two parts, the algorithms base their navigation on the PRM method. In Chapter 2 we propose a variant of this method that serves as a basis for path planning in changeable environments. To save costly collision checks, a PRM roadmap usually does not contain cycles. As a result there is at most one path from a configuration to another. If this path is blocked by a moved obstacle, the method ends in failure. Adding additional edges (and thus creating cycles) seems a good solution, but is computationally expensive. Therefore we need to carefully select the edges and only add those edges that contribute to making the roadmap more robust against placement changes of obstacles. An additional advantage is that often paths become shorter because the robot has the choice between several alternative routes. Our algorithm is capable of creating roadmaps that only contain "useful" cycles at the expense of only a small increase in running time. Useful cycles are cycles that have a high probability of contributing to creating alternative paths. Experiments are presented that show that our roadmaps are more robust against placement changes of obstacles then roadmaps without useful cycles. We also provide an upper bound on the length of the path in the query phase compared to the theoretical optimal path. The work in Chapter 2 is largely based on the work published in the proceedings of the *IEEE International Conference on Robotics and Automation* 2004 (Nieuwenhuisen and Overmars, 2004).

Part I consists of two chapters. Chapter 3 describes an algorithm that is capable of creating roadmaps that guarantee to find a path regardless of the positions of the obstacles (provided that a path exists). While in Chapter 2 we do not assume anything about the environment, in Chapter 3 we do; we use the observation that although obstacles can be moved, they are often confined to small areas. For example a chair can be placed anywhere in a room, but is rarely taken out of it. Even more confined is a door that can be open or closed. An example of such an environment is shown in Figure 1.9. Using this

Figure 1.9: Example of an environment in which obstacles can be moved but are usually confined to small areas.

observation, we describe a theoretical framework that guarantees the existence of a path in the roadmap provided one exists regardless of the configurations of the obstacles. A roadmap is created such that if an obstacle blocks a path, an alternative path is available. The results can be combined with those of Chapter 2 to create a roadmap that not only guarantees to find a path but is also capable of creating relatively short paths. We also present experiments that show the effectiveness of our approach. The major part of the results of Chapter 3 is based on the paper published at the proceedings of the *IEEE International Conference on Intelligent Robots and Systems* 2005, (Van den Berg et al., 2005).

Although in Chapter 3 we propose a straightforward method to implement the framework, it only works in simple environments. If the number of obstacles gets too high, the running time increases drastically. In Chapter 4 we propose a different approach of implementing the framework by introducing an algorithm based on Boolean logic that is capable of efficiently creating robust roadmaps. First we describe how the problem can be translated to the domain of Boolean logic, next we show how this translation can be used to efficiently decide which edges should be allowed in the roadmap and which not. Experiments show that our algorithm scales very well to larger environments. The results of this chapter have been previously unpublished.

Part II deals with environments in which the robot is allowed to move obstacles that obstruct its path out of the way. The goal of a query is defined in terms of a configuration for the robot in an environment that consists, besides stationary obstacles, also of movable obstacles. These obstacles are not capable of moving by themselves, but they can be moved by the robot. The robot needs to decide which obstacles to move such that it is able to reach its goal. An example of such an environment is shown in Figure 1.10. Note the difference with *rearrangement* problems in which the goal is defined in terms of a goal configuration for the movable obstacles. Path planning among movable obstacles is a challenging problem because most obstacles will not block the path of the robot directly but may block other obstacles from being moved. First, in Chapter 5 we describe a framework that creates a structured action space. This action space is a description of all possible actions the robot can perform in all possible orders. Extensively searching

Figure 1.10: Example of a door that is blocked by obstacles.

the action space will provide a solution to a problem provided one exists. Unfortunately, in all but the simplest problems this is computationally infeasible. Therefore, using the framework, in Chapter 6 we describe heuristics that drastically decrease the search space such that a broad range of problems can be solved efficiently. The roadmaps generated with the algorithm of Chapter 2 serve as a basis for the path planning of the robot. This is combined with local solutions for the manipulation of the obstacles while a global planner coordinates which and when obstacles are manipulated. Our experiments show that the approach works for a broad range of realistic problems. The major part of the results presented in this part has been previously published in the proceedings of the *Seventh International Workshop on the Algorithmic Foundations of Robotics* 2006, (Nieuwenhuisen et al., to appear 2006).

The final part, Part III, describes a novel approach that combines pushing and compliant motions. Pushing has been a widely studied subject. Usually an object, incapable of moving by itself, is pushed by a *pusher* that is capable of moving by itself. A pushing motion can serve different goals. In Part II a robot can push (or pull) an obstacle out of the way in order to create room to maneuver. In Part III, we look at the problem where the goal is defined in terms of a configuration for the object that is pushed. Thus the task of the pusher is to push an object to a certain goal configuration. In real world applications, pushing is a difficult problem for several reasons. First, since many parameters are not exactly known, the motions of the object while being pushed by the pusher are hard to predict. These parameters include friction between the pusher and the object, but also between the object and the environment. Also, the mass distribution of the object is often not exactly known. Another source for uncertainty is sensor errors. For example odometers can have small deviations such that the actual position of a robot is different from the intended one. A well-known technique used to compensate for uncertainty is the usage of *compliant motions*. These are motions in which an object slides along the boundaries of

the environment. Using this type of motions fixes certain degrees of freedom of the object such that no uncertainty can occur. Previously compliance has, for example, been used to solve the peg-in-hole problem. In this part we use compliant motions to compensate for uncertainty in pushing problems. An additional advantage of compliant motions is that the method suffers much less from the narrow passage problem.

Compliance combined with pushing raises interesting questions of which some are described in Chapter 7. In addition previous work is described and preliminaries are given that are used in the next chapters. The most important question is: given a goal position for the object, what is the path for the pusher such that if the pusher follows this path, the object is pushed to its goal. We show that creating an exact solution for this problem is infeasible. Therefore, in Chapter 8 we present an exact algorithm that is capable of solving a subproblem, namely creating a path for the pusher given a path for the object. This object path is allowed to consist of both compliant and non-compliant path segments. An example of such a path is shown in Figure 1.11.

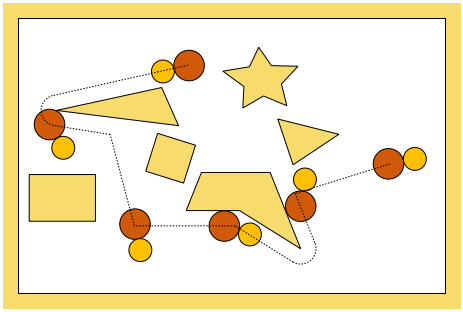


Figure 1.11: Example of a push path (top view). Starting from the right, the object (larger disk) follows the dotted line along which it is pushed by the pusher (the smaller disk). At certain points the object and pusher are shown. At some parts of the path the object slides/rolls along the boundary of the environment.

In Chapter 9 we build on the results of Chapter 8 to create an efficient algorithm for the more general case in which only a start and goal configuration for the object are given. The algorithm is then capable of creating a push plan such that the object is pushed to its goal. In this push plan, the pusher is allowed to make use of compliance between the object and the environment. Since an exact algorithm is infeasible, we combine random exploration of the non-compliant space with an exact computation of compliant space. This random exploration is based on the RRT algorithm. The RRT also creates compliant configurations that are used as starting points for compliant paths. We present experiments that show the effectiveness of using compliance.

The results in this part are based on the work published in the proceedings of the *IEEE International Conference on Intelligent Robots and Systems* 2005, (Nieuwenhuisen et al., 2005) and the proceedings of the *IEEE International Conference on Robotics and Automation 2006*, (Nieuwenhuisen et al., 2006). In addition the results are published in the *IEEE Transactions on Robotics*, (Nieuwenhuisen et al., to appear 2007).

CHAPTER 2

Probabilistic Roadmaps and Cycles

To be able to manipulate its environment, first of all a robot needs a way to navigate through this environment. A popular method for navigation in static environments is the Probabilistic Roadmap Method. It creates a graph of the free space of the environment that can be used to navigate a robot. This graph can be re-used for multiple queries. Since the lions share of the work is done in a preprocessing phase, the method provides for a very fast query phase. In a changeable environment, not all obstacles are stationary to a certain configuration but rather are allowed to change their position. If a graph is used as a basis to navigate a robot in such an environment, it may be that parts of the graph become infeasible because of the motions of the obstacles. Finding solutions locally (e.g. if an obstacle is encountered, try to avoid it) would cause the query time to increase dramatically. If the robot would have the choice between several alternative routes through the graph however, then if one of the routes is blocked (e.g. a door is closed), an alternative can be used. Unfortunately standard implementations do not provide for such alternative routes because, to save expensive collision checks, graphs usually do not contain cycles.

In this chapter we propose an extension to the Probabilistic Roadmap Method that makes the graph more robust against changes in the environment by providing alternative routes between configurations. An additional advantage is that the path length in the query phase is reduced without any additional computations, thus preserving the property of a fast query phase. The graphs created in this chapter serve as a building block for the algorithms in parts I and II.

First, in Section 2.1 we introduce the Probabilistic Roadmap Method. In Section 2.2 existing techniques to add cycles to the graph are discussed and in Section 2.3 our cycle-adding method is described. Next, in Section 2.4 we prove an upperbound on the length of the extracted paths. Finally we conduct experiments in Section 2.5 and draw conclusions in Section 2.6.

2.1 Introducing the PRM Method

The Probabilistic Roadmap Method (PRM) (Kavraki et al., 1996) has become one of the leading path planning techniques in the field of robotics, both in virtual and real-world contexts. Recently its applications have extended to the domain of computer-assisted training, advanced gaming (Kamphuis and Overmars, 2004, Nieuwenhuisen et al., 2007) and biology (see e.g. Song and Amato (2001)). Its main features are its simplicity, allowing for almost instantaneous queries, extensibility to higher dimensions and the broad range of problem types to which it is applicable.

The general idea of the PRM method is to create a topological graph $G = (V, E)$, the *roadmap*. Here, V is a set of vertices and E is a set of paths between those vertices in C_{free}. The roadmap aims at representing the connectedness of C_{free}. Ideally if the roadmap adequately represents the connectedness of C_{free} then given a configuration $c \in C_{\text{free}}$ a path from c to a configuration in G can be computed efficiently. The roadmap can then be used to efficiently answer path planning queries.

The main loop of PRM generates random configurations $c \in C_{\text{free}}$ (*sampling*). These collision-free configurations form the vertices V in the roadmap. Next, a set $N_c \subset V$ of neighbor configurations of c, is selected (Kavraki et al., 1996). If a path exists between two vertices in G, these vertices are said to be in the same *connected component* of G. To save costly collision checks, at most one neighbor per connected component is allowed in N_c. A *local planner* is used to try to connect c to each $c_i \in N_c$. If the local planner succeeds, the connection is added to the list of edges E in G. Thus, an edge in G represents the existence of a collision-free path between two vertices. The construction of the roadmap is shown in pseudo code in Algorithm 2.1.

Usually, the local planner is kept as simple as possible, allowing for fast collision detection. Typically the local planner checks if the straight line connection between two configurations in C is entirely contained in C_{free}. Practically this means that using the collision checker, a recursive binary strategy (Geraerts and Overmars, 2004) is used to step along the line between two configurations to check if the path is collision-free. For the translational degrees of freedom this is straightforward, for the other degrees of freedom, a number of different strategies can be used. For a path between two configurations with two translational and one rotational degree of freedom for example, the rotation can be evenly interpolated along the path, but also the *rotate-at-s* (Amato et al., 1998a) approach can be used. Rotate-as-s first translates the robot to an intermediate configuration s somewhere along the straight line path, then rotates and finally translates to the endpoint. More details and a comparison of local planners can be found in (Dale, 2000, Geraerts and Overmars, 2004, Sánchez and Latombe, 2001).

In order to find the set of neighbor configurations, a distance measure $d(c, c_i)$ is needed to define the distance between configurations c and c_i. For translational degrees of freedom, the Euclidean distance is an appropriate measure. For rotational degrees of freedom this is not so trivial. A lot of effort has been put in finding an appropriate distance measure (see for example Amato et al. (1998a)), but often the choice is problem dependent. Note that we assume that the distance measure satisfies the triangle inequality as

Algorithm 2.1 CONSTRUCTROADMAP

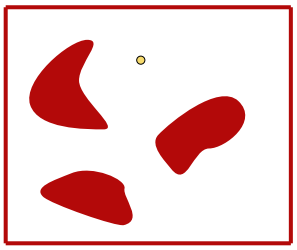
Let: $V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
1: **loop**
2: $c \leftarrow$ a (random) configuration in C_{free}
3: $V \leftarrow V \cup \{c\}$
4: $N_c \leftarrow$ a set of neighbor vertices chosen from V
5: **for all** $c_i \in N_c$ **do**
6: **if** the line (c, c_i) is collision-free **then**
7: add the edge (c, c_i) to E

is usually the case. If a distance measure has been found, connections are typically only tried to neighbors within a certain distance, the *maximum neighbor distance*. Moreover the number of neighbors is often limited (e.g. at most n connections are added to a configuration). An example of the creation of a PRM is shown in Figure 2.1(a)…(e).

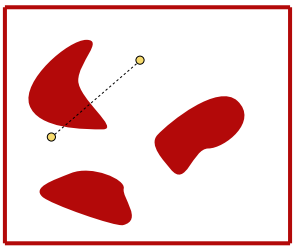
After adding a number of vertices and edges to G, the roadmaps tends to get connected and reflects the connectedness of C_{free}. Now G can be used to solve path planning queries. First the start c_s and goal c_g configurations of the query are added to G using the local planner. Then a simple shortest path algorithm (like Dijkstra's algorithm or A* (Hart et al., 1968)) can be used to find the shortest path $\tau : [0,1] \rightarrow C_{\text{free}}$ in the roadmap such that $\tau(0) = c_s$ and $\tau(1) = c_g$ (Figure 2.1(f)).

Since τ consists of straight line segments, if it is executed by the robot it usually results in erratic and redundant motions in the sense that the path is not smooth and may take detours. Therefore often a post processing step called *smoothing* is applied to improve the quality of the path. The most basic form of smoothing repeatedly tries to create shortcuts in the query path, thus removing redundant local detours from the path. This is done by selecting two random configurations on τ and using the local planner to check if a shortcut exists, this is shown in Figure 2.2. More advanced smoothing techniques try to remove C^1 discontinuities from τ (Nieuwenhuisen et al., 2007) such that the derivative of τ is continuous or try to create high quality roadmaps in the preprocessing phase (Geraerts and Overmars, 2006).

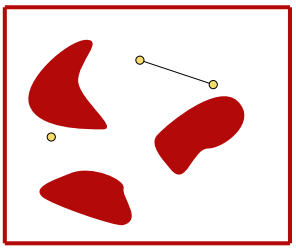
Over the years many extensions to the original PRM method have been proposed. Many of these address the narrow passage problem by using a non-uniform sampling distribution that aims at creating more samples in "difficult" areas. Obstacle Based PRM, developed by Amato et al. (1998b) works the same as standard PRM with the exception that when a forbidden configuration is sampled, a random direction is chosen and the robot is moved in that direction in an attempt to find a collision-free configuration. If this succeeds, the configuration is added as a vertex to the roadmap. This procedure results in additional vertices close to obstacles. Another method to generate more vertices close to obstacles is the Gaussian sampler (Boor et al., 1999). The idea is to only allow configurations in the roadmap that are close to a colliding configuration. This results in a Gaussian distribution of the vertices in the roadmap with the number of vertices decreasing with the distance to the obstacles. The bridge test (Hsu et al., 2003) aims at focusing the sam-



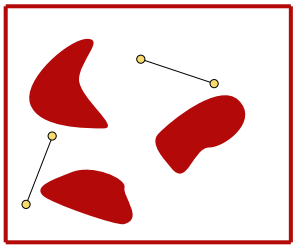
(a) A first random collision-free configuration has been found.



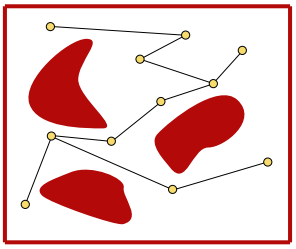
(b) A second collision-free configuration has been found. The path between the two configurations is not collision-free.



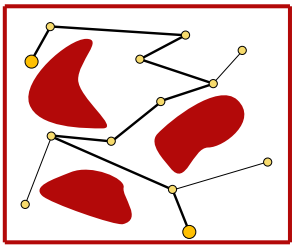
(c) A third configuration is found. A collision-free path exist to another configuration. This path is added to the roadmap as an edge.



(d) Another configuration is added.



(e) The process is repeated until a roadmap has been created.



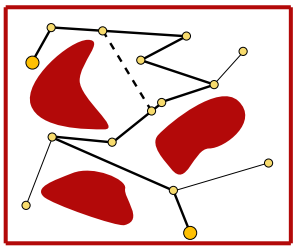
(f) Two query points are added to the roadmap and a path is extracted.

Figure 2.1: The creation of a PRM (a)…(e) and an example of a query (f).

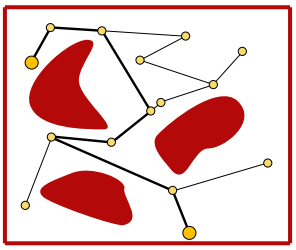
pling process toward narrow passages. Configurations that collide with obstacles are used in pairs. These pairs should consist of configurations that are close together. If there is a configuration between them that is collision-free, then we know this configuration is in a narrow passage. In that case, the configuration is added as a vertex to the roadmap.

Another key issue has been to keep the number of vertices in the roadmaps small. The visibility-based PRM was developed by Siméon et al. (2000) to only add nodes to the roadmap that extend the visibility of C_{free}. The idea is that a configuration is necessary only if it can be connected to either zero or more than one other connected components. The reachability roadmap method (Geraerts and Overmars, 2005) adds a minimal set of vertices to the roadmap. The computation of the set is based on coverage criteria.

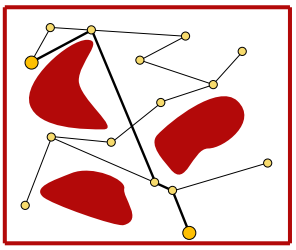
Other improvements that have been suggested include the one of Bohlin and Kavraki (2000) who postpone collision checks to the query phase, effectively turning the PRM into a single shot method. Hsu et al. (1999) create a roadmap in a dilated C_{free} such that the robot is allowed to penetrate the obstacles somewhat. Next the roadmap is modified by sampling around configurations that do not lie in C_{free}. Wilmarth et al. (1999) retract configurations to the medial axis to increase the probability of passing narrow passages. Branicky et al. (2001) investigate the usage of quasi-random sampling instead of true random sampling. They claim better performance and hope to be able to derive deterministic



(a) On the path τ two random configurations are chosen and using the local planner a shortcut is tested for collision.



(b) The shortcut was collision-free so it is incorporated in τ.



(c) The path τ after repeatedly creating shortcuts.

Figure 2.2: Example of the basic smoothing procedure.

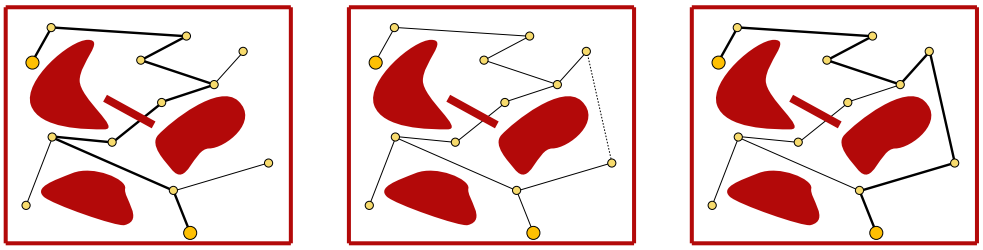
bounds on the performance of the planner. An analysis of the performance of various PRM variants can be found in the paper by Geraerts and Overmars (2002).

2.2 Connection Strategies

Since the local planner is by far the most computationally expensive step of the PRM method, its number of executions should be kept as low as possible. If there is already a path in G from one vertex to another, adding an edge between these vertices does not contribute to the connectivity of the roadmap, so usually only edges that connect two different *connected components* are allowed in G. Since collision checks are costly to perform, maintaining connected components is relatively cheap. The result of this strategy is that G is a forest and there is at most one path between two vertices in G. This makes such a roadmap unsuitable to be used in changeable environments because G does not provide an alternative route if an obstacle changes its position and invalidates one of the edges of a path.

To introduce alternative routes in the roadmap, we need the algorithm to allow cycles such that if a path is blocked by an obstacle, an alternative path is likely to exist (Figure 2.3). Previously cycles have been allowed in the roadmap to provide for better (i.e. shorter) paths. To add cycles to the roadmap, a straightforward approach is to connect every vertex to multiple neighbor vertices. This is called the *nearest-n* method. It simply tries to connect to all n nearest neighbor vertices. While this method succeeds in creating alternative routes between vertices, the running time of the PRM method increases quickly because of the increased number of calls to the local planner. *Component-n* tries to connect to n vertices in each connected component in the neighbor set. Both methods suffer from the drawback that while they do add cycles to the roadmap, these cycles are usually small compared to the size of the environment and they often do not attribute to providing alternative routes to the roadmap. Geraerts and Overmars (2004) provide an overview and comparison between different connection strategies.

To add cycles to the roadmap while keeping the increase in running time low, we need



(a) The path τ is blocked by an obstacle.

(b) If the dotted edge would have been part of G, then an alternative route would exist.

(c) The alternative route.

Figure 2.3: Example in which a cycle in the roadmap can provide a useful alternative route.

to make a distinction between useful and useless cycles. Previously *homotopy* has been used to make this distinction. Schmitzberger et al. (2002) propose a technique to list all homotopy classes of an environment.

Definition 2.2.1 (Homotopy)**.** *Two paths τ_1 and τ_2 are said to be in the same* homotopy *class if and only if τ_1 can be continuously deformed in τ_2 in C_{free}.*

Unfortunately, homotopy is not always a good method to make a distinction between useful and useless cycles in higher dimensions than $\mathbb{R}^2$. This is clarified in Figure 2.2. In this figure, while both the solid-line and dotted-line paths are in the same homotopy class, both paths are useful because they both provide a different path around the obstacle.

In the next section we will state a different definition to distinguish between useful and useless cycles.

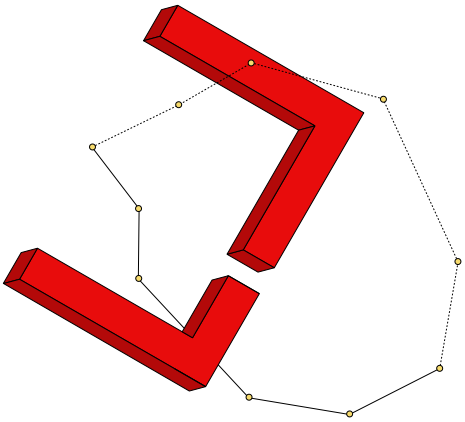


Figure 2.4: While both the solid and the dotted paths are part of the same homotopy class they are both useful.

2.3 Useful Cycles

To create alternative routes between vertices in G, we need to add extra edges (creating cycles) while minimizing the increase of the running time. Edges that provide for an alternative route that is only marginally different from the existing routes are not favored. Assume that, given two configurations, the local planner creates straight line connections in C between those configurations. Let $d(c,c')$ be the shortest Euclidean distance between configurations c and c' in C. Let $G(c,c')$ be the graph distance between vertices c and c' in the roadmap which is defined as the length of the shortest path in the roadmap from c to c'. If there is no path in G from c to c' we set $G(c,c')$ to ∞. We will now introduce the term *K-useful edge*.

Definition 2.3.1 (K-useful edge)**.** *Let c be a randomly chosen configuration in C_{free} and N_c its set of neighbors. For a neighbor $c_i \in N_c$ we say that the edge (c,c_i) is K-useful if:*

$$K \cdot d(c,c_i) < G(c,c_i) \text{ or } G(c,c_i) = \infty. \tag{2.1}$$

We will talk about useful edges, when the dependence on K is implicitly understood. If no path exists between two vertices c and c' in G, the edge (c,c_i) is useful by definition. If a path already exists in G between two vertices, the edge (c,c_i) is only added if it is useful. In the latter case a cycle is created which is called a *useful cycle*. So only cycles that improve the graph distance between c and c_i by a substantial factor K are added to G. A *simple cycle* has no repeated vertices except for its start and end vertices. Its length is the sum of the lengths of its edges. Our definition of a useful cycle is adaptive to the local density of the sampling. If a variant of PRM is used that does not use a uniform sampling distribution, but for example adds additional vertices in narrow passages, then the length of the simple cycles in those areas will be shorter on average. Simple cycles in areas with fewer vertices will have larger lengths on average. Changing the value of K influences the number of cycles that is added to the roadmap. A small value of K adds more cycles, a large value of K adds less. If $K \leq 1$, all edges are allowed, since $G(c,c_i)$ can never be smaller than $d(c,c_i)$. If $K = \infty$, the algorithm behaves as regular PRM and the resulting roadmap is a forest.

A major algorithmic question is how to compute $G(c,c_i)$ efficiently. Since adding a new edge to the roadmap can influence all existing graph distances, these cannot be maintained easily and a shortest path algorithm (like Dijkstra) has to be executed for every potential connection, in order to calculate $G(c,c_i)$. When the number of vertices in the graph becomes large, the (re-)computation of Dijkstra's shortest path algorithm will become a dominating factor in the running time. A solution is to use a dynamic all-pairs shortest path algorithm (e.g. Demetrescu and Italiano (2001, 2003)). However, since the search can be pruned, we use a simpler approach.

We proceed in a way similar to the A* algorithm. Suppose we want to know if edge (c_i,c_j) is a useful edge. Dijkstra calculates the shortest graph distance from the source to all other vertices. But we do not need the exact distance; we are only interested to know whether $K \cdot d(c_i,c_j) < G(c_i,c_j)$. Since Dijkstra's algorithm visits vertices by increasing graph distance from c_i, we know that when vertex c_k is selected by Dijkstra's algorithm (and c_j has not yet been selected), $G(c_i,c_j) \geq G(c_i,c_k)$. We can use this property to prune the

search, let

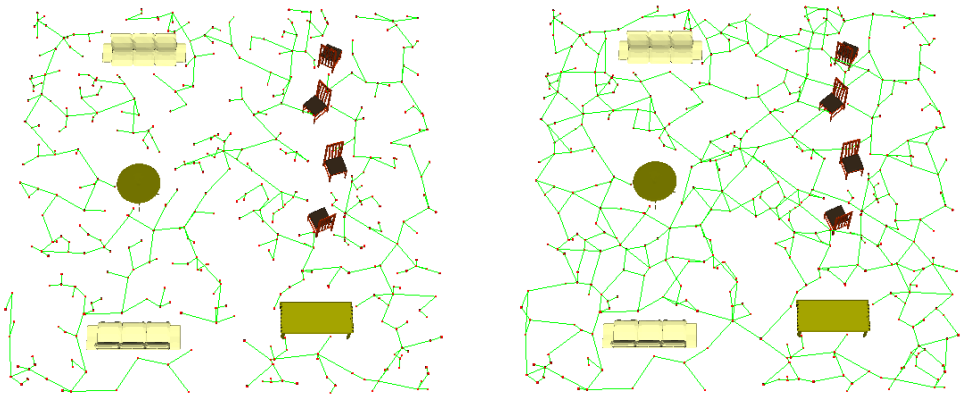
$$G_d(c_i, c_k) = G(c_i, c_k) + d(c_k, c_j). \tag{2.2}$$

We now let Dijkstra's algorithm select vertices based on the value of G_d. If vertex c_k is selected, it gets a value equal to $G(c_i, c_k)$. $G_d(c_i, c_k)$ is a lower bound of $G(c_i, c_j)$ because, once Dijkstra selects c_k, we know that it takes at least another $d(c_k, c_j)$ to reach c_j.

If $G_d(c_i, c_k)$ of a vertex c_k is larger than $K \cdot d(c_i, c_j)$ we can stop the execution of Dijkstra's algorithm, because we know that $G(c_i, c_j) > G_d(c_i, c_k)$. In this case the edge (c_i, c_j) will be added to G. If Dijkstra's algorithm selects c_j before the threshold is reached, we also stop the search and do not add the edge.

The resulting graph is no longer a tree because of the cycles. We will refer to this pruned version of Dijkstra's algorithm as the *usefulness test*. In the query phase the normal version of Dijkstra's algorithm can be used to find the shortest path. Finally, standard smoothing can be used to optimize the path.

In practice there is a *maximum number of neighbors* M_n to which a vertex is connected. If V is the collection of vertices, then, after preprocessing, the maximum number of edges in the graph, is $M_n \cdot |V|$. During preprocessing the worst case running time of Dijkstra's algorithm is dependent on $|V|\log(|V|)$. Examples of graphs both without and with cycles are shown in Figure 2.5.



(a) A graph without cycles.

(b) A graph consisting of the same vertices but with cycles.

Figure 2.5: Examples of a graph without and with cycles.

2.4 Theoretical Results

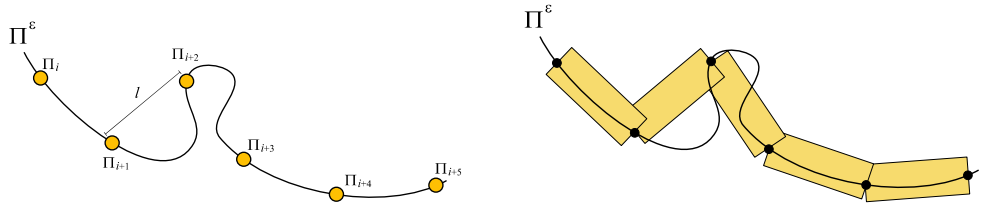
The basic PRM algorithm is known to be *probabilistically complete*. This means that if a path between two configurations exists, the algorithm is guaranteed to find it if time goes to infinity (Švestka, 1997). Nothing can be guaranteed about path length however if edges

are only allowed between different connected components. In this section we will show that, using the usefulness property, we can prove that the path length resulting from our algorithm converges to K times the optimal path length.

Let $\epsilon > 0$ be a small clearance and let Π^ϵ be the shortest path with distance at least ϵ between a configuration of Π^ϵ and its closest obstacle using the distance measure. We assume that ϵ has been chosen small enough such that Π^ϵ exists. Let $L(\Pi^\epsilon)$ denote the length of this path. We will show that, when time goes to infinity, our algorithm will find a path with a length converging to $K \cdot L(\Pi^\epsilon)$. Let $\delta > 0$ be a sufficiently small constant. We will approximate the path Π^ϵ with (hyper-)cylinders of radius δ. The endpoints of the axes of the cylinders connect configurations of Π^ϵ. Choose the cylinder axis-length l such that

$$\sqrt{l^2 + \delta^2} < \epsilon. \tag{2.3}$$

We pick points Π_i on Π^ϵ as follows: Π_0 is the start point of the path. Next, as Π_1 we pick the first point on the path that has a straight-line distance l from Π_0. Π_2 is the first point at distance l from Π_1, etc. (Figure 2.6(a)). Finally we add the goal point of the path. Assume that we added $n+1$ points in total and thus approximated Π^ϵ with n cylinders. The centers of their bottom and top are the points Π_i (Figure 2.6(b)) and the cylinders have radius δ.



(a) The path Π^ϵ divided in parts with straight line distance l using points Π_i.

(b) Using the points Π_i to approximate Π^ϵ with cylinders.

Figure 2.6

Lemma 2.4.1 (Collision-free cylinder)**.** *The cylinders as defined above are guaranteed to be collision-free.*

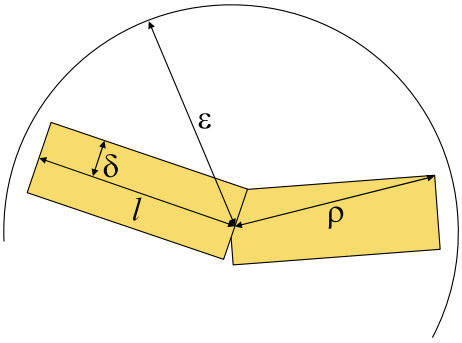
Proof: Let Π_i be the start point of the cylinder. Let ρ be the distance from Π_i to the furthest point in the cylinder. By the choice of l and δ we have

$$\rho = \sqrt{l^2 + \delta^2} < \epsilon. \tag{2.4}$$

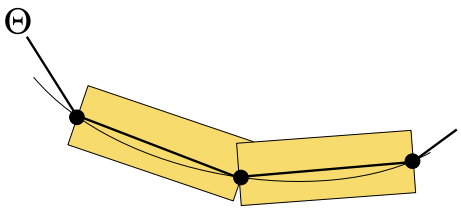
As point Π_i lies on the path Π^ϵ it has by definition a clearance of at least ϵ. Hence the cylinder must be collision-free (Figure 2.7(a)). ■

Consider the piecewise linear path Θ that is created by connecting every Π_i to Π_{i+1} by a straight line section in configuration space (Figure 2.7(b)). It is easy to see that

$$L(\Theta) \leq L(\Pi^\epsilon). \tag{2.5}$$



(a) Because ρ is smaller or equal than ϵ, the cylinders are collision-free.

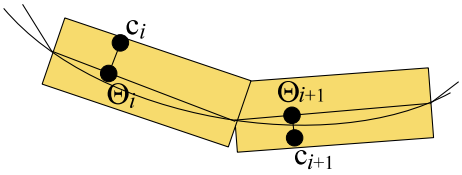


(b) Connecting the points Π_i creates Θ.

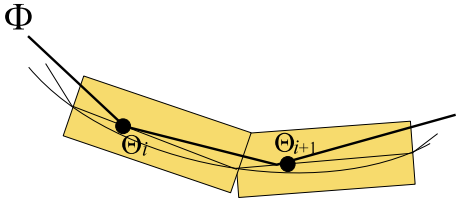
Figure 2.7

Now assume that after long enough sampling, each cylinder i contains a sample c_i. Let Θ_i be the closest point on Θ from c_i (Figure 2.8(a)). Consider path Φ that is constructed by connecting every Θ_i to Θ_{i+1} with a straight-line motion (Figure 2.8(b)). Since Θ_i and Θ_{i+1} are both within the collision-free sphere of Figure 2.7(a), the straight line path (Θ_i, Θ_{i+1}) is also collision-free. It is easy to see that

$$L(\Phi) \leq L(\Theta). \tag{2.6}$$



(a) The point Θ_i is the closest point on Θ from c_i.



(b) Connecting the points Θ_i to create the path Φ.

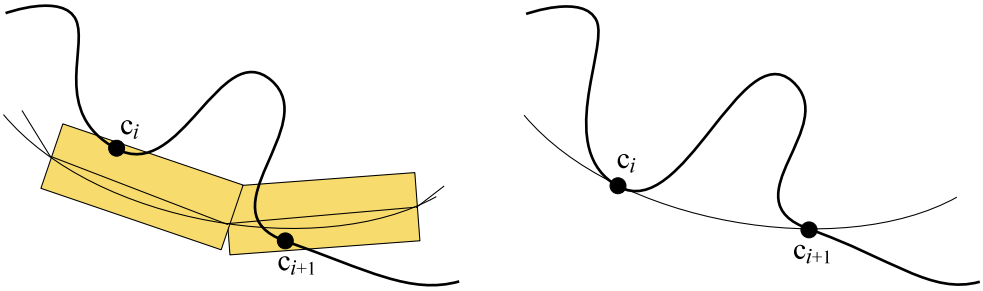
Figure 2.8

Let c_i and c_{i+1} be the samples in two consecutive cylinders. Clearly their distance can be bounded by:

$$d(c_i, c_{i+1}) \leq 2 \cdot \delta + d(\Theta_i, \Theta_{i+1}). \tag{2.7}$$

Since the straight-line connection between c_i and c_{i+1} is collision-free, we know from the usefulness property that there is a bound on their graph distance (Figure 2.9(a)), combining with Equation 2.7 yields

$$G(c_i, c_{i+1}) \leq K \cdot d(c_i, c_{i+1}) \leq K \cdot (2 \cdot \delta + d(\Theta_i, \Theta_{i+1})). \tag{2.8}$$



(a) The graph distance between c_i and c_{i+1} is bounded.

(b) If the radius δ of the cylinders approaches 0, the points c_i approach Π^ϵ.

Figure 2.9

Because Equation 2.8 holds for every segment we can bound the graph distance as follows using Equations 2.5 and 2.6.

Theorem 2.4.1 (Graph distance)**.** *The graph distance $G(c_0, c_n)$ is bounded by*

$$G(c_0, c_n) \leq K \cdot (2 \cdot n \cdot \delta + L(\Phi)) \leq K \cdot \left(2 \cdot n \cdot \delta + L(\Pi^\epsilon)\right). \tag{2.9}$$

If we decrease the radius of the cylinders and thus let δ approach 0 (Figure 2.9(b)), the graph distance between c_0 and c_n approaches $K \cdot L(\Pi^\epsilon)$. When ϵ approaches 0 as well, the graph distance approaches K times the length of the optimal path Π^0.

For each $\epsilon > 0$ and $\delta > 0$ the cylinders have an equal constant-size volume. So if time goes to infinity the probability that each cylinder indeed contains a sample approaches 1. (Actually, this is not true for the last cylinder but that cylinder already contains the goal configuration as a sample.)

2.5 Experimental Results

Our algorithm has been implemented in (Visual) C^{++} using a Pentium 2.4GHz with 1GB internal memory. To test our algorithm we have used several benchmark environments (see Figure 2.10(a)…(d)) each of which will be detailed below.

Environment 1 This is an environment with a number of boxes having small passages between them. We have used this environment to create a two dimensional roadmap. The robot used is a small, complicated object consisting of 11290 triangles. The query we have used was to move the robot from the bottom left to the top right position in the environment.

Environment 2 This environment consists of 500 randomly distributed tetrahedra. The robot is an L-shaped object which needs rotation to maneuver through the environment. The whole environment consists of one homotopy class. The task for the robot is to maneuver from one corner of the environment to the opposite corner.

Environment 3 The robot is a flamingo consisting of 7049 triangles. It has to maneuver out of a cage (1032 triangles). To exit the cage, it has the choice between several holes.

Environment 4 A sphere-shaped robot has to move from one position to another in a complex house environment consisting of 1600 polygons. There are roughly two solutions for this environment: a short path that uses the interior of the house, and a longer one that uses the "garden".

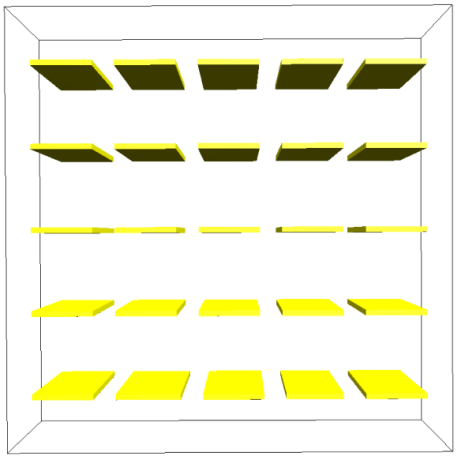
In all environments we used the same parameters for generating the roadmap: all vertices were attempted to be connected to at most 10 neighboring vertices. The maximum neighbor distance was about 10% of the longest diagonal of the bounding box. We used uniform sampling in all of our experiment.

2.5.1 Determining an Optimal Value for K

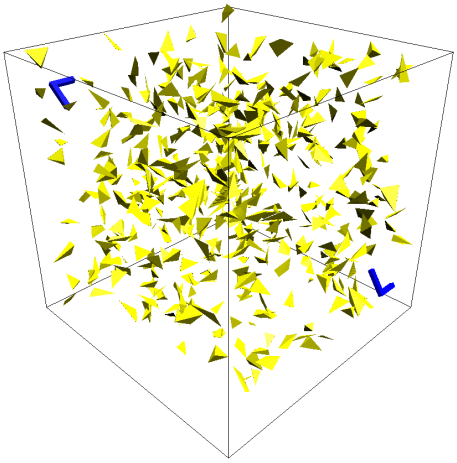
First we needed to establish the best value for K. Connecting every configuration to all of its neighbors (equal to $K = 1$) results in the best roadmap in terms of connectivity and thus the shortest paths. The price to pay for the increased running time however, is huge. Although theoretically no guarantees can be given, in practice the value of K has a direct correlation to the length of the cycles if uniform sampling is used. Our algorithm aims at improving *robustness* against changes in the environment. Suppose an obstacle changes its position. Because of this change, a number of edges of the roadmap will become invalid. Given a query, the roadmap should provide an alternative route around the obstacle that changed position if such a route exists. Very small cycles (compared to the size of the obstacle that moved) in the roadmap have a high probability of being intersected multiple times by the obstacle and do therefore often not contribute to providing an alternative route around the obstacle, this is shown in Figure 2.11(a)…(c). Thus K should be chosen such that cycles that are too small to significantly contribute to providing alternative routes have a low probability of being added to the roadmap. The best value of K is therefore related to the size of the obstacles that change their position within the environment.

To determine an optimal K we conducted the following experiment using environment 1: an obstacle at a random position was added to the environment and the roadmap edges that intersected the obstacle were invalidated. The size of the obstacle was in the same order as the existing obstacles. Next, we tested whether the query was still feasible. If so, another obstacle was added etc. We measured robustness by counting the number of obstacles we could add until the query was not feasible anymore. The experiment was repeated for several different values of K. In every experiment the total number of calls to the collision checker was kept equal; the creation of the roadmap was stopped when some maximum number of collision checks was executed. Therefore we could determine the increase in running time due to our algorithm. For every value of K we took the average over 20 runs, the results are depicted in Figure 2.12. The running times are reported as the percentage they differ from the running time of the standard PRM method, the *forest method*, which had a running time of 17.17s on average using the same amount of collision checks. Therefore, the running times can be seen as the additional computation time needed for our useful cycle method.

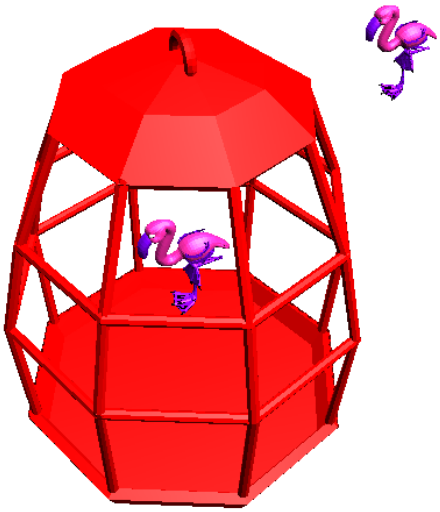
As can be seen from the results, except for small values of K, the additional running



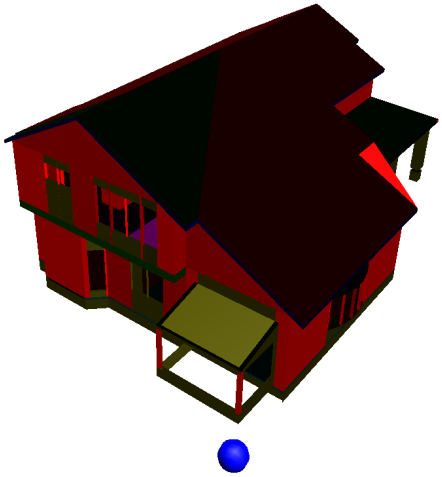
(a) Environment 1.



(b) Environment 2.

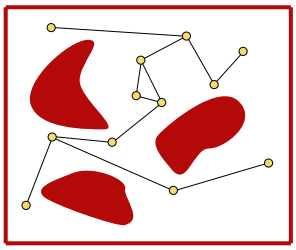


(c) Environment 3.

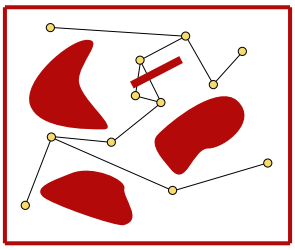


(d) Environment 4.

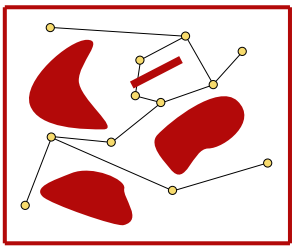
Figure 2.10: The various test environments.



(a) The roadmap contains a relatively small cycle.



(b) If an obstacle is added close to the cycle, the roadmap does not have a high probability of providing an alternative route.



(c) A larger cycle however, is able to provide an alternative route.

Figure 2.11: Relation between the size of the cycles and the probability of finding an alternative path.

time increases linear with K. If K is small, the robustness was very low. This can be explained by the fact that if K is small, vertices are connected to almost all of their neighbors, adding many edges that do not contribute to the connectedness of the environment and thus preventing more useful edges to be added. If $K > 4$, the robustness slowly decreases while the running time increases. This increase in running time can be explained by the fact that it takes our useful cycle method more time to decide if an edge is useful. The best values for K are in the range $[2,4]$ where, in this environment, the additional computation time is about 12%. Since the running time of deciding whether a cycle is useful is independent of the cost of collisin checks, this percentage drops if collision checks are more expensive. Although the optimal K depends on the environment and also on the preferences of the user, $K = 3$ was a good value for all our experiments.

2.5.2 Comparing the Robustness

To determine the quality of our roadmaps, we conducted experiments in which we compared our method to a variation on standard PRM in which we randomly allow cycles, the *random cycle method*. If two connected components are already connected in the roadmap, standard PRM does not allow another edge between them. We did allow the addition of such edges using a probability. The probability was choses such that the total number of cycles added to the roadmap was equal to the number of cycles the useful cycle method adds. In both methods we determined the robustness by adding random obstacles using the procedure described in Section 2.5.1. As a reference, we also added the results for the forest method. Again for every result we took the average of 20 runs. The results are shown in Table 2.1. In environment 1 our algorithm outperforms the random cycle method while the running times are comparable. On average about 36 random obstacles could be added to the environment until the query was not feasible anymore. For the random cycle method this number was only 19. Without cycles, only about 4 random obstacles could be added on average before the query became infeasible.

In environment 2, we observed similar results. Almost 17 random obstacles could be

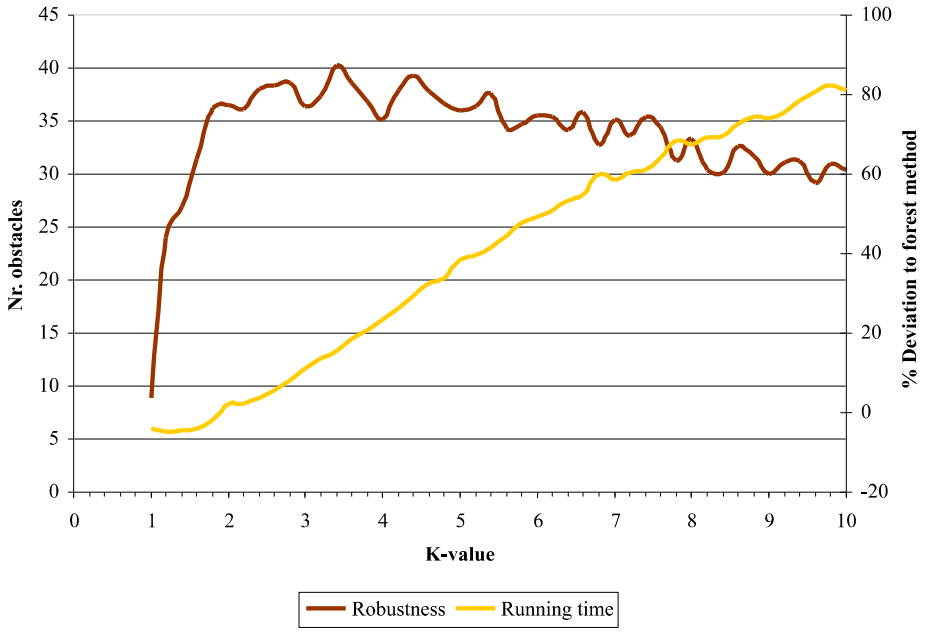


Figure 2.12: Finding an optimal K. The robustness is the average number of obstacles that could be added to the environment before the query became infeasible. The running time is the additional percentage of running time compared to the forest method.

added before the query failed using our algorithm. Using the random cycle method this number was only about 10 and for the forest method it was less than 3. The running time of the useful cycles method was higher though and this is because collision checks in this environment are cheap and therefore the useful cycle method is relatively expensive in terms of computation time.

In environment 3, there are multiple holes in the cage that the flamingo can use to escape. If only one of these contains a path, as is the case in the forest method, then blocking this path prevents the flamingo from reaching its goal. On average it takes about 7 random obstacles to block this path if the forest method is used. The random cycles method performs better, it is able to avoid almost 16 obstacles on average before the query becomes infeasible. Finally, for the useful cycles method this number is more than 28. Since collision checks are relatively expensive in this environment because of the complexity of both the robot and the stationary obstacles, the difference in running time is very small.

2.5.3 Comparing Query Path Length

As has been profen in Section 2.4, an additional advantage of the useful cycles method is that the length of the shortest path in the roadmap converges to K times the shortest

	Environment 1		Environment 2		Environment 3	
	rob	rt(s)	rob	rt(s)	rob	rt(s)
Forest	4.36	17.17	2.9	4.49	6.74	23.31
Random cycles	19.08	18.95	10.37	3.34	15.9	22.14
Useful cycles	36.44	19.07	16.95	5.54	28.6	23.02

Table 2.1: Results of the experiments. The robustness (rob) is the average number of obstacles that could be added to the environment before the query became infeasible. The running time (rt) is the average running time in seconds to create the roadmap.

path having minimal clearance ϵ if time goes to infinity. After extracting a path from the roadmap, smoothing is often used to optimize the path. The result of smoothing is that the path is significantly shortened. Smoothing usually performs well to remove small redundancies in a path, but is often not able to radically change a path (e.g. changing the homotopy class of the path). We calculated the length of the paths of the query phase both with and without smoothing. We compared our useful cycle method to the forest method using environments 2, 3 and 4. Also the variation in path length is reported.

In every environment we established for how long smoothing was necessary until the query path length did not change anymore. The reported query path lengths are given as a percentage of the shortest path.

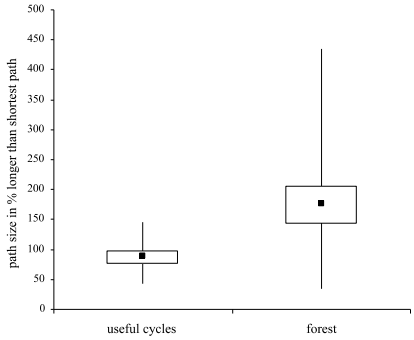
In environment 2 we smoothed for 1 second. As can be seen from the results, the useful cycle method outperforms the forest method in terms of path length. However, standard smoothing performs well and therefore the average path length after smoothing for both methods does not differ a lot.

In environment 3, the choice for the right hole in the cage is crucial for a short query path. If the query path for the forest method choses the "wrong" hole, then smoothing is not able to correct this. Thus there is a huge difference in query path lengths between the two methods, even after smoothing which was run for 2 seconds. The variation in path length is much lower in the useful cycle method.

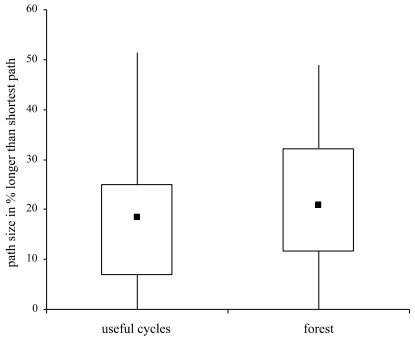
Environment 4 shows similar results as environment 3. Again, if the forest method results in a query path with a long detour, smoothing (run for 1 second) is not able to correct this.

2.6 Concluding Remarks

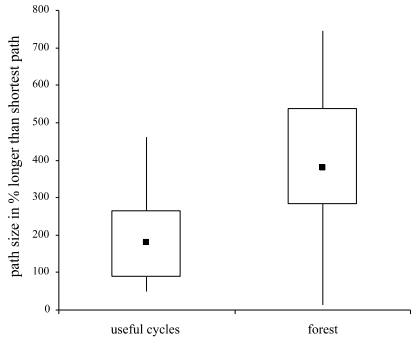
In this chapter we have presented a new connection algorithm for probabilistic roadmaps that provides alternative routes by adding useful cycles to the roadmap. Such roadmaps are an important prerequisite for path planning among changeable obstacles. If an obstacle blocks the path of the robot, the roadmap can be used to search for an alternative path before deciding to manipulate or to try to avoid the obstacle. Also in the postprocessing phase it is often not possible or too computationally expensive to remove long detours



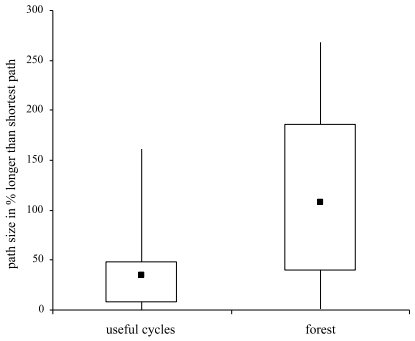
(a) Environment 2: query path.



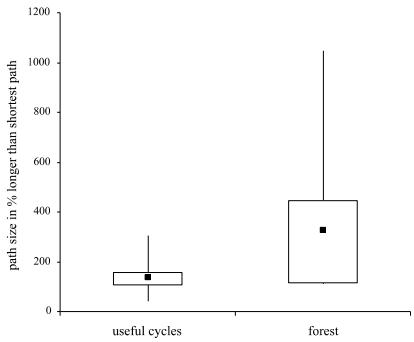
(b) Environment 2: smoothed path.



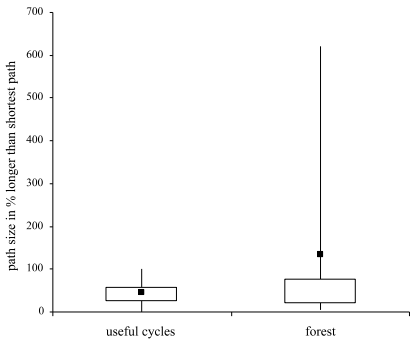
(c) Environment 3: query path.



(d) Environment 3: smoothed path.



(e) Environment 4: query path.



(f) Environment 4: smoothed path.

Figure 2.13: The results of the query path length comparison. The boxes show the area between the first and third quartiles. The lines show the highest and lowest values. The black squares show the average values.

from the path. A roadmap with cycles can therefore provide for shorter paths.

The value of K provides an easy way to change the behavior of the algorithm. A small K results in many cycles and is suitable for dynamic environments in which the robot has to avoid small obstacles or other robots. For environments in which the number of changes is fixed (e.g. an environment in which there are many doors), a larger K, resulting in less small cycles, is more appropriate.

We showed that the amount of additional computation time is small except in environments where collision checks are computationally cheap. Since the useful cycle method is not dependent on collision checks, the relative cost of the method decreases if the collision checks get more expensive. We compared the robustness of the roadmaps created by the useful cycle method to a method that randomly adds cycles to the roadmap. We showed that the useful cycle method is more robust against the addition of random obstacles, which makes it suitable to be used in changeable environments.

Unlike the forest method, in which no guarantees can be given on the shortest query path length, we provided a theoretical bound on this path length. Experiments showed that this property also holds in practical settings. These experiments showed that the useful cycle method outperforms the forest method in all environments; it provides short query paths without any additional collision checks in the query phase, allowing for fast queries. If smoothing (that needs collision checks) was applied after calculating the query path, the differences were smaller in some environments, but still significant. These results show that the useful cycle method can also be applied in static environments to improve overall path quality. Finally it can be used in environments where multiple robots share the same roadmap to avoid collisions between robots.

Recently Jaillet and Siméon (2006) created an algorithm that adds cycles to a roadmap based on visibility information. If paths are easily deformable into each other, they are considered redundant and one of them is not added to the roadmap. With their method they are able to create high quality roadmaps at the expense of some additional collision checks. Since they do not allow cycles between easily deformable paths, their method is less suited for use in changeable environments.

Part I

Changing Environments

CHAPTER 3

Creating Robust Roadmaps

In this chapter a variation on the Probabilistic Roadmap Planner is introduced that aims at constructing robust roadmaps for path planning in changing environments. Besides stationary obstacles, changing environments contain moving obstacles that can have a different position for each query. The algorithm of the previous chapter adds cycles to the roadmap which help in making the roadmap more robust against changes in the placements of the obstacles by providing alternative routes. Such a roadmap could be created with respect to the stationary obstacles only and not to the moving obstacles. If a query has to be solved then, given the positions of the moving obstacles, edges intersecting the moving obstacles have to be invalidated for the course of that query. These edges can be detected by checking these edges for collision against the moving obstacles. This approach has one important drawback: at query time the invalidated edges of the roadmap have to be identified. Expensive collision checks are involved in this process. Even though the process can be sped up using Kd-trees (Bentley, 1975) that, given the position of a moving obstacle, quickly find potential invalidated edges, query time increases dramatically.

We observe that the motion of the moving obstacles is often not unconstrained, but is restricted to some confined area, e.g. a door that can be open or closed or a chair whose position is bounded to a room. This property is exploited by assuming that a moving obstacle has a predefined set of potential placements. We present an algorithm that builds on the results of the previous chapter by adding edges to the roadmap (and thereby creating cycles) based on their necessity with respect to the moving obstacles. Information about the validity of the edges against the placements of the moving obstacles is inherently available during preprocessing. Therefore queries can be preformed almost instantaneously regardless of the placements of the moving obstacles. In addition the roadmap remains compact because only edges that contribute to the robustness of the roadmap are added. Our implementation shows that after a roadmap is created in the preprocess-

ing phase, queries can be solved instantaneously, thus allowing for on-the-fly replanning to anticipate changes in the environment.

3.1 Introduction

Realistic environments are often not static, but contain obstacles that change their positions over time. For such environments the PRM method is not guaranteed to work, because the premise that planning will occur many times in the same environment does not hold anymore. In this chapter we propose an algorithm that, besides useful cycles also adds cycles such that if a path exists, it is represented by a path in the roadmap independent of the positions of the moving obstacles.

For this we use the fact that in many applications the environment contains information about which changes in the placements of the obstacles are possible. For example, we usually know where the doors are that can be opened, and in which states they can be (anywhere between open and closed). Another example is a chair of which we know that it is located "somewhere" in a room. We refer to obstacles for which the set of *potential* placements is known a priori as *moving obstacles*. These include obstacles that can be removed from (and re-added to) an environment. An example of such a environment is shown as Figure 3.1.

Note that we are not concerned here with path planning in *dynamic* environments where continuously moving obstacles are avoided using temporal coordination (see e.g. Van den Berg and Overmars (2005)). We rather focus on *changing* environments, in which obstacles occasionally change their placements instantaneously. We assume that we know for each moving obstacle its set of potential placements, and will exploit this knowledge specifically. Although we aim at providing efficient solutions to problems in which the motion of the moving obstacles is restricted to some confined area, we put no constraints on the nature of this set.

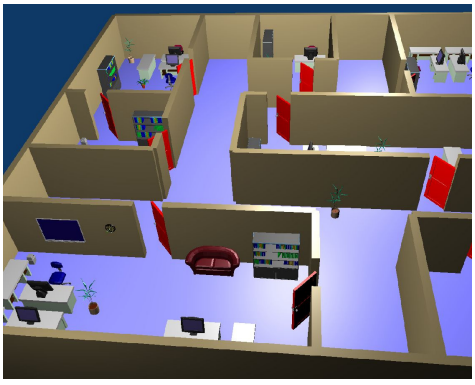


Figure 3.1: A typical example of an environment containing moving obstacles. Although most objects can change their position, it is expected that they are always situated in the same room.

The knowledge of the potential placements of the moving obstacles can be employed to create roadmaps that incorporate the possible changes of the environment, such that during the query phase the changes can be anticipated quickly. A straightforward approach would be to create the roadmap as if none of the moving obstacles is present and then mark parts of the roadmap as temporarily blocked as long as they are occupied by a moving obstacle. This approach is roughly followed by Leven and Hutchinson (2002). A problem with this approach is that it requires the roadmap to cover the topology of the free space regardless of the placements of the moving obstacles, yet this cannot be not guaranteed.

In this chapter we introduce a method that *does* guarantee the above, and produces roadmaps for changing environments that are *robust*. By robust we mean that for any placement of the moving obstacles the roadmap contains a path between any pair of query configurations, provided that such a path actually exists.

Our approach directly follows the PRM paradigm. Even though PRMs with cycles allow multiple connections between different connected components, we also need to add *necessary* edges to guarantee the creation of a robust roadmap. If already present paths between vertices may possibly be invalidated by some placements of the moving obstacles while a new edge is not, the new edge is a necessary edge. Of all edges in the roadmap we know for which placements of the moving obstacles they are valid. This ensures that queries can be performed quickly, and facilitates on-the-fly replanning to anticipate changes in the environment.

Our method is as generic as PRM, which means that it can be used for a broad range of robot types in configuration spaces of any dimension, and in combination with any sampling scheme or local planner. We have implemented our method and experiments show that in reasonable construction time, robust roadmaps can be produced for realistic environments. Queries are solved instantaneously, allowing for real-time on-line planning.

Only a few papers so far deal with path planning in changing environments. Leven and Hutchinson (2002) use a voxel grid that covers the workspace and stores for each roadmap edge which voxels are intersected by the robot during the path associated with that edge. When an obstacle changes position, the voxels are used to identify the edges that are invalidated by the obstacle. This is done by checking the voxels covered by the obstacle for overlap with the swept voxels of each of the edges. Using only the edges of the roadmap that are not invalidated, a path can quickly be found.

A more ad-hoc approach is used by Jaillet and Siméon (2004). Their planner first constructs a cycle-free roadmap for the static part of the environment. If during a query an edge crucial for finding a path appears to intersect with a moving obstacle, a variant of the Rapidly-exploring Random Trees (RRT) approach (LaValle and Kuffner, 2001) is used to reconnect the vertices of the invalidated edge. If this procedure fails, the two disconnected components of the roadmap are attempted to be reconnected by additional global sampling.

The main difference between these methods and ours is that they try to solve the problem in the query phase rather than in the preprocessing phase. They do however not

assume any knowledge about the potential placements of the moving obstacles. We do assume this although our approach would still work if we define the set of all potential placements of the moving obstacles to be the entire configuration space. Our approach creates roadmaps that are guaranteed to contain a path for any feasible query. Also obstacle changes can be anticipated quickly during runtime, as we store for each edge for which placements of the obstacles it is valid.

The remainder of this chapter is organized as follows. In Section 3.2 we give a general solution for creating robust roadmaps. The problem is discretized in Section 3.3, and we give some simple examples involving one moving obstacle. In Section 3.4 we extend the method to work with multiple moving obstacles. The query phase is discussed in Section 3.5, and we conduct experiments and show results in Section 3.6.

3.2 General Solution

A changing environment consists of a robot, stationary obstacles and a set of moving obstacles. For now, we restrict ourselves to one moving obstacle M (in Section 3.4 we lift this restriction). Let C_{M} be the configuration space of M. For M, a set of potential placements $P(M) \subset C_{\mathrm{M}}$ is defined. This set contains, in general, an infinite number of continuous placements, but it may also contain a finite number of discrete placements. Each $p \in P(M)$ maps the moving obstacle to a placement in the environment and effectively corresponds to a different path planning problem among stationary obstacles.

A query is defined as a triple (s, g, p), where s and g are the start and goal configurations of the robot, and p denotes a placement of M. Here, we devise an algorithm, based on the PRM-planner, which creates a roadmap G that is robust against the placement of M. The edges in this roadmap are undirected. As a consequence the robot should be symmetric in its motions. With each edge in our roadmap is associated a set describing for which placements of M it is valid. The algorithm is probabilistically complete, i.e. if the method runs long enough, the roadmap contains a solution for every feasible query.

3.2.1 Definitions

The PRM planner randomly samples configurations in the collision-free configuration space C_{free}. These configurations form the vertices of the roadmap. Since, in changing environments, we also have the moving obstacle M to take into account, we extend the definition of collision-free to the case of changing environments:

Definition 3.2.1 (Collision-free)**.** *For a configuration $c \in C$ of the robot and a placement $p \in P(M)$ of M, we define the function $CF(c, p)$ to be true iff the robot placed at c neither collides with the stationary obstacles nor with M placed at p.*
For two configurations $c, c' \in C$, and a placement $p \in P(M)$, we define the function $CF(c, c', p)$ to be true iff the path generated by the local planner between c and c' neither collides with the stationary obstacles nor with M placed at p.

A configuration c is added as a vertex to the roadmap if there exists a $p \in P(M)$ for which $CF(c,p)$. Subsequently, in standard PRM, connections are tried to other vertices that are not in the same connected component as c. Here, we need new definitions for "path existence" and "component" in order to take M into account.

Definition 3.2.2 (Path existence)**.** *For two vertices c_i and c_j in the roadmap G and a placement $p \in P(M)$ for M, we define the function $PE(c_i, c_j, G, p)$ to be true iff a path exists in G connecting c_i and c_j when the moving obstacle is placed at p. Since G is undirected: $PE(c_i, c_j, G, p) = PE(c_j, c_i, G, p)$.*

Instead of connected components, we use the notion of *labeled components*. A labeled component consists of vertices between which a path exists regardless of the placement of M.

Definition 3.2.3 (Labeled component)**.** *Two vertices c_i and c_j belong to the same* labeled component *L in G iff $PE(c_i, c_j, G, p)$ for all $p \in P(M)$.*

The collection $\{L_1, L_2, \ldots\}$ of all labeled components is a *partition* of the set of vertices V in the roadmap, so $L_1 \cup L_2 \cup \ldots = V$ and $L_i \cap L_j = \emptyset$ $(i \neq j)$. Note that two vertices that belong to different labeled components can still be connected by an edge in the roadmap. This means that such an edge only connects the vertices for a subset of $P(M)$. A consequence of this is that between two labeled components a path also only exists for certain placements of $P(M)$. Therefore, for each pair of labeled components, we need to maintain the set of placements of M for which a path exists between them. Such a set is called a *connection set*.

Definition 3.2.4 (Connection set)**.** *For each pair of labeled components L_i and L_j in G, a* connection set *$CS(L_i, L_j, G) \subset P(M)$ is defined as the set of placements of M for which a path exists between L_i and L_j, if $c_i \in L_i$ and $c_j \in L_j$ then*

$$CS(L_i, L_j, G) = \{p \in P(M) \mid \exists_{c_i \in L_i, c_j \in L_j} PE(c_i, c_j, G, p)\}.$$

In our algorithm we only add an edge to the roadmap if it connects two different labeled components *and* if it extends the connection set of those two labeled components. To state this formally, we introduce the notion of *necessary* edges.

Definition 3.2.5 (Necessary edge)**.** *For two vertices $c_i \in L_i$ and $c_j \in L_j$, we define the edge (c_i, c_j) to be* necessary *iff it extends the connection set between the two labeled components in G, i.e.*

$$\{p \in P(M) \mid CF(c_i, c_j, p)\} \not\subset CS(L_i, L_j, G).$$

If a necessary edge is added to the graph, then by definition it extends the connection set of its two incident labeled components, so we need to update this connection set. This update may affect other connection sets as well, because new routes through the roadmap may have become available. The new information needs to be *propagated* through the roadmap to update the other connection sets. The procedure is detailed in Section 3.2.3.

We now have all the ingredients for an algorithm that creates a robust roadmap for changing environments. This algorithm in shown in pseudo code as Algorithm 3.1. Just as with standard PRM, the algorithm repeats until some stop-criterion is met, for example a pre-specified amount of time. Algorithm 3.1 shows a general solution to our problem. It is not yet a practical solution, as in lines 3 and 9 a possibly infinite set is evaluated. In Section 3.3 we resolve this by discretizing the set of placements of M. In line 12, the propagation algorithm is called to update the connection sets. If a connection set contains every potential placement of M, the two associated labeled components can be merged. This is done in line 13. The propagation and merging algorithms are detailed in Section 3.2.3.

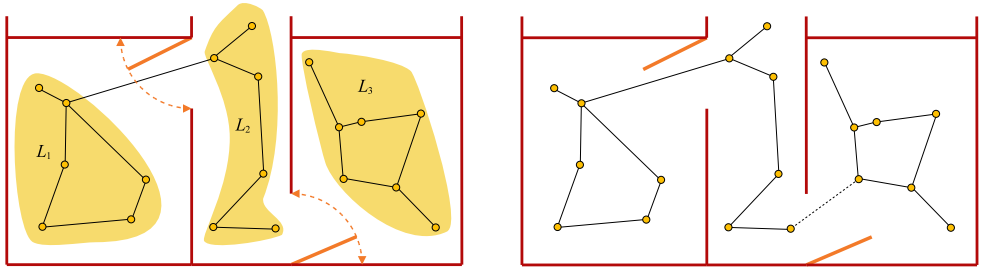
Algorithm 3.1 CONSTRUCTROBUSTROADMAP

Let: $G = (V, E)$
1: **repeat**
2: sample a random configuration $c \in C$
3: **if** $\exists(p \in P(M) \mid CF(c, p))$ **then**
4: $V = V \cup \{c\}$
5: initialize $L \leftarrow c$
6: **for all** $(L_i \neq L)$ **do**
7: initialize $CS(L, L_i, G) \leftarrow \emptyset$
8: **for all** $(c' \in V \mid c' \neq c)$ **do**
9: **if** edge (c, c') is necessary (see Definition 3.2.5) **then**
10: $E = E \cup (c, c')$
11: $P_f \leftarrow (p \in P(M) \mid CF(c, c'))$ {placements of M for which (c, c') is free}
12: PROPAGATE$(L(c), L(c'), G, P_f)$ {$L(c)$ and $L(c')$ are the labeled components of vertices c and c' respectively}
13: MERGELABELEDCOMPONENTS
14: **until** some stop-criterion is met

3.2.2 Probabilistic Completeness

Our method is probabilistically complete. That is, if time goes to infinity the roadmap is guaranteed to contain a solution path for any feasible query. We sketch a proof here, which roughly follows the proof for standard PRM as given by Švestka (1997).

Given is an environment with stationary obstacles, a moving obstacle and a query (s, g, p) for which there exists a path. We cover this path with overlapping free balls. The connection made by the local planner between configurations in two neighboring balls is collision-free with respect to the stationary obstacles and to placement p of the moving obstacle. There are two possibilities: either this connection is necessary and added to the roadmap, or this connection is not necessary because there already exists a path in the roadmap between the two configurations that is collision-free with respect to p (see Definition 3.2.5). In either case a path will exist between the two configurations. As the probability that each ball contains a configuration approaches 1 when the running time



(a) Three labeled components are present in the graph. For certain placements of the leftmost door there is a connection between L_1 and L_2.

(b) Adding the dotted edge to G does not only extend $CS(L_2, L_3, G)$ but also $CS(L_1, L_3, G)$.

Figure 3.2: An environment consisting of two rooms and a hallway separated by doors.

goes to infinity, it is guaranteed that a path will be found between s and g for placement p.

3.2.3 Propagation and Merging

After adding an edge between two labeled components, we need to update the connection set and propagate the new information to all other connection sets (line 12 of Algorithm 3.1). An example is shown in Figure 3.2. For this purpose we need the notion of *neighboring* labeled components.

Definition 3.2.6 (Neighboring)**.** *Two labeled components L_i and L_j are* neighboring *each other iff*

$$\exists_{c_i \in L_i, c_j \in L_j} \mid (c_i, c_j) \in E.$$

If we add an edge between L_i and L_j, thus extending the set of placements of M for which a connection exists between L_i and L_j, we first update the connection set $CS(L_i, L_j, G)$. Then, to propagate this new information to the other connection sets, we recursively update all neighbors of L_i and then all neighbors of L_j. The propagation algorithm is shown as Algorithm 3.2. Its input consists of the two labeled components for which connectivity is extended and the placements $P_f \subset P(M)$ for which the edge is collision-free. Every connection set is updated at most once after the addition of an edge.

After running Algorithm 3.2 it is possible that vertices that used to belong to different labeled components now belong to the same labeled component according to Definition 3.2.3. Stated differently, for every placement of M there is a connection between the labeled components of these vertices. We call the connection set of such labeled components *complete*.

Definition 3.2.7 (Complete)**.** *The connection set $CS(L_i, L_j, G)$ is* complete *iff for every placement of M, a path exists between L_i and L_j:*

$$CS(L_i, L_j, G) = P(M).$$

Algorithm 3.2 PROPAGATE (L_i, L_j, G, P_f)

1: $CS(L_i, L_j, G) \leftarrow CS(L_i, L_j, G) \cup P_f$
2: **for all** neighbors L_k of L_i **do**
3: $P_f \leftarrow CS(L_i, L_j, G) \cap CS(L_i, L_k, G)$
4: **if** $P_f \not\subset CS(L_j, L_k, G)$ **then**
5: PROPAGATE (L_j, L_k, G, P_f)
6: **for all** neighbors L_k of L_j **do**
7: $P_f \leftarrow CS(L_i, L_j, G) \cap CS(L_j, L_k, G)$
8: **if** $P_f \not\subset CS(L_i, L_k, G)$ **then**
9: PROPAGATE (L_i, L_k, G, P_f)

If a connection set is complete, we need to *merge* the associated labeled components. This process is detailed in Algorithm 3.3.

Algorithm 3.3 MERGELABELEDCOMPONENTS

1: **for all** $CS(L_i, L_j, G)$ that have been updated by PROPAGATE **do**
2: **if** $CS(L_i, L_j, G) = P(M)$ **then**
3: $L_i \leftarrow L_i \cup L_j$
4: Remove the connection sets incident to L_j
5: Append neighbor list of L_j to neighbor list of L_i
6: Relabel the vertices in L_j.

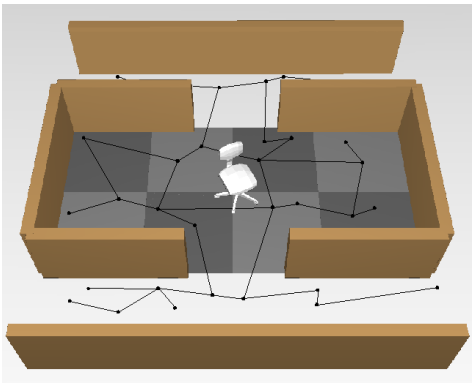
3.3 Discretization

In our algorithm, we need to check for which placements of M the vertices and edges are collision-free (see lines 3 and 9 of Algorithm 3.1), provided that they do not collide with the static obstacles (Definition 3.2.1). In case of a discrete set of placements $P(M)$, we can check against each individual element of this set. For continuous sets however, it is practically impossible to check for which placements the vertices and edges collide. This is due to the complexity of the sweep volume of a path of M; for all but the simplest shapes of M, creating C exactly is infeasible. Therefore, we partition $P(M)$ into a finite set of n *chunks* $\Delta(M) = \{\delta^1, \delta^2, \ldots, \delta^n\}$.

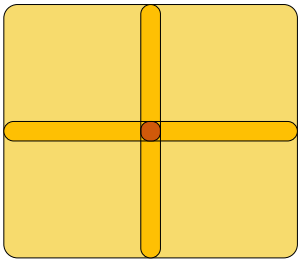
Definition 3.3.1 (Chunk)**.** *A* chunk $\delta \in \Delta(M)$ *is a subset of the placements of obstacle M:* $\delta \subset P(M)$.

The definition of $\Delta(M)$ as a partition of $P(M)$ implies the following: $\bigcup \Delta(M) = P(M)$ and $\delta^i \cap \delta^j = \emptyset$ when $i \neq j$. Instead of checking for which *placements* of M a vertex or edge is collision-free, we now check for which *chunks* of M the vertex or edge is collision-free.

We observe that, in order to preserve completeness, we need to choose the chunks such that if the problem is solvable for a placement $p \in P(M)$ of M, the problem is also



(a) The subdivision of the set of placements in 8 chunks, and the associated robust roadmap for a cylindrical robot.



(b) The sweep volumes of the chunks overlap, this is shown for the leftmost 4 chunks.

Figure 3.3: An example of the creation of chunks. The chair is allowed to be placed anywhere in the room, but it may not rotate.

solvable for the chunk that contains p. If we would partition $P(M)$ into an infinite number of chunks, then every chunk contains only one placement, which would restore the continuous situation. In most cases, however, it is sufficient to partition $P(M)$ into only a small number of chunks.

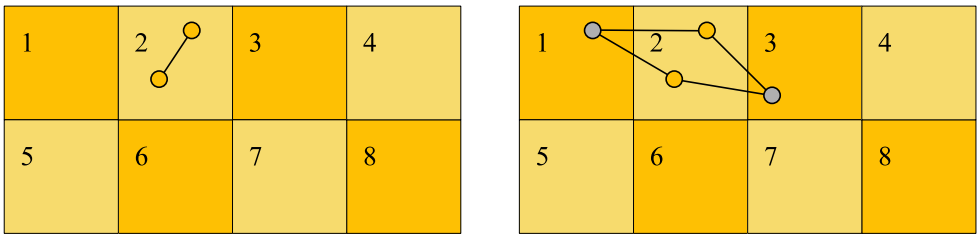
Figure 3.3 shows an example of the subdivision of the potential placements of a moving obstacle. The obstacle is a chair with two degrees of freedom that can be placed freely inside a room. The robot is a small cylinder. The set of placements of the chair is subdivided into 8 different chunks. A resulting roadmap is shown. Between the two parts of the roadmap above and below the room, a path always exists regardless of the chunk the chair is in, so they belong to the same labeled component. Note that the sweep volumes of the chunks in *workspace* will overlap. This overlap is omitted in Figure 3.3(a), but is shown in Figure 3.3(b).

Vertices whose associated configuration collides with certain placements of M (thus are "inside" a chunk as is the case with some vertices of Figure 3.3(a)) will never have collision-free connections to other vertices for *all* placement of M. This means that every vertex inside a chunk is a labeled component by itself for which we need to maintain connection sets to all other labeled components. Luckily we can merge certain colliding vertices to one labeled component by relaxing Definition 3.2.3: If the sets of chunks for which two vertices are free, are both equal to the set of chunks for which a path exists between the vertices, we can merge their labeled components (Figure 3.4). To this end, the definition of labeled component (and thus line 2 of Algorithm 3.3) should be adapted. Note, that this new definition does not change the definition of labeled components for vertices that are free for all chunks.

Definition 3.3.2 (Labeled component)**.** *Two vertices c_i and c_j belong to the same* labeled

component L *iff*

$$\{\delta \in \Delta(M) \mid CF(c_i,\delta)\} = \{\delta \in \Delta(M) \mid CF(c_j,\delta)\} = \{\delta \in \Delta(M) \mid PE(c_i,c_j,G,\delta)\}.$$



(a) **(b)**

Figure 3.4: A (part of) an environment consisting of 8 chunks. Since in both examples the two vertices in chunk 2 are collision-free for the same chunks ({1,3,4,5,6,7,8}), and a path exists between them for those chunks, the two can be merged to one labeled component in both examples.

To find out whether an edge between two labeled components is necessary (see Definition 3.2.5), we do not need to collision-check the edge with all chunks. Only the chunks for which there is no connection yet need to be checked. If for one or more of those the edge is free, the edge extends the connection set and hence it is necessary.

An important issue is how to create the chunks automatically. A straightforward solution is to partition the range of every degree of freedom of the moving obstacle into a collection of segments. The Cartesian products of these segments then form the chunks. This procedure works well with the examples used in our experiments. There may however be situations in which more complex procedures are necessary. To be able to collision-check against a chunk, we must create an object that represents (a superset of) the workspace sweep volume of the chunk. For this purpose we use principles used by Leven and Hutchinson (2002), by covering the workspace with a voxel grid. The object is then formed by the collection of voxels that is swept by a number of obstacle placements sampled evenly over the chunk. The sampling density should correspond to the resolution of the voxel grid, which determines the accuracy of the representation.

3.4 Multiple Obstacles

The previous section considers only one moving obstacle, but the generic description of our solution applies to the case of multiple moving obstacles as well. Multiple obstacles can be regarded as one composite obstacle whose set of degrees of freedom is just the concatenation of the degrees of freedom of each individual obstacle.

Assume that we have k moving obstacles $M_1, M_2, \ldots, M_k$ and that the set of placements $P(M_i)$ of each individual obstacle M_i is partitioned into a set of chunks $\Delta(M_i) = \{\delta_i^1, \delta_i^2, \ldots\}$, just as we did in the previous section. The set of chunks D of the composite obstacle is then the Cartesian product of the sets of chunks of the individual obstacles:

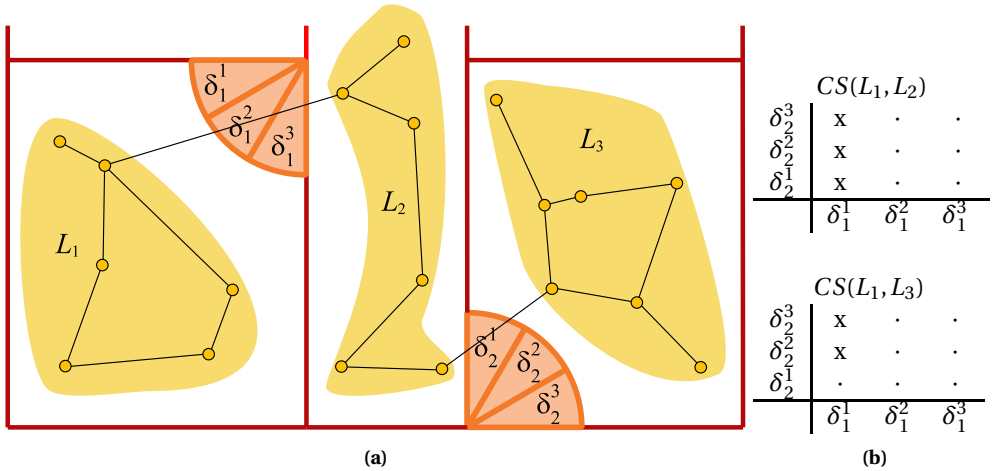


Figure 3.5: Example of composite chunks. (a) An environment with two moving obstacles (the two doors). The placements of both doors are partitioned in three chunks. (b) Two corresponding connection sets. The connection sets show for all combinations of chunks whether there is a connection or not between the labeled components. An "x" means there is a connection, a dot means there is no connection.

$D = \Delta(M_1) \times \Delta(M_2) \times \cdots \times \Delta(M_k)$. An element of D, a *composite chunk*, is denoted by $(\delta_1^{q_1}, \delta_2^{q_2}, \ldots, \delta_k^{q_k})$, where the q's are indices of individual obstacle chunks.

Using the set D, we can use Algorithm 3.1 without any changes, that is, a connection set now maintains for which combination of individual obstacle placements (e.g. for which composite chunks) a path exists between two labeled components. In Figure 3.5 an example representation is given of two connection sets in an environment with two moving obstacles of which each placement set is partitioned into three chunks. Since two moving obstacles are involved, we can think of the connection sets as two-dimensional grids of Booleans, where the chunks of each individual obstacle are listed along the axes. A grid cell models a composite chunk of the respective individual chunks along the axes.

If we have k moving obstacles, the grids become k-dimensional and if the placement set of each obstacle M_i is partitioned into m_i chunks, the total number of chunks of the composite obstacle is $\prod_{i=1}^{k} m_i$. However, when an edge is considered for addition to the roadmap, we do not need to perform collision checks with all chunks of the composite obstacle to check whether the edge is free. It suffices to check whether the edge is free with respect to the chunks of each of the individual obstacles. So in total only $\sum_{i=1}^{k} m_i$ collision checks have to be performed. From these checks, we can easily deduce for which composite chunks the edge (c_i, c_j) is free:

$$CF(c_i, c_j, (\delta_1^{q_1}, \delta_2^{q_2}, \ldots, \delta_k^{q_k})) = CF(c_i, c_j, \delta_1^{q_1}) \wedge CF(c_i, c_j, \delta_2^{q_2}) \wedge \cdots \wedge CF(c_i, c_j, \delta_k^{q_k}).$$

For each connection set $CS(L_i, L_j, G)$ we need to maintain for all $\prod_{i=1}^{k} m_i$ composite chunks whether a path exists between two labeled components L_i and L_j. A connection set is straightforwardly represented by an array of Booleans, and the operations on connection sets we need in the algorithm (union, intersection, test for subset and completeness) are easily implemented for such a representation.

Another issue concerns how to store the total collection of connection sets. In principle a connection set is maintained for every pair of labeled components. From analysis on the standard PRM method we know that the number of connected components increases quickly in the first part of the execution of the algorithm, and decreases later on as the components get connected. We are likely to see the same behavior for labeled components in our method. When the number of components is maximal, many of them are not connected to each other. This also means that the associated connection sets are empty. Therefore, we choose to only store non-empty connection sets.

In the current algorithm, connection sets are also maintained for labeled components that are connected in G via many other labeled components. These connection sets are not so interesting, as the probability that a direct edge is added between these labeled components is small. Yet, storing such connection sets takes as much memory as others.

We can overcome this problem by changing the propagation algorithm (Algorithm 3.2). Instead of propagating the information through the roadmap until no expansions of connection sets are achieved any more, we may put a maximum on the depth of the propagation, i.e. connection sets between labeled components that are connected to each other via many other components are not updated and propagated. This means that these connection sets remain empty, and therefore do not need to be stored. Also, the propagation algorithm becomes faster. The drawback is that the information stored in all connection sets is not complete anymore. This may cause some extra edges to be added, but we expect this number to be very limited.

An example roadmap created by our algorithm is shown in Figure 3.6. The environment consists of 4 doors each of which has two potential placements (chunks). Although not trivial to observe, the goal position g can be reached from the start position s for every combination of the placements of the doors. Therefore the vertices of s and g belong to the same labeled component and G is robust for the query. (It is even robust for every query.)

3.5 Query Phase

The roadmaps created by our algorithm are robust against changes of placements of obstacles. During the construction of the roadmap we store for each edge for which placements of the moving obstacles it is free (see line 11 of Algorithm 3.1). We exploit this property to quickly compensate for placement changes of obstacles during the execution of a query. Experiments show that after preprocessing, queries can be performed instantaneously. To find a path between a given start and goal position, we first search for a path in the roadmap using a shortest path algorithm, given the current placements of the ob-

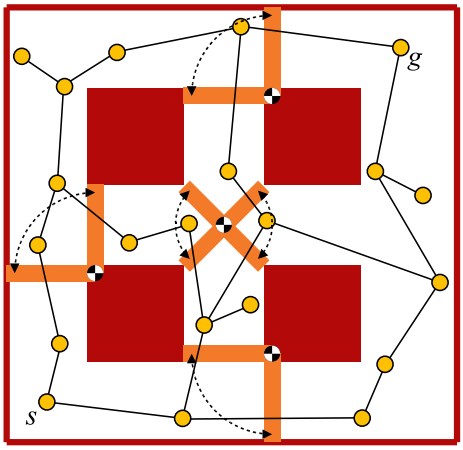


Figure 3.6: An example of a robust graph in an environment consisting of 4 moving obstacles. Each of the moving obstacles has two distinct placements. There is always a path between s and g independent of the placements of the moving obstacles.

stacles. Next, the robot starts executing the resulting path. If, during the execution of the query an obstacle changes its placement, we immediately know if one of the edges of the query path is invalidated. If this is the case, we re-plan by querying from the current robot position to the goal position. Since our roadmap is robust, we are again guaranteed to find a new path if such a path exists.

3.6 Experiments

We implemented our algorithm in C^{++} and conducted experiments on 3 changing environments each representing a realistic example. For each experiment we used a cylinder-shaped robot having two degrees of freedom. We will now describe the experiments in detail.

The environment of the first experiment (Figure 3.3) consists of a room in which a chair can be freely placed (**office with chair**). A robot needs to move from the lower corridor to the upper corridor. Recall that we need to choose the chunks such that if the problem is solvable for a particular placement, the problem is also solvable for the chunk that contains this particular placement. With this in mind, we partitioned the set of placements of the chair in 8 chunks. In workspace, each chunk overlaps with its neighboring chunks as shown in Figure 3.3(b).

The second environment (**puzzle**), shown as Figure 3.6, consists of 4 moving obstacles that can all be placed in two different positions, hence the number of chunks is 8. In contrast to the first experiment, the connection sets consist of composite chunks. There are 16 combinations of moving obstacle placements (2^4), and thus we have 16 composite chunks. The robot needs to move from one corner to another.

The third experiment (**double puzzle**), shown as Figure 3.7 is a variation on the second, but since the environment is repeated twice, the number of chunks is 16 and the number of composite chunks is $2^8 = 256$. Again the query points are chosen far apart.

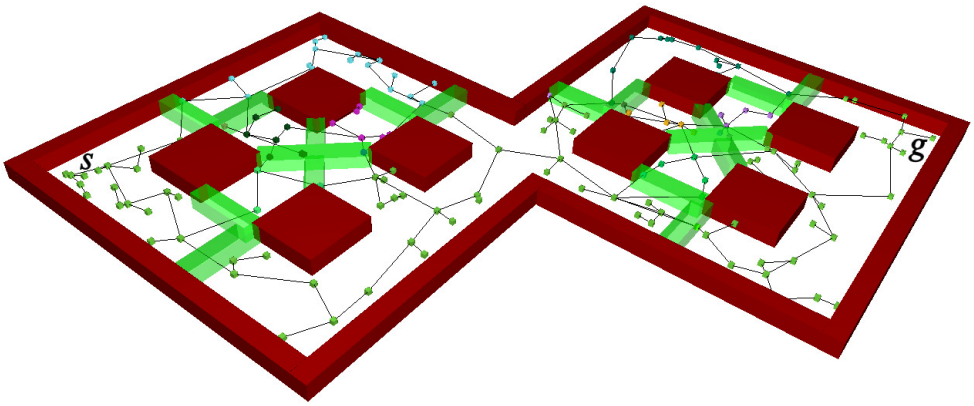


Figure 3.7: A more complicated puzzle environment, consisting of 256 potential combinations of the moving obstacle placements.

Since the roadmap is created by using random sampling, for each environment we conducted 25 experiments for which the running times were averaged. We ran the algorithm until a roadmap was created that contains a path between the query points for all potential obstacle placements. In these experiments this is a good criterion because the query points are far apart and the robot has to pass many moving obstacles on its path. For all experiments we used a cylinder as the robot, representing a moving body.

To check a vertex or an edge for collision, we always first check the robot against the stationary obstacles. If there is a collision, we reject the edge or vertex immediately. For checking an edge for collision we used a recursive binary strategy (Geraerts and Overmars, 2004). Let the *size of the environment* be the length of the longest straight line segment that fits within the bounding box of the environment. Connections are only tried between vertices that are at most half the size of the environment apart. We implemented the described algorithms including all suggested optimizations, except that we did not use a limited depth on the propagation. Experiments were conducted on a Pentium 3.0GHz equipped with 1GB of memory. The results of our experiments are shown in Table 3.1.

As can be seen from the results, creating a robust roadmap took only little time in the first two experiments. In the third environment (which has 16 times more composite chunks than the second environment), preprocessing took a little longer but still less than 2 seconds. After preprocessing, queries can be performed instantaneously.

	Avg. running time (s)
Office with chair	0.23
Puzzle	0.19
Double puzzle	1.94

Table 3.1: Roadmap creation times for the three experiments.

3.7 Concluding Remarks

In this chapter we have introduced an algorithm that builds on the ideas presented in Chapter 2. There we added cycles to the roadmap based on the current graph distance. While this method is relatively cheap, it does not take the specific properties of the obstacles into account. If we know the potential placements of an obstacle in advance, we can use this information to create robust roadmaps that are guaranteed to find a path provided one exists independent of the placement of the obstacles.

Given the set of potential placements of an obstacle, only those edges that extend the probability a path can be found during query time are added. For this, first the current connectivity of the graph is evaluated. If for certain placements of an obstacle no path exists between two parts of the graph, while the edge is collision-free for these placements, the edge is called necessary and added to the graph. In addition to the necessary edges, useful cycles of Chapter 2 can be used to provide for shorter and more efficient paths.

Previous methods often try to find a solution during query time. A drawback of this is that the query phase may become computationally expensive, while many applications demand fast and predictable response during query time. Our method tries to create a robust roadmap during preprocessing time such that queries can be solved almost instantaneous. While our method is primarily aimed at path planning in changing environments in which the placements of the obstacles is fixed during query time, because of the fast queries it can also be used in environments in which obstacles change their placement during the execution of the query. If such a change takes place, a new query can be quickly executed to provide for a new path.

We conducted experiments that showed that our method performs well within runtime demands. Even in environments with multiple moving obstacles, roadmaps could be created within seconds. We also proposed some heuristics to speed up roadmap construction. These heuristics provide a trade-off between completeness and efficiency. Unfortunately there is an exponential dependency between the number of chunks and the size of the connection sets. If the number of chunks gets too high, the running time increases quickly. This issue is the main topic of the next chapter.

We have presented our method independent of any postprocessing techniques or optimizations that can be used during preprocessing. Since we do not assume anything about the sampling technique or the local planner, existing optimization techniques can be used without restrictions. To keep the query phase quick, our method can, for example, be combined with the work of Geraerts and Overmars (2006), which present a method for creating

high quality paths in the preprocessing phase. The combination is potentially powerful because the amount of work in the query phase is small while the extracted paths are of high quality.

CHAPTER 4

Improving Roadmap Construction

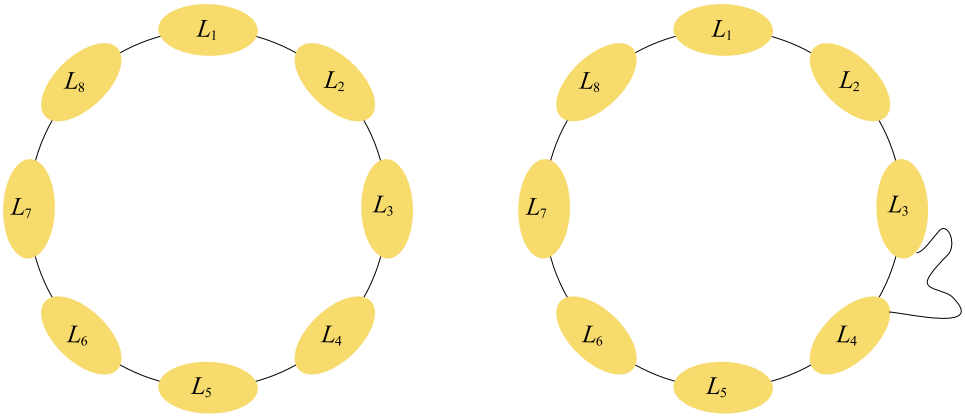
While the algorithm introduced in Chapter 3 is very successful in creating roadmaps that are robust against placement changes of the obstacles it has some drawbacks. The most important one is that when the number of chunks gets large, the decision whether an edge is necessary gets computationally expensive. In this chapter we will first analyze this problem, pinpoint its causes and then state solutions. The solutions fit within the framework presented in Chapter 3. We will conduct experiments to compare our algorithm with (a slightly adapted version of) standard PRM.

4.1 Analysis of the Algorithm

The algorithm presented in Chapter 3 uses connection sets to represent the placements of the moving obstacles for which a path exists between labeled components. Since such a connection set represents all potential combinations of placements of the moving obstacles, the size of the connection sets and thus the amount of time spent on the operations manipulating the connection sets increases exponentially with the number of obstacles and chunks. If the set of placements of a moving obstacle M_i is partitioned in m_i chunks, then the total number of elements in each connection set equals $\prod_{i=1}^{k} m_i$ where k is the total number of moving obstacles. Stated differently, there is an exponential dependency between the number of chunks and the size of the connection sets. For example, if an environment consists of 10 obstacles, each consisting of 4 chunks, a connection set consists of $4^{10} = 1{,}048{,}576$ elements. If between every pair of labeled components a connection set is defined, the number of connection sets maintained is equal to the square of the number of labeled components. (Note that this number should be divided by 2 if undirected roadmaps are used because then connection sets $CS(L_i, L_j, G)$ and $CS(L_j, L_i, G)$ are

equal.) Thus if the number of moving obstacles increases, the algorithm quickly becomes computationally infeasible.

A second problem occurring with the propagation algorithm is that if a connection set between two labeled components L_i and L_j is extended, this can potentially affect all other connection sets. Consider the following example shown in Figure 4.1: the roadmap G consists of n labeled components $(L_1, L_2, \ldots, L_n)$. If the labeled components form a closed chain, i.e. labeled component L_i is only neighboring L_{i-1} and L_{i+1} if $1 < i < n$ and L_1 is neighboring L_n, then if a necessary edge is added between L_i and L_{i+1}, this could potentially affect all other n^2 connection sets. In this case the propagation algorithm is relatively expensive. In Figure 4.1 there are 8 labeled components each having two neighboring labeled components. Now the connection set between L_3 and L_4 is extended. Since from each labeled component there are two paths to every other labeled component (clockwise and counterclockwise) the extension of $CS(L_3, L_4, G)$ could potentially influence all existing connection sets and propagation may be necessary to $n^2 = 8^2$ connection sets.



(a) A closed chain of labeled components each neighboring only two other labeled components.

(b) The connection set $CS(L_3, L_4, G)$ is extended.

Figure 4.1: An example in which propagation is expensive.

For many pairs of labeled components (especially those close together) there is only a limited number of obstacle placements for which no path exists. The connection sets however store for which combination of placements of the obstacles a path *exists*. The result is that connection sets are exponentially large while only for a small number of obstacle placement combinations no path might exist. Because we need to perform many operations on these connection sets (check completeness, merge) the running time of the algorithm is for a large part related to these operations.

4.2 Improving the Algorithm

Before we propose solutions to the observed problems of the previous section we translate our problem to the domain of Boolean logic (Boole, 1848). For this, we use the variables of the chunks as literals (a literal is an atomic formula or its negation in Boolean logic). For example, δ_i^2 means that obstacle M_i is positioned inside chunk δ_i^2. The negation of a chunk, e.g. $\overline{\delta_i^3}$, means that obstacle M_i is not positioned within that chunk. The combination of chunks for which a path is obstructed can now be described using an *obstruction function.*

Definition 4.2.1 (Obstruction function)**.** *Given two configurations $c_i, c_j \in G$, an* obstruction function *$O(c_i, c_j)$ is a Boolean function that states for which combinations of literals (chunks) the path created by the local planner between c_i and c_j is obstructed. If the values for the literals are known (and thus the positions of the moving obstacles), it evaluates to* TRUE *if the path is obstructed, else it evaluates to* FALSE.
For two labeled components L_i and L_j, the obstruction function $O(L_i, L_j, G)$ states for which combinations of literals no path exists in G between the two labeled components.

Suppose the local planner reports intersections on the path between configurations c_i and c_j in Figure 4.2 with the following chunks: δ_1^3, δ_2^1 and δ_2^2, then the obstruction function for the corresponding edge is: $O(c_i, c_j) = (\delta_1^3 \vee \delta_2^1 \vee \delta_2^2)$. Given the placements of the obstacles, we know which literals are TRUE and which ones are FALSE. If an obstruction function of an edge evaluates to TRUE then the edge collides for the given placements of the obstacles. Note that the obstruction function of a single edge always consists of only one clause in disjunctive normal form.

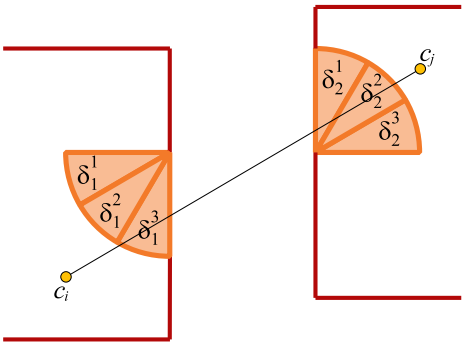


Figure 4.2: An example of an edge (c_i, c_j) intersecting multiple chunks. The corresponding obstruction function is: $O(c_i, c_j) = (\delta_1^3 \vee \delta_2^1 \vee \delta_2^2)$.

Using the literals we can also describe the connection sets that state for which combinations of chunks a path exists between two labeled components. However, considering that between two labeled components there is often only a limited amount of chunks that can block the path between them, we will switch from maintaining connections to maintaining obstructions. Obstructions between labeled components will be maintained in

obstruction functions. For each pair of labeled components L_i and L_j, an obstruction function $O(L_i, L_j, G)$ is defined.

Suppose we have two labeled components L_i and L_j. The corresponding obstruction function $O(L_i, L_j, G)$ is initialized with TRUE. This signifies that no path exists for any combination of the literals between L_i and L_j. If an edge is added between $c_i \in L_i$ and $c_j \in L_j$ then the corresponding obstruction function is extended by taking the conjunction of the obstruction function of the edge and the current obstruction function between the labeled components: $O(L_i, L_j, G)$ becomes $O(L_i, L_j, G) \wedge O(c_i, c_j)$. It is also possible that no direct edge is added between two labeled components but connectivity is extended via other labeled components using propagation. Such an example is shown in Figure 4.3. Here an obstruction function $O(L_1, L_3, G)$ exists for labeled components L_1 and L_3. Next, an edge between L_2 and L_3 is added to G. Since a path through G is now possible from L_1 to L_3 via L_2, the obstruction function $O(L_1, L_3, G)$ changes.

Given the values of the literals (e.g. the chunks in which the obstacles are) at query time, an obstruction function can be evaluated to check if a path exists between two labeled components for a given set of placements of the obstacles. If it evaluates to FALSE, a connection between the labeled components exists.

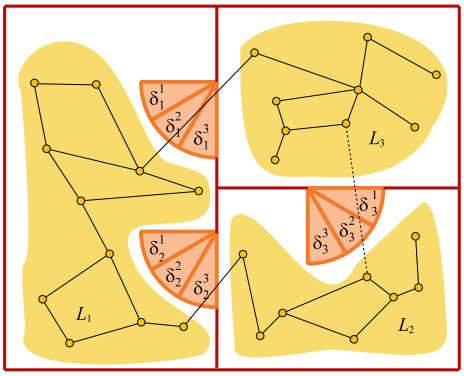


Figure 4.3: Example of a more complex obstruction function. Before the dotted edge is added to the roadmap, $O(L_1, L_3, G) = (\delta_1^2 \vee \delta_1^3)$. After the addition $O(L_1, L_3, G) = (\delta_1^2 \vee \delta_1^3) \wedge (\delta_3^1 \vee \delta_3^2 \vee \delta_2^3)$.

4.2.1 Implementing the Operations

We will now describe how the different operations of Chapter 3 can be implemented using obstruction functions. To check if the edge between vertices $c_i \in L_i$ and $c_j \in L_j$ is necessary, we need to verify that the conjunction of $O(c_i, c_j)$ and the current obstruction function $O(L_i, L_j, G)$ yields more valid paths between the labeled components than the current obstruction function $O(L_i, L_j, G)$. This is the case if $O(c_i, c_j)$ evaluates to FALSE (i.e. a path exists) for certain combinations of literals while $O(L_i, L_j, G)$ evaluates to TRUE for these combinations. Stated differently the new edge is necessary if $O(L_i, L_j, G)$ is *not* an implication of $O(c_i, c_j)$. To check for a subset relation we create the function

$(\overline{O(L_i, L_j, G)} \vee O(c_i, c_j))$. If this is a *tautology* (i.e. it is true for every combination of literals), then the edge is not necessary. Checking whether such a function is a tautology can be an expensive process because of the large number of combinations of literals involved.

To speed up the evaluation of the obstruction functions, we will transform them to a satisfiability test. Such a test checks whether a combination of literals exists that makes a function TRUE. If this is not possible the test outputs *unsatisfiable*. Testing whether a function is a tautology is equivalent to testing if its negation is unsatisfiable. Now we can restate the definition of a necessary edge:

Definition 4.2.2 (Necessary)**.** *For two vertices $c_i \in L_i$ and $c_j \in L_j$ and obstruction function $O(L_i, L_j, G)$, we define the edge (c_i, c_j) having obstruction function $O(c_i, c_j)$ to be* necessary *if the following function is unsatisfiable:*

$$\overline{(\overline{O(L_i, L_j, G)} \vee O(c_i, c_j))}$$

which simplifies to

$$(O(L_i, L_j, G) \wedge \overline{O(c_i, c_j)}).$$

The satisfiability problem is well-known in Boolean logic and is NP-complete (Cook, 1971, Garey and Johnson, 1979). Fortunately many high performance heuristics have been developed, see e.g. the proceedings of the International conference on Theory and Applications of Satisfiability Testing (Biere and Gomes, 2006).

Recall that in the previous chapter, after updating a connection set, new information was propagated to other connection sets by recursively updating all neighbors. Because we now use obstruction functions the propagation algorithm needs to be adapted slightly. To check if propagation is useful, we can use the new definition for necessary edges. The adapted propagation algorithm is shown as Algorithm 4.1. In the initial call, O_n is the obstruction function of the newly added necessary edge.

Algorithm 4.1 PROPAGATE (L_i, L_j, G, O_n)

```
1: O(L_i, L_j, G) ← O(L_i, L_j, G) ∧ O_n
2: for all neighbors L_k of L_i do
3:     O_n ← O(L_i, L_j, G) ∨ O(L_i, L_k, G)
4:     if (O(L_j, L_k, G) ∧ ¬O_n) is unsatisfiable then
5:         PROPAGATE (L_j, L_k, O_n)
6: for all neighbors L_k of L_j do
7:     O_n ← O(L_i, L_j, G) ∨ O(L_j, L_k, G)
8:     if (O(L_i, L_k, G) ∧ ¬O_n) is unsatisfiable then
9:         PROPAGATE (L_i, L_k, O_n)
```

After propagation, labeled components that are connected for every combination of chunks need to be merged. For this Algorithm 3.3 is used. Two labeled components were merged if their connection set was *complete*. Because obstruction functions represent combinations of chunks for which no path exists, we need to redefine completeness.

Definition 4.2.3 (Complete)**.** *The obstruction function* $O(L_i, L_j, G)$ *is* complete *iff for every combination of placements of the obstacles* $O(L_i, L_j, G) = \text{FALSE}$.

Stated differently, to check if an obstruction function is complete we need to check whether it is *unsatisfiable*.

4.2.2 Preliminary Functions

Since multiple chunks are associated with one obstacle, the associated literals are not independent. Besides the information that is contained within the obstruction functions, there is also a list of assumptions that we can add to those functions. Given moving obstacle M_i, partitioned in m_i chunks $\Delta(M_i) = \{\delta_i^1, \delta_i^2, \ldots, \delta_i^{m_i}\}$, we know the following two facts:

1. We know that M_i will be present in one of the chunks, thus $\bigvee_{p=1}^{m_i} \delta_i^p = \text{TRUE}$.
2. We also know that an obstacle will never be in two chunks at the same time. Therefore $\overline{\delta_i^p \wedge \delta_i^q} = \text{TRUE}$ for all $p \neq q$.

The above two rules apply to all k moving obstacles $M_1, M_2, \ldots, M_k$. These preliminary functions are always added as assumptions when a function is tested for satisfiability. For example in the environment shown in Figure 4.4 we know the following 11 functions to be true:

$$\delta_1^1 \vee \delta_1^2 \vee \delta_1^3 \vee \delta_1^4 \quad \overline{\delta_1^1 \wedge \delta_1^2} \quad \overline{\delta_1^1 \wedge \delta_1^3} \quad \overline{\delta_1^1 \wedge \delta_1^4} \quad \overline{\delta_1^2 \wedge \delta_1^3} \quad \overline{\delta_1^2 \wedge \delta_1^4} \quad \overline{\delta_1^3 \wedge \delta_1^4}$$
$$\delta_2^1 \vee \delta_2^2 \vee \delta_2^3 \quad \overline{\delta_2^1 \wedge \delta_2^2} \quad \overline{\delta_2^1 \wedge \delta_2^3} \quad \overline{\delta_2^2 \wedge \delta_2^3}.$$

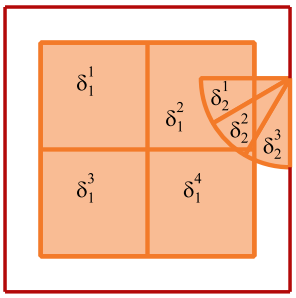


Figure 4.4: An environment consisting of two moving obstacles: a door (3 chunks) and a chair (4 chunks). The overlap of the chunks in workspace is omitted (see Figure 3.3 for details).

4.2.3 Limiting the Number of Obstruction Functions

The number of obstruction functions (and in the previous chapter the number of connection sets) is an important factor in the running time of our algorithm. The more data is

propagated throughout the roadmap, the more obstruction functions will be created. Because of the Boolean operations, the obstruction functions also get more complex. However, the larger the propagation depth, the further two labeled components are apart in terms of graph distance. If no information is propagated, our algorithm behaves the same as standard PRM. The first level of propagation is between neighboring labeled components. The next level is between labeled components that are connected via a third labeled component etc. To guarantee that no unnecessary edge is added, propagation needs to continue as long as obstruction functions are extended (see Algorithm 4.1).

Our ultimate goal is to create roadmaps that represent the connectivity of the environment without getting too dense to provide for a fast query phase. We have conducted experiments to investigate the relation between the maximum propagation depth and the density of the roadmap. The algorithm was run until a certain number of vertices were added to the roadmap. With this fixed number of vertices, the number of edges in the roadmap is a good measure of its density. We compared the maximum propagation depth with the number of edges in G. The environment consists of 10 obstacles whose placement sets were all represented by 4 chunks. The results of this experiment are shown in Table 4.1.

Propagation depth	Time (s)	Number of edges	Number of obstruction functions
0	6.6	4529	0
1	9.6	867	352
2	17.3	676	706
3	51.2	652	1525
4	133.1	652	1458

Table 4.1: Relation between propagation depth and number of edges.

As can be seen from the results, the number of edges decreases quickly when the maximum propagation depth is increased. If the propagation depth gets too large however, the number of edges hardly decreases anymore while the running time increases dramatically. This can be easily explained by realizing that an obstruction function between two labeled components is only used when a direct edge between those labeled components is tested for usefulness. The further labeled components are apart, the smaller the probability this will happen. Therefore we can limit the propagation depth such that only a very small number of useless edges is added to the roadmap while the running time decreases drastically.

We establish the maximum propagation depth automatically. As stated in Chapter 2, in standard PRM connections are usually only tried to vertices within a certain distance, the *maximum neighbor distance*. We use this distance to establish the maximum propagation depth. If two labeled components are further apart than the maximum neighbor distance then propagation stops. The distance between two neighboring labeled components can easily be defined as the distance between their center points where a center point is defined as the point with coordinates that are the average of all points in the labeled compo-

nent. If the two labeled components are not neighboring we use the cumulative distance.

Early during the execution of our algorithm there are many different labeled components consisting of only one or a few vertices. In a later stage, as the number of edges in the roadmap increases, many of these will be merged into larger labeled components. To speed up this process we start the roadmap creation process by *seeding*. By seeding we mean using standard PRM in the free space only (i.e. not colliding with the stationary obstacles nor any of the chunks). This creates labeled components between the chunks of the moving obstacles that can later act as bridges between the other labeled components. The number of vertices that are added in the seeding process is determined automatically. Ideally it should be related to the difficulty of the problem. If a problem consists of many chunks it is considered difficult. Using information gathered during the seeding process we can estimate this difficulty. In the seeding process an edge is rejected if it intersects with a chunk. We keep track of the ratio between accepted and rejected edges and use that to determine the number of vertices used for seeding. (Note that, to determine this ratio, we ignore edges that are rejected because they collide with one of the stationary obstacles.) Using the maximum number of vertices that we use as a stop-criterion for the algorithm, the ratio is used to decide how many vertices are added in the seeding phase. For example if the stop criterion is a maximum of 1000 vertices and the ratio between edges that collide with one or more chunks and non-colliding edges is 0.2 then we end the seeding phase after the addition of 200 vertices.

4.3 Experiments

Our algorithm has been implemented in C^{++}. To check the satisfiability of the Boolean functions we used MiniSat (Eén and Sörensson, 2005). This is a very efficient SAT solver that is publicly available. We have performed a few different experiments. We compared the algorithm of this chapter with a standard implementation of PRM (*standard* PRM) and with the algorithm as described in the previous chapter (the *straightforward method*). To provide for a fast query phase, we need to know which edges of the roadmap collide with which (placements of) the moving obstacles. This information is inherently available in our algorithm and for a fair comparison it should also be available to the standard PRM. Therefore, during preprocessing we collision check the vertices and edges of the standard PRM algorithm both with the stationary obstacles and the chunks. The potential collisions with one or more chunks are stored within the roadmap such that at query time this information is readily available. To gain insight in the performance gain of seeding we did all experiments both with and without seeding.

For the experiments we used three environments: **Puzzle** and **double puzzle** from the previous chapter and a new environment that resembles an office, called **office** environment. The latter is shown as Figure 4.5(a) and consists of a number of different moving obstacles. The environment represents 5 different rooms around a central staircase. The two top and two bottom rooms can be reached using ordinary doors all represented by 3 chunks. Also between the rooms are doors. Inside the rooms are desks and chairs that can

be positioned in 4 different chunks. The left room can be reached by means of a sliding door. Inside this room there are two boxes that can both be placed in 4 different chunks. In total this environment consists of about 12 million different combinations of chunks. Because of the high number of chunks, the office scene was computationally infeasible for the straightforward method.

All experiments ran until a predetermined number of vertices were added to the roadmap. In this way it is easy to compare our algorithm to standard PRM. Also all other parameters were kept constant to provide for a fair comparison; they are shown in Table 4.2. Besides the running time we also report the number of edges in the roadmap since this number relates closely to the query time (since the number of vertices is the same in all algorithms).

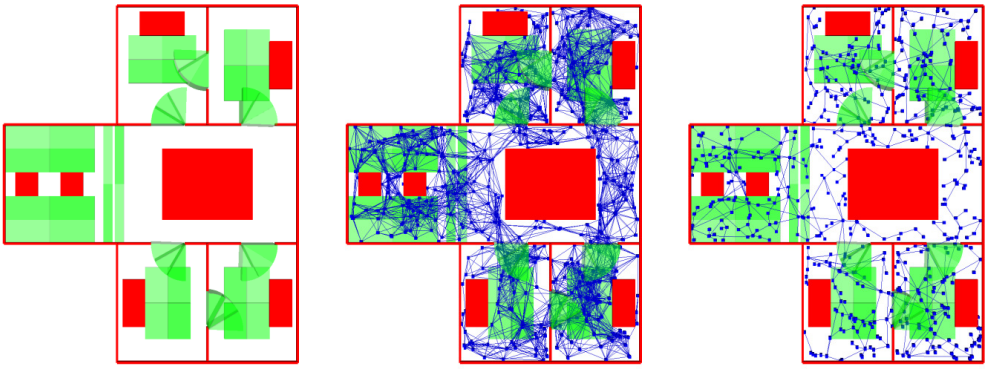
	Max. nr. vertices	Max. neighbor distance	Max. nr. neighbors
Puzzle	200	26%	10
Double puzzle	400	49%	10
Office	800	15%	10

Table 4.2: Parameter settings. The maximum neighbor distance is a percentage of the diameter of the environment.

Every experiment was repeated 50 times and the results were averaged. Results are shown in Table 4.3. As can be seen from the results, in the simple environments our algorithm outperforms standard PRM in terms of time. This can be mainly attributed to the low number of chunks and therefore the relative high cost of collision checking as compared to the time spend in manipulating obstruction functions. Because standard PRM does not have the advantage of labeled components, it is not able to save collision checks and therefore its running time is higher. In the simple puzzle environment the overhead of the satisfiability tests is relatively high. Therefore the straightforward method outperforms the Boolean logic method by a small factor. If the number of chunks gets larger, the method using Boolean logic is faster. This is already the case in the double puzzle environment that has 16 chunks. In the office environment, which is much more complex, the running times of standard PRM and Boolean logic are comparable. In all cases the complexity of the roadmaps was much lower in our algorithm, as can be seen from the number of edges, resulting in a better query time. Figure 4.5(b)+(c) shows examples of the roadmaps generated by standard PRM and our method. The connectivity of the second and third roadmaps is comparable but since in the third roadmap only useful edges were allowed, the roadmap is much sparser. As can be seen from all experiments, seeding helps by lowering the running time while keeping the number of edges equal.

	Puzzle		Double puzzle		Office	
	Time(s)	Nr. edges	Time(s)	Nr. edges	Time(s)	Nr. edges
Standard PRM	2.03	1652	7.61	3130	5.74	6368
Straightforward	0.54	207	4.03	430		
Boolean logic	0.79	207	2.42	410	5.20	902
w/o seeding	1.22	218	3.52	424	9.32	981

Table 4.3: Results averaged over 50 runs.



(a) The office environment. The chunks are shown in slightly different shades.

(b) The roadmap created by standard PRM.

(c) The roadmap created by our algorithm.

Figure 4.5: Example roadmaps.

4.4 Concluding Remarks

In this chapter we have improved the basic algorithm of Chapter 3 for motion planning in changing environments. We first pinpointed its bottlenecks and then suggested improvements. A first problem was the memory consuming connection sets that kept path existence information for every combination of chunks. If the environments become more complex the memory consumption becomes too high for this approach. Therefore we have switched from maintaining full connection information to maintaining only collision information. While this makes no difference from a theoretical perspective, in practice it turns out to be a major gain. Secondly, the operations concerning the connection sets were rather brute force. To speed up these operations, we translated the problem to the domain of Boolean logic. While the problem of checking the completeness of an obstruction function (that replaced the connection set) still remains difficult, within this domain many very fast heuristics have been created. To determine if two labeled components can be merged, we need to check whether the related obstruction function is complete. This problem was translated to a satisfiability test that could be performed quite efficiently in

practice.

We also presented two heuristics. The first heuristic limited the propagation depth. Because of this limitation, information in the obstruction functions is not complete anymore and thus unnecessary edges could be added to the roadmap. In our experiments we showed that the number of unnecessary edges was very low while the gain in performance was huge. We also described a method to automatically determine the propagation depth without introducing additional parameters. The second heuristic speeds up the roadmap creation process by first adding some vertices in the free space. This heuristic prevents that in the early phases of the algorithm many small labeled components are created that later need to be merged.

We conducted experiments that showed that our algorithm performs efficiently in realistic environments. Compared to standard PRM, our algorithm creates much sparser roadmaps (while maintaining the same connectivity) that provide for fast queries and replanning.

Part II

Movable Obstacles

CHAPTER 5

Creating a Framework

This chapter addresses the problem of navigating an autonomous moving robot in an environment with both stationary and movable obstacles. If a movable obstacle blocks the path of the robot attempting to reach its goal configuration, the robot is allowed to alter the placement of the obstacle by manipulation (e.g. pushing or pulling), to clear its path. In this chapter we present a framework for solving path planning problems amidst movable obstacles.

5.1 Introduction

In Part I we have looked at path planning problems in which the environment was subject to changes by other factors than the robot (e.g. humans, the wind that closes a door etc.). We have presented an effective algorithm that makes the path planning for a robot robust against those changes. In this chapter we look at problems in which the robot itself is able to change the environment. The goal is defined in terms of a configuration for the robot. While navigating to this goal, the robot is allowed to move obstacles out of the way if necessary.

While the ability of being able to move obstacles out of the way is a clear extension to the domain of path planning, the problem of navigating a robot in an environment inhabited by both stationary and movable obstacles has been largely unaddressed. Our motivation comes from an ultimate wish to automatically generate visually-convincing motions for computer-controlled entities in virtual environments and games. Although our methods may be applicable to other domains, in this chapter we aim at creating convincing paths for moving entities. By convincing we mean that the navigation and manipulation strategy of the robot resembles human behavior. A rigorous way to plan motions

among stationary and moving obstacles would be to consider the problem in the high-dimensional composite configuration space of the robot and the movable obstacles. Unfortunately, in all but the simplest instances the complexity is too high to efficiently find a solution. However, with our motivation in mind, we believe that the above costly approach is not required. A human-like robot moving in a realistic (e.g. office) environment will plan his or her motions on the basis of knowledge about the layout of the stationary features of the environment. While executing the path, the robot may encounter movable obstacles such as a chair or a trolley with supplies standing in the way, or a door that is closed. If the robot encounters such movable obstacles it will try to move them out of the way by manipulating them or by getting around them, so that it will be able to continue its predetermined path. Only if the required manipulations get truly complicated or require a lot of effort, the robot may start to explore alternative paths toward the goal. Since it is our aim to provide convincing human-like motions of robots, we want the planner to follow a similar strategy.

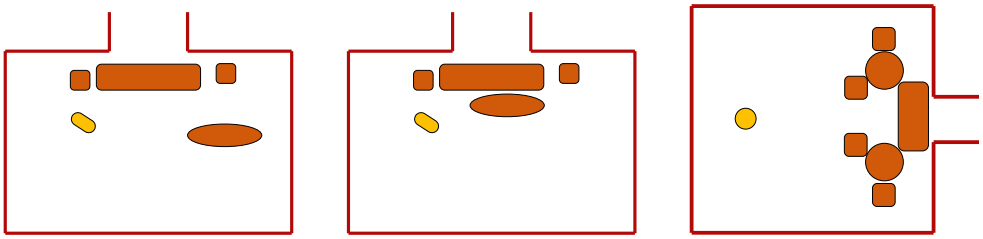
The difference between our approach and that in the composite configuration space resembles the difference between *decoupled* (Erdmann and Lozano-Pérez, 1987) and *centralized* (Schwartz and Sharir, 1983c) planning for multiple robots. As in decoupled planning we approach the problem in a lower-dimensional configuration space taking into account stationary features only. The resulting path is subsequently adjusted to resolve collisions with non-stationary features.

Rearrangement planning (Alami et al., 1994, Ben-Shahar and Rivlin, 1998) is a problem closely related to path planning among movable obstacles. In rearrangement planning, a robot also navigates in an environment among movable obstacles, but the goal is not defined in terms of a configuration for the robot, but rather in terms of configurations for the movable obstacles. Even though rearrangement planning has had considerable attention over the years, path planning among movable obstacles has not.

Wilfong (1988) showed that path planning among movable obstacles is NP-hard by creating a reduction from 3-SAT. He does so by showing that the clauses of a 3-SAT instance can be represented in a path planning problem among movable obstacles. Chen and Hwang (1991) created a grid based planner that heuristically tries to minimize the cost to move obstacles out of the way. To reduce the cost, they only considered a very limited number of different states. With their planner they solved some simple but realistic problems.

Another planner is the one developed by Stilman and Kuffner (2005). Their global approach uses the fact that the free space of the robot consists of multiple connected components. If start and goal are not in the same connected component then the robot uses manipulation to move obstacles to try to join connected components. To detect if a manipulation action has succeeded, a grid based approach is used. To manipulate an obstacle, contact points are sampled and a set of primitive actions is applied to those points. The candidate obstacles for manipulation are found by using an A* search on a grid from the current position of the robot to the goal. Obstacles encountered during this search are the candidates for manipulation. In case of failure, backtracking is used. The authors prove that their planner is resolution complete for a class of problems they call LP_1, analogous to the LP_1 class in rearrangement planning (Ben-Shahar and Rivlin, 1998). The LP_1

class contains problems in which disjoint components of the free space can be merged by moving a single obstacle. Stated differently, only if an obstacle blocks the path of the robot directly, it will be manipulated. Problems that require the manipulation of an obstacle that blocks another obstacle will not be solved. An example of an LP_1 problem is shown as Figure 5.1(a).



(a) Moving only the couch is enough to open the path to another room (LP_1).

(b) First the oval table has to be moved before the couch can be moved (LP_2).

(c) The couch is blocked by the round tables which in their turn are blocked by the chairs (LP_3).

Figure 5.1: Rooms consisting of couches, tables and chairs. The robot is represented as the light colored object. The goal for the robot is to leave the room.

Solving a path planning problem among stationary and movable obstacles can be regarded as finding an alternating sequence of motions toward a movable obstacle and manipulation of this obstacle until the goal configuration is reachable without further manipulation, we call such a sequence a *manipulation plan*. In this chapter a novel framework is presented based on the expansion of a so-called *action tree*. This action tree represents the complete set of motions toward movable obstacles and manipulations of movable obstacles in every possible order. Finding a manipulation plan is equivalent to finding a path in the action tree. The framework considers problems in the class of *LP*, i.e., the set of problems that can be solved by a sequence of manipulations (Figures 5.1(b)+(c)). In contrast to the LP_1 class, the *LP* class contains problems for which movable obstacles have to be manipulated that do not directly block the path of the robot. For example, the manipulation of a movable obstacle can be blocked by another movable obstacle for which the manipulation is blocked by yet another movable obstacle etc. During the manipulation of a movable obstacle, the other movable obstacles are assumed to be stationary. An example of a manipulation plan is shown in Figure 5.2.

This part is arranged as follows. In this chapter a framework is presented for motion planning amidst movable obstacles. First in Section 5.2, we will formally define the problem and state some definitions. Next, in Section 5.3 the action tree and its properties are presented. In Chapter 6 we will describe how to create an efficient planner based on the framework. In Section 6.1 we distinguish some building blocks necessary to realize the action tree and describe how to implement these. Heuristics to guide the search process in the action tree are detailed in Section 6.2. Techniques to improve path quality are described in Section 6.3. Finally, results of our experiments are presented in Section 6.4.

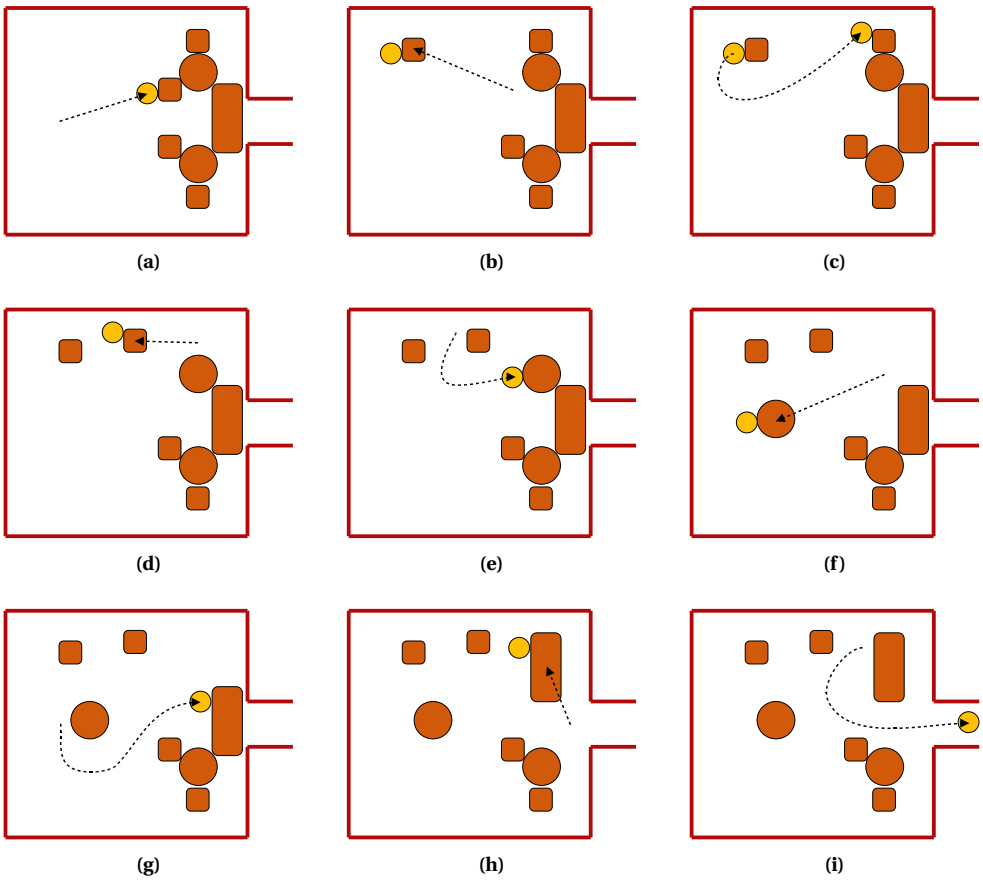


Figure 5.2: Complete example in which the robot needs to maneuver out of the room. Note that the manipulation plan consists of an alternating sequence of motions toward a movable and the manipulation of a movable.

5.2 Problem Statement and Preliminaries

Let R be a robot defined in a workspace that, besides stationary obstacles, contains k movable obstacles (or movables for short) $M = \{M_1, M_2, \ldots, M_k\}$. The robot and the movables are assumed to be closed sets. A movable M_i cannot move by itself, but can only be moved by R if M_i is first grasped by R. The distance between two closed objects O_i and O_j is denoted by $d(O_i, O_j)$, which is the Euclidean distance between the two points of O_i and O_j that are closest together. If the Euclidean distance between points p and q is denoted by $e(p,q)$ and $\text{INT}(O)$ denotes the interior of object O, then:

$$d(O_i, O_j) = \begin{cases} \min\limits_{p \in O_i, q \in O_j} e(p,q) & \text{IF } \text{INT}(O_i) \cap \text{INT}(O_j) = \emptyset \\ \infty & \text{IF } \text{INT}(O_i) \cap \text{INT}(O_j) \neq \emptyset \end{cases}$$

M_i is only said to be grasped by R if $d(R, M_i) = 0$. If M_i is grasped by R, the combination of R and M_i is denoted by M_i^R. In this chapter, manipulation of M_i is defined as a joint motion of R and M_i during which the point at which M_i is grasped remains the same; M_i^R acts as a rigid body. A physical model is used to define the set of possible motions of M_i^R, depending on the forces that R is able to apply to M_i. This model can also describe the potential positions on the boundary of M_i where grasps are allowed. Since the physical model is highly dependent on the specific capabilities of R and the properties of the environment, we use a simplified model in which R is capable of manipulating a movable in any direction and grasp it whenever $d(E, M_i) = 0$. Other models can be used without affecting the algorithm. We do not allow R to manipulate two movables at the same time.

Our goal is to create a manipulation plan for R from a given start to a given goal configuration. The behavior of R should be convincing compared to the behavior of its real (e.g. human) counterpart. For example, if a human has the choice between moving multiple obstacles that block a door and taking a small detour, it will most likely do the latter. Moreover most problems will be solved locally, i.e. a blocking movable will often not have to be pushed a long distance before R will be able to move around it.

In this chapter we introduce the concept of an action tree. The action tree is a general framework for solving path planning problems among movable obstacles. In contrast to a roadmap graph, which represents the free configuration space, the action tree represents the different actions R can perform given the configurations of the movables. Using this framework, a planner is presented that uses heuristics to guide the search process through the action tree in order to efficiently find a solution.

5.3 Action Tree

In the basic path planning problem all obstacles are stationary and only the robot itself changes its configuration. One single description for the configurations of the obstacles suffices during the execution of the algorithm. In our problem setting the movables can also change their placements during the execution of the planning algorithm. Therefore R operates in a constantly changing environment but is itself responsible for the changes. To

describe the configurations of the movable obstacles we introduce the notion of a *worldstate*. A worldstate encodes the placements of all non-stationary obstacles, i.e. robot R and movables M.

Definition 5.3.1 (Worldstate)**.** *A* worldstate $W = (w_r, w_1, \ldots, w_i, \ldots, w_k)$ *describes the configuration* w_r *of* R *and the configurations of all* k *movables* $(w_1, \ldots, w_i, \ldots, w_k)$ *in* M*.*

If R is placed at configuration w_r this is denoted by $R[w_r]$, analogous if movable M_i placed at configuration w_i, this is denoted by $M_i[w_i]$. A worldstate is essentially a point in the composite configuration space of R and M. As stated before however, we will not solve the planning problem in this composite configuration space.

We will define two basic actions that are used to transform one worldstate into another. The first action is GRASP$(M_i[w_i])$. A successful call to GRASP$(M_i[w_i])$ transforms a worldstate $W = (w_r, w_1, \ldots, w_i, \ldots, w_k)$ into a worldstate $W' = (w'_r, w_1, \ldots, w_i, \ldots, w_k)$ satisfying $d(R[w'_r], M_i[w_i]) = 0$, else it reports failure. To grasp a movable, R tries to move from its current configuration w_r to a randomly selected configuration w'_r satisfying $d(R[w'_r], M_i[w_i]) = 0$. If M_i cannot be grasped (because R is not able to reach the boundary of M_i) the action reports failure. Note that if $d(R[w_r], M_i[w_i]) = 0$ (R is already grasping M_i), the action GRASP$(M_i[w_i])$ is by no means forbidden. Grasping the currently grasped obstacle has the effect of re-grasping M_i at another location. A re-grasp can be useful if the current grasp does not suffice to clear the path, for example if the room for M_i^R to maneuver is limited (Figure 5.3).

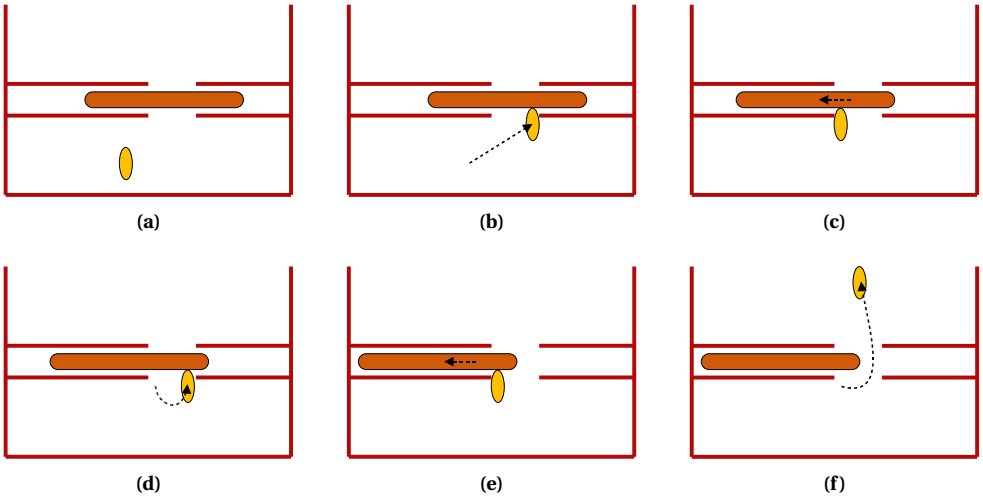


Figure 5.3: An example in which the robot has to re-grasp a movable obstacle in order to clear its path.

The second action is MANIP(M_i^R), which attempts to manipulate the currently grasped movable M_i. If a call to MANIP(M_i^R) is successful, it transforms worldstate $W = (w_r, w_1, \ldots, w_i, \ldots, w_k)$ into another worldstate $W' = (w'_r, w_1, \ldots, w'_i, \ldots, w_k)$. Recall

that manipulating M_i^R does not change the relative positions of R and M_i, i.e. the grasp is not changed. Stated differently, MANIP(M_i^R) results in a joint motion of M_i and R satisfying the constraints imposed by the underlying physical model from their current configuration to a randomly selected configuration. If MANIP(M_i^R) does not succeed, for example because the physical model forbids the manipulation or a collision occurs, MANIP(M_i^R) reports failure. We now define the *action tree* T_A.

Definition 5.3.2 (Action Tree)**.** *An* action tree T_A *is a (directed) tree that describes the space of all potential valid actions. Every node of a tree corresponds to a worldstate. The edges between the nodes represent the action* (GRASP() or MANIP()) *that results in the transformation from one worldstate to another. Every child node succeeds its parent in time, i.e. moving from child to parent does not result in a valid path.*

With every node η of T_A, a worldstate $W(\eta)$ is associated. An action tree T_A consists of two types of nodes: *manipulation nodes* are the result of a call to MANIP(), *grasp nodes* are the result of a call to GRASP(). A manipulation node will never need to have a child node that is also a manipulation node. This can be seen as follows. Suppose manipulation node η has a child node η' that is also a manipulation node. Then node η' could also have been a direct child of the parent of η (i.e. a sibling of η). The same holds for grasp nodes, a grasp node will never have another grasp node as a child because the second movable could have been grasped at once without first grasping the first one. Therefore a grasp node will always have a manipulation node as a parent and vice versa. Recall that we do not allow R to grasp or manipulate two movables at the same time.

After initializing the algorithm by associating the initial worldstate with the root node η_s of T_A, the algorithm constructs T_A by *expanding* nodes. The expansion of a manipulation node η adds one or more child nodes to η in which a movable is grasped at a random configuration; this may also be a re-grasp of the currently grasped movable. The expansion of a grasp node η adds one or more child nodes to η that manipulate the currently grasped movable.

After the addition of a manipulation node η to T_A, R may be able to reach its destination. If a grasp node is added, this will never be the case, since no movables have changed their configuration. After manipulating a movable though, a new path may have emerged that brings R to its goal. Therefore, after the addition of a manipulation node to T_A, the algorithm checks whether R can reach its goal configuration. If this is the case, a special node is added to the tree that represents the motion of R from η to the goal. Now the algorithm can terminate. If no path to the goal can be found, the construction of T_A continues.

A manipulation plan is a finite alternating sequence of grasp and manipulation actions. In an action tree such a plan is represented by a path from the root to a leaf node. An example of an environment with an accompanying action tree is shown as Figure 5.4. Here, the robot has to maneuver from the top left to the top right of the environment. The root node η_s of the tree represents the initial worldstate. In this example, each child node of a manipulation node tries to grasp a different movable obstacle. Thus since there are four movable obstacles, η_s has four child nodes, each representing a random grasp of one of the four movables. Here, only one of these nodes is feasible, the other movable obstacles are not reachable by R, and thus the corresponding nodes result in a dead end.

Therefore only the node grasping M_1 can be expanded. A number of manipulation nodes are added that represent push/pull actions on M_1. Two of these fail to open the passage to the hallway, thus only M_1 can be (re-)grasped from this node. From the node that successfully opens the passage to the hallway, three movables (M_1, M_2 and M_3) can be grasped and manipulated. If M_2 is chosen for manipulated, the passage to the goal is opened and the goal can be reached. In Figure 5.5 the nodes that lead to a solution are shown together with their corresponding worldstates.

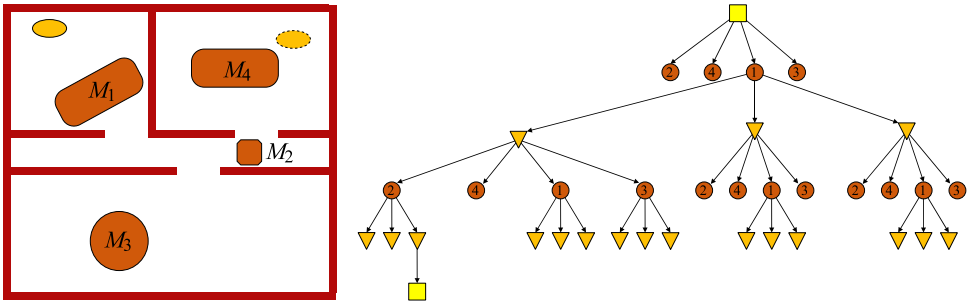


Figure 5.4: An environment consisting of 4 movables and a corresponding action tree.

If nodes are selected for expansion randomly and every grasp is approximated by a grasp node in T_A at most a distance $\epsilon > 0$ away, then we conjecture that our framework is probabilistically complete. This means that if the random expansion process is repeated for a sufficient amount of time, an action tree will eventually contain a manipulation plan (represented by a path in T_A from the root node to a leaf node) for a feasible query.

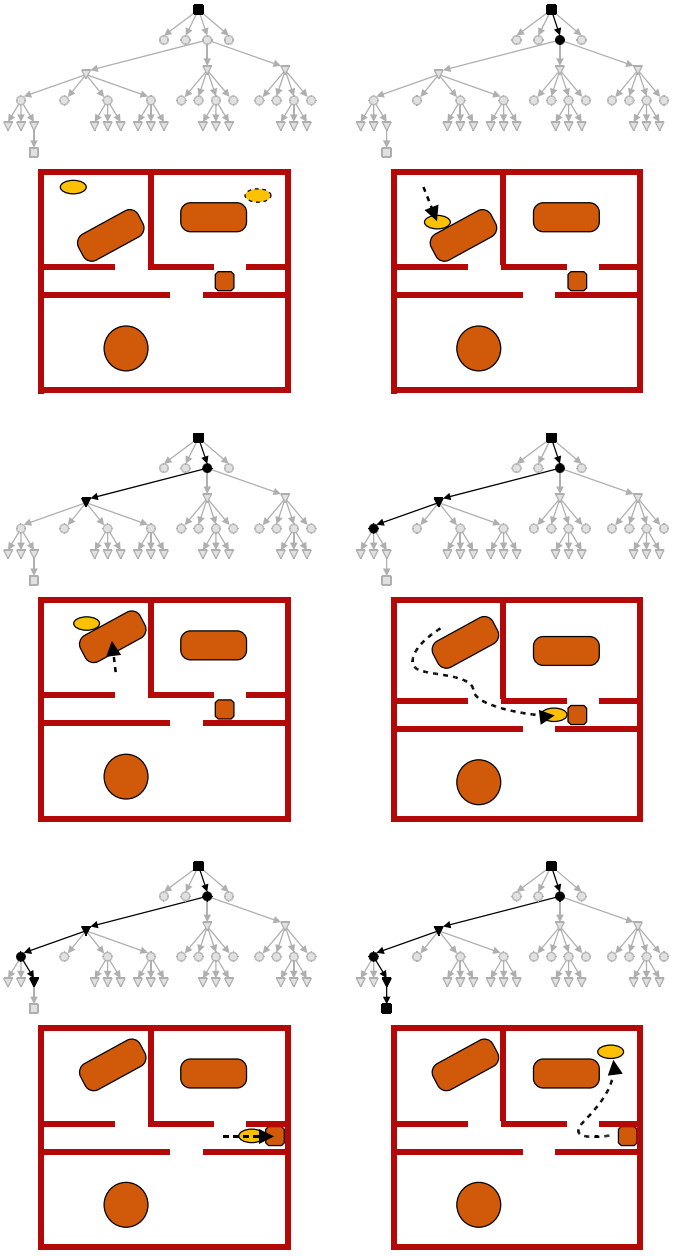


Figure 5.5: The path through the tree that leads to a solution together with the corresponding world-states.

CHAPTER 6

Planning a Path

In this chapter we create a planner based on the framework of the previous chapter. First we will describe how to implement such a planner. If the problem gets more complicated (i.e. more movable obstacles are involved), the construction of an action tree until it contains a solution (e.g. using a breadth-first expansion process) may become infeasible. Therefore, we present heuristics inspired by problems encountered in practical applications that guide the construction process of the action tree in favor of promising tree nodes.

6.1 Realization of the Action Tree

In this section we will describe how to implement the necessary building blocks for the construction the of action tree. These consist of the following four types:

- Navigating the robot.
- Checking if the robot can reach its goal given a certain worldstate.
- Grasping a movable.
- Manipulating a movable.

6.1.1 Navigating the Robot

To be able to grasp an obstacle the robot needs a means to navigate. In principle any path planning technique resulting in feasible (i.e. collision-free with respect to the static obstacles) paths can be used. Here, we choose to use the PRM method as presented in Chapter 2.

A roadmap G is created for the robot regardless of the movable obstacles, since their configurations can change while executing the algorithm. During the execution edges and vertices in G that collide with one or more movables will be temporarily invalidated. To make G more robust against invalidation of edges we add cycles to it using the methods described in Section 2.3. The cycles help in reaching grasp configurations and also provide for alternative paths if R is not able to continue its path (e.g. because R is unable to manipulate a movable that blocks its path).

The action that changes the environment is the MANIP(M_i^R) action. A manipulation transforms a worldstate $W(p(\eta)) = (w_r, w_1, \ldots, w_i, \ldots, w_k)$ into a worldstate $W'(\eta) = (w'_r, w_1, \ldots, w'_i, \ldots, w_k)$, where $p(\eta)$ is the parent of node η. To be able to quickly update G for a given worldstate, at every manipulation node, a list of invalidated edges of G is stored. This list is equivalent to the list of $p(\eta)$ except for edges that represent a path that intersects with either $M_i[w_i]$ or $M_i[w'_i]$. Only these edges have to be checked for collision against M_i. To find these efficiently, the endpoints of the edges are stored in a Kd-tree (Bentley, 1975). A Kd-tree allows for quickly identifying the edges that are close to w_i (in workspace) so that they can be checked for collision. When the root node is added to T_A, the graph G needs to be checked once against all movables in order to create the initial list of invalidated edges. Since a grasp node only affects the configuration of R, grasp nodes simply copy the list of invalidated edges from their parent since they do not change the environment.

6.1.2 Checking whether the Robot can Reach its Goal

If the environment has been changed by R, it may be able to reach its goal configuration. Therefore after a manipulation node has been added to T_A, the roadmap G is updated according to the new configuration of the manipulated movable. Next, the current configuration of R can be added to G. Finally a shortest path query can be executed to check whether R can reach its goal. If this is the case, a special (i.e. not a MANIP() nor GRASP()) node is added to T_A that represents the last part of the manipulation plan moving R to its goal configuration.

6.1.3 Grasping a Movable Obstacle

The GRASP($M_i[w_i]$) action creates a path for R from its current configuration w_r to w'_r where $d(R[w'_r], M_i[w_i]) = 0$. Here, w'_r is a randomly chosen configuration in which R grasp M_i. Depending on the application, the selection of grasp configurations can be customized. As previously described, G is updated with respect to the current worldstate. Next, w_r and w'_r are attempted to be connected to G. If this succeeds, a query can be executed between w_r and w'_r. If the query is successful, a grasp node is added to T_A.

6.1.4 Manipulating a Movable Obstacle

After R has grasped M_i by means of a GRASP($M_i[w_i]$) it will try to manipulate M_i by executing the MANIP(M_i^R) action. To jointly navigate the robot and the movable M_i^R we will

use an approach based on the Rapidly-exploring Random Trees (RRT) algorithm developed by LaValle and Kuffner (2001). An RRT is aimed at growing a tree from a given start configuration in an attempt to cover the free space. Here, the RRT operates in the configuration space of M_i^R where all movables M_j with $i \neq j$ are considered stationary. An RRT quickly generates many different paths away from the start configuration. This property is very useful in our situation because our target is to move M_i^R away from its current configuration and consider multiple goal configurations.

The vertices in the RRT represent configurations for M_i^R. (nee Note the difference between the action tree that contains nodes, and the RRT that contains vertices.) The edges represent paths between them. The RRT algorithm works as follows. First the start configuration is added to the RRT as a vertex. Next, a random (not necessarily collision-free) configuration $c_r = (w_r, w_i)$ with $d(R[w_r], M_i[w_i]) = 0$ for M_i^R is generated. The nearest configuration c_n in the RRT to c_r is found (not necessarily a vertex of the RRT) and a path from c_n to c_r is tested for collision. If c_r is reached, it is added as a vertex to the RRT together with the edge (c_n, c_r). If a collision occurs before c_r is reached, the last collision-free configuration c_s is added to the RRT together with the edge (c_n, c_s). This process is repeated until some maximum number of attempts to extend the RRT is performed. This number does not need to be high since if the algorithm fails in finding an appropriate RRT extension while one exists then deeper in T_A other manipulation nodes for the same movable may succeed. An example of an RRT is shown as Figure 6.1.

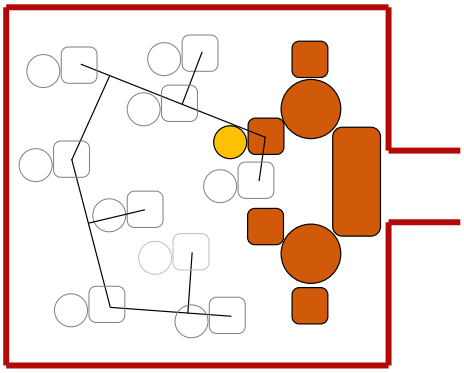


Figure 6.1: Example of an RRT. At the vertices the corresponding configuration of M_i^R is shown.

The RRT algorithm needs to check whether a path exists between c_n and c_r. To verify the existence of such a path a *local planner* is used as is the case in PRM (Section 2.1). Recall that given two configurations, the task of the local planner is to check whether the path between them is collision-free. In our algorithm, the generated path also needs to comply with the physical model for the specific type of manipulation (e.g. pushing/pulling). The local planner is allowed to use any physical model as long as it is capable of deciding whether a manipulation path exists between two configurations or, in case of a collision, what the closest reachable configuration is to c_r.

Since the RRT algorithm works incrementally by nature, it is easy to implement our MANIP() action using an RRT. Every grasp node η contains an RRT. The child nodes of η (which are all manipulation nodes) represent the vertices of that RRT. At such a vertex R is no longer on G. Therefore the manipulation node is only valid if R can be reconnected to G.

We have now described all building blocks necessary for expanding a node. Algorithm 6.1 summarizes this by showing how a node is expanded. Its parameters are the node η, the robot graph G and the goal configuration for the robot w_g. In the next section, we will describe heuristics to tailor this concept to practical problem settings.

Algorithm 6.1 EXPANDNODE (η, G, w_g)

```
1:  if type[η] = MANIPULATIONNODE then
2:    for all M_i ∈ M do
3:      w_r ← RANDOMGRASP(M_i) {generate random grasp}
4:      η' ← GRASP (η, w_r) {grasp M_i at a random position}
5:      if η' ≠ NULL then
6:        add η' to children[η] {add a grasp node as a child to η}
7:  else {η is a grasp node}
8:    for i ← 0 to MAXRRTVERTICES do
9:      η' ← EXTENDRRT (η) {η is container of the RRT }
10:     if η' ≠ NULL then
11:       add η' to children[η] {add a manipulation node as a child to η}
12:       UPDATEROBOTGRAPH(G, η') {store invalidated edges in η'}
13:       if PATHEXISTS(G, η', w_g) {is there a path to w_g?} then
14:         return PATHFOUND {the goal can be reached}
```

6.2 The Planner

Although breadth-first expansion of T_A guarantees an exhaustive exploration of all possible sequences of grasps and manipulations, this strategy becomes computationally expensive or even infeasible when the number of movables is large. In practical settings however, only a small subset of the movables are involved in the final manipulation plan of R. In that respect, many nodes of T_A will most likely not contribute to the final solution. For example, if a manipulation plan needs to be created that moves R from room A to B, then often the movables present in room C will not be part of the final manipulation plan. On the other hand, a movable that is not directly impeding the path of R, may be blocking the manipulation of another movable and because of that be part of a feasible manipulation plan. Because of the above considerations, expanding nodes in a breadth-first search manner is not the most efficient way to find a manipulation plan. In this section we will describe how to focus the node expansion process toward promising nodes, such that a solution can be found rapidly without affecting the probability of finding a solution. To be

able to focus the expansion process we will expand nodes only once, i.e. only leave nodes are expanded.

6.2.1 Selecting a Node for Expansion

Instead of expanding T_A in a breadth-first manner, we will use heuristics to guide the expansion process. To be able to do this, every node η is assigned a probability $q(\eta)$. Let $\text{CHILDREN}(\eta)$ be the child nodes of η. If node η has one or more children, we require that

$$\sum_{\nu \in \text{CHILDREN}(\eta)} q(\nu) = 1. \tag{6.1}$$

Nodes are now selected for expansion by creating a path from the root node to a not yet expanded node (a leaf). Starting at the root (having probability 1), a child node is selected based on its probability. This process is repeated until a leaf is reached. Then that leaf is selected for expansion.

After expanding a node η, all its children are initially assigned equal probabilities. Later on in the process we will increase or decrease probabilities for a node depending on the progress that is made. For example, if η is successful in getting closer to the goal, $q(\eta)$ is increased. Increasing or decreasing $q(\eta)$ should not violate Equation 6.1. In addition we must assure that $q(\eta)$ never equals 0 as this would exclude its selection. If $q(\eta)$ is already high (e.g. $q(\eta) = 0.95$), there is little reason to increase it further. Using the above requirements, we use the following procedure to adapt the probabilities: a fraction $f \in [0,1]$ is used to increase $q(\eta)$ of a node η. We denote the updated value of the probability of η by $q'(\eta)$. The siblings of node η are denoted by the set $S(\eta)$.

We define *update rule 1* as

$$q'(\eta) = (1 - q(\eta))f + q(\eta) \tag{6.2}$$

$$\forall_{\nu \in S(\eta)} \; q'(\nu) = q(\nu)(1-f). \tag{6.3}$$

Lemma 6.2.1. $\sum_{\nu \in S(\eta)} q'(\nu) + q'(\eta) = 1$ *after update rule 1.*

Proof: Combining Equations 6.2 and 6.3 yields

$$\begin{aligned}
\sum_{\nu \in S(\eta)} q'(\nu) + q'(\eta) &= \sum_{\nu \in S(\eta)} q(\nu)(1-f) + (1 - q(\eta))f + q(\eta) \\
&= \sum_{\nu \in S(\eta)} q(\nu) - \sum_{\nu \in S(\eta)} q(\nu)f + f - q(\eta)f + q(\eta) \\
&= \sum_{\nu \in S(\eta)} q(\nu) + q(\eta) - f\left(\sum_{\nu \in S(\eta)} q(\nu) + q(\eta)\right) + f. \qquad (6.4)
\end{aligned}$$

We know from Equation 6.1 that

$$\sum_{\nu \in S(\eta)} q(\nu) + q(\eta) = 1$$

which after the combination with Equation 6.4 yields

$$\sum_{v \in S(\eta)} q'(v) + q'(\eta) = 1.$$

■

Similarly if, after the expansion of a node, no or little progress is made, the probabilities of selecting that node should be decreased. The following procedure is used to decrease the probabilities of node η if η has at least one sibling. We define *update rule 2* as

$$q'(\eta) = q(\eta) - f\,q(\eta) \tag{6.5}$$

$$\forall_{v \in S(\eta)} \; q'(v) = \frac{1 - q(v)}{|S(\eta)| - 1 + q(\eta)} f\,q(\eta) + q(v). \tag{6.6}$$

Lemma 6.2.2. $\sum_{v \in S(\eta)} q'(v) + q'(\eta) = 1$ *after update rule 2.*

Proof: Combining Equations 6.5 and 6.6 yields

$$\begin{aligned}
\sum_{v \in S(\eta)} q'(v) + q'(\eta) &= \sum_{v \in S(\eta)} \left(\frac{1 - q(v)}{|S(\eta)| - 1 + q(\eta)} f\,q(\eta) + q(v) \right) + q(\eta) - f\,q(\eta) \\
&= \frac{f\,q(\eta)}{|S(\eta)| - 1 + q(\eta)} \sum_{v \in S(\eta)} (1 - q(v)) + \sum_{v \in S(\eta)} q(v) + q(\eta) - f\,q(\eta).
\end{aligned}$$

Combining with Equation 6.1:

$$\begin{aligned}
\sum_{v \in S(\eta)} q'(v) + q'(\eta) &= \frac{f\,q(\eta)}{|S(\eta)| - 1 + q(\eta)} \sum_{v \in S(\eta)} (1 - q(v)) + 1 - f\,q(\eta) \\
&= \frac{f\,q(\eta)}{|S(\eta)| - 1 + q(\eta)} \left(|S(\eta)| - \sum_{v \in S(\eta)} q(v) \right) + 1 - f\,q(\eta) \\
&= \frac{f\,q(\eta)}{|S(\eta)| - 1 + q(\eta)} (|S(\eta)| - (1 - q(\eta))) + 1 - f\,q(\eta) \\
&= f\,q(\eta) \frac{|S(\eta)| - 1 + q(\eta)}{|S(\eta)| - 1 + q(\eta)} + 1 - f\,q(\eta) \\
&= 1.
\end{aligned}$$

■

Selecting a (leaf) node for expansion is now a process of starting at the root node and repeatedly selecting one of the child nodes based on its probability until a leaf is reached. We have conducted experiments with two different procedures of selecting a child node.

The first method selects a child node based on a biased random procedure: *random selection*. The bias is dependent on $q(\eta)$. Thus if a node contains three child nodes with respective probabilities $\{0.3, 0.5, 0.2\}$ then those values equal the probabilities that these nodes are selected. The second method is to simply select the node with the largest probability: *deterministic selection*. We conducted experiments comparing both methods, the results are shown in Section 6.4. If $q(\eta)$ is increased (at the expense of its siblings), this only increases the probability that η is selected given that $p(\eta)$ has been selected first. Thus, increasing $q(\eta)$ only locally changes T_A. Such a local change does not have an influence on the node selection process of nodes closer to the root even though increasing $q(\eta)$ indicates that the node made progress toward the goal (as will be detailed in the next sections). To solve this issue, the increased probability of η is *propagated* along the path from η to the root node η_s, i.e. the probabilities of all ancestors of η are increased as well. We must make sure however that after a few updates, the probabilities of nodes higher in the tree do not become too high. If we consider the path from η to the root, the further away a node η' is from η, the less similarity between $W(\eta)$ and $W(\eta')$. A simple measure of similarity between nodes is their difference in depth in T_A. Let $\Delta(\eta)$ denote the depth of a node η in T_A. While propagating, we define a local f value by multiplying f by a factor: $f_l = f\frac{\Delta(\eta')}{\Delta(\eta)}$. The value f_l is used to adapt the probability of a node $q(\eta')$. This ensures that the change in probability is decreased when the distance between η' and η in the tree increases. The procedures to increase and decrease the probabilities of nodes are shown as Algorithms 6.2 and 6.3. The value Δ_s is the depth of the leaf node that initiated the propagation.

Algorithm 6.2 INCREASEPROBABILITY (η, f, Δ_s)

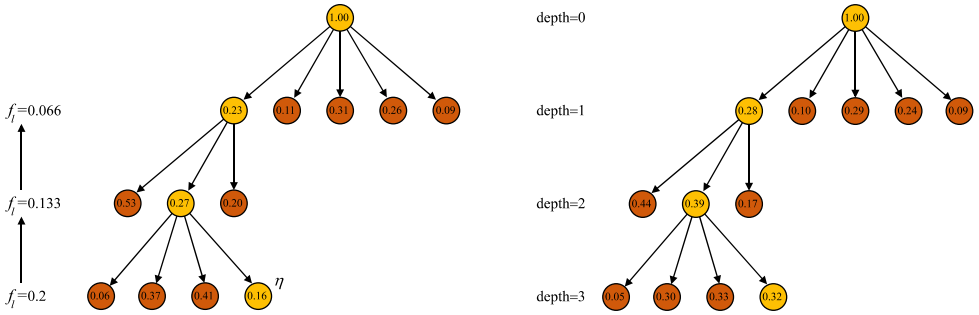
if $\eta = \eta_s$ **then**
  return {root node reached}
$f_l = f \times \frac{\Delta(\eta)}{\Delta_s}$ {define local f}
$q'(\eta) = (1 - q(\eta))\, f_l + q(\eta)$ {use f_l to increase probability}
for all $v \in S(\eta)$ **do**
  $q'(v) = q(v)(1 - f_l)$ {use f_l to decrease probability}
INCREASEPROBABILITY $(p(\eta), f, \Delta_s)$

Algorithm 6.3 DECREASEPROBABILITY (η, f, Δ_s)

if $\eta = \eta_s$ **then**
  return {root node reached}
$f_l = f \times \frac{\Delta(\eta)}{\Delta_s}$ {define local f}
if $|S(\eta)| > 1$ **then**
  $q'(\eta) = q(\eta) - f_l\, q(\eta)$ {use f_l to decrease probability}
  for all $v \in S(\eta)$ **do**
    $q'(v) = \frac{1-q(v)}{|S(\eta)|-1+q(\eta)} f_l\, q(\eta) + q(v)$ {use f_l to increase probability}
DECREASEPROBABILITY $(p(\eta), f, \Delta_s)$

An example of increasing the probability of a node is shown in Figure 6.2. Since only

the nodes (and their children) on the path from η to the root are updated, the cost of the propagation is $O(\Delta(\eta))$.



(a) The probability of node η is increased using a fraction of 0.2. While propagating, f_l is used to adapt the node probabilities. The value of f_l varies with the tree depth.

(b) The resulting probabilities after propagation using Algorithm 6.2.

Figure 6.2: Increasing the probability of a node and propagating through T_A. The values in the nodes are the probabilities.

The value of the fraction f determines the global behavior of the algorithm. If f is small, the differences between the probabilities of the nodes will not be large, resulting in a breadth-first type of expansion. If f is high however, a small number of nodes will receive a high probability, resulting in a depth-first type of expansion.

6.2.2 Adapting Probabilities

As stated before, often only a small subset of the movables M will be blocking the path of R. Therefore we will increase the probability of manipulating movables that actually block the path of R. We distinguish two types of blocking: a movable M_i can block the path of R either *directly* or *indirectly*. Directly blocking means that the shortest path in G to the goal of R actually collides with M_i, indirectly blocking means that some M_j, $i \neq j$ blocks the manipulation of M_i.

Directly Blocking Movables

After the expansion of a manipulation node η, it is checked whether R can reach its destination (Section 6.1). If no path to the goal is found, it is necessary to manipulate movables. Selecting a good movable candidate for manipulation involves taking into account our target of creating convincing paths for R. One of the properties of such a path is that a small detour is favorable over manipulating many movables (Figure 6.3). Therefore a second version of G is created in which no edges are invalidated but rather get a penalty when colliding with one of the movables. This graph is denoted by G^p. The cost of traversing an edge with a penalty in G^p is the sum of the cost of traversing that edge in G and a

penalty value. Using G, G^p can be constructed by assigning a penalty to edges instead of invalidating them. No additional collision checks are necessary to construct G^p.

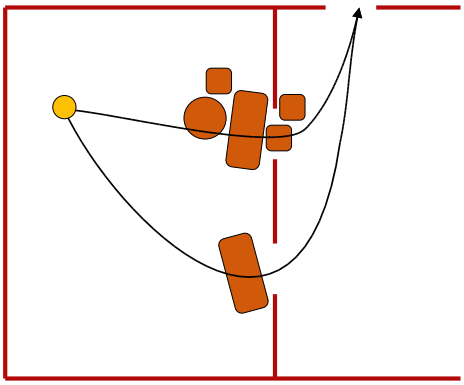


Figure 6.3: The robot needs to move through the top passage. While the topmost path is the shortest, a natural choice would be to take a small detour because it involves less manipulation of obstacles.

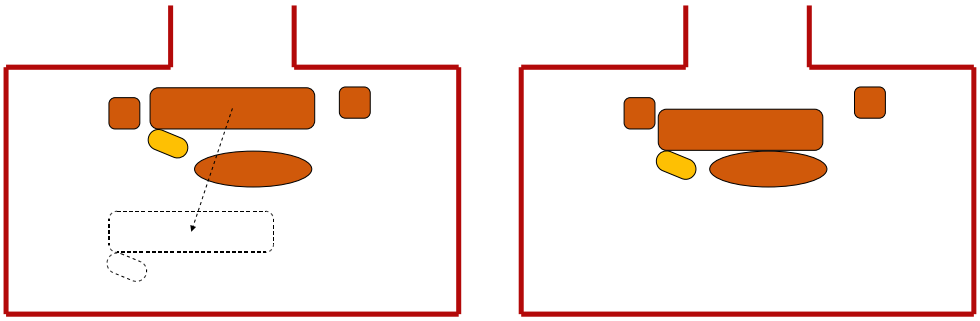
After reconnecting R to the graph, a shortest path query on G^p, using for example A* (Stenz, 1995) or D* Lite (Koenig and Likhachev, 2002) yields a path that prefers to avoid movables. By using the result of the query, the *first colliding movable* M_i on the path to the goal in G^p can be easily determined. M_i is called a *directly blocking movable*.

The above procedure is repeated a few times to make sure that the expansion process does not become too focused toward one path. For this, the edges in G^p that collide with M_i are invalidated and a new query is initialized. If successful, again the first colliding movable is determined. In all our experiments we have repeated this procedure three times which was a sufficient amount for our environments. Only the child nodes of η that grasp one of the directly blocking movables receive a probability larger than 0 (the sum of their probabilities is 1). Thus if we find two directly blocking movables using the above procedure, they both receive a probability of 0.5.

Indirectly Blocking Movables

Movables that block the path of R directly, can be determined by a shortest path query. However, a movable can also block the manipulation action of another movable, thus blocking the path of R indirectly (see Figure 6.4 for an example). Using the properties of the RRT algorithm, these can be determined easily. Recall that an RRT is extended by trying to create a path for M_i^R from the RRT to a randomly chosen configuration. If this random configuration is not reached, M_i^R collides with either a stationary obstacle or another movable M_j (Figure 6.4(b)). In both cases the RRT is extended but in the latter case M_j blocks the manipulation path of M_i^R and M_j is identified as an *indirectly blocking movable*.

During the expansion of a grasp node η in which M_i is grasped an RRT is created and the configurations of the RRT are added to T_A as child nodes of η. If a blocking movable M_j is encountered, the probability of the sibling of η that grasps M_j is increased using



(a) The start of the manipulation action. A random node for the RRT is generated.

(b) The oval table blocks the manipulation path. This configurations is added to the RRT and the probability of the node in which the oval table is grasped is increased.

Figure 6.4: Example of an indirectly blocking movable.

Algorithm 6.2. The more often M_j acts as an indirectly blocking movable, the more likely it becomes that M_j is selected for manipulation.

Decreasing Probabilities

If a node η in T_A has many descendants, the value $q(\eta)$ will increase because of the propagation algorithm. However, if its descendants cease to make progress, its probability should be lowered. For this, we need a method to measure progress. A simple measure of progress is the graph distance of R to the destination in G. The current position of R is connected to G and using a shortest path query to the destination provides an estimate of the current distance to the goal. The distance estimate is saved in the node such that the progress between a node and its parent can easily be determined. If no or little progress is made, the probability of the node is decreased by a small fraction using Algorithm 6.3.

6.2.3 Lazy Expansion

Expanding a node (Algorithm 6.1) can be a costly operation. Luckily many operations can be postponed until a node is actually selected for expansion. Manipulation nodes have children that grasp a movable. Checking the feasibility of the grasp (i.e. can the grasp configuration be reached by R from its current configuration) is not necessary until the child node is selected for expansion. So if a manipulation node is chosen for expansion, it adds child nodes that represent grasps of movables without verifying that such grasps actually exist.

If a child node is added to a grasp node, the robot graphs (G and G^p) need to be updated. This update involves collision checks and is thus relatively expensive. Only when the child node is selected for expansion this update is necessary. Therefore the computation of the update of G and G^p can be postponed until the node is actually chosen for expansion.

If lazy expansion is used then it is uncertain if η is feasible at the moment it is added to T_A. Only if η is selected by the random process for expansion, its feasibility is checked. If it is not feasible, η is declared a *dead end* and $q(\eta)$ is set to 0. The probabilities of $S(\eta)$ are updated such that Equation 6.1 will hold. Subsequently another leaf node is selected for expansion.

6.3 Smoothing

As described in Chapter 2 smoothing is a well-known postprocessing step to improve the path quality of a sampling-based path planning algorithm. Since the navigation of the robot is based on such an algorithm, when moving from one grasp to another we can apply all types of smoothing developed for sampling-based planners. Since the paths for the robot in the grasp nodes consist of straight line path segments, simply choosing two random configurations on the path and trying to create a straight line shortcut usually suffices to remove redundant motions. Alternative methods to create higher quality paths are developed by Geraerts and Overmars (2006).

For smoothing manipulation paths in the manipulation nodes, the same techniques can be applied since manipulation paths also consist of straight line segments created by the RRT. The physical model has to be used to check if shortcuts are applicable for the combined motion of the robot and the movable.

Another issue is removing redundant nodes from the final path. Recall that a manipulation plan is a path from the root node of T_A to one of the leaves. Since the node expansion process is guided by random choices (either in picking the nodes or generating the RRT) the final manipulation path may consist of nodes (and thus of actions) that are not strictly necessary to guide the robot to its goal. An example is shown in Figure 6.5.

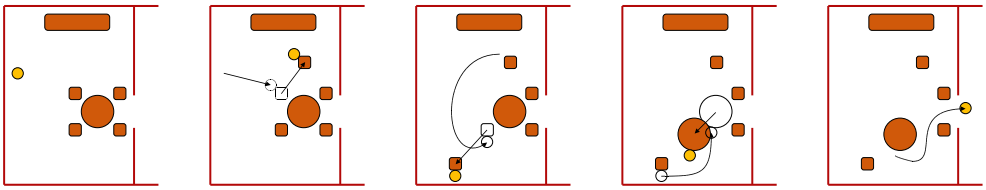
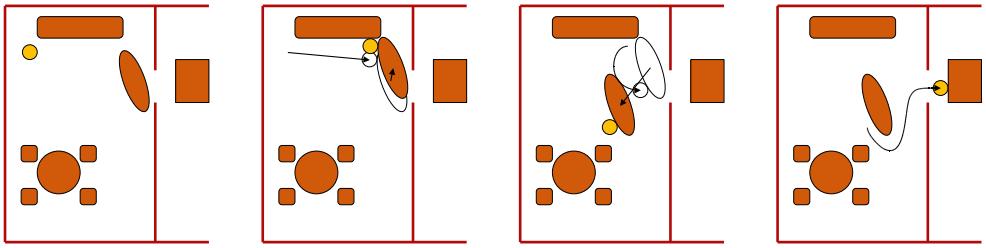


Figure 6.5: Example of a redundant node in the action tree. The robot first moves two chairs (b) and (c). Next the table is moved (d), which is sufficient to leave the room (e). Moving the first chair is not necessary and thus, the nodes representing (b) are redundant.

Checking if a node is redundant is an expensive operation. It may be that a node may seem redundant early in the manipulation plan, but turns out to be helpful later in the plan. Removing such a node makes the manipulation plan invalid. Therefore, checking if a node is redundant involves re-checking the rest of the manipulation plan. Since this procedure has to be repeated for every node in the final manipulation plan makes this an

computationally expensive operation. There is however a subset of redundant nodes that are much easier to detect. These involve nodes in which the same movable is manipulated twice in a row. An example of such a redundant node is shown in Figure 6.6.



(a) The initial situation.

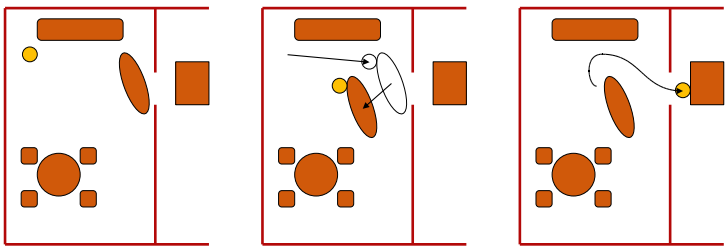
(b) The movable is grasped and moved.

(c) The same movable is re-grasped and moved again.

(d) The robot is able to leave the room.

Figure 6.6: Another example of a redundant node. Here the same movable is manipulated twice in a row. In this example the robot could move the movable directly to the configuration shown in (c), therefore bypassing the manipulation and grasp nodes shown in (b).

The procedure of checking for this type of redundant nodes is as follows. Suppose we have the following sequence of nodes: $\{\eta_1^G, \eta_2^M, \eta_3^G, \eta_4^M, \eta_5^G\}$ (the superscripts show whether the node is a GRASP() or MANIP() node). If η_1^G and η_3^G grasp the same movable M_i, we have a candidate for redundancy. We create a configuration $c = (w_r, w_i)$ for R and M_i with w_r equal to the grasp of η_1^G and w_i equal to the configuration of M_i in η_4^M. Next, we check if a path exist for M_i^R from its configuration in η_1^G to c using the local planner. If this is the case, we know there is a shortcut from η_1^G to η_4^M. Since configuration c grasps M_i at a different position than η_4^M, we attempt to reconnect w_r to G (as in Section 6.1.4). Finally we check if a path for R exists in G that brings R to the grasp position of η_5^G. Now we can safely remove η_2^M and η_3^G from the manipulation plan. The result of the procedure on the situation of Figure 6.6 is shown as Figure 6.7. The procedure is repeated for all consecutive pairs of nodes in the manipulation plan.



(a) **(b)** **(c)**

Figure 6.7: The same example as in Figure 6.6 but with the redundancy removed.

6.4 Experiments

We have implemented our algorithm in C^{++} and conducted different experiments with a number of environments on a Pentium 2.40GHz with 1GB of memory. In all environments the heuristics as described in Section 6.2 were used.

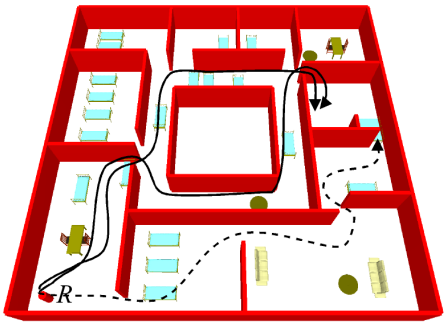
We used a number of environments to conduct the different experiments. The first environment, **LP3** is shown in Figure 5.1(c). It is a simple environment in terms of number of obstacles that shows the capabilities of the algorithm to solve an LP3 problem. For the second environment, **3xLP3** we took three versions of the environment of Figure 5.1(c) and connected them together, effectively creating a problem involving a series of three LP_3 problems. For environment three, **3xLP2** we created a series of three LP2 problems by repeating the environment shown in Figure 5.1(b). The fourth environment, **Hospital** is shown in Figure 6.8. The pitfall in this environment is that the shortest path (shown as the dotted arrow) is blocked by an immovable obstacle close to the goal. Thus, to reach a solution, the algorithm probably needs backtracking (depending on the random choices). The fifth environment, **Office** (Figure 6.9) has two indirectly blocking movables. Finally we conducted experiments with the environment shown in Figure 6.10, **Random tables**. The solution to this last environment consists of solving a long series of LP1 problems in a very constrained environment. We used it to show that our algorithm also scales to large numbers of movables. This environment contains 400 movable obstacles. All experiments were repeated 100 times and the average values are reported.

In the first experiment we compared the two node selection methods. Both methods select a child node based on its probability. The first method is the *random selection* method. The probability of a node directly corresponds to its probability of being selected. The second method is the *deterministic selection* method. It simply selects the node having the highest probability. We used environments LP3 and Office. To increase the probability of a node, we used a value for f of 0.05, to decrease the probability of a node we used an f value of 0.1. We report the running time, the tree size and the depth of the solution node. The results are shown in Table 6.1. As can be clearly seen from the results, selecting the child node having the highest probability results in lower average running times and smaller tree sizes.

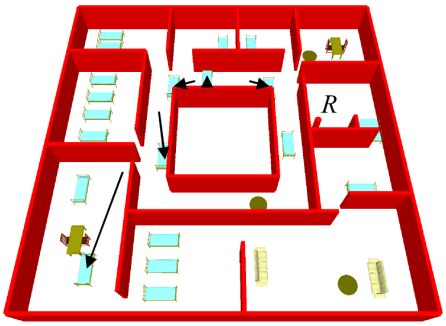
	Running time(s)		Tree size		Solution Depth	
	Random	Det.	Random	Det.	Random	Det.
LP3	3.9	0.6	365	74	9.6	15.2
Office	22.0	5.2	104	400	23.3	39.4

Table 6.1: Results of the two node selection methods, random selection and deterministic selection (det.).

In the second experiment we established the optimal f value for increasing node prob-

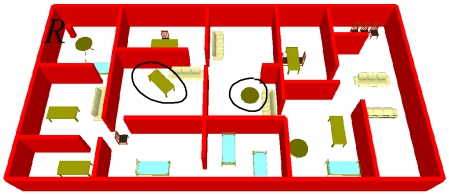


(a) The two solid arrows show feasible paths, the dotted path results in a dead end (behind the first movable is another that blocks the manipulation of the first).

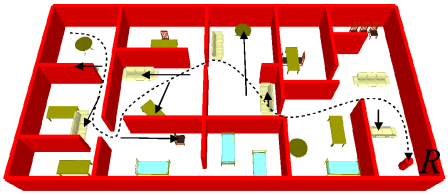


(b) The situation after the robot has reached its destination.

Figure 6.8: The hospital environment. Robot R is represented by a cylinder.

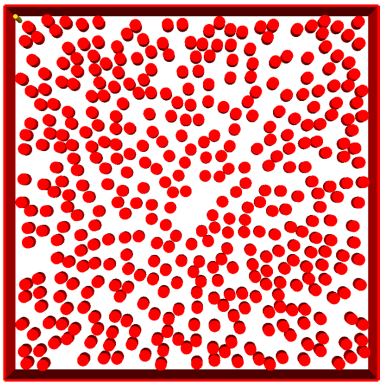


(a) The two encircled movables are indirectly blocking movables.

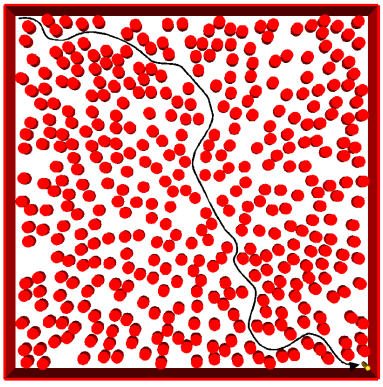


(b) The situation after the robot has reached its destination.

Figure 6.9: The office environment. The robot needs to move from the top left to the right bottom.



(a) The robot (top left) needs to maneuver to the bottom right in a very constrained environment.



(b) An example of a solution path. While moving to its goal many movables are manipulated by the robot.

Figure 6.10: The random tables environment.

abilities. First we fixed the f value for the decrease algorithm to 0.1 and let the f value for the increase algorithm increase from 0 to 0.5. We used the 3xLP3 environment and the Office environment. The results are shown in Figure 6.11.

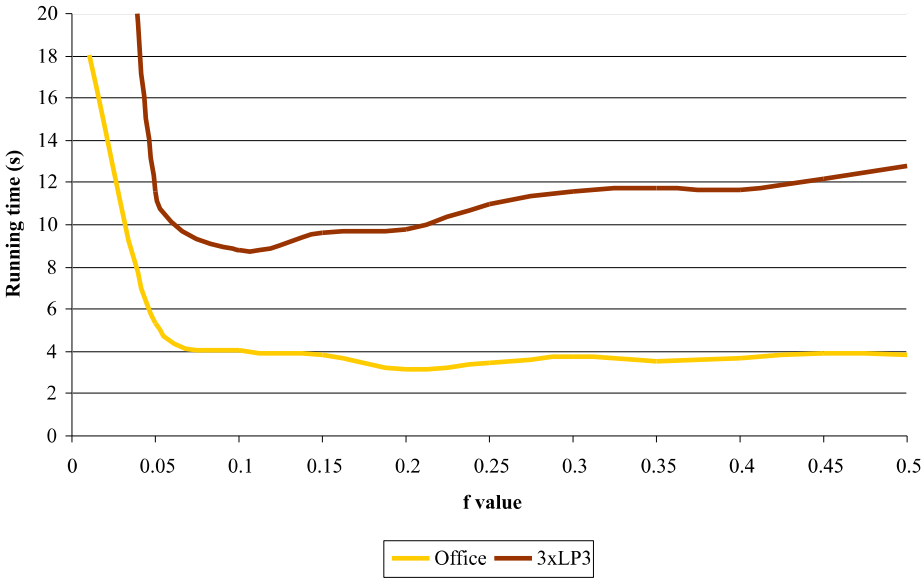


Figure 6.11: Varying the f value when increasing probabilities.

As can be seen from the results, a very small f value results in long running times. This can be explained by realizing that if the probabilities of nodes are not increased, the algorithm will expand nodes in a breadth first manner and thus running time suffers. Too high f values result in focusing too much on one branch of the tree (depth-first behavior). In case of the Office environment, this does not make a lot of difference since the solution path is a series of LP1 problems and only two LP2 problems. Since there are not many indirectly blocking movables, the influence of the value of f is small. In case of the 3xLP3 environment, the increase focuses the search in the tree toward the right indirectly blocking movables, therefore the influence of a proper f value is larger. In both experiments an f value of 0.2 is appropriate.

The third experiment established an optimal f value when decreasing node probabilities. We conducted experiments with environments 3xLP2 and Hospital. We fixed the f value to 0.2 for increasing probabilities and increased the f value for the decrease probability algorithm from 0 to 0.3. The results are shown in Figure 6.12.

Probabilities are decreased when the robot ceases to make progress. This is particularly useful if the robot has chosen the wrong route to the goal and runs into a dead end. In environment 3xLP2 there is only one route leading to the goal, so not much

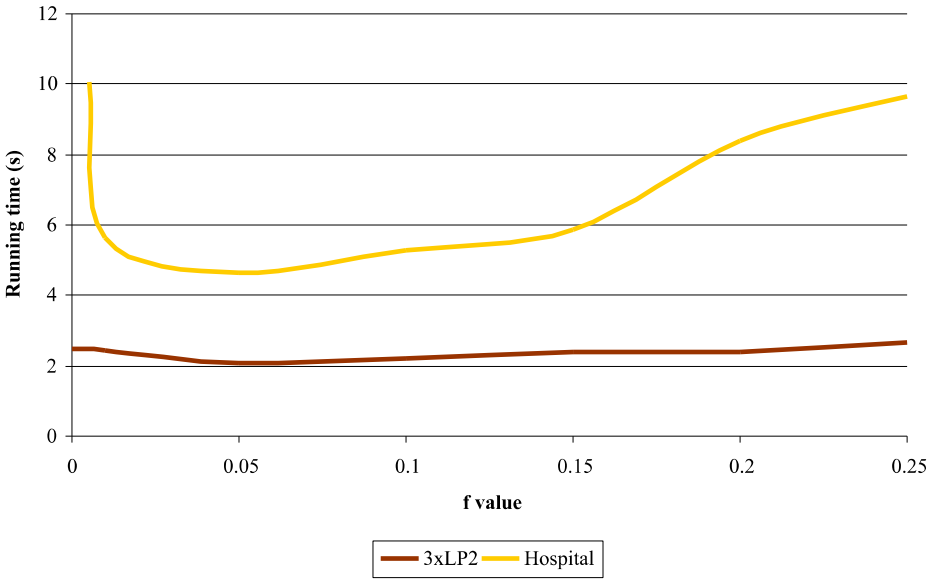


Figure 6.12: Varying the f value when decreasing probabilities.

influence of decreasing probabilities is observed. In the Hospital environment, the shortest path (when ignoring the movable obstacles) runs into a dead end. Decreasing probabilities is very useful in such a situation. Varying the f value shows that there is a large domain of f values that leads to good results. As long as the f value is not too small.

	Running time(s)	Tree size	Solution depth
LP3	0.8	98	18.4
3xLP3	6.9	388	81.4
Hospital	3.8	119	33.8
Office	4.7	101	34.9
Random tables	94.8	820	154.8

Table 6.2: Results of the experiments.

For the rest of the experiments we chose an f value of 0.2 for the increase probability algorithm and an f value of 0.1 for the decrease probability algorithm. As node selection method we chose to use the deterministic selection method. Next, we conducted experiments with almost all environments to shown the performance of the algorithm in

a variety of environment types. The results are shown in Table 6.2. As can be seen from these results with our algorithm we are able to create solutions efficiently for a range of different types of environments.

	Path length		Number of grasps	
	Basic	Advanced	Basic	Advanced
3xLP3	86.6	87.4	46.5	39.6
Office	31.9	32.0	28.0	21.6

Table 6.3: Comparing basic smoothing and advanced smoothing.

Finally we conducted experiments to show the effectiveness of our smoothing method. We calculated path length which is the total Euclidean distance the robot has to move from its start to its goal. In addition we counted the number of grasps which is also a measure for the number of manipulations (recall that a manipulation plan is an alternating sequence of grasps and manipulations). The first results were obtained without removing redundant nodes (*basic*) method. The second results were obtained with removing redundant nodes (*advanced*) method. In both experiments, we used basic smoothing (shortcutting robot paths and manipulation paths). The environments we used were 3xLP3 and Office. The results are shown in Table 6.3.

As can be seen from the results, the difference in path length is not large. This can be explained by the fact that redundant nodes in the manipulation plan often correspond to small motions in confined areas. Removing such nodes does not decrease the path length by a large amount. However the number of grasps does decrease significantly and therefore the quality of the manipulation plans increases by removing redundant nodes.

6.5 Concluding Remarks

In this chapter we have presented a planner capable of solving path planning queries among movable obstacles. Our approach is based on the framework presented in Chapter 5. This framework creates an action tree in which the nodes represent worldstates describing the configurations of the robot and the movable obstacles. The edges represent the transitions between the worldstates. During those transitions, the robot either manipulates one of the movables or (re-)grasps a movable. A path in the action tree represents a manipulation plan. Constructing the complete action tree (in a breadth-first manner) can be infeasible because of the huge number of nodes.

In environments encountered in practical settings often only a small subset of the manipulation plans in the action tree are useful. Inspired by this observation we have presented heuristics that focus the node expansion process toward these manipulation plans. This process is solely guided by adapting the probabilities of selecting nodes. By continuously adapting these probabilities, using information gathered during the process, an

efficient algorithm is obtained that is capable of solving realistic problems in reasonable running times.

Even though the heuristics are often successful in guiding the probabilistic process in realistic problems, in certain situations the process may become slow. This happens if many movables are close together and need to be manipulated in a certain order. If n movables are involved, there are $n!$ sequences of manipulating the movables. Since none of these result in overall progress, many nodes get comparable probabilities resulting in breadth-first behavior. Experiments have also shown that sometimes computation time is spent on parts of the action tree that are quite similar (e.g. they only slightly differ in the configuration of a movable). A solution could be to use information gathered in the process not only locally in the action tree but rather in all nodes that resemble the current node in some aspects. For example, if a movable is impeding the robot in a node, then its probability of manipulation should not only be increased in that node but in all nodes that contain a similar subproblem. If the subproblem is solved in one node, then a shortcut in the action tree could be added to all other nodes. This type of extensions are interesting topics for future research.

Part III

Pushing using Compliance

CHAPTER 7

Introduction and Preliminaries

In this part we consider the path planning problem in which an object, incapable of moving by itself, is pushed through an environment by the robot. In practical problem settings, planning a path and navigating a single robot is already a difficult task because of the many sources of errors involved. Pushing a passive object makes this situation even more difficult. In this chapter we study the combination of pushing and compliant motions. A compliant motion is a motion in which the robot touches the boundary of the environment while navigating. Often this type of motions is used to compensate for uncertainty. If the robot keeps contact with the environment, it is much easier to control and keep track of its configuration. We use compliant motions to create manipulation plans. Such a manipulation plan is a navigation path for a robot that, if followed, pushes an object to a certain goal configuration. In contrast to most previous contributions we use compliance for our passive object and not for the robot itself. The object is allowed to slide along the boundaries of the environment, thus compensating for uncertainty. Another important advantage of compliant motions is that they can help to solve the narrow passage problem that is encountered when operating in a constrained environment. In this chapter we state the problem, describe previous work and give preliminaries that are used in the coming chapters.

7.1 Introduction

In previous chapters we have addressed the problem in which the environment changes and problems in which the robot is allowed to change the environment to make it possible to navigate to its goal configuration. In this chapter we will look at another variant of changeable environments namely problems in which the objective is not defined in terms

of a configuration for the robot, but rather defined in terms of a configuration for a passive object. By manipulating the object by a robot, it is moved to its goal. An example of this type of manipulation is a robot arm in a manufacturing plant that needs to insert a part (the passive object) into an engine. Because of the complex structure of the engine, the goal configuration for the part may be difficult to reach, requiring a very complex and precise motion of the robot arm. Also in virtual environments used for e.g. computer assisted training in which the user must perform a complicated manipulation task, difficulties often occur because of the fine motions required.

A number of causes for the problems that arise while executing such a complex task can be distinguished. Firstly, while the motion required by the passive object alone may be feasible, the manipulation as a whole may be infeasible because the manipulator itself also needs room to maneuver. Secondly, manipulations may be highly constrained, therefore even small errors in sensor data can lead to failure in the planning process or execution of the path. Finally, finding a manipulation plan in highly constrained environments is often computationally expensive because of the well-known narrow passage problem. We will elaborate on these aspects in more detail later in this section.

To address the above issues, we propose a novel approach that combines manipulation planning with compliant motions. Here we define compliant motions as motions in which the manipulated object is in contact with the obstacles of the environment. Note that here we do not see obstacles as something to avoid, but rather something to exploit by letting the object slide along an edge of an obstacle while it is manipulated.

Objects can be *manipulated* in numerous ways, and each type of manipulation implies different constraints on the combined motion of manipulator and object. All types of manipulation have in common that the manipulator needs to apply force to the object. Research on how to apply this force has led to a broad range of different forms of manipulation that include grasping (Reuleaux, 1876), squeezing (Goldberg, 1993), rolling (Arai and Khatib, 1994) and even throwing (Mason and Lynch, 1993) an object. However, pushing (Mason, 1986) is one of the most widely studied types of manipulation. An extensive overview of all forms of manipulation can be found in the book of Mason (2001).

To take advantage of compliance, pushing is the most suitable of the above methods. So as a case study on the use of compliance we have looked at the pushing problem. The objective of pushing is to maneuver an *object*, incapable of moving by itself, from an initial configuration to a goal configuration by pushing with an autonomous robot, the *pusher*. Because of the many parameters involved, e.g. mass distribution of the object, different types of friction and limitations of the pusher, planning and controlling a pushing motion is often a difficult problem.

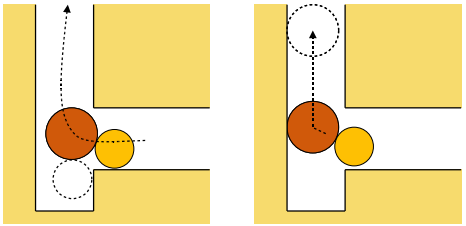
Compliance has often been used to compensate for uncertainty, for example to solve the peg-in-hole problem. Lozano-Peréz et al. (1983) introduced the *preimage backchaining* approach to solve robot path planning problems. The idea is to compute the points from which the goal is reachable with a single manipulation (the preimage). Next, the preimage is iteratively treated as a new goal until the initial robot configuration is within a preimage. Compliance is used to confine the motions to be tangent to the constraining surfaces. Briggs (1992) improved an algorithm introduced by Donald

(1988) to find a trajectory from a start region to a goal region amidst planar polygonal obstacles where control is subject to uncertainty. Compliance was used to find a trajectory.

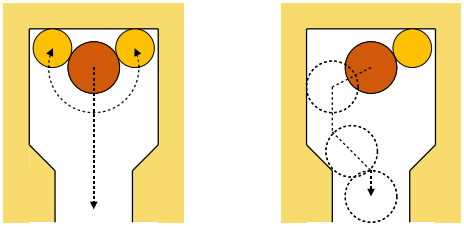
Our algorithms combine manipulation by pushing with compliance. Given a query, consisting of a start and a goal configuration of an object, a *manipulation plan* is created for the pusher such that if the pusher satisfies this plan, it pushes the object from its start to its goal configuration. Such a manipulation plan is allowed to consist of both compliant and non-compliant sections. In the compliant sections the object slides along the boundary of the obstacles while being pushed by the pusher. Such motions are only possible if the force of the pusher in the direction parallel to the obstacles is large enough to overcome the friction between the object and the obstacles ((Lynch and Mason, 1996)). The resulting motion of the object is thus parallel to the boundary of the obstacles in the environment.

Allowing compliant sections in the manipulation plan has several advantages. Firstly, if no compliance is used, there is often only one position on the boundary of the object from which it can be pushed to move in a specific direction. If this position is not reachable by the pusher (because it is obstructed by obstacles as it operates in a highly constrained environment) no manipulation plan may be found. If the object moves compliantly to the obstacles of the environment there is a range of positions on the boundary of the object that all cause it to move in the same direction parallel to the edges of the obstacles. Thus it is more likely that the pusher can reach such a position. This also leads to the additional advantage that the complexity of the manipulation plan is reduced. Secondly, traditional path planning problems often have difficulties passing narrow passages. These are caused by highly constrained areas of the environment. In these areas, compliance often helps to pass narrow passages. Thirdly, since in compliant motions most degrees of freedom are fixed, compliance helps in compensating for uncertainty. Uncertainty can occur because of errors in sensors. A well-known example is a small error in an odometer that can cause a large deviation in the localization of the pusher and object. Another source of uncertainty is the actual amount of friction, because the exact coefficients of friction and center of mass are often not known. This makes it difficult to predict the exact outcome of a manipulation and thus to fully incorporate the role of friction in the planning process.

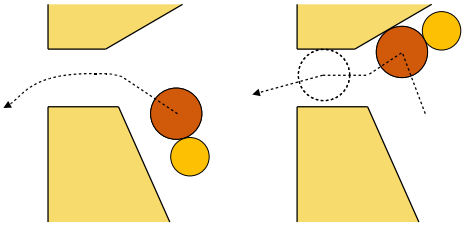
A few examples in which compliance is advantageous are shown in Figure 7.1, the small disk is the pusher, the large disk the object. The first example, shown in Figure 7.1(a), shows a situation where no manipulation plan exists without compliance because of the lack of space for the pusher to maneuver. With compliance a simple plan can be created. In the situation of Figure 7.1(b), a complicated manipulation plan can be created in which the pusher alternates between its current position and the dotted position to approximate the almost straight line path necessary to let the object escape the caving. With compliance, a much simpler solution exists. Figure 7.1(c) shows a narrow passage. Many sampling-based manipulation algorithms have problems finding a path through such a passage. Using compliance often helps in finding such paths. Finally in Figure 7.1(d) an example is shown where without compliance the pusher needs to break contact with the object and make a long detour.



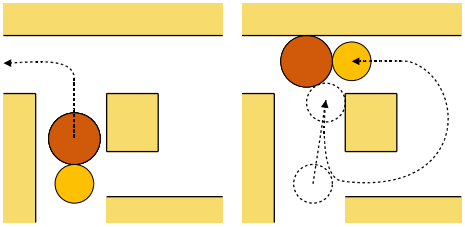
(a) Without compliance the pusher is not able to reach the necessary push position. With compliance, the object is pushed against the edge of the environment and compliance is used to push the object to its intended target.



(b) To move the object down without compliance the pusher needs a long sequence of alternating between two push positions while slowly pushing the object down. With compliance, a solution exists consisting of only a few pushes.



(c) A narrow passage. Many algorithms have trouble finding a path through such a passage. With compliance a path is easily found.



(d) The object needs to enter the corridor on the left. Without compliance, the pusher has to let the object loose and make a long detour. With compliance the object is simply pushed against the back wall and compliance is used to continue.

Figure 7.1: Four examples in which compliance helps finding a manipulation plan for a disk pushing another disk. The left figures show the intended path toward the goal for the object. The right figures show the solutions if compliance is allowed.

As a first step toward exploring the potentially powerful combination of pushing and compliance and to gain insight into this combination, we look at the simplified problem of two disks. One disk is an autonomous pusher, capable of pushing the other (passive) disk (the object). The goal is to push the object from its initial placement to a given goal placement in an environment of non-intersecting line segments (which includes the case of polygonal obstacles). While being pushed by the pusher, the object is allowed to slide along the boundary of the environment (compliant motion). Friction between the object and the environment and between the pusher and the object are taken into account. Even in this reduced problem setting the encountered problems are challenging and their solutions provide useful insights for future research.

If the radius of the pusher is larger than the radius of the object, the possibilities for the pusher to push the object away from a compliant configuration are limited. In that case, environments can even be constructed such that pushing the object away from a compliant position is impossible. Therefore, to explore the combination of compliance and pushing, we look at the more challenging case where the radius of the object is larger

than the radius of the pusher. Nevertheless, with a few exceptions, the results of this part can also be applied to the case in which the radius of the pusher is larger than the radius of the object.

7.2 Preliminaries

Given disks O (the object) and P (the pusher) with radii of r_o and $r_p < r_o$ respectively, in an environment consisting of n non-intersecting line segments $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, we create a manipulation plan for P such that if P satisfies this plan, it pushes object O from a given start to a goal configuration. We assume that P always keeps contact with O. As a consequence, the configuration space is three dimensional: two parameters specifying the position $q = (q_x, q_y)$ of the center of O and one that specifies the relative position α of P, the *push position*. Push position α is the angle the directed line from the center of O to the center of P makes with the positive x-axis. The configuration space is the space of all tuples (q, α). A configuration $c = (q, \alpha)$ is said to be *free* if both O and P do not intersect with Γ (touching is allowed). Configuration c is *compliant* with obstacle γ_i if the shortest distance between q and γ_i is r_o. It is assumed that the friction between O and the supporting plane is large enough such that there is no motion of O after pushing ceases (in other words we assume the pushing to be quasi-static). The center of friction of O is the center of the disk.

7.2.1 Friction

We will now describe the role friction plays in the outcome of a push action in which O is compliant. If P pushes O while O is in a compliant configuration, it can either slide or roll along the boundary of an obstacle γ_i. This behavior is dependent on two coefficients of friction: μ_1 is the coefficient of friction between O and γ_i and μ_2 is the friction between O and P. For P to stay at the same relative position with respect to O, it needs to push O in the direction of the center of O. The angle the line through the centers of the disks makes with γ_i is called β. Let F be the force that P uses to push O. Let the force orthogonal to F be called F_f. Furthermore the vertical and horizontal components of F are called F' and F'' respectively. Finally, the force due to the friction between O and γ_i is called F'_f. The situation is sketched in Figure 7.2. It is easy to see that if the force $F_f \geq F'_f$, object O will slide along γ_i, else it will roll. The moment arms of both forces are equal and do not need to be taken into account. Since $F_f = \mu_2 F$ and $F'_f = \mu_1 F \sin(\beta)$, we have that

$$\frac{\mu_2}{\mu_1} \geq \sin(\beta). \tag{7.1}$$

If Equation 7.1 holds, O will slide along the boundary of the obstacle. In other cases it will roll. However, not every value of β results in a motion of O and P. If $F'' \geq F'_f$, O and P will move, else they will stand still. Since $F'' = \cos(\beta) F$, we can derive the following equation:

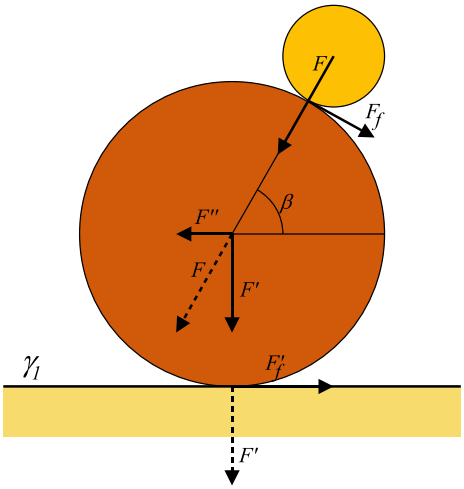


Figure 7.2: The role of friction on the behavior of O and P. F is the force applied by P on O. The friction between O and γ_i is denoted by μ_1, the friction between O and P by μ_2.

$$\tan(\beta) < \frac{1}{\mu_1}. \tag{7.2}$$

If Equation 7.2 holds, O and P will move. In all other cases they will stand still.

7.2.2 Definitions

If the direction of motion of O needs to be changed, the relative position of P needs to be changed while keeping contact with O. Such a transit is called a *contact transit*.

Definition 7.2.1 (contact transit)**.** *A* contact transit *is a motion in which P rolls/slides along the stationary O, thus a contact transit transforms a configuration $c = (q, \alpha)$ into a configuration $c' = (q, \alpha')$.*

Suppose $c = (q, \alpha)$ is non-compliant. With one exception explained in Chapter 9 we will only allow *stable* pushes (Lynch, 1992, Lynch and Mason, 1996), i.e. P and O remain fixed to each other. Thus, if O is pushed by P, the path O will follow is on the line through the centers of O and P.

The *racetrack* $R^{r_o}_{\gamma_i}$ of a single obstacle $\gamma_i \in \Gamma$ is defined as the boundary of the Minkowski sum of γ_i and a disk of radius r_o (Figure 7.3), so $R^{r_o}_{\gamma_i} = \partial(\gamma_i \oplus D(r_o))$, where $D(\rho)$ is a disk of radius ρ and $\oplus$ denotes the Minkowski sum. (The Minkowski sum $A \oplus B$ of two objects A and B is defined as $A \oplus B = \{(a+b) | a \in A \wedge b \in B\}$.) If $q \in R^{r_o}_{\gamma_i}$, then q is called a *compliant position* on obstacle γ_i.

There is only one push position to push O in a certain direction if O is non-compliant. If O is compliant, this does not hold. Its behavior when pushed by P will then depend on

the obstacle to which it is compliant. Therefore we make a distinction between *compliant space* and non-compliant space. Compliant space UB^{r_o}, consists of the union boundary of the Minkowski sum of Γ and a disk of radius r_o. Stated formally:

$$UB^{r_o} = \partial\Big(\bigcup_{\gamma\in\Gamma}(\gamma \oplus D(r_o))\Big).$$

The individual racetracks are pseudodisks. Therefore the racetracks for all $\gamma \in \Gamma$ together form a collection of pseudodisks. Such a collection has a complexity of $O(n)$, see the work by De Berg et al. (2000). Thus also the complexity of UB^{r_o} is $O(n)$. If O is compliant, its center follows a curve in compliant space. An example of compliant space is shown in Figure 7.4.

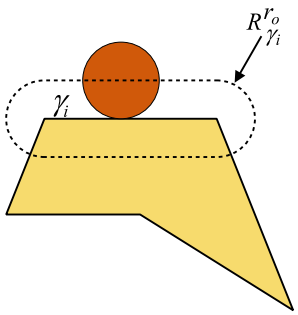


Figure 7.3: Example of the racetrack $R^{r_o}_{\gamma_i}$ of obstacle γ_i.

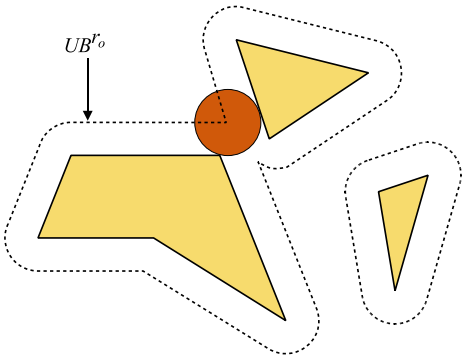


Figure 7.4: Example of compliant space, consisting of UB^{r_o} shown as the dotted lines.

As stated before, if q is compliant, there is a range of push positions that all cause O to follow the same path in the same direction. Such a range is called the *push range*. The size of this range is dependent on the friction.

Definition 7.2.2 (push range)**.** *At compliant position $q \in UB^{r_o}$, the* push range *describes the continuous range of push positions that cause O to follow the same compliant path (regardless of O and P being collision-free). Since there are two directions of compliant motions around γ_i, there are two push ranges:* PR^+ *for the clockwise motion and* PR^- *for the counterclockwise motion around γ_i.*

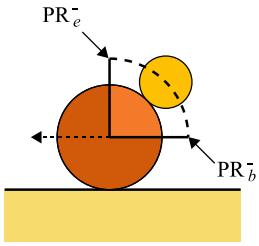
Since the size of a push range is always equal to or smaller than $\frac{1}{2}\pi$, we can define the two extreme values of PR^+ as starting at PR^+_b and ending at PR^+_e using the shortest counterclockwise rotation. Similarly we can identify PR^-_b and PR^-_e as the extreme values for PR^-. The push range is defined such that the push position closest to the obstacle is 0 and increases while rotating to the other extreme value of the push range. If friction is not taken into account, the size of $\text{PR}^+ = [\frac{1}{2}\pi, 0]$ and $\text{PR}^- = [0, \frac{1}{2}\pi]$. If friction is involved then (using Equation 7.2) the size of the push range is dependent on the value of μ_1: $\text{PR}^+ = [\arctan(\frac{1}{\mu_1}), 0]$ and $\text{PR}^- = [0, \arctan(\frac{1}{\mu_1})]$.

Three examples without friction are shown in Figure 7.5(a)…(c), an example with friction is shown in Figure 7.5(d). Using a contact transit, it is not guaranteed that P can reach either PR^+ or PR^- because one or more obstacles can impede the transit, Figure 7.5(e). While pushing O, P can be forced outside the push range such that pushing cannot continue (Figure 7.5(f)). For the rest of this part we will, without loss of generality, assume that $\mu_1 = 0$, $\text{PR}^+ = [\frac{1}{2}\pi, 0]$ and $\text{PR}^- = [0, \frac{1}{2}\pi]$.

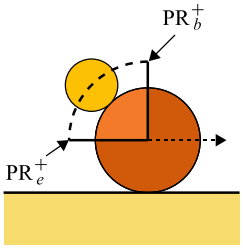
Suppose O and P are placed at configuration c. Let l be the line segment between the centers of O and P. We define $\Omega = (l \oplus D(r_p)) \backslash (O \cup P)$ as the *wedge area*. The wedge area consists of two separate areas. If O is compliant, these are denoted by the *lower wedge*, closest to the obstacle O is compliant to and the *higher wedge*, furthest away from the obstacle. An example of the wedge area is shown in Figure 7.6.

7.3 Complexity

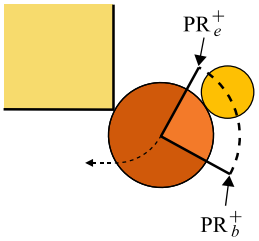
Intuitively one might try to solve the disk pushing problem with compliance using a complete and (hopefully) polynomial time algorithm. For the compliant parts of the path we have found such an algorithm, which will be described in the forthcoming chapters. However, a complete manipulation plan usually consists of compliant path sections connected by bridges through non-compliant space. The complexity of the bridges in terms of number of path segments and number of contact transits can get very high. This is illustrated in Figure 7.7. Here, P needs to push O out of a confined area (note that O does not have to follow the dotted line exactly, it rather shows the intended direction). Using compliance does not provide a solution to this problem since O will get stuck. The situation is even worse as *every* path using compliance will fail in bringing O outside the area. When pushing O away from its start configuration then, even when using a maximum angle, it is inevitable that it will end up in a compliant configuration and therefore fail (Figures 7.7(b)…(d)). The only way to push O out of the area is by means of a non-compliant path. This path however, needs to be very complicated. Because P needs room to revolve, it can only push O a very small distance, then do a contact transit and repeat. This process needs to be repeated



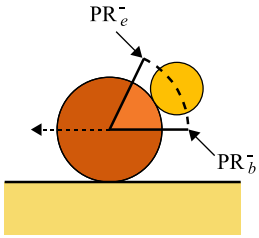
(a) $\text{PR}^- = [0, \frac{1}{2}\pi]$ if no friction is assumed between the object and the obstacle.



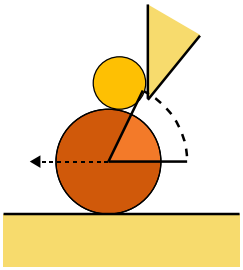
(b) $\text{PR}^+ = [\frac{1}{2}\pi, 0]$ if no friction is assumed between the object and the obstacle.



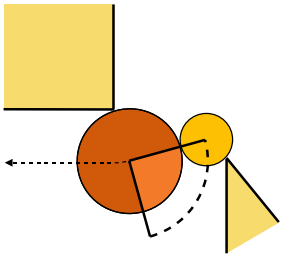
(c) $\text{PR}^+ = [\frac{1}{2}\pi, 0]$ if O is pushed around a vertex of an obstacle (no friction).



(d) $\text{PR}^- = [0, \arctan(\frac{1}{\mu_1})]$ if there is friction between O and the obstacle.



(e) Pushing in the counterclockwise direction is not possible because P is not able to reach PR^-.



(f) Pushing cannot continue because P is forced outside PR^+ by an obstacle.

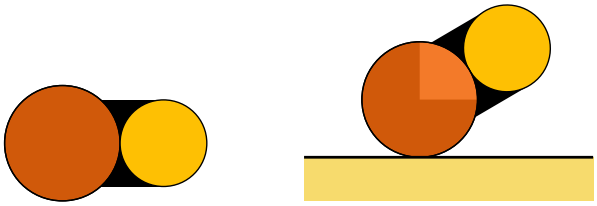
Figure 7.5: The push range. The intended path of the object is shown by the dotted arrow.

until P has enough room to push O out of the area. This is shown in Figures 7.7(e)+(f). If the dimensions of the obstacles are chosen such that P has very little room to revolve then the manipulation plan quickly gets more complicated. This is shown in Figure 7.7(g), if ϵ approaches 0, then the desired manipulation plan can be made arbitrarily complex.

The above shows that a polynomial time algorithm for the complete problem does not exist. We can however create an efficient solution for a subproblem. Therefore we present two algorithms. One complete algorithm for a subproblem (Chapter 8) and a solution for the whole problem that combines a complete algorithm with a probabilistically complete algorithm (Chapter 9). In this solution we combine complete exploration of the compliant parts of the path with an RRT based approach for the non-compliant parts. Both algorithms are sketched in the next section.

7.4 Our Contribution

In this part, we propose two algorithms that combine pushing with compliant motions. First, in Chapter 8 we investigate the problem of first deciding whether a pusher plan exist that causes O to follow a specified path. This path for O is part of the problem definition. If



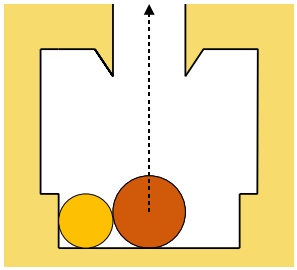
(a) The wedge area for a non-compliant configuration.

(b) If O is compliant, the wedge area can be subdivided by the lower wedge (closest to the obstacle) and the higher wedge.

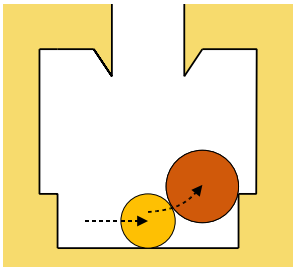
Figure 7.6: Examples of a wedge area.

a solution exists it is reported else the algorithm reports failure. The object path is allowed to consist of both compliant and non-compliant sections. To create the path for the object, any compliant path planning technique can be used as long as the resulting paths consist of straight lines and circular arcs (the circular arcs are used to rotate compliantly about a vertex). We present a complete algorithm that calculates a *push plan* for the pusher such that the object is pushed along the given path or reports failure if no push plan exists. Given an environment consisting of n line segments, our algorithm takes $O(n^2 \log n)$ preprocessing time and reports a push plan in $O(kn \log n)$, where k is the complexity of the object path. Under the realistic assumption of low obstacle density (Section 8.5), the query time is reduced to $O((k+n)\log n)$. Non-compliant sections are allowed, for example to act as bridges between two compliant sections. Even for this restricted case it turns out the problem is complicated and provides useful insights in the combination of pushing and compliance.

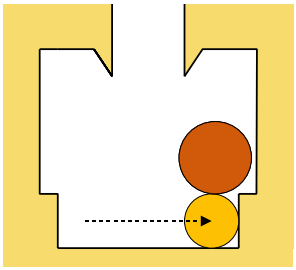
In the second algorithm (presented in Chapter 9) we build on the results of Chapter 8 to create a *manipulation plan* for both the pusher and the object, given a start and goal configuration for the object. It is based on the Rapidly-exploring Random Tree (RRT) algorithm and combines random exploration of the open space with a complete computation of reachable compliant configurations. For the non-compliant areas, a tree is created by generating random configurations. For every random configuration a path is tried from the tree to that random configuration. If the random configuration cannot be reached because of a collision, a configuration as close as possible to the obstacle that caused the collision is added to the tree. This configuration is then used as a starting point to explore the compliant configurations using a complete approach, eventually capturing the structure and connectivity of all compliant configurations. The results of compliant exploration are added as configurations to the tree.



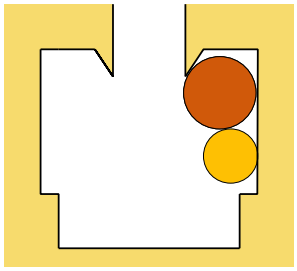
(a) The intended direction of motion for O.



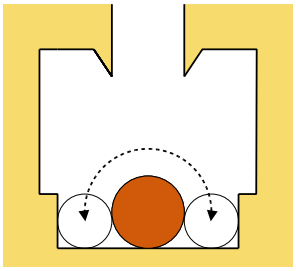
(b) Pushing O until it hits an obstacle.



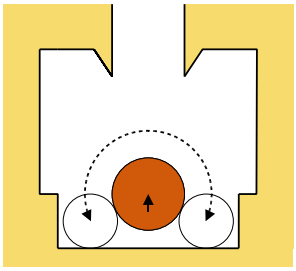
(c) Finally, P can get under O.



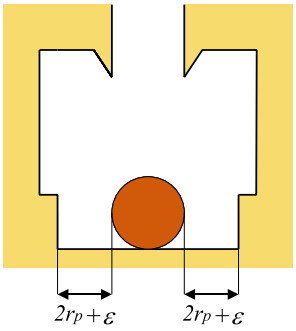
(d) But O inevitably gets stuck.



(e) The only solution is to continuously switch between two push positions.



(f) And slowly push O up.



(g) If ϵ approaches 0, the complexity of the necessary push path increases dramatically.

Figure 7.7: An example in which a very complicated manipulation plan is necessary.

CHAPTER 8

Creating a Push Plan

In this part we consider a disk shaped robot that pushes another disk in an environment among obstacles. Instead of only allowing the object to move through open spaces, we also allow the object to slide along the boundaries of obstacles in the environment using compliance, extending the possibilities for the robot to find a push plan. As has been shown in Chapter 7, solving the complete push planning problem in polynomial time is infeasible. We are, however, capable of solving a subproblem in polynomial time. In this chapter we present an efficient algorithm that is capable of creating a push plan, for a given path of the object. The path for the object is allowed to consist of both compliant and non-compliant sections. We present an algorithm that, given a path for the object consisting of k sections, preprocesses the environment consisting of n non-intersecting line segments in $O(n^2 \log n)$ and reports a push plan in $O(kn \log n)$ time or reports failure if no path exists. Under the weak assumption of low obstacle density, the query time is reduced to $O((k+n)\log n)$.

8.1 Introduction and Problem Statement

In this chapter, we present a complete algorithm that, given a path for O, computes a push plan for P such that O is pushed by P along the given path or reports failure if no such push plan exists. To create the path for O, any motion planning technique resulting in compliant paths can be used as long as the resulting paths consist of straight lines and circular arcs. The circular arcs can be used to rotate compliantly about a vertex. In addition non-compliant sections are allowed to act as bridges between two compliant sections.

Given disks O and P having radii r_o and $r_p < r_o$ in an environment of n disjoint line

segments $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ and a trajectory $\tau : [0,1] \rightarrow \mathbb{R}^2$ for O, we compute a push plan σ for P such that if P satisfies this plan it pushes O along τ or we report that no path for P exists. We allow the contact point between O and P to slide around the boundary of O. It is assumed that the friction between O and the supporting plane is large enough such that there is no motion of O after pushing ceases. In other words, pushing is assumed to be quasi-static.

8.2 Definitions

The object path τ consists of k sections $I = \{i_1, i_2, \ldots, i_k\}$ that jointly partition the unit interval $[0,1]$. The index of the section containing placement $s \in [0,1]$ is indicated by $I(s) : [0,1] \rightarrow \{i_1, i_2, \ldots, i_k\}$. The start and endpoints of section $i \in I$ are called i^s and i^e respectively. Every section is open in its endpoint: $I(i^s) = i$ and $I(i_t^e) = I(i_{t+1}^s) = i_{t+1}$. The end and start points of two subsequent sections are connected such that the path of O is C^0 continuous i.e. $\tau(i_t^e) = \tau(i_{t+1}^s)$. Throughout this chapter we will refer to a path section simply as *section*. Every section $i \in I$ is either compliant or non-compliant. The compliant sections are parts of the compliant space. Thus, there are two types of compliant sections: a straight line compliant section that is compliant with an obstacle edge and a circular compliant section that is compliant with an endpoint of an obstacle edge. The non-compliant sections are straight line segments through non-compliant space. The function $d(a,b)$ denotes the shortest Euclidean distance between objects a and b. A non-compliant section $i \in I$ does not touch any obstacle, i.e. $\forall(\gamma \in \Gamma) : d(\gamma, i) > r_o$. An example showing a complete path for O is shown in Figure 8.1.

Within one section, the push range PR does not change. It only changes when moving from one section to the next. Since we assume that O and P are in contact for all $s \in [0,1]$, we represent a push plan σ for P as $\sigma : [0,1] \rightarrow [0, 2\pi)$. The push plan σ defines the position of P for a corresponding position on the trajectory τ. The functions $\text{PR}_b(\tau(s))$ and $\text{PR}_e(\tau(s))$ map a position of the path of O to the extreme values of the push range at s. For the intended object path τ our goal is to compute a corresponding push plan σ such that if P satisfies this plan, it pushes O along τ. The position for P at the start of the object path, i.e. $\sigma(0)$ is part of the problem definition.

In addition to Section 7.2.2 we need some definitions. The pusher P may not be able to reach (parts of) the push range PR because one ore more obstacles may impede PR. The subset of PR that is collision-free for P is called the *valid push range* VPR.

Definition 8.2.1 (valid push range)**.** *The* valid push range *at position s on τ (Figure 8.2(b)) is the part of* PR *that is collision-free. If $p(\tau(s), \alpha)$ gives the world coordinates for P at configuration $(\tau(s), \alpha)$, then*

$$\text{VPR}(\tau(s)) = \{\alpha \in \text{PR}(\tau(s)) | d(p(\tau(s), \alpha), \Gamma) > r_p\}.$$

$\text{VPR}(\tau(s))$ may consist of multiple intervals, split up by obstacles if $I(s)$ is a compliant section (Figure 8.2(b)). Since we do not allow P to lose contact with O, at most one of these intervals is reachable for P from its current position $\sigma(s)$. Therefore we define the *reachable valid push range.*

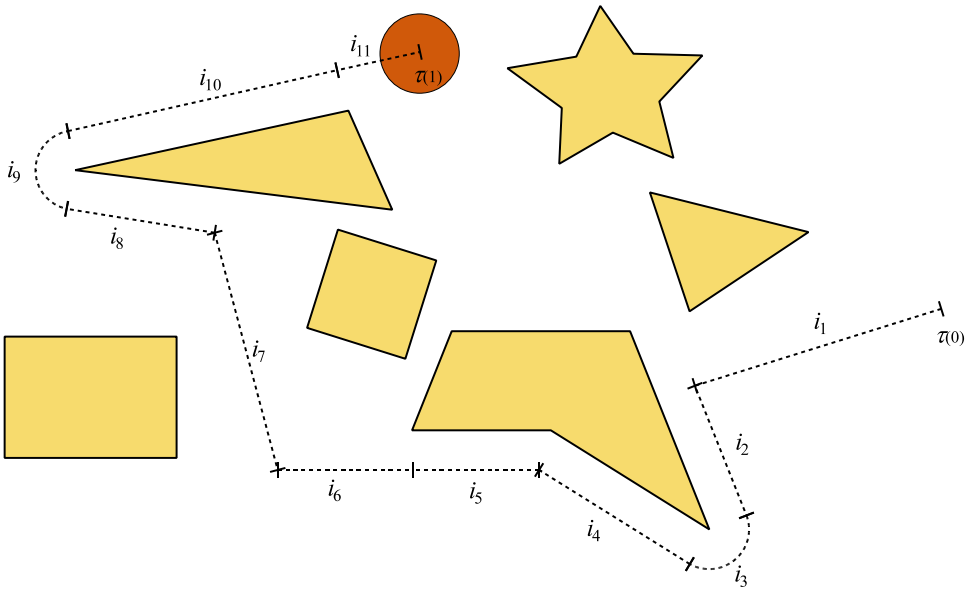


Figure 8.1: An example of an object path τ consisting of 11 sections. There are 4 non-compliant sections: i_1, i_6, i_7 and i_{11}. The rest of the sections are compliant. The start $\tau(0)$ and goal $\tau(1)$ of the path are also shown.

Definition 8.2.2 (reachable valid push range)**.** *The* reachable valid push range *(Figure 8.2(c)),* $\text{RVPR}(\tau(s), \sigma(s))$ *is the subset of* VPR *that is reachable for P from its current position* $\sigma(s)$ *using a contact transit.*

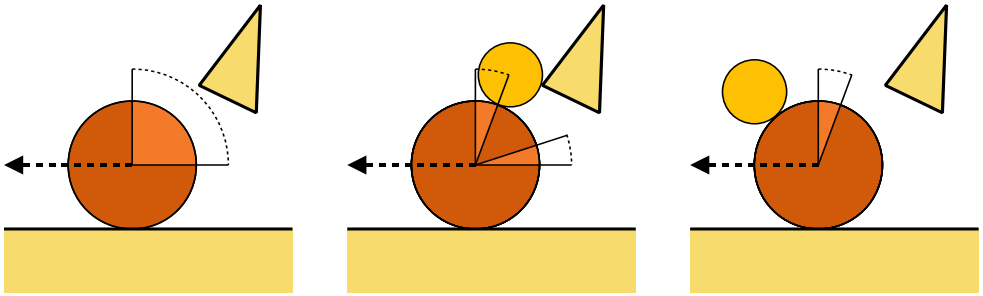
An interesting property of RVPR is that all push positions α inside this range result in the same motion for O.

The object path, given by the user, consists of k_c compliant sections and k_n non-compliant sections ($k_c + k_n = k$). The only restrictions on this path are that its non-compliant sections consist of straight lines and the compliant sections are within compliant space that consists of line segments and circular arcs of radius r_o.

In this chapter we will frequently use $UB^{r_p} = \partial\Big(\bigcup_{\gamma\in\Gamma}(\gamma \oplus D(r_p))\Big)$. This union boundary consists of all placements for the center of P where P is in contact with an obstacle. The complexity of UB^{r_p} is $O(n)$, as it is the boundary of n pseudodisks.

8.3 Global approach

Without loss of generality, we divide the path of O in four specific types of path sections. To reduce the search space we only look at paths of a specific nature, limiting the set of solutions. For each type of path section we will create a push plan separately.



(a) The push range PR if the object needs to move in the counterclockwise direction around the obstacle.

(b) The subset of the push range that is collision-free, the VPR.

(c) The subset of the VPR that is reachable for P from its current position, RVPR.

Figure 8.2: Illustrations of the ranges for a compliant section.

- sections in which O slides along an edge of an obstacle γ_i to which we will refer to as *straight line compliant sections*,
- sections in which O rotates about an endpoint of an obstacle γ_i, referred to as *circular compliant sections*,
- sections in which O is not in contact with Γ, referred to as *non-compliant sections* and
- finding a contact transit to reach $\text{VPR}(\tau(i^s))$ at the start of section $i \in I$.

At the start of a section, the position of P is determined by the position of P at the end of the previous section. As a result also the reachable valid push range at the start of a section is determined by the position of P at the end of the previous section. Only for the first section, the position for P is given by the query under consideration. We will explain the solutions to the four cases for the counterclockwise compliant motions. The clockwise compliant motions are analogous.

8.3.1 Straight Line Compliant Sections

For one straight line compliant path section $i \in I$ on the domain $[i^s, i^e]$, in which O moves compliant with obstacle γ, the corresponding push plan σ for P on this domain generally consists of multiple sections because P may need to avoid obstacles. We call a (part of a) push plan *descending* when P only moves in the direction of PR_b^- while pushing O, this is shown in Figure 8.3. Analogously, an *ascending* push plan is a plan in which P only moves in the direction of PR_e^-.

Suppose a push plan for the straight line compliant section exists. This means that the width of the reachable valid push range is larger than 0 for all $s \in [i^s, i^e]$. Recall that P is free to choose a push position within the reachable valid push range at every point of the section since these all result in the same motion for O.

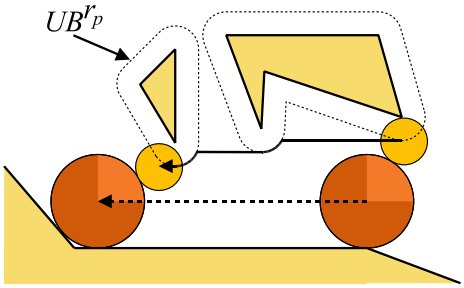
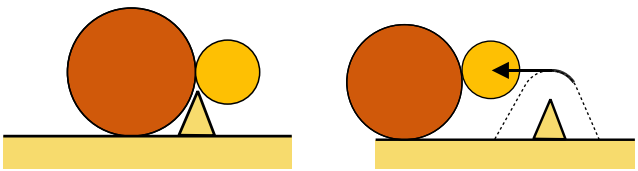


Figure 8.3: An example of a part of a push plan. The path for O is shown as the dotted arrow. The positions for O and P are shown for both i^s and i^e. At i^s, P is as far away from PR_b^- as possible. The resulting push plan for P is shown as the solid black arrow. As can be seen, there is no need for P to rotate away from PR_b^-; a descending path suffices.

We start by a contact transit for P to a position as close to PR_e^- as possible (this is the position in the reachable valid push range furthest away from γ). Pusher P starts pushing O until P hits an obstacle. Next, we let P follow UB^{r_p} for as long as the path of P is descending (after that, P fits underneath the obstacle). Next, we again find the first obstacle that will be hit by P (which is now at a different position). We repeat this procedure until we have reached the end i^e of the section. This approach ensures that P only changes its position if necessary. The path $\sigma(s)$ for P now consists of straight line sections and circular sections.

Since P starts as close to PR_e^- as possible, it is forced to revolve toward PR_b^-, if it hits an obstacle. However, since an obstacle cannot enter the lower wedge while O slides along an edge of an obstacle (it would intersect with O), the pusher will never be forced to revolve in the direction of PR_e^-. Thus if a push plan exists, also a descending push plan exists. There is one exception though. If at i^s an obstacle is present in the lower wedge (Figure 8.4), P may be forced to move in the direction of PR_e^-. If this is the case, we simply follow UB^{r_p} as long as the path for P is ascending.

Using the above method, P always stays within RVPR. Therefore we are guaranteed to find a push plan for the straight line compliant section provided one exists. Because in each section $i \in I$, every obstacle can be hit at most once by P, in total at most n changes of the pusher position are required per straight line compliant section using this method.



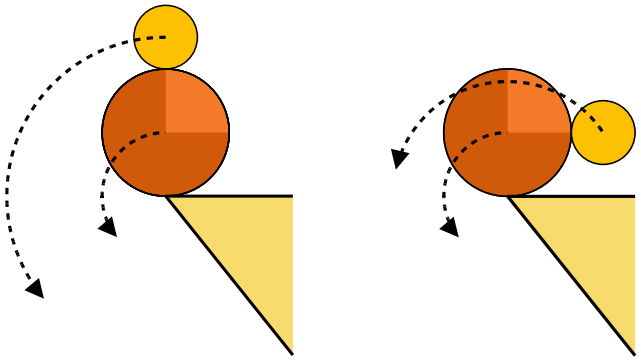
(a) The triangular obstacle intersects with the wedge area.

(b) P follows an ascending path.

Figure 8.4: Example of an obstacle in the wedge area.

8.3.2 Circular Compliant Sections

When O is pushed compliantly around a vertex in a certain section $i \in I$, i is circular shaped. Let $s \in [i^s, i^e]$, then if P holds the same relative position inside $\text{PR}^-(\tau(s))$ during the section, the path of the pusher is also circular. The radius of this pusher path depends on the exact position of P within $\text{PR}^-(\tau(s))$. If P is positioned at PR^-_e, the radius is maximal and equal to $2r_o + r_p$. If P is positioned at PR^-_b, the radius of its path is minimal and equal to $\sqrt{r_p^2 + 2r_p r_o + 2r_o^2}$. The smallest and largest values of this radius are shown in Figure 8.5(a)+(b). The smaller the radius of the path of P, the smaller the area swept by the combination of O and P (the *combined swept area*). This is because a larger part of P moves inside the area swept by O. Since the area swept by O is collision-free by default, the more P moves inside this area, the likelier it is that P is collision-free.



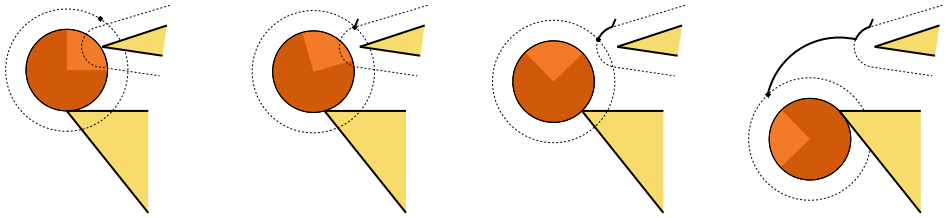
(a) The largest value of the radius of the path for P. The radius equals $2r_o + r_p$.

(b) The smallest value of the radius of the path for P. It can be easily seen that this radius equals $\sqrt{r_p^2 + 2r_p r_o + 2r_o^2}$.

Figure 8.5: Pushing O around a vertex.

Using the above observation, we use an approach that aims at minimizing the combined swept area such that the probability an obstacle is encountered is minimized. The pusher tries to reach PR^-_b; however, we do not use a contact transit, but rather use the following approach as illustrated in Figure 8.6(a)…(c). Starting at i^s, we try to move P along a straight line toward the vertex to which O is compliant. As a result, P will start pushing O around the vertex. If an obstacle is encountered, we follow UB^{r_p}. This process is continued until $\sigma(s) = \text{PR}^-_b(\tau(s))$ the end of the section is reached. If we call this position s', we have now found a valid push plan for the domain $[i^s, s']$. If i^e has not yet been reached then, for the rest of the section (the domain $[s', i^e]$) we let $\sigma(s) = \text{PR}^-_b(\tau(s))$. The combined swept area (if no obstacle is encountered) is shown in Figure 8.7.

As can be seen from the figure, the approach minimizes the combined swept area of O and P. For the second part of the push plan, where $\sigma(s) = \text{PR}^-_b(\tau(s))$ the combined swept area S is always a subset of that of any other valid push plan for section i. If we would use a



(a) The situation at $s = i^s$.

(b) The pusher starts pushing in the direction of the vertex.

(c) P follows UB^{rp} as long as it is moving toward the vertex to which O is compliant or, as is the case here, until $\sigma(s) = \text{PR}_b^-(\tau(s))$.

(d) After P has reached PR_b^- it can simply follow the path of PR_b^- for the rest of the section.

Figure 8.6: Example of a circular compliant section. The pusher is shown as a dot that represents its center.

different push position for P it would be closer to PR_e^- and the combined swept area would always be a superset of S.

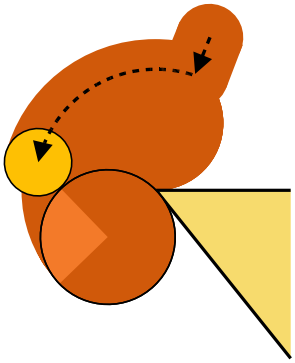
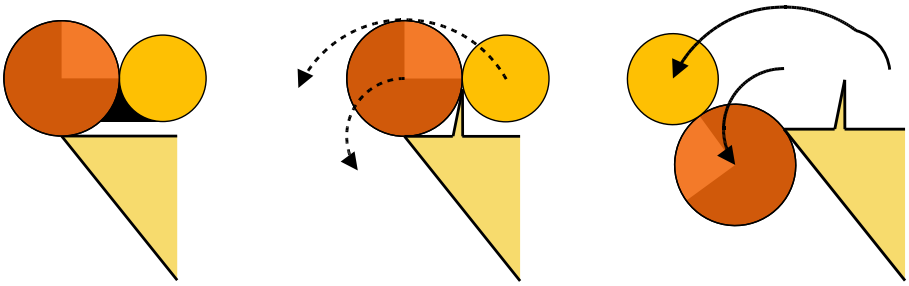


Figure 8.7: The combined swept area if no obstacle is encountered.

Note that an exception occurs when at $s = i_b$, $\text{PR}_b^-(\tau(i^s))$ is reachable but an obstacle is present in the lower wedge area (Figure 8.8(a)). In this case, we partially follow UB^{rs} until $\sigma(s) = \text{PR}_b^-(\tau(s))$ in order to avoid the obstacle (Figures 8.8(b)+(c)).

8.3.3 Non-Compliant Sections

If a section $i \in I$ on the domain $s \in [i^s, i^e]$ is non-compliant, then $\text{PR}_b(\tau(s)) = \text{PR}_e(\tau(s))$. Thus for a non-compliant section the push range consists of only one single push position. If $\text{PR}_b(\tau(s))$ is collision-free for the entire section, a push plan exists for that section. Since P moves largely in the area swept by O, only if an obstacle is present in the wedge area at



(a) If an obstacle is present inside the wedge area between O and P, the intended path for P may be blocked.

(b) The intended path for P collides if P follows the path of PR_b^-.

(c) The solution is to first partially follow UB^{r_s} until $\sigma(s) = \text{PR}_b^-(\tau(s))$.

Figure 8.8: Even though P can reach PR_b^- at i^s, its path is blocked by an obstacle.

i^s and the section is long enough such that P actually collides with this obstacle, then no push plan for this section exists and we report failure.

8.3.4 Contact Transits

Contact transits are necessary at the start of any section $i \in I$ if $\sigma(i^s) \notin \text{VPR}(\tau(i^s))$. In this case, we need to construct a contact transit for the pusher such that $\sigma(i^s) \in \text{VPR}(\tau(i^s))$. P can only be outside VPR at the start of a section. (Recall that if P is forced outside PR inside a section, then no push plan exists and the algorithm reports failure.)

When a contact transit is necessary, P has the choice between a clockwise or counterclockwise path. If the current or next section is compliant, then one of these paths is blocked by the obstacle to which O is compliant. During a contact transit, the path of P is always a circular arc with radius $r_p + r_o$. If both paths are blocked by an obstacle then the transit is not possible and failure is reported.

8.4 Data Structures

In the previous section we have described the global approach. A number of collision queries must be answered to actually compute a push plan. In this section we will describe the data structures that are needed for this and provide time bounds for the operations. Again we describe the algorithm for the counterclockwise compliant paths, the clockwise paths are analogous.

8.4.1 Straight Line Compliant Sections

A straight line compliant section $i \in I$ is defined on the domain $[i^s, i^e]$. The position for P at i^s is as far from PR_b^- as possible. We need to determine the first obstacle that forces P to change its position. To find this obstacle, we use ray shooting. Ray shooting considers

the problem of determining the first intersection between a ray (a directed line segment emanating from a point) and a collection of obstacles. A ray is shot from the center of P at, parallel to the edge to which O is compliant. If the ray hits UB^{r_p}, this indicates a collision between P and an obstacle. We can safely push O to the point that P actually hits the obstacle. Now P can follow UB^{r_p} for as its path is descending (as shown in Figure 8.3). If the boundary ceases to descend, the pusher is at a position where it fits underneath the obstacle. Since the obstacles consist of straight line segments, we cannot encounter this obstacle again in section i. We repeat the ray shooting procedure until O reaches i^e. If an obstacle is present in the wedge area (Figure 8.4), the same procedure is used except that UB^{r_p} is followed until P ceases to *ascend*. Then P has reached a position where it is able to fit above the obstacle.

In order to find a colliding obstacle we need to perform a ray shooting query in a space consisting of circular arcs and line segments (that form UB^{r_p}). A procedure for ray shooting amidst possibly intersecting algebraic arcs in the plane is given by Koltun (2001). The paper describes a data structure that can be constructed in time $O(n^2 \log n)$ and has size $O(n^2)$. Ray shooting queries can then be answered in $O(\log n)$ time. The total number of ray shooting queries necessary is $O(k_c n)$ since every element of UB^{r_p} can be encountered at every straight line compliant section in worst case.

Lemma 8.4.1. *We can preprocess an environment in $O(n^2 \log n)$ time such that a push plan for a straight line compliant section can be found in $O(n \log n)$ time. Since there are k_c compliant sections, the total query time of all straight line compliant sections is $O(k_c n \log n)$.*

8.4.2 Circular Compliant Sections

As stated before, the preferred position of P during a circular compliant section $i \in I$ is $\text{PR}_b^-(\tau(s))$. We divide the algorithm in two parts. First, we try to find an $s \in [i^s, i^e]$ for which $\text{PR}_b^-(\tau(s)) \in \text{RVPR}(\tau(s), \sigma(s))$. If such an s exists, then $\sigma(s) = \text{PR}_b^-(\tau(s))$ and from then on the radius of the circular path of P equals $\sqrt{r_p^2 + 2r_p r_o + 2r_o^2}$.

In order to reach $\text{PR}_b^-(\tau(s))$ (Figure 8.6), we proceed similar to Section 8.4.1. First, we shoot a ray from the current position of P to the vertex to which O is compliant. Then, we push O until P collides. Next, we let P follow UB^{r_p} as long as P is moving toward the vertex of the edge to which O is compliant. This procedure is repeated until $\sigma(s) = \text{PR}_b(\tau(s))$ or O reaches the end of the section. For ray shooting we can reuse the structure of Section 8.4.1. In worst case all obstacles can be present in the wedge area.

If we did not reach i^e, we continue with a circular path for P having radius $\sqrt{r_p^2 + 2r_p r_o + 2r_o^2}$. For this second part of the section, we preprocess the environment by finding all intersections of all possible circular paths of P with UB^{r_p}. For this we add circular arcs with radius $\sqrt{r_p^2 + 2r_p r_o + 2r_o^2}$ to the environment at every vertex. The total number of elements in this environment is $O(n)$. We find intersections by using a plane sweep algorithm. The original version of this algorithm is described by Bentley and Ottmann (1979). Balaban (1995) described an algorithm that also works for curved segments. His algorithm takes $O(n \log n + q)$ time and uses $O(n)$ space to list all q intersections. We observe that $q = O(n^2)$ and this bound can be achieved when the obstacles are cluttered. The

algorithm of Balaban requires each segment to have at most one intersection with any vertical line, which means that we may need to split certain circular path sections of P into two separate pieces, which does not influence the time bounds. If an intersection with an obstacle is found, then, since P is already at PR_b^-, no push plan exists. This is the situation of Figure 7.5(f).

Lemma 8.4.2. *Finding all intersections of circular compliant sections together costs $O(n^2)$ time. For the ray shooting part, we can preprocess the environment in $O(n^2 \log n)$ time such that a push plan can be found in $O(n \log n)$ time. In worst case we encounter every obstacle at every circular section. Thus, the ray shooting queries for all circular sections together cost $O(k_c n \log n)$ time.*

Note that since the straight line compliant sections and the circular compliant sections both use the same type of ray shooting, the results of preprocessing can be shared.

8.4.3 Non-Compliant Sections

For a non-compliant section $i \in I$, there is only one valid push position. The only way to invalidate the push plan is if an obstacle is present in one of the wedge areas as shown in Figure 7.6 and i is long enough for the collision to actually occur. We can check this by shooting a ray from the center of P at the start of the section in the direction of $\tau(i^s)$. If the ray collides with UB^{r_p} before the end of the section, then no push plan for this section exists. We can use the data structure created in Section 8.4.1. Since there are k_n non-compliant sections, the total query time is $O(k_n \log n)$.

Lemma 8.4.3. *The total query time used for the non-compliant sections together is $O(k_n \log n)$.*

8.4.4 Contact Transits

Since the object path τ is known in advance, we also know all positions where contact transits may be necessary. Since the total number of sections of τ is k, so is the number of contact transits. The path of P during a contact transit is always circular with radius $r_p + r_o$. In the case of two subsequent non-compliant sections, P can revolve about O both in the clockwise and counterclockwise directions and thus two possible transits exist. If the circular arc of the path of P intersects UB^{r_p}, then no transit following this path is possible. We can again preprocess the environment using Balaban's algorithm to find all intersections of the contact transit circular arcs with UB^{r_p} as described in 8.4.2. In worst case we encounter every obstacle at the start of every section resulting in kn intersections.

Lemma 8.4.4. *In order to find all possible intersections during contact transits, in the query phase we use $O((k+n)\log(k+n) + kn)$ time.*

8.4.5 Run Time Analysis

For the two types of compliant sections, we need ray shooting. Also for the non-compliant sections, one ray shooting query is necessary. The preprocessing time for the algorithm is

$O(n^2 \log n)$ (Lemma 8.4.1). We can also preprocess the data structure necessary to find the intersections for the circular compliant sections. According to Lemma 8.4.2, this preprocessing takes $O(n^2)$ time. Combining Lemmas 8.4.1, 8.4.2 and 8.4.3 results in a total ray shooting query time of $O(kn \log n)$. For the contact transits we need a total query time of $O((k+n)\log(k+n)+kn)$ (Lemma 8.4.4).

Theorem 8.4.1. *A push plan for P such that O is pushed along a path consisting of both compliant and non-compliant sections during which P remains in contact with O can be computed in $O(kn \log n)$. The preprocessing takes $O(n^2 \log n)$ time. The complexity of the push plan is $O(kn)$. If no path exists, this is reported within the same time bounds.*

8.4.6 Path Existence

The procedures of the previous sections describe how to efficiently find a push plan for P. It may however be that such a plan does not exist. The path τ for O is given. It is assumed that this path does not intersect with any obstacle. Because P is often able to move in the area swept by O and since P is not allowed to lose contact with O, there is only a limited number of events that can cause a push plan to fail. A number of these are naturally discovered when creating the push plan as described in this section. Here, we will briefly summarize all cases.

The first case occurs if at the start of a section $i \in I$ of τ an obstacle is present in the wedge area. If $\tau(i^s)$ is non-compliant, then an obstacle in the wedge area prevents P from pushing O along the intended path. This is easily detected by the ray shooting query as described in Section 8.4.3.

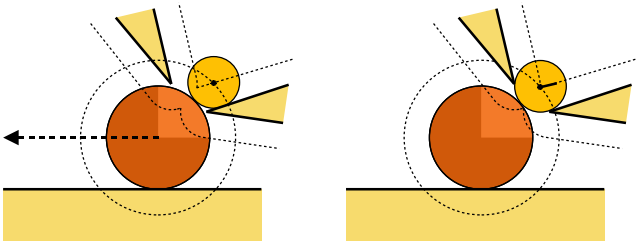
If $\tau(0)$ is compliant, P can get stuck if it encounters an intersection of racetracks. Such a situation is shown in Figure 8.9. Here, P follows UB^{r_p}. If UB^{r_p} forces P to move away from O then we know P is stuck. Again this can be easily detected without additional cost.

If P is forced outside PR in a straight line compliant section, then also no push plan exists. Simply checking if P is still within PR after every section of $\sigma(s)$ detects this. For the first part of a circular compliant section the same holds. For the second part, it suffices to check if PR_b^- is collision-free as described in 8.4.2. Finally no push plan exists if P is not able to complete a contact transit.

8.5 Low Obstacle Density

In practical settings, the complexity of the free space tends to remain far below the theoretical worst-case complexity bounds. A realistic assumption about the complexity of the free space is *low obstacle density*. Path planning in environments having low obstacle density has been well studied, for example by Van der Stappen et al. (1998). Before we formally define low obstacle density, we first devise a useful property of the object path τ.

If a straight line compliant section $i \in I$ is longer than some threshold d and O has been pushed at least a distance d on the current section, we are certain that for all $s \in [i^s + d, i^e]$, $\text{PR}_b^-(\tau(s)) \in \text{RVPR}(\tau(s), \sigma(s))$. Thus, if $s = i^s + d$ we are certain to be able to transit P to

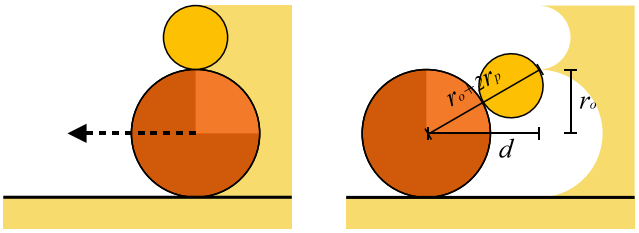


(a) The dotted lines show UB^{rp}. Also around O, the pusher positions are shown. The intended path for O is shown as the dotted arrow.

(b) Following UB^{rp}, P is forced to move away from O; P is stuck.

Figure 8.9: The pusher can get stuck at the start of a compliant section.

$\text{PR}_b^-(\tau(i^s+d))$ and stay at that position until i^e has been reached. This can be seen as follows. Since it is not possible for an obstacle to enter the sweep area of O after the start of the section, an obstacle that prevents $\text{PR}_b^-(\tau(s))$ from being in $\text{RVPR}(\tau(s),\sigma(s))$ already does so at the start of the section, therefore there must be a d for which the obstacle no longer prevents P from reaching PR_b^-. If the distance between the center of O and the obstacle closest to this center is at least r_o+2r_p, we are certain that P can reach PR_b^-. Using this observation it is easy to see that if $d > \sqrt{4r_p^2+4r_p r_o}$, the pusher is guaranteed to be able to reach PR_b^- (Figure 8.10).



(a) The area that represents the union of all possible obstacles.

(b) The first position from which P is certain to be able to reach $\text{PR}_b^-(\tau(i^s+d))$. This is the case if $d > \sqrt{4r_p^2+4r_p r_o}$.

Figure 8.10: Establishing the minimum distance d that O has to move to guarantee that $\text{PR}_b^-(\tau(i^s+d)) \in \text{RVPR}(\tau(i^s+d),\sigma(i^s+d))$.

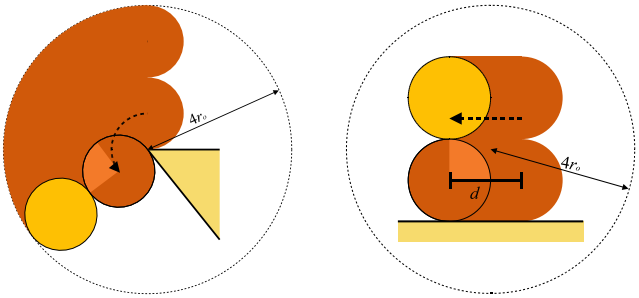
Lemma 8.5.1 (Reachability of PR_b)**.** *If $(i^e-i^s) > \sqrt{4r_p^2+4r_p r_o}$ for compliant section i, the following holds for all $s \in [i^s+\sqrt{4r_p^2+4r_p r_o}, i^e]$:*

$$\text{PR}_b(\tau(s)) \in \text{RVPR}(\tau(s),\sigma(s)).$$

This property is useful if the environment satisfies the definition of *low obstacle density*. There are several types of definitions for low obstacle density, but the most general one is defined as follows.

Definition 8.5.1 (Low obstacle density)**.** *Let* $\mathbb{R}^2$ *be a space with a set* Γ *of obstacles. Then* $\mathbb{R}^2$ *is said to be a* low obstacle density *space if any region with minimal enclosing circle of radius* λ *intersects at most a constant number of obstacles* $\Gamma' \subset \Gamma$ *with minimal enclosing circle of radius at least* $c\lambda$ *for some constant* $c > 0$*.*

In a circular compliant section, the largest combined swept area is observed if $\forall s \in [i^s, i^e] : \sigma(s) = \text{PR}_e^-$ and $r_p = r_o$. Thus, the total combined swept area always fits within a disk of radius $4r_o$, see Figure 8.11(a). For a straight line compliant section, using Lemma 8.5.1 it is easy to see that the combined swept area on the domain $[i^s, i^s + \sqrt{4r_p^2 + 4r_p r_o}]$ also fits within a disk of radius $4r_o$ (Figure 8.11(b)). Recall that for the rest of the domain $[i^s + \sqrt{4r_p^2 + 4r_p r_o}, i^e]$, P cannot encounter an obstacle because it moves completely within the swept area of O.



(a) The combined swept area in a circular compliant section always fits within a disk of radius $4r_o$.

(b) Also the combined swept area of the first part of a straight line compliant section fits within the same disk.

Figure 8.11: The combined swept area of compliant sections always fits within a disk of radius $4r_o$.

Let λ be the length of the smallest obstacle $\gamma \in \Gamma$. If our environment Γ with n non-intersecting line segments satisfies the low obstacle density property, then at most a constant number of obstacles intersects with a disk shaped region of radius $c\lambda$ for some $c \geq 0$. If we assume that $r_o < c\lambda$ for some $c \geq 0$, then also a disk shaped region of radius r_o intersects at most a constant number of obstacles. This implies that a disk shaped region of radius $4r_o$ also intersects at most a constant number of obstacles. Since each obstacle can be encountered at most once every section, we know that P can encounter at most a constant number of obstacles in a compliant section. Therefore at most a constant number of ray shooting queries is needed for a compliant section if the environment satisfies the low obstacle density. We now partially restate Theorem 8.4.1.

Theorem 8.5.1. *If the environment satisfies the low obstacle density property and if the radius of O is at most a constant times the size of any obstacle, then a push plan such that O is pushed along a path consisting of both compliant and non-compliant sections can be computed in $O((k+n)\log n)$ time. The complexity of the push plan is $O(k)$.*

8.6 Concluding Remarks

In this chapter we have studied the problem in which a disk pushes another (passive) disk while remaining in contact. The passive disk is allowed to make use of compliance in order to extend the number of pushable paths. It is a first step in exploring the potential of the combination of pushing and compliance. Exploiting the boundaries of the environment increases the possibilities of finding a push plan. In Chapter 7 we have already concluded that an efficient polynomial time algorithm for the problem does not exist. However, we have created an algorithm that solves a subproblem: given a path for a passive object, create a push plan for a pusher such that the object follows this intended path. We have created an algorithm that solves the problem in $O(kn\log n)$ time and, using a realistic assumption about the environment, it can even be solved in $O(k\log n)$ time.

Using compliance has several advantages. A overview of which can be found in Chapter 7. For the algorithm in this chapter the most important advantage is that in the compliant sections of the path, there is a range of push positions that all cause the object to move in the same direction. This property allowed us to create an algorithm in which the pusher is able to avoid obstacles by revolving around the object without affecting the direction in which the object is pushed. The width of the range is determined by the friction. If friction exists between the object and the environment, this can easily be incorporated by narrowing the push range according to the friction cone. More explanation about the role of friction was given in Section 7.2.1.

The results of this chapter provide promising insights for the development of an algorithm in which also the path for the object is created. The bridges between the compliant sections of the path will need to be created using an approximate algorithm because of the complexity.

CHAPTER 9

Creating a Manipulation Plan

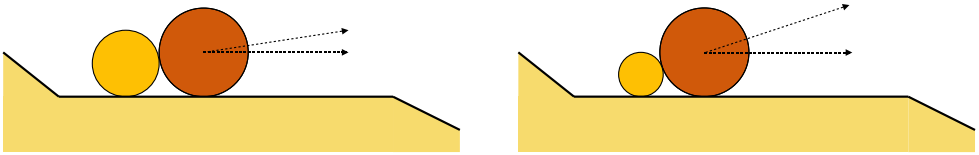
In the previous chapter we have created a push plan given the path of an object. In this chapter we will build on these results by generating a path for both the object and the pusher. A manipulation plan is created such that if the pusher satisfies this plan the object is pushed from a start to a goal configuration. In this plan both compliant and non-compliant paths are allowed. To create the compliant paths we use a complete computation of reachable compliant configurations. For the non-compliant paths we will use an approach based on Rapidly-exploring Random Trees (RRTs).

9.1 Introduction

Our approach is based on the Rapidly-exploring Random Trees (RRTs) (LaValle and Kuffner, 2001) algorithm. It combines random exploration of open spaces with an exact computation of reachable compliant configurations. For the open areas, a tree is created by generating random configurations. For every random configuration a path is tried from the tree to that random configuration. If the random configuration cannot be reached because of a collision, a configuration as close as possible to the obstacle that caused the collision is added to the tree. This configuration is then used as a starting point to explore the compliant configurations using an exact approach, eventually capturing the structure and connectivity of all compliant configurations. The results of compliant exploration are added as configurations to the tree.

9.2 Preliminaries

In the previous chapter we used straight line sections to act as bridges between compliant path sections. Using only straight line segments as non-compliant sections however limits the pusher in pushing the object away from an obstacle. This problem gets worse as the size of the pusher grows. The obstacle to which O is compliant limits P in reaching certain push positions. The larger P is, the more push positions are unreachable for P, see Figure 9.1. This prevents P from pushing O in certain directions. The planner may even fail because of this limitation. To efficiently push O away from a compliant configuration, we will use a method in which O slides along the boundary of P (an *unstable* push). If sensor errors are involved, then the final position of O may not be exactly known beforehand. Because of the way our planning works in non-compliant space (Section 9.3.1), such sensor errors are of little influence on the performance of the planner.



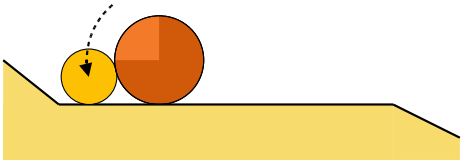
(a) With a large P the range of directions to which O can be pushed is small.

(b) With a smaller P the range increases but is still not large.

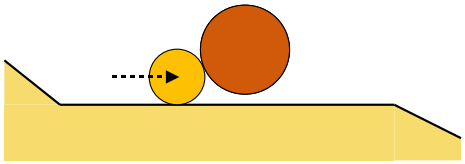
Figure 9.1: Pushing O away from a compliant configuration using a straight line push.

Recall that the environment consists of n non-intersecting line segments $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$. The most effective way of pushing O away from a compliant configuration on obstacle γ_i (i.e. limiting the length of the path), is by maximizing the angle in which O is pushed away from γ_i. Therefore, we start by revolving P to a position as close to γ_i as possible. If P then follows a path parallel to γ_i, it pushes O away from γ_i, while continuously maximizing the angle in which O is pushed away from γ_i. If no other obstacles impede the path, the resulting motion of the object is a curve called a *hockey stick curve* (see Figure 9.2). A hockey stick curve minimizes the distance needed to push O away from an obstacle. Agarwal et al. (1997) gave a mathematical description of a hockey stick curve for a point pushing a disk. The resulting motion of a pusher of radius r_p pushing an object of radius r_o is equivalent to that of a point pushing a disk of radius $r_p + r_o$ where the point represents the center of P. The coordinates of O as a function of time are shown as Equation 9.1, β is the initial angle the line through the centers of O and P makes with γ_i.

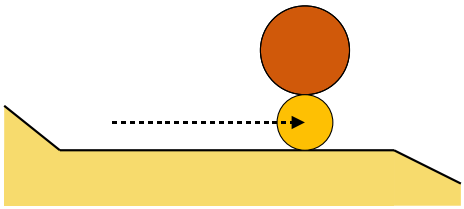
$$
\begin{aligned}
x(t) &= t + \frac{1 - \tan^2\left(\frac{\beta}{2}\right) \cdot e^{2t}}{1 + \tan^2\left(\frac{\beta}{2}\right) \cdot e^{2t}} - \cos(\beta) \\
y(t) &= \frac{2\tan\left(\frac{\beta}{2}\right) \cdot e^{t}}{1 + \tan^2\left(\frac{\beta}{2}\right) \cdot e^{2t}} - \sin(\beta)
\end{aligned}
\tag{9.1}
$$



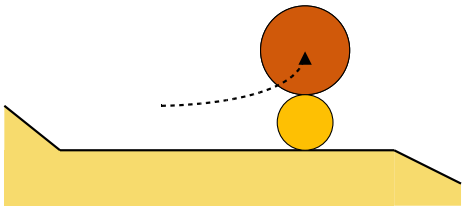
(a) To maximize the angle, first P revolves to a position as close to the obstacle as possible.



(b) P starts pushing. O is pushed away from the obstacle.



(c) The distance between the obstacle and O is now equal to $2r_p$.



(d) The path O has traversed is a hockey stick curve.

Figure 9.2: Pushing the object away from a compliant configuration using a hockey stick curve.

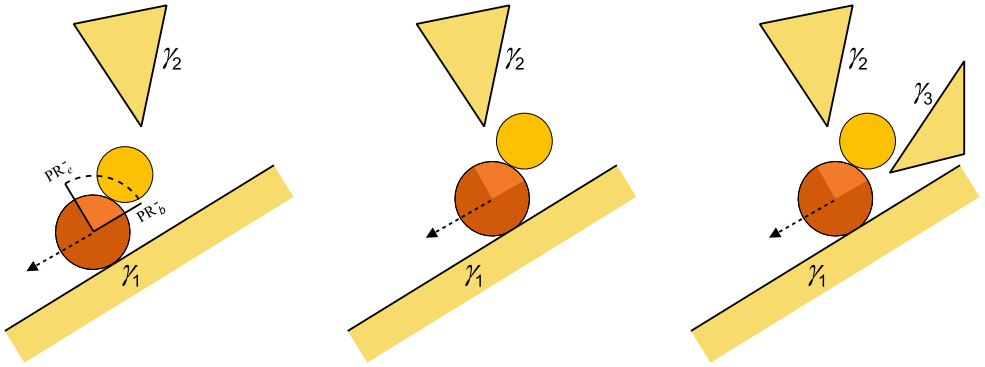
We need some definitions in addition to the definitions of Section 7.2.2. We will group subsets of compliant space (consisting of UB^{r_o}) that share certain properties. Given a compliant configuration, we define the *bounding obstacles* (Figure 9.3).

Definition 9.2.1 (Bounding obstacles)**.** *Given a compliant configuration $c = (q, \alpha)$, the* bounding obstacles $BO(q, \alpha)$ *are the ordered pair of obstacles (γ_j, γ_k) that P hits first if it rotates from α to either $\text{PR}_b^+/\text{PR}_b^-$ and $\text{PR}_e^+/\text{PR}_e^-$ respectively. The absence of a bounding obstacle, is denoted by $\perp$.*

We introduce the notion of *compliant intervals*. A compliant interval is a continuous subset of the racetrack of an obstacle in which the bounding obstacles are fixed.

Definition 9.2.2 (Compliant interval)**.** *A compliant interval $I_{\gamma_i}(\gamma_j, \gamma_k) = \{q \in R_{\gamma_i}^{r_o} | \exists \alpha : BO(q, \alpha) = (\gamma_j, \gamma_k)\}$ of obstacle γ_i is maximal connected subset of $R_{\gamma_i}^{r_o}$ that shares the same bounding obstacles. The compliant start position of $I_{\gamma_i}(\gamma_j, \gamma_k)$ is denoted by $q_b(I_{\gamma_i}(\gamma_j, \gamma_k))$, its end position by $q_e(I_{\gamma_i}(\gamma_j, \gamma_k))$.*

Because the bounding obstacles form an ordered pair, an interval is directed, i.e. both the clockwise and counterclockwise paths on the racetrack have their own unique set of intervals. At one compliant position of an obstacle, multiple (possibly overlapping) intervals can exist; an example is shown in Figure 9.4(a)…(g). We observe that a compliant interval is unique, i.e. an interval with the same bounding obstacles cannot occur twice on a racetrack of an obstacle. Exceptions are the compliant configurations where no obstacles bound PR^+/PR^- (as is the case in the interval shown in Figure 9.4(b)), but since these cannot overlap, they can still be uniquely defined by their start and end positions. The following lemma follows directly from Definitions 9.2.1 and 9.2.2.



(a) If P rotates to either PR_b^- or PR_e^- it does not encounter any obstacle, therefore the bounding obstacles are $(\perp, \perp)$.

(b) P cannot reach PR_e^-, the bounding obstacles are $(\perp, \gamma_2)$.

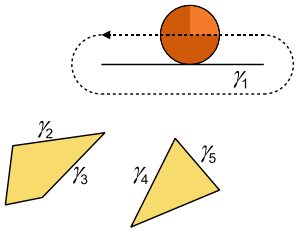
(c) The bounding obstacles are (γ_3, γ_2).

Figure 9.3: Bounding obstacles. For the relevant obstacles, the labels are given.

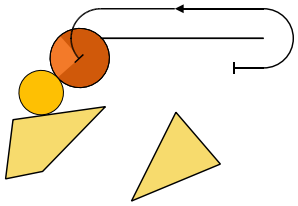
Lemma 9.2.1 (Reaching the end of an interval)**.** *Given a compliant configuration* $c = (q, \alpha)$ *with* $q \in I_{\gamma_i}(\gamma_j, \gamma_k) \wedge BO(q, \alpha) = (\gamma_j, \gamma_k)$*, it is assured that* P *can push* O *compliantly to* $q_e(I_{\gamma_i}(\gamma_j, \gamma_k))$*.*

9.3 Rapidly-exploring Random Trees

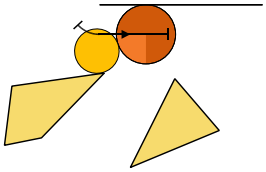
Many path planning algorithms are based on the generation of collision-free samples (Chapter 1). Between these samples, connections are tried and a graph is built that can be used to solve path planning queries. Since standard random sampling does not result in compliant samples, we need a method to create them. Retracting non-compliant random samples to the obstacles (Amato et al., 1998a) seems a straightforward solution. However, this approach introduces the problem of deciding when and how samples need to be retracted which is a difficult problem. In addition, retraction of samples tends to be a costly operation because of the additional collision checks involved. After a random sample has been created, most algorithms try to connect it to already existing vertices in the graph. If the connection fails (because of a collision), an obstacle must be impeding the path. Usually such connections are discarded. In our case however, these collisions are valuable in creating compliant samples. Since we aim at creating a path from a single start to a single goal configuration (single shot approach), we use Rapidly-exploring Random Trees (RRTs) introduced by LaValle and Kuffner (2001) to create the necessary graph. We will first briefly explain how the basic RRT algorithm works and then elaborate on how to adapt the RRT such that it is suited to solve our problem.



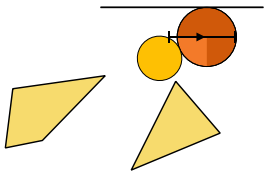
(a) Pushing O counterclockwise around γ_1, every compliant position can be reached



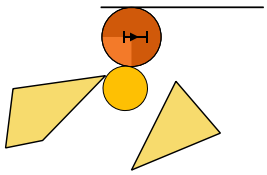
(b) The interval in which no obstacles bound the push range $(\perp,\perp)$.



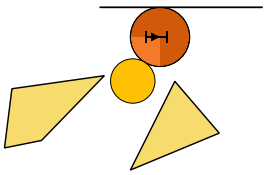
(c) The interval $(\perp,\gamma_2)$.



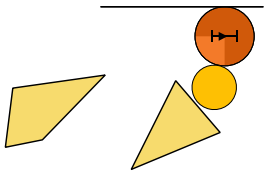
(d) The interval $(\perp,\gamma_4)$.



(e) The interval $(\gamma_3,\perp)$.



(f) The interval (γ_3,γ_4).



(g) The interval $(\gamma_5,\perp)$.

Figure 9.4: The compliant intervals of the counterclockwise path on the racetrack of obstacle γ_1. Note that some intervals overlap.

9.3.1 The Basic RRT Algorithm

As we use a version of the RRT algorithm that is slightly different from the one used in Section 6.1.4, we will briefly restate the algorithm. Recall that the RRT is a single shot approach in which a tree T is constructed that gradually improves resolution. In its most basic form, it grows a tree from the start configuration c_s until the goal configuration c_g can be connected to T.

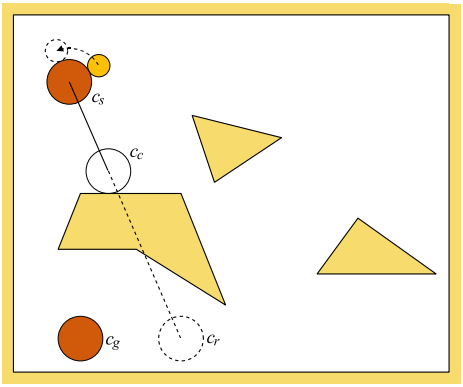
Initially the start configuration c_s is the only vertex of T. Next, a random configuration c_r is generated. The nearest configuration c_n in T to c_r is found (using a Euclidean distance metric). Configuration c_n can either be associated with a vertex in T or it can be a configuration in the interior of an edge in T. In either case, a local planner is used that tries to create a path from c_n to c_r. If on this path an obstacle is encountered, the closest configuration c_c to the boundary of the obstacle is determined. If no obstacle is encountered and thus c_r is reached, then $c_c = c_r$. Next, c_c is added to T as a vertex, together with the edge (c_n, c_c). By repeating this process, T gradually explores the free space. To make

sure that c_g will be connected to the tree, occasionally $c_r = c_g$. This will force the algorithm to try if a connection to c_g is possible. If this is the case, a solution has been found. For the rest of this chapter we have chosen to try a connection to c_g every 20^{th} iteration of the algorithm. This is a value that works well in all our experiments.

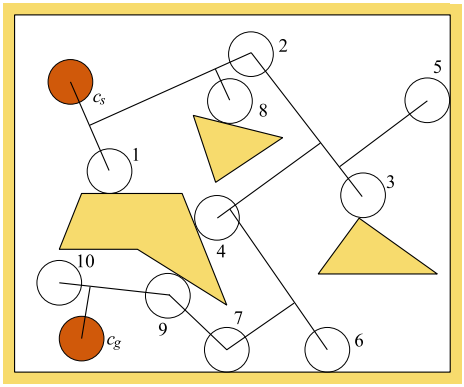
9.3.2 Tailoring the RRT

The basic RRT algorithm is not suited for pushing problems since it is unable to deal with the joint motion of an autonomous P and a passive O. We need a special version of the local planner that takes this joint behavior into account (Section 9.4). In our version of the algorithm, we only generate random positions for O, the positions for P are determined by the local planner (Figure 9.5(a)).

Using RRTs as a basic planning concept, leads to compliant configurations in a natural way. If the local planner encounters an obstacle when connecting two configurations, the point of collision is a compliant configuration. The more obstacles are present between the start configuration c_s and the goal configuration c_g, the more compliant configurations will be found by the RRT, thus a natural balance exists between compliant and non-compliant configurations. An example of the RRT algorithm creating compliant configurations is shown in Figure 9.5(b).



(a) The first iteration, the position of P is determined by the direction of the path from c_s to c_c.



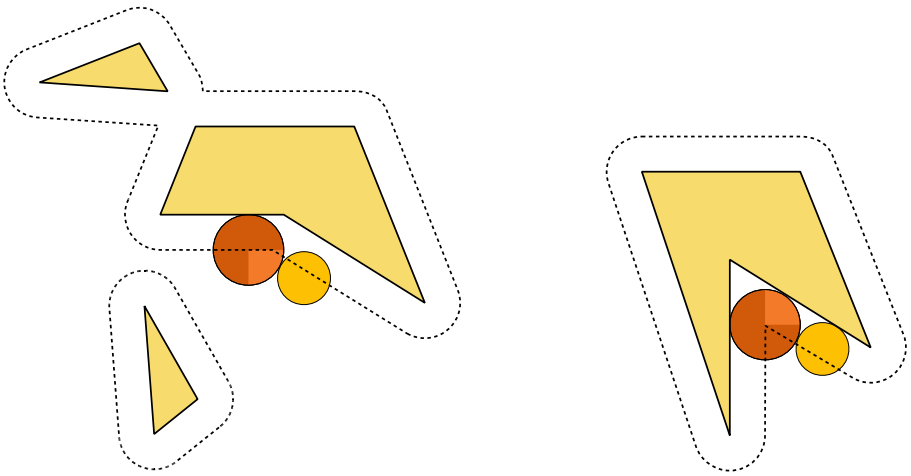
(b) The final RRT. As can be seen, a number of compliant configurations have been found.

Figure 9.5: Example of the creation of an RRT. The order in which the vertices were added to the tree is shown.

To handle compliance, the RRT algorithm needs to be extended. If a compliant configuration has been found we need to discover which part of the compliant space is reachable from this configuration. We call this procedure *compliant exploration* (Section 9.5). Every obstacle γ_i has two distinct compliant exploration directions: one in the clockwise and one in the counterclockwise direction along the racetrack induced by γ_i.

During compliant exploration, O is pushed around the obstacle until it returns at the start position or until no further pushing is possible. A *compliant component* is a

connected component in compliant space (see Figure 9.6(a) for an example). If obstacles belong to the same compliant component, a path through compliant space may exist between them. The fact that two obstacles belong to the same compliant component by no means guarantees that a path through compliant space can be created as can be seen from Figure 9.6(b). Here, the object O gets stuck between two obstacles that belong to the same compliant component. The ability to create a path also depends on the size and configuration of the obstacles and the size of P. The complete RRT algorithm as used in this chapter is shown in pseudo-code as Algorithm 9.1.



(a) A scene consisting of 3 obstacles (11 line segments). As can be seen from the compliant space UB^{r_O} (the dotted lines), the top 2 obstacles form a compliant component while the bottom obstacle forms a compliant component on its own.

(b) Even though line segments form a compliant component, the object runs into a dead-end.

Figure 9.6: Examples of compliant components.

In line 4, the nearest configuration in the tree needs to be found. If the nearest configuration (c_n) in the tree is non-compliant, the shortest path is simply the straight line path. If a compliant configuration is involved, the shortest path between two configurations may however consist of a (partial) hockey stick curve.

9.4 Local Planner

As in most sampling-based motion planning algorithms, the local planner (line 5 of Algorithm 9.1) plays an important role. In our setting, we need a local planner that is able to verify if the random position q_r for O can be reached from the nearest configuration $c_n = (q_n, \alpha_n)$ in the tree or else it needs to report the first obstacle with which there will be

Algorithm 9.1 PUSHINGRRT(T, c_s, c_g)

```
1:  T.ADDVERTEX (c_s)
2:  repeat
3:      q_r ← random position for O, occasionally q_r is the goal position
4:      c_n ← GETNEARESTNEIGHBOR(T, q_r)
5:      c_c = (q_c, α_c) ← LOCALPLANNER(T, c_n, q_r)
6:      if c_c ≠ NULL {did we find a valid vertex?} then
7:          T.ADDVERTEX (c_c)
8:          T.ADDEDGE (c_n, c_c)
9:          if PATHEXISTS (c_s, c_g) then
10:             RETURN FOUND
11: until stopping criterion is met
```

a collision. If the local planner succeeds in finding a path, a connection in the tree between the corresponding vertices is created.

As our configurations consist of a position of O along with a relative orientation for P, and configurations may or may not be compliant, the local planner needs to adapt its approach to the local situation.

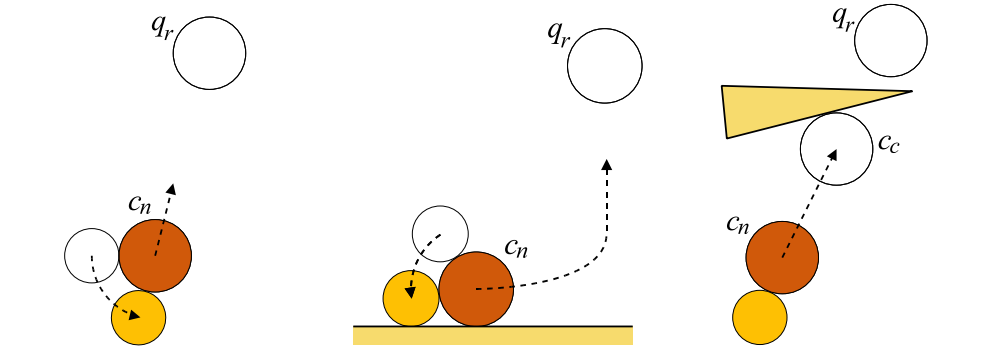
9.4.1 Local Planner Cases

There are three different types of local paths the local planner has to deal with: paths through non-compliant space, paths starting in compliant space and paths ending in compliant space. We will discuss each of them.

Suppose the nearest neighbor configuration c_n resulting from line 4 of Algorithm 9.1 is non-compliant (Figure 9.7(a)). The local planner has to verify whether a straight line path exists that connects the nearest neighbor configuration $c_n = (q_n, \alpha_n)$ to a configuration $c_r = (q_r, \alpha_r)$ where q_r is the sampled position. There is only one push position for P such that O follows a straight line path from q_n to q_r. To reach this push position, a contact transit is applied at q_n; this can either be a clockwise or a counterclockwise transit.

Figure 9.7(b) shows the situation in which c_n is compliant. It depends on the position of q_r whether P is able to transit to the desired push position for a straight line path. It is likely that such a transition is impossible because the obstacle to which O is compliant will probably impede the contact transit. Thus, we use a hockey stick curve to push O away from the obstacle. The hockey stick push maximizes the angle in which P pushes O away from the obstacle. Before the end of the hockey stick is reached, P may encounter another obstacle. If this happens, a new hockey stick push is started and the procedure is repeated until a hockey stick curve can be completed. This situation is shown in Figures 9.8(a)+(b). The process can continue indefinitely e.g. if O is in a small confined area or if O and P have almost similar radius. Although it is unlikely that such situations will occur in practice, restricting the total length of the consecutive hockey stick curves prevents that the process of appending hockey stick curves does not terminate.

In both cases (c_n being compliant or non-compliant), P has reached the desired push

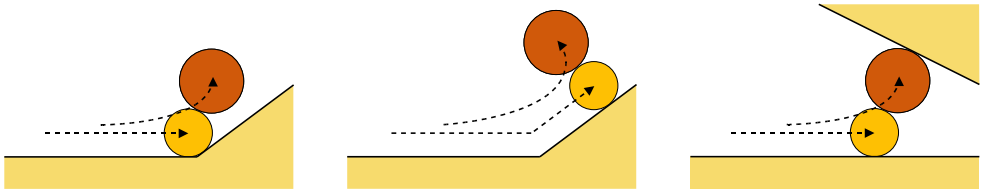


(a) The non-compliant case. A contact transit is followed by a straight line push.

(b) The compliant case. A hockey stick curve is followed by a straight line push.

(c) Hitting an obstacle. While pushing O to q_r, an obstacle is hit and thus a compliant configuration has been found.

Figure 9.7: The different cases the local planner has to deal with.



(a) During the hockey stick curve, P encounters another obstacle.

(b) A second hockey stick curve is initiated.

(c) Example in which O hits another obstacle during a hockey stick curve.

Figure 9.8: Multiple hockey stick curves.

position to try to push O in a straight line from its current position to q_r. If q_r is reached, then a (non-compliant) vertex positioned $c_r = (q_r, \alpha_r)$ with the appropriate α_r (corresponding to the position of P) is added to T together with an edge from c_n to c_r. If an obstacle is encountered at c_c before q_r is reached (Figure 9.7(c)), a compliant configuration is found and is used as a starting point for the compliant exploration algorithm of Section 9.5. It is also possible that during the hockey stick curve O hits an obstacle (Figure 9.8(c)), also in that case, c_c is the collision position and compliant exploration is started at c_c. The complete LocalPlanner algorithm is shown as Algorithm 9.2.

9.4.2 Geometric Primitives

To efficiently check whether $c_r = (q_r, \alpha_r)$ can be reached from the nearest configuration $c_n = (q_n, \alpha_n)$ for some α_r, or to report the first obstacle with which there will be a collision, we can use basic geometry. Before O can be pushed from q_n to q_r, P needs to reach α_n by means of a contact transit. To check the validity of the contact transit, we construct,

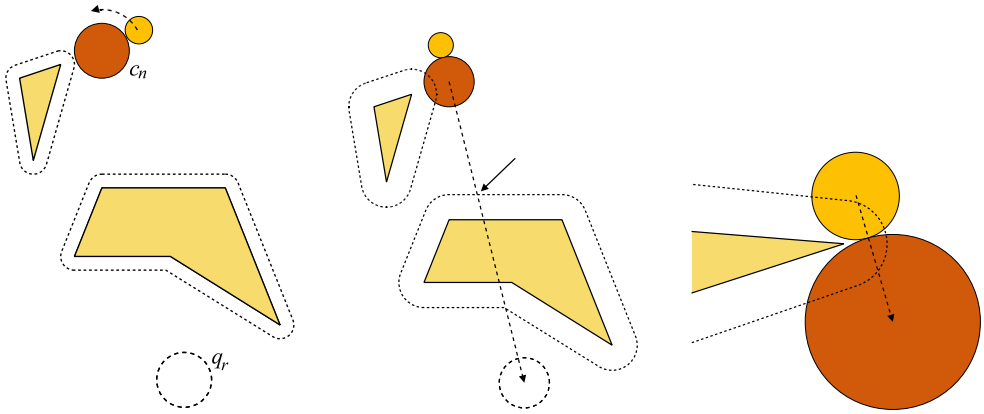
Algorithm 9.2 LOCALPLANNER(T, c_n, q_r)

```
1:  c_c ← c_n
2:  if COMPLIANT(c_n) then
3:      c_c ← CREATEHOCKEYSTICK(c_n)
4:      if c_c = NULL then
5:          RETURN NULL {failure}
6:  c_c ← CREATECONTACTTRANSIT(c_c, q_r)
7:  if c_c = NULL then
8:      RETURN NULL {failure}
9:  c_c ← PUSH(c_c, q_r) {push O from c_c to q_r}
10: if COMPLIANT(c_c) {found compliant configuration} then
11:     EXPLORECOMPLIANTCW(T, c_c) {extends T}
12:     EXPLORECOMPLIANTCCW(T, c_c) {extends T}
13:     RETURN c_c
14: else
15:     RETURN c_c {q_r reached by PUSH}
```

$UB^{r_p} = \partial\Big(\bigcup_{\gamma\in\Gamma}(\gamma \oplus D(r_p))\Big)$. An example of UB^{r_p} is shown in Figure 9.9(a). In this figure, P is a point that represents its center. As in Section 8.4, we will use ray shooting to check the validity of a contact transit. Recall that ray shooting considers the problem of determining the first intersection between a ray (a directed line segment emanating from a point) and a collection of obstacles. In contrast to Section 8.4, the rays are now circular. To check the validity of a contact transit, we need to solve a circular ray shooting problem. This circular path is formed by the path of the center of P having radius $r_o + r_p$. We need to find the first intersection of the ray in an environment consisting of circular arcs and line segments (the features that UB^{r_p} consists of). If the ray intersects UB^{r_p} before the desired push position is reached, then no contact transit is possible. Note that both the clockwise and counterclockwise contact transits for P need to be checked. If P succeeds in reaching the desired push position, it can start pushing O in the direction of q_r. To check the validity of the path of O or to find the first obstacle O collides with, straight line ray shooting is used from q_n to q_r in the environment of UB^{r_o}, Figure 9.9(b). If a collision occurs, then we have found a compliant configuration c_c at the collision point. Note that on the path from q_n to q_r, P moves in the area swept by O and thus does not need to be collision checked. An exception occurs when after the contact transit an obstacle is present in one of the wedges between O and P. This can be detected by a straight line ray shoot among the features of UB^{r_p} from the center of P to the center of O, Figure 9.9(c).

Efficient algorithms to perform circular ray shooting are given by Koltun (2001) and Cheng et al. (2004). To check a hockey stick curve for collision for both P and O requires numerical analysis. This can be implemented using techniques from e.g. Burden and Faires (2001).

In line 4 of Algorithm 9.1 we need to find the configuration in the tree nearest to q_r. This nearest configuration is not necessarily a vertex of the tree, but can also be a config-



(a) Circular ray shooting among the features of UB^{rp} to collision check the contact transit.

(b) Straight line ray shooting among the features of UB^{ro} to collision check the path of O. The arrow shows the point of collision.

(c) A linear ray shoot in UB^{rp} to collision check the wedges.

Figure 9.9: Ray shooting in the contact spaces. The rays are the dotted arrows.

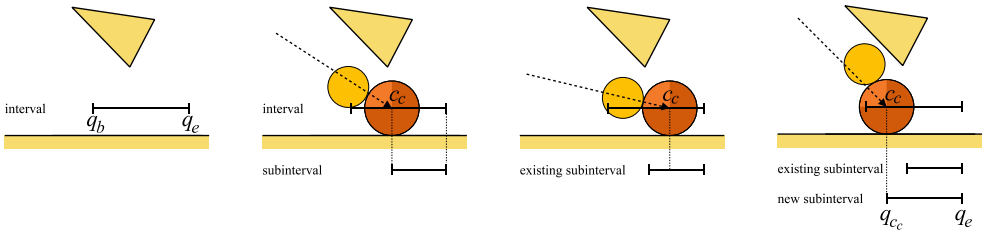
uration on an edge. Because it is not essential to find the nearest neighbor configuration exactly and because our edges do not solely consist of straight lines (but also hockey stick curves), we choose to use an approximate solution. Every edge in the tree is approximated by adding intermediate configurations along the edge that are used only for nearest neighbor searching (LaValle, 2006).

9.5 Compliant Exploration

If the local planner is not able to reach the random configuration q_r but instead hits an obstacle γ_i at configuration c_c, this collision point is used as a starting point for compliant exploration. Compliant exploration is a procedure to capture the topological structure of the set of compliant configurations (that can be reached through compliant motions only) starting at c_c. The results of this exploration are compliant configurations that are added to T. As a result T is a hybrid tree consisting of non-compliant configurations and continuous sets of compliant configurations. Starting at c_c, we initiate the compliant exploration in two directions: clockwise and counterclockwise. For one of these a preceding contact transit is necessary. After that, the two are similar.

To capture the topology of compliant space and to distinguish between explored and not yet explored parts of compliant space, intervals are used (see Definition 9.2.2). A vertex $v \in T$, compliant to obstacle γ_i stores the compliant position $q_{c_c} \in R^{ro}_{\gamma_i}$ of O along with the corresponding interval $I_{\gamma_i}(\gamma_j, \gamma_k)$. The interval implicitly defines the range of valid positions of P. Because of Lemma 9.2.1 we know that, once a configuration that belongs to an interval has been reached, P is guaranteed to be able to push O to the end of that

interval. Therefore, v does not represent a single position for O, but rather the continuous subset $[q_{c_c}, q_e(I_{\gamma_i}(\gamma_j,\gamma_k))]$. This subset is referred to as *subinterval* of v. If v is used in a planning query, P is free to choose any position between the bounding obstacles of the associated interval $I_{\gamma_i}(\gamma_j,\gamma_k)$.



(a) An interval $[q_b, q_e]$ associated with the bottom obstacle.

(b) The dotted arrow shows how O arrived on the interval. A vertex v is created, the subinterval that is associated with v is shown.

(c) O arrives in an already explored part of the interval, no new vertex is added to T.

(d) O arrives in a not previously explored part of the interval, the subinterval is extended.

Figure 9.10: Subinterval associated with a vertex of the tree.

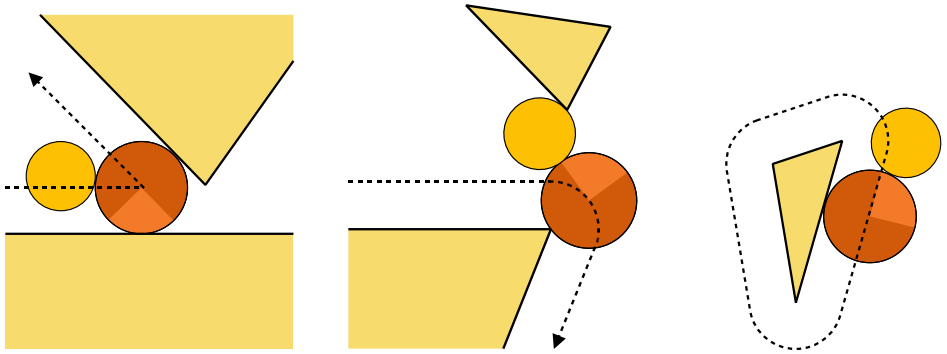
9.5.1 Creating Compliant Vertices

If the RRT algorithm generates a compliant configuration $c_c = (q_{c_c}, \alpha_{c_c})$ on obstacle γ_i, the corresponding interval $I_{\gamma_i}(\gamma_j,\gamma_k)$ is identified by determining γ_j and γ_k. An example of an interval is shown in Figure 9.10(a). Three cases can occur. The first case occurs when $I_{\gamma_i}(\gamma_j,\gamma_k)$ has not been encountered before. A new vertex v is created and added to T. The subinterval associated with v is $[q_{c_c}, q_e(I_{\gamma_i}(\gamma_j,\gamma_k))]$. This case is shown in Figure 9.10(b). In the second case, there is already a vertex v' in T associated with $I_{\gamma_i}(\gamma_j,\gamma_k)$, and q_{c_c} is inside the subinterval of v'. All compliant positions reachable from c_c were already reachable from v', thus c_c can be discarded and no new vertex is added to T, Figure 9.10(c). The third case occurs when there is already a vertex v' in T associated with $I_{\gamma_i}(\gamma_j,\gamma_k)$, but q_{c_c} is outside the subinterval of v', Figure 9.10(d). A new vertex v is added to T that is a copy of v' but with an associated subinterval $[q_{c_c}, q_e(I_{\gamma_i}(\gamma_j,\gamma_k))]$ which also contains the old interval. Vertex v' now becomes redundant and is removed from T.

If a new vertex v has been added to T, we check whether a subsequent interval can be reached at position $q_e(I_{\gamma_i}(\gamma_j,\gamma_k))$ (possibly after a contact transit). Since P and O always maintain contact, there is at most one such interval and it will be associated with the same compliant component. If another interval can be reached, the procedure is repeated.

There are a number of situations in which compliant exploration ends. As stated before, we can encounter an already explored part of an interval. In that case, compliant exploration ends because no new parts of compliant space will be discovered. Also, at the start of a new interval, the contact transit to a position in the push range may fail (Figure 9.11(a)). If P is not able to push O any further because it gets stuck between two obstacles

or is forced outside the push range, exploration ends, Figure 9.11(b). If O reaches the position where exploration started (i.e. it was pushed entirely around a compliant component) exploration also ends, Figure 9.11(c).



(a) O hits another obstacle and P is not able to reach the new push range.

(b) P is forced outside the push range by an obstacle.

(c) O has been pushed completely around a compliant component.

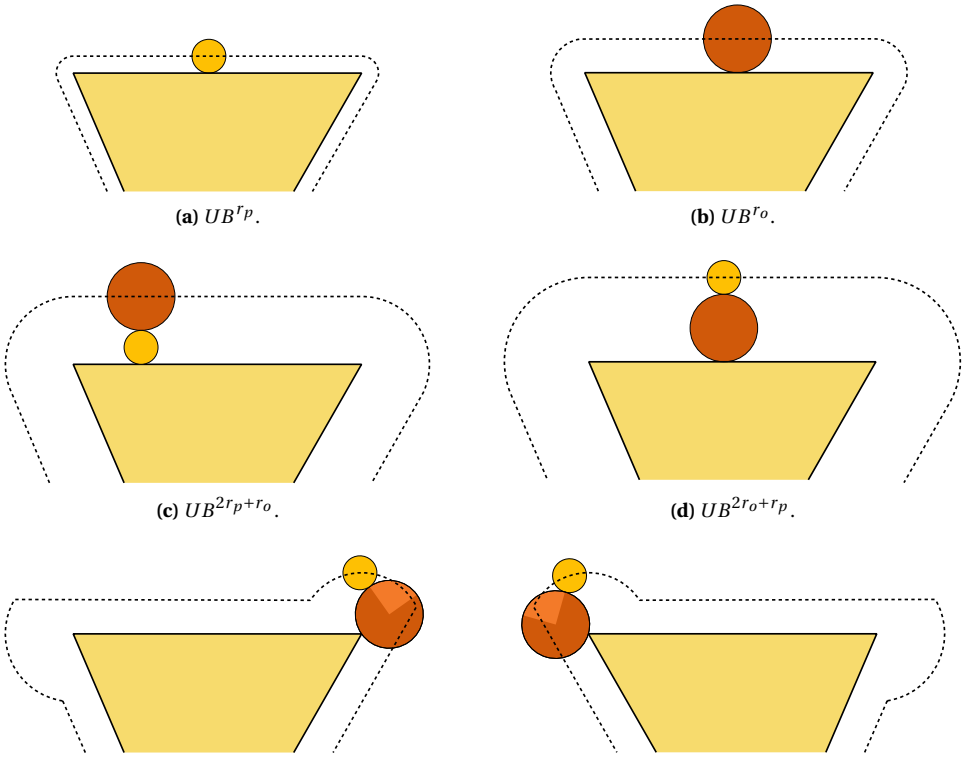
Figure 9.11: Compliant exploration ends.

9.5.2 Geometric Aspects

Compliant exploration is a purely geometric process. After a compliant configuration c_c has been found by the local planner, we first need to detect to which interval c_c belongs. An interval is identified by the bounding obstacles. To find the bounding obstacles at c_c, circular ray shooting among the features of UB^{r_p} is used. These rays represent the path of the center of P. If such a ray hits UB^{r_p} this indicates that P hits an obstacle. One ray is shot from the center of the current position of P in the direction of $\mathrm{PR}_b^+/\mathrm{PR}_b^-$ and another one in the direction of $\mathrm{PR}_e^+/\mathrm{PR}_e^-$. Recall that, if P is outside the push range, a preceding contact transit may be necessary.

After determining the interval $I_{\gamma_i}(\gamma_j,\gamma_k)$ to which c_c belongs, the object is certain to be able to reach $q_e(I_{\gamma_i}(\gamma_j,\gamma_k))$. To find $q_e(I_{\gamma_i}(\gamma_j,\gamma_k))$, a number of *events* can be distinguished that trigger the end of an interval. To determine these events, we use 6 structures (shown in Figure 9.12(a)…(f)):

- UB^{r_p}
- UB^{r_o}
- $UB^{2r_p+r_o}$
- $UB^{2r_o+r_p}$
- $LPCW$: the path of the lowest endpoint of the push range when moving in the clockwise direction around a compliant component



(a) UB^{r_p}.

(b) UB^{r_o}.

(c) $UB^{2r_p+r_o}$.

(d) $UB^{2r_o+r_p}$.

(e) The path of the lowest position of the push range clockwise $LPCW$.

(f) The path of the lowest position of the push range counterclockwise $LPCCW$.

Figure 9.12: The various structures necessary to determine the events.

- $LPCCW$: the path of the lowest endpoint of the push range when moving in the counterclockwise direction around a compliant component

We will now describe the events for compliant exploration in the clockwise direction using the structures. Compliant exploration in the counterclockwise direction is analogous.

1. Since an interval is associated with an obstacle, the interval ends by definition when O collides with a different obstacle. The collisions of O with another obstacle are easily determined by finding the vertices of UB^{r_o} (Figure 9.13(a)).

2. There is a change in the bounding obstacles. We subdivide the events that trigger these changes into 3 different types.

   (a) If an obstacle enters PR^+, an event occurs. An obstacle can only enter PR^+ in two ways, either via PR_b^+ or PR_e^+. If it enters via PR_b^+, the event can be found by

calculating intersections between $UB^{2r_o+r_p}$ and UB^{r_p}. Such an intersection is shown in Figure 9.13(b). If the obstacle enters via PR_e^+, the event can be found by finding the intersections between $LPCW$ and UB^{r_p} (Figure 9.13(c)).

(b) An obstacle can also leave PR^+. An obstacle γ_j leaves PR^+ if its distance to the center of O is such that P fits between O and γ_j. This event can be determined by finding intersections between UB^{r_o} and $UB^{2r_p+r_o}$, Figure 9.13(d).

(c) If O moves without an obstacle entering or leaving the push range, an event can also occur. These can also be determined using the intersections of 2a. In Figure 9.13(e) an obstacle forces P outside the push range and an interval ends. Figure 9.13(f) shows an example of the start of a new interval. Since P is able to reach PR_b^+, a new interval originates.

Using the events in the order in which they are encountered by O and P, it is easy to determine the chain of intervals reachable from a compliant start position in both the clockwise and counterclockwise exploration direction. If O can be pushed completely around a single obstacle without any event occurring, an interval also ends. Note that since intervals can overlap, an event denotes the end of *an* interval which is not necessarily the interval O and P are currently in. Storing the vertices of the structures in Kd-trees (Bentley, 1975) allows for efficient determination of the events.

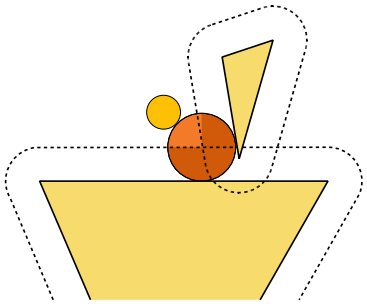
To find the events in case of non-zero friction μ_1 between O and Γ, only $UB^{2r_o+r_p}$ needs to be adapted accordingly. Instead of determining the intervals after a compliant configuration has been found, preprocessing the events and intervals and storing them allows for even faster compliant exploration.

Figure 9.14 shows a complete example. At all non-compliant vertices and some compliant vertices the object and pusher are shown. The vertices that represent the compliant paths in the clockwise direction are omitted for clarity.

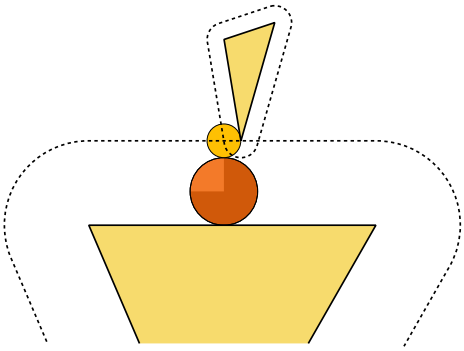
9.6 Probabilistic Completeness

The basic RRT algorithm is known to be probabilistically complete (LaValle and Kuffner, 2001). If an RRT is used for pushing through non-compliant space, the position of P at a vertex is dictated by the direction of the connection of the vertex to the tree. Stated differently, if O is pushed to a vertex, then the position of P at that vertex is dependent on the direction O and P came from. Because of the random nature of the RRT, every potential configuration will eventually be approximated by a configuration at most a distance ϵ away from this configuration.

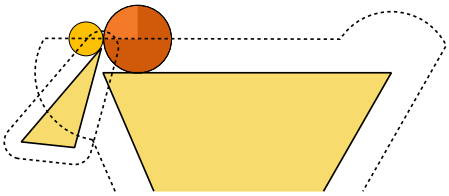
Compliant exploration is complete by nature. We need to show that every reachable compliant configuration will eventually be found. Because the RRT algorithm is probabilistically complete, every pushable path through non-compliant space will eventually be approximated by a path that is at most a distance ϵ away from this path. This implies that every possible compliant configuration reachable from non-compliant space will eventually be approximated by a configuration that is at most a distance ϵ away from



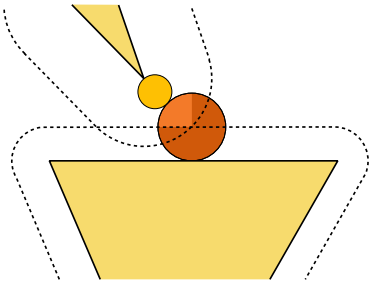
(a) The collision of O with another obstacle.



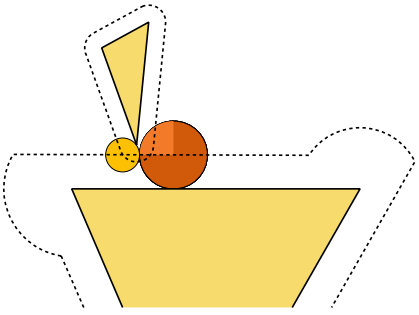
(b) An obstacle entering PR^+ via PR^+_b.



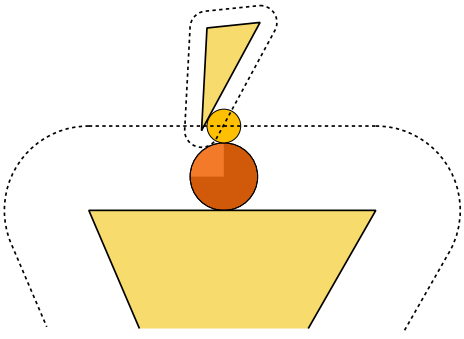
(c) An obstacle entering PR^+ via PR^+_e



(d) An obstacle leaving PR^+.



(e) P is forced outside PR^+; an interval ends.



(f) P is able to reach PR^+_b; a new interval starts.

Figure 9.13: Overview of the various events.

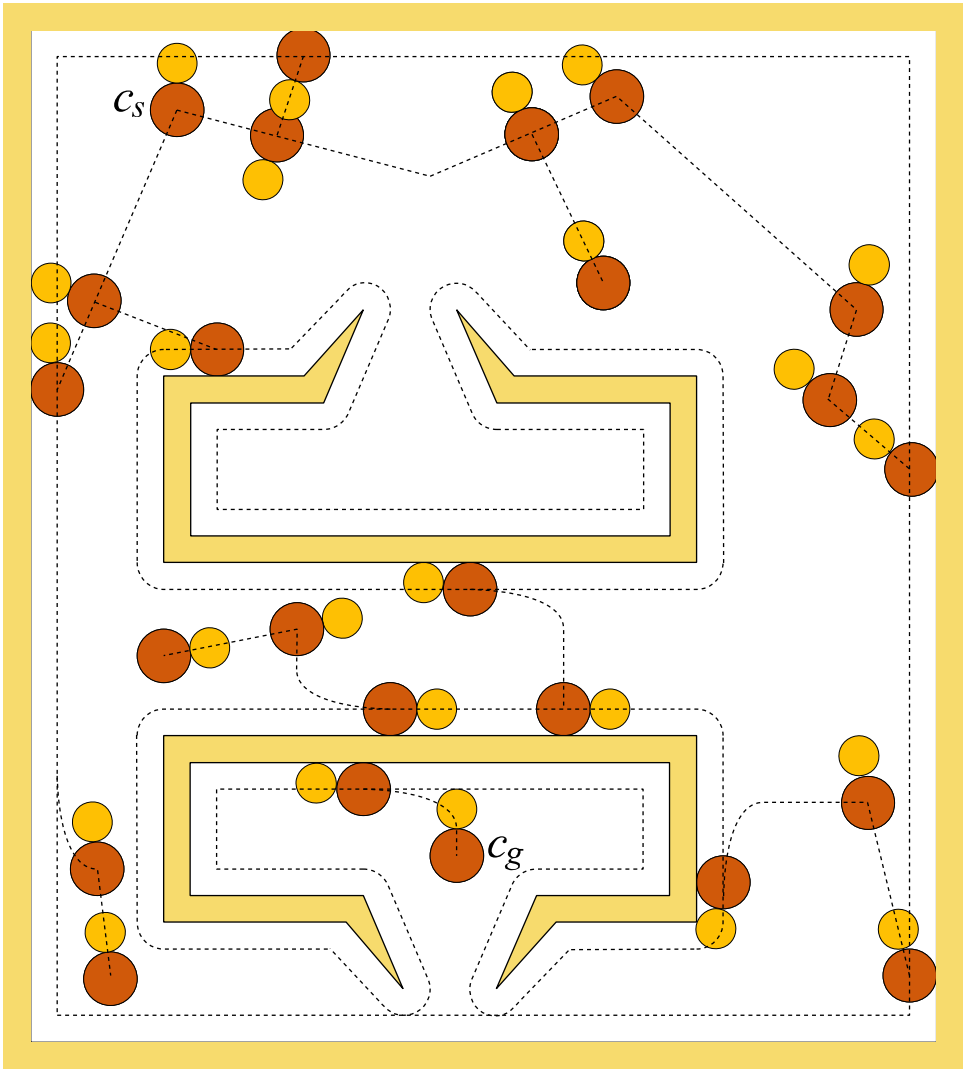


Figure 9.14: Complete example in which the pusher and object are shown at every non-compliant vertex of the tree. Also some counterclockwise compliant vertices are shown (clockwise vertices are omitted). The dotted line shows the complete tree.

this configuration. Since the RRT uses random sampling, every compliant configuration can be selected as nearest neighbor and thus be used as a starting point for a hockey stick push.

9.7 Experiments

We have implemented our algorithm in Visual C^{++}. Obstacles were preprocessed to create the list of events, since these are equal for every query. In this preprocessing phase, we have first created the structures (UB^{r_o}, UB^{r_p}, $UB^{2r_p+r_o}$, $UB^{2r_o+r_p}$, $LPCW$ and $LPCCW$). These structures consist of circular arcs and straight lines. Their intersections are used to create the list of events associated with every obstacle, as described in Section 9.5. To simplify the implementation, we have not used a Kd-tree to store the intersections, but rather identified them in a brute force manner. Using the events, the list of intervals for every obstacle was created. During the execution of a query, this list was used for the compliant exploration procedure if a compliant configuration was found. Hockey stick curves were checked for collisions by taking small steps ($1/200^{th}$ of the diagonal of the bounding box) along the curve. Finally, for the distance of the intermediate points of the RRT we have used a value of $1/80^{th}$ of the diagonal of the bounding box. These values were established experimentally and were adequate in all our experiments.

We have conducted several experiments on a Pentium 2.4GHz system with 1GB of memory. The results are compared with a basic RRT algorithm (RRT *only*), that does not use compliance. This algorithm simply follows the description given in Section 9.3.1, with the addition that contact transits were incorporated to ensure that connections were valid. The results are shown in Tables 9.1 and 9.2. All results were averaged over 10 runs.

First, we looked at the (simple) examples of Figure 7.1. Recall that no path can be found in Figure 7.1(a) without compliance. With compliance a solution can easily be found. After preprocessing (which takes about 0.02s) queries can be executed quickly in 0.0025s on average.

In Figure 7.1(b), the RRT only algorithm could not find a valid path even after the creation of hundreds of configurations. This is due to the fact that a very specific sequence of random samples is necessary. With compliance, it is easy to reach the goal once a compliant configuration has been found. Preprocessing takes about 0.007s after which queries can be executed in 0.004s on average.

In the experiment with the environment shown in Figure 7.1(c), the RRT only algorithm is able to find a path. Because the object has to pass a narrow passage, it takes more time to reach the destination then when using compliance. After preprocessing which takes about 0.01s, our algorithm is able to find a path in 0.0005s on average as opposed to 0.01s for the RRT only algorithm. The RRT only algorithm needs many more vertices to find a solution 14 versus 141.

Finally, we have conducted experiments with the environment shown in Figure 9.15 consisting of 19 obstacle line segments. Preprocessing took 0.08s on average and queries were executed in 0.02s on average. Using the RRT only algorithm, a query took 0.27s. No-

tice also the difference in the number of vertices: 88 for the compliant algorithm versus 506 for the RRT only algorithm.

	Figure 7.1(a)		**Figure 7.1(b)**	
	Compliance	RRT only	Compliance	RRT only
Prepprocessing time (s)	0.02	-	0.007	-
Query time (s)	0.0025	-	0.004	-
Number of vertices	40	-	30	-

Table 9.1: Results of the experiments.

	Figure 7.1(c)		**Figure 9.15**	
	Compliance	RRT only	Compliance	RRT only
Preprocessing time (s)	0.01	0	0.08	0
Query time (s)	0.0005	0.01	0.02	0.27
Number of vertices	14	141	88	506

Table 9.2: Results of the experiments.

The number of mutual intersections of the various structures increases quadratically with the number of obstacles in the worst case. However, in practical environments we have encountered a linear increase in the number of intersections and thus, if a Kd-tree is used, a linear increase in preprocessing time.

9.8 Concluding Remarks

In this chapter we have introduced a novel manipulation planning algorithm in which pushing is combined with compliant motions. We have used the simplified problem of two disks as a first step to explore the potential of this combination. Building on the results of Chapter 8, the resulting manipulation plans use compliance to extend the range of problems for which a solution can be created. Also in environments or in subsets of environments in which the density of the obstacles is high, compliance helps in lowering the complexity of the solution and to pass narrow passages.

In the previous chapters we have seen that complete methods are infeasible because of the complexity of the non-compliant subspaces of the solution space. In this chapter we used the RRT algorithm that creates paths that act as bridges between the compliant parts of the path. RRTs also provide a balance between the number of compliant and non-compliant path segments that reflects the importance of compliant configurations to the solution of the problem. If the environment is preprocessed then it is easy to check to

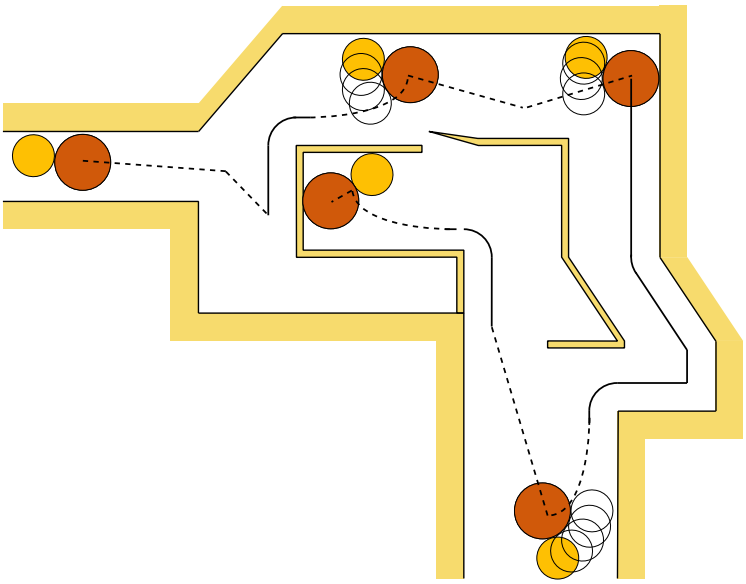


Figure 9.15: Experimentation environment. At the left the start configuration is shown, in the center the goal configuration. An example of a final path is shown. The dotted lines represent non-compliant path segments, the solid lines represent compliant path segments. At some points the motion of the pusher is shown.

which interval a configuration belongs and thus what part of the compliant space is reachable from it using geometric operations. A useful addition can be the support of multiple trees as described by Strandberg (2004). This may be helpful in difficult parts of the environment where compliance does not work.

Compliance is very useful in compensating for uncertainty. Because compliant motions are not dependent on a single push position but rather have a push range, the pusher is free to choose a push position within this range without affecting the motion of the object. Thus a small deviation in the push position does not affect the motion of the object. For the non-compliant motions, the algorithm chooses a random configuration to which a path is tried. If the object is not pushed precisely in the direction of this random configuration but rather has a small deviation, this does not affect the algorithm.

We have not allowed the pusher to lose contact with the object. An interesting topic for future research is to extend our algorithm by allowing these non-contact transits. This can for example be done by using existing path planning techniques in which the pusher moves and the object is considered an obstacle. Configurations in which a contact transit is not possible (because of collision of the pusher), are candidates for such non-contact transits.

CHAPTER 10

Conclusions

This thesis has addressed path planning in changeable environments. In contrast to traditional path planning that deals with static environments, in changeable environments objects are allowed to change their configurations. In many cases, path planning algorithms must facilitate quick answers to queries in order to be useful. For example, an opponent in a training simulation needs to respond to the actions of the user without any significant delay. To achieve such performance, path planning methods usually use a preprocessing phase in which the environment is explored. As much computation time as possible is moved to this preprocessing phase such that at query time only little time is needed to solve an actual path planning query. This approach has led to many successful methods that are applicable to a broad range of problems.

The methods described in this thesis can be seen as a next step in the evolution of path planning algorithms using preprocessing. Although in the last decades no real general-purpose path planning algorithm for static environments has been developed, many successful planners have been developed that work on specific classes of problems. Often these are variations on the standard PRM method. A broad range of planners is available nowadays all having their own properties and application areas. Attention therefore slowly shifts to other aspects of path planning problems (e.g. path quality) and to more difficult types of problems including planning for multiple robots or groups of entities, planning for a camera following a character and also planning in non-static environments which is the focus of this thesis.

Because of the nature of preprocessing it is difficult to cope with unanticipated changes that occur in the environment in a later stage. A straightforward approach is to use the information gathered during preprocessing for the planning and if a problem is detected (e.g. an obstacle has changed position and blocks the path of the robot) then it is attempted to solve this problem locally. This type of solution is usually computationally

expensive and may fail if no local solution exists.

In Chapter 2 and in Part I we have described a solution to problems in which the obstacles are manipulated by external factors (i.e. not by the robot). The robot then has to deal with these changes when navigating. Part II has addressed the problem of obstacles blocking the path of the robot. The robot then needs to manipulate these obstacles to clear its path. Finally, in Part III we have studied the problem of pushing an object through an environment by the robot. We will draw separate conclusions for each of these scenarios.

10.1 Creating Cycles in the Roadmap

Roadmap methods were developed to be able to re-use computations to provide for a fast query phase. Until recent years however, the quality of the paths in the roadmap has not had much attention. Most research was aimed at speeding up the roadmap creation process and improving the coverage (i.e. each configuration $c \in C_{\text{free}}$ can be connected to G using the local planner) of the roadmap. In this respect allowing cycles in the roadmap is not beneficial. A cycle in the roadmap will never contribute to the coverage but will rather slow down the roadmap creation process and therefore in most existing solutions the roadmap is a forest. As a consequence there is at most one path in the roadmap between two vertices. In terms of path quality this means that the resulting query path may take a long detour while a shorter path is available. If the roadmap is used for non-static environments then having a forest has another disadvantage: roadmaps without cycles do not provide for an alternative route if a path is blocked by an obstacle. If only a few obstacles change their position or if some new obstacles are added to the environment, this may render the roadmap useless. A straightforward solution is to repair the roadmap locally if a problem is encountered while executing a query, but this affects the speed of the query phase.

In Chapter 2 we have introduced a new method that allows a limited number of cycles in the roadmap such that the robustness against moving or adding obstacles increases by a large amount. Although simple methods have previously been described to add cycles (the classic proof of probabilistic completeness of PRM even requires adding all potential cycles), checking the usefulness of a cycle before considering it for addition to the roadmap is new. Useful in this respect means that the cycle has a high probability of contributing to proving an alternative route in the roadmap. In addition the check should be fast such that the running time of the roadmap generation does not increase significantly. We have presented a method that adds useful cycles based on the current graph distance between vertices. If the cycle sufficiently improves this graph distance it is added to the roadmap. This approach naturally adapts to the local sampling density which is often focused toward difficult areas of the environment. In our experiments we have shown that our approach is very successful. We have conducted experiments that show the effectiveness of checking the usefulness of a cycle before it is considered for addition to the roadmap. Because the method is relatively cheap and does not need additional collision checks (which may be expensive depending on the application) it serves as a basis for path planning problems in changeable environments. Having useful cycles in the roadmap may in many cases

prevent the need of executing more sophisticated algorithms to cope with changes in the environment and therefore speeds up the process of creating a path.

A novel property of roadmaps containing useful cycles is that there is a bound on the length of a path in the roadmap. Although the probabilistic completeness proof of PRMs implies that the length of the shortest path in the roadmap approaches the shortest path, this is dependent on the property that every vertex is connected to all other vertices if the local planner finds such a path. In practical implementations of PRM methods this is never the case and thus no guarantee can be given on the length of the paths in the roadmap. We have shown that in our method such a bound exists and has a direct correlation with our criterion that decides when to add a cycle and as such makes it suitable to be used in static environments as well as to improve path quality.

In the future more sophisticated methods may be developed to determine the usefulness of a cycle. Regardless of the method used, there will be a trade-off between the number of changes in the environment that the method is robust against and the increase in roadmap size. If the difference between the environment that was used in the preprocessing phase and the one in the query phase gets too large (e.g. if too many obstacles change their configuration) any method will have difficulty coping with these changes and will need a very dense (and thus computationally expensive) roadmap in the preprocessing phase.

If the roadmap consists of directed instead of undirected edges then determining whether an edge is useful is more difficult since it is harder to detect whether an edge is redundant. A thorough description of this problem and a solution is described by Švestka (1997). Combining this solution with useful edges would make an interesting extension to the algorithm.

10.2 Changing Environments

In part I we have proposed an algorithm that deals with the class of problems in which obstacles can change their configuration between the time the roadmap was created and the query. Examples of such obstacles are doors, chairs and boxes.

Until recently, it has been beneficial to use scripting for the path planning of robots in games and training simulations (in this context often referred to as non-playing characters). Using scripting, the potential set of motions of the characters was known beforehand and conflicts could be accounted for during the development of the application. As virtual worlds get larger and more complex (e.g. because the number of characters increases), doing the path planning manually gets harder and more tedious. Also users of these environments are getting used to more sophisticated behavior of the virtual characters and thus also the path planning needs to be upgraded to more convincing techniques. Another reason to switch from manual path planning to automatic planning is that it is to be expected that in the near future virtual worlds will be generated on the fly. The reason for this is that a user can experience a different environment every time he or she runs the application. For example in disaster training simulations this will prevent the user to get to know the environment too well thus allowing for a more realistic training. Moreover it will

allow the developers to provide the user with much larger environments, extending the life-cycle of the application. A side effect of this automatic world generation is that scripting will no longer be a solution for path planning. Another evolution (of which the first signs are already visible in some applications) is that objects will no longer be stationary but users will be allowed to manipulate them as is the case in the real world (an example is shown as Figure 10.1). The above developments require automatic path planning techniques that can be run after the world is generated (a preprocessing step) that allow for the fast creation of a path at runtime. The PRM method seems a promising technique to provide for this type of planning. Standard PRM implementations are not suited to deal with obstacles that change their position since they invalidate the edges of the roadmap.

Figure 10.1: Moving an object using the "gravity gun" in the game Half Life (Courtesy of Valve).

We have observed that in many environments the obstacles are confined to small areas. In Part I we have proposed a method that exploits this property. The method assumes that the set of potential placements of obstacles are known beforehand (in the case of a door this is trivially so, but a chair can be confined to a room) and incorporates solutions to obstacle placement changes during the preprocessing phase by adding necessary edges to the roadmap. Chapter 3 starts by identifying the problem and describing how to create a theoretical optimal solution. Next, it continues by stating a practical approach by subdividing the potential sets of placements of the obstacles into so-called chunks. Some heuristics were presented to speed up the process and finally a proof of concept was given in the experimentation section.

Although the results are very promising, the speed of the above approach suffers when the number of obstacles that can change position increases. A solution to this was presented in Chapter 4. We have observed that the decision of adding a necessary edge resembles the well-known problem of satisfiability in Boolean logic. We have described how to convert the necessary edge problem to a satisfiability problem in Boolean logic. While the satisfiability problem is NP-complete, very powerful heuristics exist that are capable of solving difficult instances of the problem quickly. Combined with certain other improvements the method showed to be efficient in robustness, speed and roadmap size.

An interesting challenge for the future is the extension of the method to work in dynamic environments in which the obstacles continuously change their configurations. By dynamic we mean that while the obstacles are still confined to a certain area, they are allowed to change their configuration during the execution of a query. The dynamic obstacles will have to be monitored (using e.g. sensors) and the path of the robot needs to adapt to the observed motions of the dynamic obstacles. If certain knowledge about the motions of the dynamic obstacles is known in advance this may be incorporated in the planning process. In "true" dynamic environments however nothing is known about these motions by definition and if the motions are independent of the motions of the robot then local solutions are necessary.

The automatic generation of chunks in more difficult situations is also an interesting topic for future research. If, for example, information is available about the probability distribution for the set of placements of an obstacle, this information can be used to generate the chunks. In areas having a low probability of containing the obstacle, the chunks may be larger than in areas where this probability is high.

10.3 Movable Obstacles

Another step in the evolution toward realistic virtual environments is the ability of robots to manipulate obstacles that block their path. Imagine, for example a simulation in which a firefighter commander is trained. The commander gives his (virtual) firefighters higher level commands (e.g. "walk around the building and enter it at the back"). For a realistic training, the firefighters should be able to move away obstacles that block their paths in order to, for example, clear the door.

In Part II we have considered the relatively unaddressed problem of path planning amidst movable obstacles. Here the environment contains obstacles which need to be moved by the robot to clear its path. Although this is a natural extension to the path planning problem, planning amidst movable obstacles has not had much attention. This may be due to the fact that even for simple instances the problem is NP-hard. Moreover, before being able to solve path planning amidst movable obstacles, first an adequate solution for the standard path planning problem is required for the navigation of the robot. For this, we have used the PRM variant of Chapter 2.

Having an adequate solution for the robot navigation has enabled us to focus on the movable obstacles. Fundamental questions are which obstacles to manipulate, how to manipulate them and in which order to manipulate them. Since solving this problem would also solve 3-SAT, this problem is NP-hard. In contrast to the decision of adding necessary edges in Part I however, we have observed that the major part of the space of potential actions does not contribute to a solution and therefore we have been able to create powerful heuristics to guide the search for a solution. We created a novel method that represents the action space as nodes in a tree. Leaves are new actions that are selected based on the success of their parent nodes. We have shown that even difficult problems could be solved in reasonable times.

Our method can serve as a basic planning algorithm for robots as a next step in making their behavior more realistic. To achieve this, some further developments are necessary. First, to be able to allow for interactive performance the speed of the planning process needs to be increased. An interesting idea in this respect may be to subdivide the planning process. Planning the whole path in advance may be unwanted because other robots (controlled by the computer or a user) can move obstacles as well. The further a path is planned ahead, the larger the probability that it will be invalid due to actions of other robots. Subdividing the planning process into smaller portions may provide a solution to this. Another interesting development is the cooperation of multiple robots to perform a task, in this case moving obstacles. To accomplish such such joint tasks, the motions of the robots need to be coordinated. An overview of different approaches to coordinate the motions of multiple robots can be found in the introduction of the paper by Van den Berg and Overmars (2005b).

10.4 Pushing using Compliance

In Part III we have addressed the problem of pushing a disk in a polygonal environment. Pushing an object by a robot is often easier or more applicable than pulling since it does not involve grasping the object. A robot arm can push an object using a single finger while pulling involves more complicated behavior. Unfortunately in addition to the usual sensor errors, pushing is also sensitive to another type of uncertainty; if the object's center of mass is not exactly known then pushing an object leads to erratic behavior leading to unstable pushes. Among other causes, these errors have been responsible for a long standing gap between the theoreticians and people working from a more practical point of view. In theory many algorithms exist to solve all kinds of manipulation planning problems but when applied to a practical situation often their benefit is small because of different types of deviations encountered. Often these problems are caused by small sensor errors that can lead to large deviations in, for example, the actual future position of the robot. Instead of assuming better sensors will eventually solve these problems (as is highly questionable), we have looked at solutions that are robust against sensor errors and therefore are more suited to be used in practical problems.

A promising solution to the above problems is to combine pushing with compliant motions. Compliant motions compensate for uncertainty. In addition, the center of mass and friction constants do not need to be known exactly as a range of push positions lead to the same motion of the object. To show the potential of the approach we have proposed a planner in which a robot is capable of pushing a disk shaped object through a polygonal environment. Compliant motions are allowed in the planning process by letting the object slide along the boundaries of the environment. After introducing the problem and stating some properties and definitions in Chapter 7 we have explored the potential of the approach in Chapter 8. We have presented a first planner capable of creating a path for the pusher given the path of the object. The object path was allowed to contain compliant parts. We have shown that such push plans can be created effectively and concluded that the results are promising for creating planners that use compliance. In Chapter 9 we

have proposed such a planner. Given a start and goal configuration the planner creates a manipulation plan such that the object is pushed to the goal. The manipulation plan consists of both compliant and non-compliant path segments. For the non-compliant parts of the manipulation plan we adapted the RRT method such that it is capable of handling pushing motions. The non-compliant path segments act as bridges between the compliant path segments. Deviations from the intended path because of sensor errors in these non-compliant segments are allowed as they are compensated for in the next compliant segment.

With our planner we have obtained some very interesting results. We have shown that our method outperforms a planner that does not use compliance by a large factor. Also the range of problems for which our planner finds a manipulation plan is larger than for a planner that does not use compliance.

The results of this part can be used as a framework for further research. A first step in this research should be to extend the method to pushers and objects that are not disk shaped. The advantage of having a push range instead of only one push position still holds if the object is polygonal. Another topic of future research is to allow the pusher to lose contact with the object. This involves discretizing the positions in which it is beneficial to lose contact with the object. Replacing the pusher by a robot arm, pushing the object using a finger is also an interesting extension to the problem domain. Depending on the robot arm used, this may involve taking the limitations of the motions of the finger into account.

Path planning in changeable environments is a natural extension to path planning in static environments. It is an important research area which is getting even more important with the development of more intelligent robots and the increasing demand for realistic behavior of virtual characters. In this thesis we have looked at three scenarios that consider path planning in changeable environments for which we have developed novel algorithms. While successful in many applications, they leave a lot of room for further developments. In particular the algorithms presented in Part III should be regarded as the first steps of creating solutions for these scenarios. Our algorithms have a lot of potential for extensions while also many open problems remain and provide interesting topics for future research.

Bibliography

P.K. Agarwal, J.-C. Latombe, R. Motwani, and P. Raghavan. Nonholonomic path planning for pushing a disk among obstacles. In *IEEE International Conference on Robotics and Automation*, 1997.

R. Alami, J.P. Laumond, and T. Siméon. Two manipulation planning algorithms. In *International Workshop on the Algorithmic Foundations of Robotics*, pages 109–125, 1994.

N.M. Amato, O. Bayazit, L. Dale, C. Jones, and D. Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. In *IEEE International Conference on Robotics and Automation*, pages 630–637, 1998a.

N.M. Amato, O. Bayazit, L. Dale, C. Jones, and D. Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces. In *International Workshop on the Algorithmic Foundations of Robotics*, pages 155–168, 1998b.

H. Arai and O. Khatib. Experiments with dynamic skills. In *1994 Japan-USA Symposium on Flexible Automation*, pages 81–84, 1994.

F. Avnaim, J.D. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles. In *Proceedings of the Workshop on Geometry and Robotics*, pages 67–86. Springer-Verlag, 1989.

I. L. Balaban. An optimal algorithm for finding segment intersections. In *11th Annual ACM Symposium Computational Geometry*, pages 211–219, 1995.

J. Barraquand, B. Langlois, and J.-C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22:224–241, 1992.

J. Barraquand and J.-C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *IEEE International Conference on Robotics and Automation*, pages 1712–1717, 1990.

O. Ben-Shahar and E. Rivlin. Practical pushing planning for rearrangement tasks. In *IEEE International Conference on Robotics and Automation*, pages 549–565, 1998.

J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computing*, pages 643–647, 1979.

J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

J. P. van den Berg, D. Nieuwenhuisen, L. Jaillet, and M.H. Overmars. Creating robust roadmaps for motion planning in changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2415–2421, 2005.

J.P. van den Berg and Mark H. Overmars. Prioritized motion planning for multiple robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2217–2222, 2005b.

J.P. van den Berg and M.H. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics and Automation*, 21:885–897, 2005.

M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry (2nd ed.)*, pages 272–276. Springer-Verlag, Berlin, Germany, 2000.

G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003. ISBN 155860801X.

A. Biere and C.P. Gomes, editors. *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-37206-7.

R. Bohlin and L.E. Kavraki. Path planning using lazy PRM. In *IEEE International Conference on Robotics and Automation*, pages 521–528, 2000.

G. Boole. The calculus of logic. *Cambridge and Dublin Mathematical Journal*, III (1848): 183–198, 1848.

V. Boor, M.H. Overmars, and A.F. van der Stappen. The Gaussian sampling strategy for probabilistic roadmap planners. In *IEEE International Conference on Robotics and Automation*, pages 1018–1023, 1999.

M. Branicky, S.M. LaValle, K. Olson, and L. Yang. Quasi-randomized path planning. In *IEEE International Conference on Robotics and Automation*, pages 1481–1487, 2001.

A.J. Briggs. An efficient algorithm for one-step planar compliant motion planning with uncertainty. *Algorithmica*, 8(3):195–208, 1992.

R.A. Brooks and T. Lozano-Pérez. A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:224–233, 1985.

R.L. Burden and J.D. Faires. *Numerical Analysis 7th ed.* Wadsworth Group, 2001.

J.F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, 1988.

P. C. Chen and Y. K. Hwang. Practical path planning among movable obstacles. In *IEEE International Conference on Robotics and Automation*, pages 444–449, 1991.

S.-W. Cheng, O. Cheong, H. Everett, and R.W. van Oostrum. Hierarchical decompositions and circular ray shooting in simple polygons. In *Discrete and Computational Geometry*, volume 32, pages 401–415, 2004.

H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, first edition, 2005.

S. Cook. The complexity of theorem proving procedures. In *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

L. Dale. *Optimization techniques for probabilistic roadmaps*. PhD thesis, Texas A&M University, 2000.

C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *IEEE Symposium on Foundations of Computer Science*, pages 260–267, 2001.

C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *ACM Symposium on Theory of Computing*, pages 159–166, 2003.

E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

B.R. Donald. The complexity of planar compliant motion planning under uncertainty. In *ACM Symposium on Computational Geometry*, pages 309–318, 1988.

N. Eén and N. Sörensson. Minisat 1.14, 2005.

M. Erdmann and T. Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

B. Faverjon. Object level programming of industrial robots. In *IEEE International Conference on Robotics and Automation*, pages 1406–1412, 1986.

M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.

R. Geraerts and M.H. Overmars. A comparative study of probabilistic roadmap planners. In *International Workshop on the Algorithmic Foundations of Robotics*, pages 43–57, 2002.

R. Geraerts and M.H. Overmars. Sampling techniques for probabilistic roadmap planners. In *Conference on Intelligent Autonomous Systems*, pages 600–609, 2004.

R. Geraerts and M.H. Overmars. Creating small roadmaps for solving motion planning problems. In *IEEE International Conference on Methods and Models in Automation and Robotics*, pages 531–536, 2005.

R. Geraerts and M.H. Overmars. Creating high-quality roadmaps for motion planning in virtual environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4355–4361, 2006.

K.Y. Goldberg. Orienting polygonal parts without sensors. *Algorithmica*, 10(2-4):210–225, 1993.

P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4(2), pages 100–107, 1968.

D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *IEEE International Conference on Robotics and Automation*, pages 4420–4426, 2003.

D. Hsu, L.E. Kavraki, J.-C. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. In P.K. Agarwal, L.E. Kavraki, and M.T. Mason, editors, *International Workshop on the Algorithmic Foundations of Robotics*, pages 141–153, 1999.

L. Jaillet and T. Siméon. A prm-based motion planner for dynamically changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1606–1611, 2004.

L. Jaillet and T. Siméon. Path deformation roadmaps. In *International Workshop on the Algorithmic Foundations of Robotics*, 2006.

S. Kambhampati and L.S. Davis. Multiresolution path planning for mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 135–145, 1986.

A. Kamphuis and M.H. Overmars. Finding paths for coherent groups using clearance. In *IEEE International Conference on Robotics and Automation*, pages 3815–3822, 2004.

L.E. Kavraki, P. Švestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580, 1996.

O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5:90–98, 1986.

P. Khosla and R. Volpe. Superquadratic artificial potentials for obstacle avoidance and approach. In *IEEE International Conference on Robotics and Automation*, pages 1778–1784, 1988.

D.E. Koditschek. Exact robot navigation by means of potential functions: Some topological considerations. In *IEEE International Conference on Robotics and Automation*, pages 1–6, 1987.

S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *IEEE International Conference on Robotics and Automation*, pages 968–975, 2002.

V. Koltun. Segment intersection searching problems in general settings. In *Seventeenth annual symposium on Computational geometry*, pages 197–206, 2001.

B.H. Krogh. A generalized potential field approach to obstacle avoidance control. In *International Robotics Research Conference, Bethlehem, Pennsylvania*, 1984.

J.-C. Latombe. *Robot Motion Planning*. Kluwer, 1991.

S.M. LaValle. *Planning Algorithms*. Cambridge University Press (or http://msl.cs.uiuc.edu/planning/), 2006.

S.M. LaValle and J.J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *B. R. Donald, K. M. Lynch, and D. Rus, editors, Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.

D.T. Lee. *Proximity and Reachability in the Plane*. PhD thesis, Coordinated Science Laboratory, University of Illinois, Urbana, 1978.

P. Leven and S. Hutchinson. A framework for real-time path planning in changing environments. *International Journal of Robotics Research*, 21(12):999–1030, 2002.

T. Lozano-Peréz, M.T. Mason, and R.H. Taylor. Automatic synthesis of fine-motion strategies for robots. In *IEEE Transactions on Computers*, pages 108–120, 1983.

K.M. Lynch. The mechanics of fine manipulation by pushing. In *IEEE International Conference on Robotics and Automation*, pages 2269–2276, 1992.

K.M. Lynch and M. T. Mason. Stable pushing: Mechanics, controllability and planning. *International Journal of Robotics Research*, 15(6):533–556, 1996.

M.T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, August 2001.

M.T. Mason. Mechanics and planning of manipulator pushing operations. *International Journal of Robotics Research*, 5(3):53–71, 1986.

M.T. Mason and K. Lynch. Dynamic manipulation. In *IEEE/RSJ International Workshop on Intelligent Robots and Systems*, pages 152–159, 1993.

D. Nieuwenhuisen, A. Kamphuis, and M.H. Overmars. High quality navigation in computer games. *Science of Computer Programming: Special issue on Game Programming*, 2007.

D. Nieuwenhuisen and M.H. Overmars. Useful cycles in probabilistic roadmap graphs. In *IEEE International Conference on Robotics and Automation*, pages 446–452, 2004.

D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Path planning for pushing a disk using compliance. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4061–4067, 2005.

D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. An effective framework for path planning amidst movable obstacles. In *International Workshop on the Algorithmic Foundations of Robotics*, to appear 2006.

D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Pushing using compliance. In *IEEE International Conference on Robotics and Automation*, pages 2010–2016, 2006.

D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Pushing a disk using compliance. *IEEE Transactions on Robotics and Automation*, to appear 2007.

C. Ó'Dúnlaing and C.K. Yap. A retraction method for planning the motion of a disc. *Journal of Algorithms*, 6:104–111, 1982.

M.H. Overmars and P. Švestka. A probabilistic learning approach to motion planning. Technical Report UU-CS-1994-03, Utrecht University, department of Computer Science, 1994.

F. Reuleaux. *The Kinematics of Machinery*. Macmilly and Company (Republished by Dover in 1963), 1876.

G. Sánchez and J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *International Symposium of Robotics Research*, pages 403–418, 2001.

E. Schmitzberger, J.-L. Bouchet, M. Dufaut, W. Didier, and R. Husson. Capture of homotopy classes with probabilistic road map. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2317–2322, 2002.

J.T. Schwartz and M. Sharir. On the piano movers' problem: I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983a.

J.T. Schwartz and M. Sharir. On the piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, 4:298–351, 1983b.

J.T. Schwartz and M. Sharir. On the piano movers' problem: III. Coordinating the motion of several independent bodies: the special case of circular bodies moving amidst polygonal barriers. *Int. Journal of Robotics Research*, 2(3):46–75, 1983c.

T. Siméon, J.-P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. *Advanced Robotics*, 14(6):477–493, 2000.

G. Song and N.M. Amato. Using motion planning to study protein folding pathways. *Journal of Computational Biology*, 9(2):149–168, 2001.

A.F. van der Stappen, M.H. Overmars, M. de Berg, and J. Vleugels. Motion planning in environments with low obstacle density. *Discrete & Computational Geometry*, 20:561–578, 1998.

A. Stenz. The focused D* algorithm for real-time replanning. In *International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

M. Stilman and J. Kuffner. Navigation among movable obstacles: Real-time reasoning in complex environments. *Journal of Humanoid Robotics*, 2(4):479–504, 2005.

M. Strandberg. Augmenting rrt-planners with local trees. In *IEEE International Conference on Robotics and Automation*, pages 3258–3262, 2004.

S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, Department of Electrical Engineering, California Institute of Technology, 1977.

P. Švestka. *Robot Motion Planning Using Probabilistic Road Maps*. PhD thesis, Utrecht University, 1997.

G. Wilfong. Motion planning in the presence of movable obstacles. In *4th Annual Symposium on Computational Geometry*, pages 279–288, 1988.

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 *IEEE International Conference on Robotics and Automation*, pages 1024–1031, 1999.

C.K. Yap. An $O(n \log n)$ algorithm for the voronoi diagram of a set of simple curve segments. Technical Report 43, Robotics Laboratory, Courant Institute, New-York University, 1985.

Relevant Publications

The publications on which this thesis is based are listed below.

- D. Nieuwenhuisen and M. H. Overmars. Useful Cycles in Probabilistic Roadmap Graphs. In *IEEE International Conference on Robotics and Automation*, pages 446–452, 2004.

- J.P. van den Berg, D. Nieuwenhuisen, L. Jaillet, and Mark H. Overmars. Creating Robust Roadmaps for Motion Planning in Changing Environments. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2415–2421, 2005.

- D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Path Planning for Pushing a Disk using Compliance. In *IEEE International Conference on Intelligent Robots and Systems*, pages 4061–4067, 2005.

- D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Pushing using Compliance. In *IEEE International Conference on Robotics and Automation*, pages 2010–2016, 2006.

- D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. An Effective Framework for Path Planning amidst Movable Obstacles. In *The Seventh International Workshop on the Algorithmic Foundations of Robotics*, to appear, 2006.

- D. Nieuwenhuisen, A.F. van der Stappen, and M.H. Overmars. Pushing a Disk using Compliance. In *IEEE Transactions on Robotics*, to appear, 2007.

Acknowledgments

Many people have contributed to this thesis in some way or the other. First of all I would like to thank my supervisors Mark Overmars and Frank van der Stappen. Having two supervisors was a big advantage not for the least because Mark and Frank proved to be complementary to each other. Mark, I really admire your discerning vision and your ability to put me back on the right track whenever I was in doubt about the directions of research. Frank, your mathematical bias and sharp eye for detail proved to be a big help in the creation of this thesis.

Next, I would like to thank the members of the reading committee Mark de Berg, Sándor Fekete, Ken Goldberg, James Kuffner and Arno Siebes for their efforts and suggestions.

A fruitful visit of Léonard Jaillet of LAAS/CNRS to Utrecht in the summer of 2004 and collaboration with Jur van den Berg led to the results of Chapter 3.

A pleasant environment to work is an important condition to stay motivated. Therefore I would like to thank all my colleagues at the department for the great atmosphere not only while working but also during lunch, while playing foosball or table tennis or when having a drink outside office hours.

Special thanks go to Jur, Arno and Reinier for not only being pleasant colleagues but also good friends outside work. Jur thanks for all the fun we had while being office mates.

Finally, I would like to thank all other friends and my family for their support. Concetta, thanks for your motivation and inspiration.

Samenvatting

Het padplanning probleem is een breed begrip met vele verschijningsvormen zowel in de “echte” wereld als in een virtuele wereld. Denk bijvoorbeeld aan een robotarm in een fabriek waarvoor een efficiënte beweging gepland moet worden om een object vast te pakken maar ook aan een karakter in een computerspel dat op een realistische manier moet bewegen door zijn omgeving. Andere voorbeelden zijn een robot die medicijnen rondbrengt in een ziekenhuis en obstakels moet vermijden of de simulatie van een biologisch proces waarbij eiwitten in elkaar gepast moeten worden. Er is de afgelopen jaren veel onderzoek gedaan naar efficiënte algoritmen om dergelijke problemen op te lossen. De resultaten van deze inspanningen zijn veelbelovend en geschikt om in de praktijk ingezet te worden. Bij de meeste algoritmen wordt echter aangenomen dat de omgeving statisch is. Dat wil zeggen dat tussen het moment van het plannen van het pad en het moment waarop het daadwerkelijk uitgevoerd is, geen veranderingen in de omgeving mogen optreden. Hoewel succesvol in veel toepassingen heeft deze aanpak zijn beperkingen aangezien veel realistische omgevingen wel degelijk veranderen. In dit proefschrift beschrijven we drie scenario’s waarin *wel* verschillende typen veranderingen van de omgeving toegestaan zijn. Voor elk van deze “veranderlijke” omgevingen beschrijven we een efficiënte oplossing voor het padplanning probleem.

De eerste twee scenario’s baseren hun navigatie op het bekende PRM (*Probabilistic Roadmap Method*) algoritme. Om deze geschikt te maken voor veranderlijke omgevingen presenteren we in **Hoofdstuk 2** een variant die dient als basis voor deze twee scenario’s. De PRM methode maakt een routekaart van de omgeving die gebruikt wordt voor navigatie van een robot. Om kostbare rekentijd te besparen bevat deze routekaart gewoonlijk geen zogenaamde *cycles*. Het resultaat hiervan is dat er hoogstens één pad bestaat tussen twee punten. Mocht dit pad geblokkeerd worden door een obstakel dat verplaatst is dan faalt de methode terwijl er misschien een alternatieve route bestaat. De routekaart uitbreiden met extra paden lijkt een voor de hand liggende oplossing maar kost veel rekentijd als dit niet met beleid gedaan wordt. Daarom moeten niet alle alternatieve routes toegevoegd worden aan de routekaart maar slechts degenen die daadwerkelijk nuttig zijn. Een bijkomstig voordeel van het hebben van alternatieve routes is dat de lengte van de paden ook afneemt omdat er keuze is tussen verschillende alternatieven.

Het algoritme dat wij presenteren in Hoofdstuk 2 is in staat onderscheid te maken tussen wel en niet nuttige alternatieve routes en kost nauwelijks extra rekentijd. We bewijzen ook dat de lengte van een pad in onze routekaart nooit langer kan zijn dan een bepaalde bovengrens. Bovendien presenteren we experimenten die de effectiviteit van onze aanpak demonstreren.

Het eerste scenario wordt behandeld in **Deel I**. Het gaat hier om omgevingen met obstakels die telkens als een pad gepland wordt een andere positie kunnen hebben. We gebruiken hierbij de observatie dat hoewel veel obstakels verplaatst kunnen worden, ze vaak alleen voorkomen binnen een besloten ruimte. Een voorbeeld is een stoel die binnen een kamer vaak verplaatst zal worden maar zelden daarbuiten zal komen. Een deur zal zelfs helemaal nooit opduiken op een verrassende plaats aangezien hij vastzit aan zijn scharnieren. Gebruikmakend van deze observatie kunnen we garanderen dat een pad altijd aanwezig zal zijn in de routekaart van de omgeving ongeacht de exacte positie van deze obstakels. In **Hoofdstuk 3** presenteren we een raamwerk algoritme dat een dergelijke routekaart kan maken door, als een pad geblokkeerd is, te garanderen dat er een alternatief pad voorhanden is, mits een dergelijk pad bestaat.

Hoewel we in Hoofdstuk 3 naast het raamwerk ook een methode beschrijven om het raamwerk te implementeren, werkt deze implementatie alleen in simpele omgevingen. Als het aantal obstakels te groot wordt, dan neemt de rekentijd voor het algoritme drastisch toe. In **Hoofdstuk 4** beschrijven we een andere aanpak om het raamwerk te implementeren door een algoritme te introduceren dat gebruikt maakt van Booleaanse logica. Eerst wordt beschreven hoe het probleem vertaald kan worden naar het domein van de Booleaanse logica en vervolgens hoe deze vertaling gebruikt kan worden om een efficiënte implementatie te maken van ons raamwerk. Ten slotte worden de resultaten van experimenten beschreven die demonstreren hoe effectief onze aanpak is.

Het tweede scenario, beschreven in **Deel II**, kijkt naar omgevingen waarin een robot obstakels die zijn pad blokkeren uit de weg mag ruimen. Het doel van het probleem wordt gedefinieerd in termen van een eindpositie voor de robot. De robot opereert in een omgeving waarin, naast vaste obstakels (zoals muren) ook verplaatsbare obstakels voorkomen (zoals stoelen, dozen, deuren etc.). Deze verplaatsbare obstakels kunnen niet uit zichzelf bewegen maar kunnen door de robot verplaatst worden door duw en/of trek bewegingen. De robot moet beslissen welke obstakels hij moet verplaatsen zodanig dat hij in staat is zijn doel te bereiken. Dit is een moeilijk probleem omdat veel obstakels de robot niet direct in de weg zullen zitten maar wel het verplaatsen van andere obstakels blokkeren. Neem bijvoorbeeld een bank die een doorgang verspert. De robot zal dan deze bank aan de kant moeten schuiven maar het kan zo zijn dat een ander obstakel dat verhindert, bijvoorbeeld als een tafel tegen de bank aangeschoven is. De robot zal dan eerst deze tafel aan de kant moeten schuiven en vervolgens pas de bank kunnen verplaatsen. Hierbij is het van belang dat de tafel op een zodanige plek wordt neergezet dat hij het verplaatsen van de bank niet in de weg zit. Naast het bepalen van welke obstakels verplaatst moeten worden spelen de volgorde en de manier van verplaatsen dus ook een grote rol. Een algoritme zal dus rekening moeten houden met deze factoren.

De resultaten van Hoofdstuk 2 dienen als basis voor het algoritme dat in dit deel beschreven wordt. Eerst, in **Hoofdstuk 5** beschrijven we een raamwerk dat een gestructureerde *action space* maakt. Deze action space is een beschrijving van alle mogelijk acties die de robot in alle mogelijke volgorden met de verplaatsbare obstakels kan doen. Uitputtend doorzoeken van deze action space garandeert een oplossing als deze bestaat. Deze aanpak is echter rekenkundig veel te duur voor de meeste problemen. Daarom presenteren we heuristieken in **Hoofdstuk 6** die de zoekruimte drastisch reduceren zodat een groot aantal problemen efficiënt opgelost kan worden. De resultaten van Hoofdstuk 2 worden wederom als basis gebruikt. Dit wordt gecombineerd met lokale oplossingen voor de manipulatie van de verplaatsbare obstakels terwijl een globale planner coördineert welke obstakels worden verplaatst en in welke volgorde. Onze experimenten tonen aan dat de aanpak goed werkt voor een groot aantal realistische problemen.

Het laatste deel, **Deel III** beschrijft een scenario waarin een specifieke vorm van manipulatie, namelijk duwen, wordt gecombineerd met bewegingen waarbij een object langs de randen van obstakels beweegt. Deze zogenaamde *compliant* bewegingen ontstaan als een object schuin tegen een obstakel (bijvoorbeeld een muur) wordt geduwd waarbij dit object dan langs deze muur gaat schuiven. Het duwen van objecten is een grondig bestudeerd onderwerp. Meestal wordt dan een object dat niet uit zichzelf kan bewegen (het passieve object) geduwd door een ander object (het actieve object) dat dit wel kan. Een dergelijke beweging kan voor meerdere doeleinden gebruikt worden. In Deel II werden deze bewegingen gebruikt om een obstakel uit de weg te duwen. In dit deel kijken we naar problemen waarbij het doel gedefinieerd is in termen van het passieve object. De taak voor het actieve object is dus om het passieve object naar zijn doel te duwen. In applicaties in de “echte” wereld is dit meestal een moeilijk probleem om een aantal redenen. Ten eerste zijn de precieze bewegingen van de twee objecten moeilijk te voorspellen aangezien vele parameters niet exact bekend zijn (denk aan wrijving en massamiddelpunten). Een andere bron van afwijkingen zijn sensorfouten. Bijvoorbeeld afstandsmeters die kijken naar het aantal omwentelingen van de wielen van een rijdende robot kunnen kleine afwijkingen hebben zodat de eigenlijke positie van de robot anders is dan de gemeten positie. Een bekende techniek om dit soort afwijkingen te compenseren is door gebruik te maken van compliant bewegingen. Dit soort bewegingen fixeert bepaalde vrijheidsgraden van het object zodanig dat er geen afwijkingen kunnen optreden. In dit deel gebruiken we compliant bewegingen om te compenseren voor afwijkingen in duwproblemen. Naast het compenseren van afwijkingen hebben compliant bewegingen nog meer voordelen. In bepaalde gevallen vergemakkelijken ze het vinden van een geschikt pad voor het actieve object en ook kunnen problemen worden opgelost die zonder compliant bewegingen niet oplosbaar zouden zijn.

De combinatie van duwbewegingen en compliance roept interessante vragen op waarvan er een aantal beschreven worden in **Hoofdstuk 7**. Tevens worden daar een een aantal definities gegeven die in de daaropvolgende hoofdstukken worden gebruikt. De belangrijkste vraag is: gegeven een eindpositie voor het passieve object, hoe ziet het pad voor het actieve object eruit zodanig dat het passieve object naar deze eindpositie geduwd wordt. Voor dit probleem tonen we aan dat een exacte oplossing ondoenlijk is in termen van re-

kentijd. Daarom presenteren we in **Hoofdstuk 8** een exacte oplossing voor een deelprobleem, namelijk het vinden van een pad voor het actieve object, gegeven een pad voor het passieve object.

In **Hoofdstuk 9** bouwen we voort op de resultaten van Hoofdstuk 8 en presenteren een efficiënt algoritme voor het meer algemene geval waarin alleen een start en eind positie voor het passieve object zijn gegeven. Het algoritme is dan in staat een pad voor het actieve object te genereren zodanig dat het passieve object naar zijn eindpositie wordt geduwd. Tijdens dit pad mag het passieve object langs de randen van de omgeving glijden. Aangezien een exacte oplossing ondoenlijk is, combineren we een exacte oplossing voor de compliant delen van het pad met een probabilistische oplossing voor het duwen door de vrije ruimte. Deze laatste oplossing is gebaseerd op het RRT (*Rapidly-exploring Random Trees*) algoritme. Dit algoritme heeft als eigenschap dat het resulteert in configuraties waarbij het passieve object de omgeving raakt. Deze configuraties gebruiken we als startpunt voor het maken van de compliant bewegingen. We presenteren resultaten van experimenten die het voordeel van het gebruik van compliant bewegingen aantonen.

Curriculum Vitae

Dennis Nieuwenhuisen was born in Gouda, the Netherlands on August 23, 1975. After moving to Apeldoorn, he received his high school diploma in 1993 at the local Koninklijke Scholengemeenschap. Starting in 1993 he worked toward a bachelor's degree in chemistry at the Hogeschool IJsselland in Deventer which he received in 1997. He then moved to Utrecht University where he studied computer science. After receiving his master's degree in 2002 he worked toward the Ph.D degree at the same university at the computer science department under supervision of prof. M. H. Overmars. In 2007 he completed this thesis.