Path Planning in Dynamic Environments

Typesetting: This thesis was typeset by the author using LaTeX 2_ϵ
Cover design: The cover was designed by Lucius, Groningen, using Adobe Illustrator CS2
Cover illustration: Lucius, **Compositie 29 in groen en geel:** *'Lentewandeling'* (detail)

Printed in The Netherlands by Drukkerij Bariet, Ruinen

ISBN: 978-90-393-4480-4

Path Planning in Dynamic Environments

Padplanning in dynamische 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 4 april 2007 des middags te 2.30 uur

door

Jur Pieter van den Berg

geboren op 27 mei 1981
te Groningen

Promotor: Prof. dr. M.H. Overmars

This work was financially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2001-39250 (MOVIE - Motion Planning in Virtual Environments).

The Road Not Taken

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.

Robert Frost, 1916

Contents

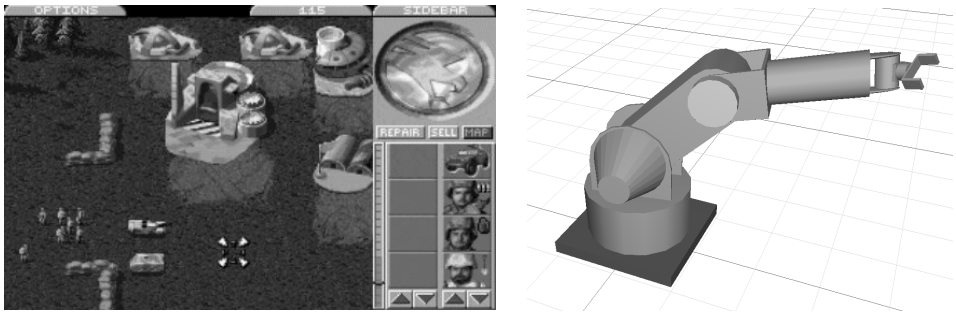
Chapter 1

Introduction

Path planning plays an important role in various fields of application and research, among which are CAD-design, computer games and virtual environments, molecular biology, and robotics. In its most general form, the task is to plan a path for some moving entity between a start position and a goal position in some environment. As an example, let us consider a well known computer game: Command & Conquer (see Fig. 1.1(a)). It is a strategy game in which the player's army has to defeat another player's army. The player can control its army by selecting one of its units (e.g. a tank or a soldier) and clicking on a location in the map to which it should move. The unit will then automatically move towards the desired goal position. At this point, the computer solves a path planning problem, that is, it has to find a path between two positions in a map that avoids obstacles (such as buildings, mountain ranges, rivers) and is preferably as short as possible. Experienced players will have noticed that the environment is discretized into a rectangular grid whose cells are either accessible or not. Each grid cell can exactly contain a single unit, and is connected to each of its 8 neighboring grid cells. A path in such a grid is a sequence of neighboring accessible cells, and it can be computed using Dijkstra's shortest path algorithm [44], or A* [70] (see Chapter 2).

A more challenging path planning problem occurs when the set of all possible states is not discrete as in the case of a grid, but continuous. An example is an industrial manipulator robot (see Fig. 1.1(b)) that has to move in a three-dimensional environment while avoiding collisions with itself and obstacles in the environment. The challenge in these cases is to discretize the problem in a sensible way, such that it becomes tractable. As we will see, using a grid is only one of many options. Although the problems of moving a military unit in a two-dimensional computer game environment and moving a manipulator robot in a three-dimensional industrial environment are seemingly very different, they have the same general formulation in terms of the *configuration space*, as we will explain below. This allows us to abstract from the specific application, and study the problem in a generic way.

The path planning problem has been studied extensively over the past decades. See, for instance, the textbooks of Choset et al. [41], De Berg et al. [26], Latombe [96] and LaValle [97] for detailed introductions into path planning and many references to related work. Most effort has been spent on path planning in *static* environments. That is, a path has to be found



(a) (b)

Figure 1.1: (a) A screenshot from the computer game Command & Conquer (Courtesy of Electronic Arts). (b) An industrial manipulator robot.

between two configurations for a movable object in an environment containing stationary obstacles whose geometry and coordinates are given. Less attention has been given to path planning in *dynamic* environments. Besides stationary obstacles, dynamic environments contain moving obstacles with which collisions must be avoided as well. For example, a mobile robot operating at a factory floor will have to navigate among humans or other robots, which can be considered as moving obstacles. Also, the path planner in Command and Conquer has to deal with a dynamic environment, as a unit for which a path is planned should not run over moving friendly units.

Path planning in dynamic environments will be the main topic of this thesis. We will study both the case where the trajectories of the moving obstacles are known beforehand, and the case where this information has to be acquired *online*, for instance by means of sensors.

In this chapter, we introduce the path planning problem and provide references to the most important earlier work. We first discuss the case of static environments, and then introduce the more advanced problem of planning in dynamic environments. Finally, we give an overview of the contents of this thesis.

1.1 Configuration Space

The path planning problem is in its most general form a geometric problem. It needs four ingredients:

- A description of the geometry of the moving entity (in this thesis called the *robot*)
- A description of (the geometry of) the environment in which the robot moves or operates (also called the *workspace*)
- A description of the degrees of freedom of the robot's motion (see below)

- A start and a goal configuration in the environment, between which a path is to be planned for the robot

Using this information, we can construct the *configuration space* of the robot, in terms of which the path planning problem is formulated generally (the notion of configuration space was first introduced by Lozano-Pérez in the late seventies [151, 109, 108]). A *configuration* of the robot in its workspace is described using a number of parameters. For instance, a configuration of a robot translating in a two-dimensional workspace can be described using two parameters, which are often denoted x and y. A configuration of a manipulator robot can be described using a number of angles that the joints make between the consecutive links. The minimal number of parameters needed to uniquely describe a robot configuration equals the number of *degrees of freedom* (dofs) of the robot. The space of all robot configurations is called the *configuration space* of the robot, which we denote by $\mathcal{C}$. Each degree of freedom of the robot accounts for one dimension of the configuration space.

In realistic scenarios, the workspace of the robot contains obstacles. These obstacles cause some configurations to be *forbidden*. For instance, a configuration c is forbidden if the robot configured at c collides with any of the obstacles in the workspace. More generally, the configuration space $\mathcal{C}$ is partitioned into a set of forbidden configurations $\mathcal{C}_{\text{forb}}$, and a set of free configurations $\mathcal{C}_{\text{free}}$. Configurations in $\mathcal{C}_{\text{forb}}$ not necessarily relate to collisions with workspace obstacles. For instance, if the robot is an industrial manipulator arm, the end-effector may collide with the base of the robot in some configurations. These are called *self-collisions*. In general, forbidden configurations can be the result of all kinds of internal robot constraints.

A path is defined as a continuous function $\pi : [0, L] \rightarrow \mathcal{C}$, parametrized by the length L of the path. The path planning problem is to find a (collision-)free path between a given start configuration $s \in \mathcal{C}$ and goal configuration $g \in \mathcal{C}$. Formulated in terms of the configuration space $\mathcal{C}$, that is finding a path π such that $\pi(0) = s$ and $\pi(L) = g$, and $\forall (t \in [0, L] :: \pi(t) \in \mathcal{C}_{\text{free}})$. If such a path does not exist, failure should be reported. In addition to this definition, one may define a quality measure on all possible paths, and require that the path that optimizes this measure is found.

1.2 Path Planning in Static Environments

The basic path planning problem deals with *static* environments, that is, workspaces solely containing stationary obstacles of which the geometry is known.[1] In this section, some elementary approaches to this problem are introduced. A common aim of the various methods is to capture the connectivity of the free configuration space into some *roadmap*.[2] This roadmap should be constructed such that each free configuration in the environment can be connected to the roadmap in a trivial way (coverage), and that for all pairs of configurations

[1] Throughout this thesis, we use the adjectives "stationary" and "moving" for obstacles and the adjectives "static" and "dynamic" for environments.

[2] Traditionally, two other approaches to path planning are distinguished besides roadmap methods: cell decomposition methods and potential field methods [96]. As cell decompositions can be seen as special forms of roadmaps (the adjacency of the cells induce a graph), we do not treat them separately here. Potential field methods are less popular nowadays, and are briefly discussed later.

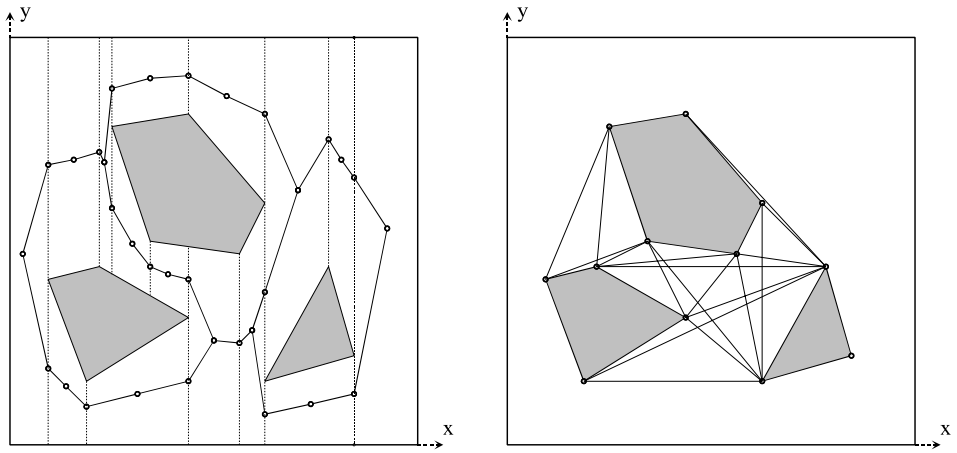


Figure 1.2: (a) The trapezoidal decomposition of a set of obstacles, whose cells' adjacencies induce a roadmap fully capturing the connectivity of the space. (b) The visibility graph of the same set of obstacles.

on the roadmap between which a valid path exists in the free space, a valid path through the roadmap exists as well (connectivity) [96]. If a roadmap can be created fulfilling these two criteria, the method is said to be *complete*, that is, a valid path can always be returned if it exists, and failure can be reported otherwise [67]. We will see that in most settings it is hard, if not impossible, to meet these criteria. In such cases we have to resort to weaker forms of completeness.

1.2.1 Two-Dimensional Configuration Spaces

The simplest instance of the path planning problem is finding a path for a *point* robot in a two-dimensional static environment. In most cases, it is assumed that the geometry of the workspace obstacles is given using a polygonal representation. As the robot is a point, the configuration space exactly resembles the workspace. Hence, the obstacles in configuration space (i.e. $\mathcal{C}_{\text{forb}}$) are *explicitly* represented. This can be used to efficiently solve the planning problem. A straightforward approach is to decompose the free configuration space into vertical trapezoidal regions, and construct a graph based on the adjacency of free cells to represent the connectivity of the free configuration space and serve as a roadmap of valid paths through the environment [39] (see Fig. 1.2(a)). It takes $O(n \log n)$ time to compute the graph, and $O(n)$ time to query it for a path between any pair of configurations (n is the total number of obstacles vertices).

Another approach is to construct the *visibility graph*, which connects the mutually visible vertices of the obstacles in the configuration space. Such a graph not just encodes some valid paths through the environment, but even the *shortest* paths between every pair of ver-

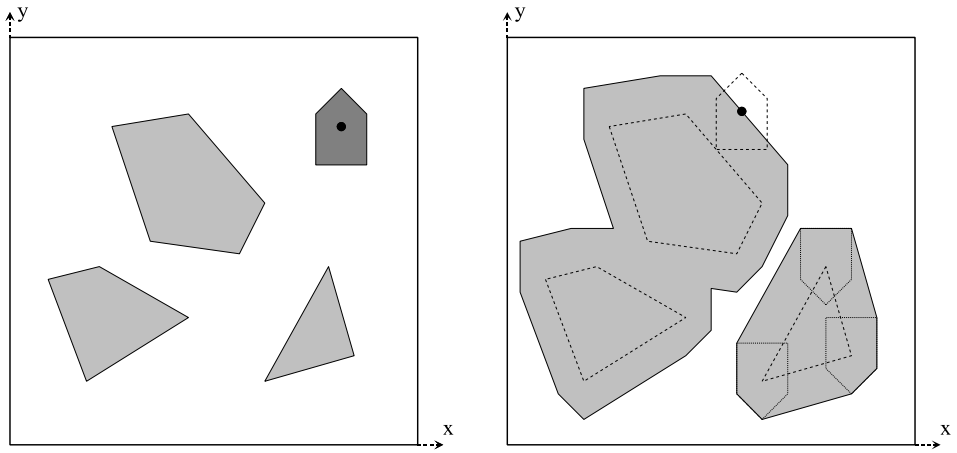


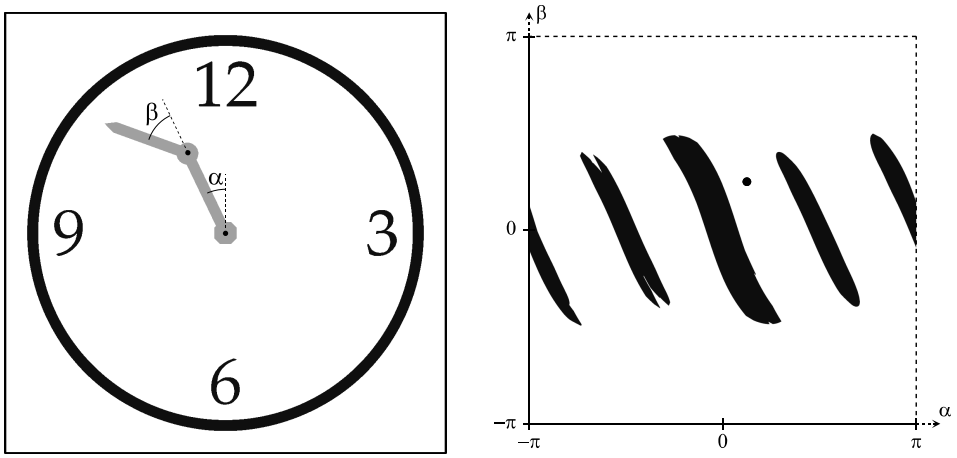
Figure 1.3: (a) A workspace with obstacles (light gray) and a translating robot (dark gray). (b) The corresponding configuration space whose obstacles are formed by taking the Minkowksi sum of the workspace obstacles with the robot reflected in its reference point (see lower-right obstacle). The robot is collision-free if its reference point is outside the configuration space obstacles (see top obstacle).

tices [116] (see Fig. 1.2(b)). It takes $O(n \log n + k)$ time to construct the graph and find a shortest path between any pair of configurations (n is the total number of obstacle vertices; k is the number of visibility edges in the output visibility graph) [64].

If the robot is not just a point, but a dimensional geometric object, the problem becomes harder, as the configuration space is no longer equal to the workspace. Let us assume that the robot can only translate in a two-dimensional workspace, then, a configuration of the robot is defined as the position of a certain *reference point* on the robot in the workspace. Now, it may very well be possible that for some configuration the reference point is in the free workspace, yet some other point on the robot is not. Hence, the configuration is forbidden. This makes that the obstacles in the configuration space are different from the obstacles in the workspace (see Fig. 1.3). The configuration space obstacles can still be constructed explicitly, by taking the *Minkowski sum* of the workspace obstacles with the robot reflected in its reference point [108]. This takes $O(n \log n)$ time using a randomized algorithm if the robot is convex and has constant complexity [26]. In the configuration space, the robot can then be treated as a point, and either of the above methods can be used.

1.2.2 Higher Dimensional Configuration Spaces

For a translating robot in a three-dimensional workspace, the configuration space is also three-dimensional and can explicitly be constructed in more or less the same way as above [8]. Also, the trapezoidal decomposition method still works [25, 141], but no such thing as a visibility graph exists in 3-D, at least not one encoding shortest paths. Canny showed that



(a) (b)

Figure 1.4: (a) A workspace with obstacles in the form of numbers (dark) and an articulated robot (light) with a fixed base and two fingers whose configurations are described using two angles α and β. (b) The corresponding configuration space, which is periodic in both dimensions. Each obstacle corresponds to one of the numbers 3, 6, 9 and 12. The dot indicates the configuration shown in (a). (Courtesy of Roland Geraerts)

finding a shortest path between any pair of vertices in a three-dimensional configuration space is an NP-complete problem [35, 129].

If the workspace is two-dimensional, but the robot can not only translate but also rotate, the configuration space is three-dimensional as well (since a configuration of a robot is described using two positional parameters and one defining the angle of rotation). In this case, however, it is much more difficult to explicitly construct the configuration space obstacles, as (assuming that both the robot and obstacle geometry is given using a polygonal representation) the facets of the obstacles become (nastily) curved, and are no longer polygonal [137, 10]. The problem can still be solved in near-quadratic running time, however [69]. Things become worse in a three-dimensional workspace, with a fully translating and rotating robot (also called a *free-flying robot*). In that case the configuration space is six-dimensional, as each configuration needs three parameters to describe the position and three to describe the orientation (roll, pitch and yaw).

In many examples, the configuration space may somehow seem to "look like" the workspace, because translation is the dominating degree of freedom. In general though, the free part of the configuration space may have a very different shape than the free workspace (see e.g. Fig. 1.4).

Not only the algebraic complexity of the facets of the configuration space obstacles is problematic in high-dimensional configuration spaces, but also the combinatorial complexity of the obstacles. Canny introduced a generic method for creating a roadmap capturing

the complete connectivity of the free space that takes time exponential in the dimension of the configuration space [36]. Reif showed that in general the path planning problem is PSPACE-hard [127].

1.2.3 Incomplete Approaches

In such high-dimensional configuration spaces, it is intractable to explicitly construct a representation of the configuration space obstacles. Instead, *implicit* representations are used. In that case, there is no geometric description of the configuration space obstacles, but given a configuration it is possible to determine whether this configuration lies either in the free space or in the forbidden space. For this, *collision-checking* algorithms are used [28, 105]. A collision-checking algorithm, or collision-checker, is able to (quickly) determine whether or not two three-dimensional objects overlap. This suffices, even though configuration spaces can be of arbitrary dimension, as the underlying geometry of the workspace and the robot remain three-dimensional.

Using such an implicit representation of the configuration space, it is impossible to construct a complete roadmap. That is, we are unable to tell whether a given roadmap completely covers and captures the connectivity of the configuration space. Yet, there are methods that appear to work very well in practice which guarantee a weaker form of completeness. Among the most popular ones is the *Probabilistic RoadMap* (PRM) method [89, 149], which creates a roadmap by *randomly* sampling configurations from the configuration space. If these configurations are collision-free (determined by the collision-checker), they are added as nodes to the roadmap. A pair of nodes is connected by an edge if a *local planner* succeeds to find a valid path between the two corresponding configurations. In most implementations, this local planner simply tries to connect the two configurations by a straight-line through the configuration space. The validity of the connection is verified by collision-checking a number of intermediate configurations, up to some preset resolution [140]. The above process goes on until a roadmap is formed that meets some user-defined criteria.

The PRM-method is not complete. That is, if a path does not exist in $\mathcal{C}_{\text{free}}$ between a given start and goal configuration, the PRM-method will never be able to prove this. However, the method is proven to be *probabilistically complete*, meaning that if a path does exist in $\mathcal{C}_{\text{free}}$ between a given start and goal configuration, the probability that a PRM-roadmap contains a solution converges to 1 as the roadmap creation time approaches infinity [148]. It is even proven that this convergence goes exponentially fast [77]. The PRM-method will be discussed in detail in Chapter 2 and will play an important role in this thesis.

1.2.4 Single-Shot vs. Multiple-Shot

The methods discussed so far have in common that some data structure (the roadmap) is created in a preprocessing phase, which can be queried multiple times for a path between arbitrary start and goal configurations. Often, the aim is to put so much effort into the preprocessing, such that queries can be answered relatively quickly, and allow –for instance– for application in real-time settings. These methods are categorized as *multiple-shot* methods.

In some applications, preprocessing is not desired, or very hard (e.g. when the environment is changeable). In such cases so-called *single-shot* methods are favorable. They aim to solve a specific query as quickly as possible, without using any preprocessing. A well-known single-shot method is the *Artificial Potential Field* method [91, 131], in which the motion of the robot is governed by an artificial potential field that is cast over the environment. Usually the goal configuration applies an attracting force and the obstacles apply a repelling force on the robot. The robot then follows the steepest descent in the resulting potential field. A drawback is that the robot is prone to get stuck in local minima of the potential field. Approaches exist to remedy this [13], but still the method does not guarantee success in general. The potential field method is particularly suitable for problems where translation is the dominating degree of freedom, as the forces are usually calculated using the workspace distance between the robot and the obstacles.

The *Rapidly-Exploring Random Tree* (RRT) approach [93, 101] is a general single-shot variant of the PRM-method. A tree of valid paths is grown outward from the start configuration by random sampling, until the goal configuration can be connected to the tree. The direction of growth is often biased towards the goal configuration [97, 80]. In other variants two trees are grown, one from the goal and one from the start, until they meet in the middle [94, 135]. As the PRM-method, the RRT-approach is proven to be probabilistically complete.

In many chapters of this thesis, we will use some form of preprocessing, in order to be as fast as possible in the query phase, except for Chapter 6, in which we use the single-shot paradigm.

1.3 Path Planning in Dynamic Environments

A natural extension to the basic path planning problem is planning in dynamic environments, in which besides stationary obstacles, also moving obstacles are present. It will be the main topic of this thesis. Just as with planning in static environments, the problem of planning in dynamic environments has many different instances, with different assumptions about the input, and different requirements on the output. In this thesis we will present approaches to a number of different instances of the problem. Here, we will discuss the broad range of instances, briefly introduce existent approaches, and provide forward pointers to chapters in this thesis.

1.3.1 Configuration-Time Space

The configuration space concept can be extended for dynamic environments by incorporating an absolute notion of time in the configuration space as an additional dimension. The resulting space is called the *configuration-time space*. Let us define $\mathcal{T} = [0, t_{\max}]$ to be the time interval of interest without loss of generality, where 0 is the initial time. Then, the configuration-time space, denoted by $\mathcal{CT}$, is formed as $\mathcal{C} \times \mathcal{T}$. It consists of pairs $\langle c, t \rangle$, where c is an element of $\mathcal{C}$ denoting the robot's configuration, and t a scalar in $\mathcal{T}$ denoting the time. If the robot configured at c collides with any moving (or stationary) obstacle at time t, the configuration-time $\langle c, t \rangle$ is element of $\mathcal{CT}_{\text{forb}}$. Else, it is in $\mathcal{CT}_{\text{free}}$. Hence, both stationary and

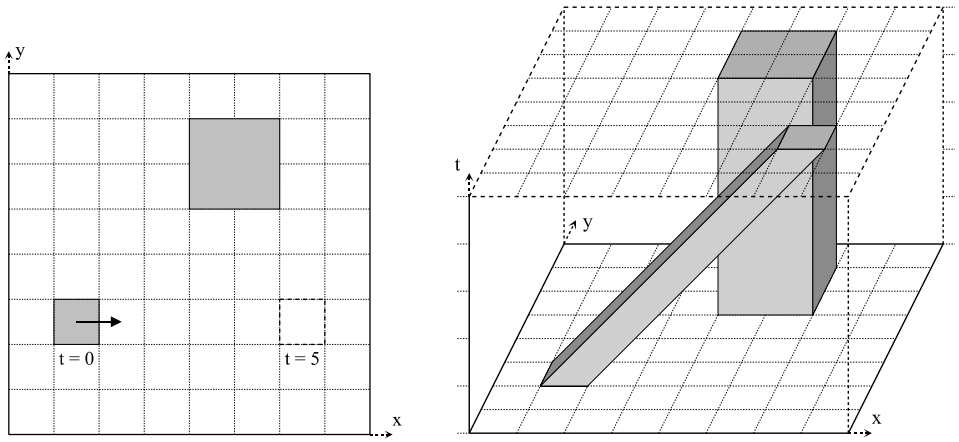


Figure 1.5: (a) A configuration space with a moving obstacle (small square with velocity vector) and a stationary obstacle (large square). (b) The corresponding configuration-time space.

moving obstacles in the workspace transform to stationary obstacles in the configuration-time space (see Fig. 1.5).

A path through a dynamic environment (in literature often called a trajectory) is defined as a continuous function $\pi : \mathcal{T} \rightarrow \mathcal{C}$, parametrized by time. The path planning problem in dynamic environments is to find a (collision-)free path between a given start configuration s and goal configuration g. Formulated in terms of the configuration-time space $\mathcal{CT}$, that is finding a path π such that $\pi(0) = s$ and $\pi(T) = g$, and $\forall(t \in [0,T] :: \langle \pi(t), t \rangle \in \mathcal{CT}_{\text{free}})$, where T is the duration (or arrival time) of the path.

Many of the methods discussed above can be applied without much difficulty on configuration-time spaces. However, as time marches forward, additional constraints are put on the validity of paths through the configuration-time space. Obviously, paths should be monotonically increasing in the time dimension in order to prevent unrealistic time travel. Note that the above definition of a path in a configuration-time space already enforces this. This may not be enough, however, as it still allows paths demanding the robot to travel with nearly infinite velocity. Often, therefore, the velocity of the robot is bounded to a maximum (say $v_{\max}$), which imposes a constraint on the slope of paths (with respect to the time dimension) through the configuration-time space: $\forall(t_1, t_2 \in [0,T] : t_1 < t_2 : d(\pi(t_1), \pi(t_2)) \leq v_{\max}(t_2 - t_1))$, where $d : \mathcal{C}^2 \rightarrow \mathbb{R}$ is a distance metric on the configuration space $\mathcal{C}$.

In many cases the notion of time in configuration-time spaces is related to the real-world time. This causes the configuration-time space to be *transitory*, that is, its representation is only meaningful during the interval in the real-world time which is incorporated in the configuration-time space. Therefore, multiple-shot approaches are not useful in dynamic environments. It is possible to construct a roadmap in a preprocessing phase that captures

the connectivity of the free configuration-time space (for instance using an adapted PRM-method), but this roadmap can never be used more than once.

Another aspect that is sometimes different for path planning problems formulated in a time-varying setting, is that the goal 'configuration-time' is not stated precisely with respect to the time. That is, it is sometimes not desired that the robot arrives at the goal configuration at a specific moment in time, but rather arrives at the goal as soon as possible, or *before* some preset moment in time. This turns the goal configuration into a line (segment) parallel to the time-axis of the coordinate frame of the configuration-time space.

1.3.2 Planning in Known Configuration-Time Spaces

The simplest instance of the planning problem in dynamic environments is when the motions of the obstacles are predictable, that is, they are fully known and given beforehand. If they are not known, or can only be estimated for a short amount of time, the problem becomes much harder. We will discuss this case later, and assume for now that we know the motions of the moving obstacles.

In case the workspace is two-dimensional, and the moving obstacles are polygons having piecewise-linear motions, the (three-dimensional) configuration-time space is polygonal and can be constructed explicitly (see e.g. Fig. 1.5). Then, using a vertical trapezoidal decomposition the problem can be solved in polynomial time [43, 138], but only when the velocity of the robot is unbounded. If the velocity of the robot is bound to a maximum, however, there is no algorithm with polynomial running time [35, 147]. In [128], a planner is presented that has a running time exponential in the number of moving obstacles. A polynomial time algorithm is presented in [57], but this result requires that all of the obstacles move slower than the robot.

All in all, complete approaches are only applicable to a limited domain. Therefore, various probabilistically complete approaches have been suggested that do not constrain the nature of the obstacle motions. As said above, constructing a PRM-roadmap in a configuration-time space is often not useful (although we will present a setting in which this is useful in Chapter 7), so a single-shot approach is natural to take. In [11, 75, 124, 154], RRT-like approaches are presented in which a tree is grown outward from the start configuration oriented along the time axis of the coordinate frame. An additional advantage of RRT approaches is that they allow for taking acceleration bounds into account as well [100]. In [152], a planner is presented based on genetic optimization.

Another approach is to decompose the problem into two parts. In the first part, a path is planned avoiding the stationary obstacles in the configuration space while ignoring the moving obstacles. In the second part, the velocity of the robot along the pre-planned path is tuned to avoid collision with the moving obstacles [85]. This results in a two-dimensional configuration-time space, as a configuration along a path can be described using one parameter. Obviously, this approach is not complete, yet it provides a way to efficiently plan a collision-free path through the configuration-time space[3]. In [85], the two-dimensional

[3]Beware that the word "path" is used both for the pre-planned path through the configuration space that is collision-free with respect to the stationary obstacles, and for the path through configuration-time space that avoids collisions with the moving obstacles by coordinating the motion of the robot along the pre-planned configuration

configuration-time space is explicitly constructed. In [53], the configuration-time space is discretized, but that allows for taking constraints on the acceleration of the robot into account as well. In this thesis, we more or less use the same paradigm at several places. Instead of a path, however, we construct a roadmap on which we coordinate the motion of the robot. This considerably enlarges the robot's maneuverability.

1.3.3 Online Planning vs. Offline Planning

An issue that is not dealt with in the above approaches is that planning a path takes time itself, during which the environment changes (the obstacles are moving). This causes a rather fundamental problem, namely if we want to plan a path that starts at time $t = 0$, then we cannot start computing the path at $t = 0$, because the computation would then immediately be outdated. To overcome this problem, often a fixed amount of time, say τ, is reserved for planning [75, 124]. The computation then starts at time $t = -\tau$, and should be finished before $t = 0$. When eventually time $t = 0$ has come, the plan can be executed.

In reality, the time to start the computation is not chosen based on a starting time of the plan (in the above example $t = 0$), but the other way around: when a path planning query is formulated in a dynamic environment at some real-world time t_w, the starting time of the plan will be equal to $t_w + \tau$, and the computation immediately starts (at time t_w).

The value of τ should not be chosen too small, because that would leave too little time for planning the path. On the other hand, it should not be chosen too large, because this would result in a latency that is unacceptable for most applications. However, for all of the previously discussed approaches (in particular the probabilistic ones) the actual time needed to plan a path is hard, if not impossible, to predict accurately. So, a different type of planner is needed to deal with the above considerations. Such planners are called *online* planners, in contrast to *offline* planners we have seen so far.

A straightforward approach to deal with the above problem is to use an offline planner and generously overestimate the typical amount of time needed to plan a path (i.e. choose a large value for τ) [75]. More sophisticated approaches use the concept of *partial planning* [124] or, similarly, *anytime planning* [152].

Partial planners take the value of τ as input, and explore different paths incrementally until time has run out. There may not be enough time to plan all the way to the goal configuration, but the planner then returns a partial plan that is the best among the ones explored in the allocated time (e.g. the path that comes as close as possible to the goal). During the execution of the motion, the available time for planning is used to extend or finalize the plan.

Anytime planners are similar. They in general, however, do plan complete paths to the goal configuration, but the quality of the path is continuously improved upon as long as time has not run out. Anytime planners are particularly suitable in situations where it is possible to generate initial plans –possibly of poor quality– very quickly, and where high quality paths are desired.

We must note that the above considerations do not imply that offline planners are not practical. They are particularly suitable in non-real-time applications, for instance in the problem of *multi-robot path planning*. In this thesis we will present an offline planner in

space path.

Chapter 3, and discuss its application to multi-robot planning in Chapter 4. Chapters 5 to 7 feature online planners.

1.3.4 Planning in Partially Known Dynamic Environments

Another issue in planning in dynamic environments is the amount of knowledge that is available of the (future) trajectories of the moving obstacles. In the previous sections we assumed that these trajectories are perfectly known in advance. In many cases, however, the motions of the moving obstacles are only predictable for the very near future, or are not predictable at all. Usually, on-board sensors provide information about the moving obstacles during the execution of the robot's path. This can be used to extend the robot's plan, or adapt a previously planned path to make it suitable for the new situation. This is repeated until the goal has been reached, that is, during the motion of the robot there is a continuous cycle of interleaved sensing and planning. Hence, only online planners are suitable for planning in partially known environments.

Let us assume that 'partially known' means that the future trajectory for the moving obstacles can be estimated based on acquired sensor data, (e.g. by extrapolating current position, velocity and direction of motion), and that this estimation can be updated when new sensor information arrives.[4] To plan a path in such an environment (say at real world time t_w), the planner often follows the online planning scheme: it takes the expected situation of the world at time $t_w + \tau$ as initial world state, and plans a path that avoids the obstacle motions as estimated. The plan is executed when time $t_w + \tau$ has come. Then during the motion of the robot, it may appear that the estimated obstacle trajectories are no longer in accordance with sensor observations. It that case, a new path must be planned to account for the new obstacle trajectories. Again, τ time is reserved for planning, and the expected situation of the world after τ time (including the position of the robot according to its *old* path) is taken as initial world state [75].

This scheme carries two fundamental problems:

- The predicted situation of the world at time $t_w + \tau$ may differ from the actual situation when some obstacles change their trajectories again during planning. This may result in invalid paths.

- The path the robot will follow between time t_w and time $t_w + \tau$ is not guaranteed to be collision-free, because this path was computed based on the old trajectories of the obstacles.

The online planners referred to in the previous subsection actually plan in partially known environments. They deal with the above problems by choosing τ as an inversely proportional function of the "dynamicity" of the environment. It remains unclear how this quantity should be quantified though.

In existing approaches, when it is detected that the estimated obstacle trajectories are not valid anymore, the information obtained in the previous planning cycle is not reused

[4] Neither sensing, nor motion prediction is the topic of this thesis, but a large amount of literature is available on these topics (see e.g. [15, 153]).

to more efficiently plan a new path: a new (partial) plan is built from scratch. In Chapter 5, we present an anytime approach for planning in partially known environments that does reuse old information, but also requires τ to be chosen based on the "dynamicity" of the environment.

In Chapter 6 we present an approach in which the obstacle predictions are chosen so conservatively that the two problems characterized above do not occur. The planner therefore is inherently safe. A drawback is that the obstacles are not allowed to move faster than the robot. Actually, in our approach there is so little information required about the trajectories of the obstacles, that this approach may also be characterized as a planner for *unknown* environments (instead of partially known). Most planners for unknown environments are very *reactive* and are often called *obstacle avoiders* [32, 91, 145, 52]. Little or no care is taken about the global characteristics of the path. Our approach does focus on the global path.

1.3.5 Acceleration Bounds vs. Velocity Bounds

An issue only slightly touched upon so far is whether the planner incorporates bounds on the acceleration or only on the velocity of the robot. Most of the planners we discussed only bound the velocity to a maximum. This means that the robot can change its velocity from, say, 0 to 1 in an instant. Hence, the acceleration at that particular instant is infinite. In most realistic scenarios the robot is subject to a maximum on its acceleration, denoted $a_{\max}$. If this is incorporated into the planner, the planning problem becomes (much) more difficult [45, 97], because for instance a series of straight-line segments through configuration-time space would not be a valid path.

Yet, some of the methods we discussed above are able to incorporate acceleration into their planner. In [75, 124], RRT-like single-shot approaches are presented in which a tree of possible paths is grown from the start configuration by sampling randomly in the *control space* of the robot, which consists of a set of valid acceleration vectors (depending on the configuration of the robot). New nodes in the tree are generated by selecting an existing configuration and integrating some action sampled from the control space over a short period of time [100].

Another approach tunes the acceleration along a pre-planned path through the configuration space as to avoid collisions with the moving obstacles [53]. The time axis is discretized, and the possible accelerations are limited to the set $\{a_{\max}, 0, -a_{\max}\}$. As a result, the possible velocities, as well as the possible positions along the path are discretized. On this grid, a path is planned to the goal.

In this thesis, we will not take acceleration into account, but are interested in the geometric problem of path planning only. We will hence restrict ourselves to setting a maximum on the velocity.

1.4 Corridors

In the settings discussed so far we assumed some kind of knowledge of the moving obstacles, either just by explicit representation, or by means of sensing. However, in many complex environments, only the stationary obstacles are precisely known at the time of planning, but

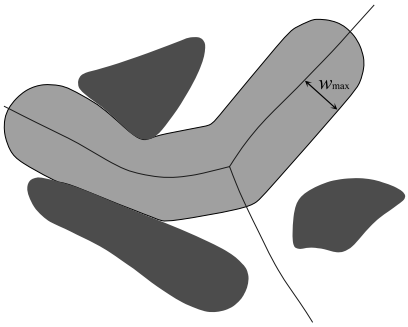


Figure 1.6: A corridor around a backbone path with an upper bound $w_{\max}$ on the disc radius.

nothing is known about the quantity and nature of the moving obstacles one will encounter. Often, a sensor is able to sense parts of the environment during the robot's motion, but the range of the sensor may be limited to only a very local area. This may cause obstacle motions to be predictable for only the very near future.

In such highly dynamic environments, it is better to separate the task of planning a path that globally leads toward the goal from the task of avoiding moving obstacles. This requires that the global path is not fixed, but contains the flexibility to locally deviate from the global path and avoid moving obstacles. On the other hand, this flexibility should not be unlimited, as the global path must remain a guideline for the global direction of motion. These considerations have lead to the development of the notion of *corridors* [34, 121, 63]. Corridors consist of a *backbone path* with on either side a free region (the corridor). The backbone path provides a global direction of motion, and the corridor leaves the room for local deviations from the global path.

Corridors also have other applications, such as path planning for coherent groups of entities [82]. There, the backbone path gives the global direction for the group, and the corridor sets a boundary for the group to stay together. Interactions between the individual members of the group (keeping appropriate distance etc.) are governed by a social potential field inside the corridor and are calculated on-line by a *local planner* (i.e. they are not preplanned).

1.4.1 Definition

A *corridor* $C = \langle \gamma(t), w(t), w_{\max} \rangle$ in a d-dimensional workspace (typically $d = 2$ or $d = 3$) is defined as the union of a set of d-dimensional balls whose center points lie along the *backbone path* of the corridor, which is given by the continuous function $\gamma : [0, L] \to \mathbb{R}^d$, where L is the length of γ. The radii of the balls along the backbone path are given by the function $w : [0, L] \to (0, w_{\max}]$. Both γ and w are parameterized by the length of the backbone path. Given a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ of length L in $\mathbb{R}^d$, the interior of the corridor is thus defined by $\bigcup_{t \in [0,L]} B\big(\gamma(t); w(t)\big)$, where $B(p; r)$ is an open d-dimensional ball with radius r that is centered at p.

Usually, the radius of the balls is determined by the clearance (i.e., the distance to the

nearest obstacle) along the backbone path. In order to guarantee that the local planner operates on a restricted environment, the radii of the balls are upper bounded by some predetermined value $w_{\max}$, which is called the *preferred width* of the corridor, or alternatively the *preferred clearance* along the backbone path (see Fig. 1.6).

1.4.2 Planning Backbone Paths for Corridors

In [82], the backbone path of the corridor is generated by extracting some path from a PRM-roadmap, and smoothing it in a post-processing stage such that it is as short as possible while keeping the preferred clearance from the obstacles. The preferred clearance cannot be obtained everywhere however, due to narrow passages. In these regions, the backbone path is retracted toward the *medial axis* of the stationary obstacles [59], such that locally the optimal clearance is obtained. The main problem of this approach is that it is unclear how to select the initial path from the PRM, as its properties (such as length and clearance) may be unrelated to the length and clearance of the post-processed and smoothed path. Smoothing all possible paths and the selecting the best one would be prohibitively expensive.

In [113], this problem is remedied by creating a roadmap of smooth paths that may serve as corridor backbones. This approach does not guarantee however, that the corridor has the preferred width everywhere where this is possible. In [62], the backbone path is extracted from the medial axis or the Voronoi diagram of the obstacles to obtain maximal clearance, but the acquired corridor may not be as short as possible.

In Chapter 8, we show how to construct a graph (or roadmap) called the *visibility-Voronoi diagram* that contains all backbone paths that would possibly result from the method of [82]. The advantage of such a graph is that the best corridor can be selected at query time based on its actual properties.

The main variables that determine the quality of a corridor are the length of the backbone path and the extent to which it were possible to obtain the preferred clearance along the backbone path. However, many scenes contain narrow passages (due to stationary obstacles) in which the preferred width cannot be obtained. So along some portions of a backbone path going through such a narrow passage, the corridor is narrower than the preferred width. There may be other routes for the corridor, however, which circumvent the narrow passage and where it is possible to sustain its preferred width, but these alternative routes may be much longer. Therefore, when planning the (backbone path of a) corridor, a trade-off should be made between length and width of the corridor.

In Chapter 9, we define a quality measure for corridors, and explore the properties of corridors that are optimal under this measure and present an approximation algorithm to plan optimal corridors.

1.5 Organization of this Thesis

In this section, we will give a brief overview of the works presented in this thesis, and put them into the perspective of previous work. The thesis is divided into two parts. The first part discusses planning in different kinds of dynamic environments, and the second part covers corridor planning.

1.5.1 Planning in Dynamic Environments

In Chapter 3, we will present an offline planner for known dynamic environments. The method has a preprocessing stage and a query stage. The preprocessing only concerns the stationary obstacles; a (dense) roadmap is created that covers the free part of the static configuration space. Then, a motion from a start to a goal avoiding collisions with the moving obstacles is computed on this roadmap using an efficient search strategy. Previous works either use a direct single-shot approach [75], or a decoupled approach where the robot motion is coordinated on a preprocessed path [53]. Our approach has advantages to both. Firstly, if the stationary obstacles produce narrow passages in the configuration space, an RRT-like approach directly applied in the configuration-time space is likely to have problems finding a path through them within a reasonable amount of time. In our approach, this problem is covered in the preprocessing. Secondly, constraining the motion of the robot to a path may be so rigorous that no valid coordination exists on this path with respect to the moving obstacles, while there do exist valid motions deviating from this path. If the roadmap we use is dense enough, and contains cycles to provide alternative routes, this problem (almost) disappears. Using a roadmap has more advantages, one of them being that the method is applicable to virtually any robot type (at least the ones for which a standard PRM-method works in a static environment). A drawback is that our approach does not allow constraints on the acceleration, in contrast to the two mentioned approaches. Experimental results show that our method achieves fast running times in complicated dynamic environments.

One of the most prominent applications of offline planners for known dynamic environments is its use in multi-robot path planning. In this related problem the task is to plan paths for a set of robots that bring each robot from some start configuration to some goal configuration without mutual collisions and collisions with stationary obstacles. In principle the set of robots can be seen as one multi-body robot with one composite configuration space. Although planning directly in this space would generally be too slow for practical application (due to the large dimensionality), this formulation of the problem shows that we are dealing with a static motion planning problem in which time does not play an intrinsic role. It is therefore that we can use an offline planner. It is applied as follows: first, each robot is given a priority. Then, in order of priority a path is planned for each robot from its start to its goal, thereby avoiding collisions with previously planned higher-priority robots. Hence, for each robot a path has to be planned in a known dynamic environment, where the previously planned robots are treated as moving obstacles. Planning may be done offline, as there is no real-time issue. If for all robots a valid path has been found, the collection of these paths is a solution for the multi-robot problem. This is a so-called *prioritized* approach. It is very fast, but not complete, as the robots are not able to coordinate their motions, which is necessary in some situations. However, in most practical settings, the prioritized approach works perfectly. The offline planner presented in Chapter 3 is pre-eminently suitable for prioritized multi-robot planning. This application is discussed in Chapter 4.

In Chapter 5, an online planner for partially known environments is presented. To deal with real-time issues, the planner features the anytime-characteristic, meaning that initially a complete path is planned quickly, which is of poor quality though. The quality of the path is improved upon as deliberation time allows. When changes are observed in obstacle trajectories, a new path must be planned. A main goal in this work, is to reuse information from

previous plans in order to update a new path fast, and hence ensure high quality output. Previous works do not reuse information, but rather plan a new path from scratch [124, 154]. The strategy chosen combines a PRM-representation of the static configuration space, with a variant of a *D** algorithm [104] to plan and replan in the roadmap-time space. There is a strong body of work on D*-like planners [146, 92], which have been used extensively to replan in static environments where changes may be observed on stationary obstacles (e.g. due to imprecise maps). In our work the aim is to extend this application to dynamic environments where changes are observed on moving obstacles (e.g. in the form of course changes).

In Chapter 6, we discuss a planner for completely unpredictable moving obstacles. The planner is in principle offline, but it can be used online as well, as it is designed specifically to overcome the fundamental difficulties that were mentioned in Section 1.3.4 inherent to planning in partially known environments. To this end, the future trajectories of the moving obstacles are estimated very conservatively. We assume that the obstacles are discs and have a known maximal velocity. Hence, the regions in which the obstacles might be are discs that grow over time. The goal in this work is to find the shortest paths avoiding these growing discs. Such paths are inherently safe, regardless of what the moving obstacles do. The problem translates into a geometric problem, for which an efficient algorithm and a fast implementation is presented. In order to keep the growing discs small, paths should be replanned continuously based on newly acquired sensor data. Also, to be able to circumvent the growing discs, the robot should move faster than all of the moving obstacles.

In the previous section we mentioned that a PRM-roadmap cannot be used directly in a configuration-time space of a (known) dynamic environment. This is because the premise that the roadmap can be used multiple times does not hold in transitory configuration-time spaces. This is unfortunate, as PRM's do have one appealing property, namely that queries can be answered very quickly, and within an accurately estimated amount of time. Hence, choosing the value of τ in a real-time setting (recall that τ is the amount of time in which a path must be planned) would be very easy. There is a class of dynamic environments, however, for which the premise *does* hold, and in which we *can* use a preprocessed roadmap and exploit all of its appealing properties. This class of environments contains the so-called *repetitive* environments, in which the motions of the obstacles are periodic. That is, after some period the obstacles return to their initial configuration and repetitively execute the same motion again. A simple example is an automated revolving door. In Chapter 7, we present a framework for path planning in such environments.

1.5.2 Corridor Planning

A good corridor backbone path is short, yet keeps a clearance of $w_{\max}$ from the obstacles. Obviously, if the preferred clearance $w_{\max}$ can be obtained everywhere, a sensible corridor backbone would be the shortest path extracted from the visibility graph of the obstacles which are enlarged by a layer of width $w_{\max}$. When the preferred clearance of the corridor cannot be obtained, the maximal width is locally obtained on the Voronoi diagram of the obstacles. Such portions of the Voronoi diagram can be connected to the visibility graph of the enlarged obstacles to produce a graph containing corridors for all homotopy classes

in the scene. We call this graph the visibility-Voronoi diagram for preferred clearance $w_{\max}$, and it is introduced in Chapter 8. Further, we present a construct called the visibility-Voronoi complex, which is a data structure containing all visibility-Voronoi diagrams for all values of $w_{\max}$, which can directly be queried for a suitable corridor (for some value of $w_{\max}$) without explicitly constructing the visibility-Voronoi diagram for the particular value of $w_{\max}$.

The visibility-Voronoi complex provides an efficient data structure encoding corridors. These corridors however, do not optimize a sensible (mathematical) quality criterion. In Chapter 9, we introduce a natural measure for the quality of corridors, trading off the length of the corridor against the width of the corridor (within the preferred width $w_{\max}$), and explore properties of corridors which are optimal according to this measure. Also, we present an approximation algorithm for planning near-optimal corridors.

Chapter 2

Preliminaries

In this chapter, we explain some elementary methods we will use throughout this thesis. First, we describe Dijkstra's algorithm [44], that finds shortest paths in a graph from a single source to all nodes in the graph. An extension to Dijkstra's algorithm is A* [70], that speeds up the calculation of a shortest path in case one is only interested in a path between a single source and a single goal.

Second, we will present the basics of the Probabilistic Roadmap Method (PRM), which is a well known and much studied method for path planning in configuration spaces of arbitrary dimension [89, 12, 149]. Many extensions have been proposed to improve the performance of the PRM in general, or in specific cases. We will discuss them briefly. The PRM method is referred to in many chapters of this thesis.

2.1 Shortest Path Algorithms

2.1.1 Dijkstra's

If one is given a graph $G = (V, E)$ consisting of a set of vertices V and a set of edges $E \subset V \times V$, and for each edge $(u, v) \in E$ an associated nonnegative cost $c(u, v)$, Dijkstra's algorithm is able to find shortest paths from a single source vertex $u_{\text{start}} \in V$ to all vertices in V. This algorithm is used in many path planning algorithms and throughout this thesis to find shortest paths in graphs (or grids, which in fact are graphs as well) that for instance represent roadmaps of valid motion paths through an environment. The cost associated with each edge is often the distance of the motion which it represents, but other criteria such as clearance, safety, etc., can be incorporated into the cost function as well.

Dijkstra's algorithm works by maintaining for each vertex u the shortest path distance $g(u)$ from the start vertex. Further, a backpointer $bp(u)$ is maintained for each vertex indicating from which neighboring vertex the shortest path from the start comes. Hence, a shortest path to some vertex can be read out by following the backpointers back to the start vertex. Initially, of all vertices the shortest path distance is set to infinity, except for the start vertex whose distance is set zero. From the start vertex, the shortest path distances are prop-

agated through the graph until all vertices have received their actual shortest path distance. To this end, the start vertex is inserted into a priority queue, with a key equal to its shortest path distance (zero in this case). In each iteration of the algorithm, the minimal element u is popped from the priority queue, and of all neighbors of u it is checked whether their shortest path distance can be decreased by changing their backpointer to u. If the distance of some vertex is decreased, it is inserted (or updated) in the priority queue, as it may enable other vertices to decrease their distances. When the priority queue becomes empty, all vertices are guaranteed to have their actual shortest path distance and correct backpointer associated with it. The algorithm is shown below (Algorithm 2.1).

Algorithm 2.1 DIJKSTRA

```
1:  for all u ∈ V do
2:      g(u) ← ∞
3:  g(u_start) ← 0
4:  Insert u_start into Q
5:  repeat
6:      u ← element from Q with minimal g(u)
7:      Remove u from Q
8:      for all neighbors v of u do
9:          if g(u) + c(u, v) < g(v) then
10:             g(v) ← g(u) + c(u, v)
11:             bp(v) ← u
12:             Insert or update v in Q
13: until Q = ∅
```

When the priority queue Q is implemented using a straightforward balanced binary sorted tree (where the elemenatary operations, such as insertion and deletion, take time logarithmically in the size of the tree) the running time of Dijkstra's algorithm is $O(E \log V)$ if E is the number of edges and V the number of vertices. The optimal running time is $O(V \log V + E)$ and can be achieved if Q is implemented using a Fibonacci heap [54]. In practice, the first implementation is often used, as is the case in the experiments we perform in this thesis. As it is a combinatorial algorithm, it runs fast compared to other, geometric operations in our algorithms that for instance need collision checks. For reasonably sized roadmaps (1000-10000 vertices) Dijkstra's algorithm only takes in a few milliseconds of computation time in general.

2.1.2 A*

Despite the speed of Dijkstra's algorithm, there are cases in which it is desirable to optimize the performance of finding shortest paths (for instance when the roadmap is very large, or the application is very time-critical). Dijkstra's algorithm finds shortest paths to all vertices in the graph, but often one is only interested in a shortest path to a specific goal vertex u_{goal}. In this case the A* method, which is based on Dijkstra's algorithm, is favorable. A* focuses the search in the graph towards the goal, whereas in Dijkstra's algorithm the shortest paths

distances are propagated breadthwise. Still, A* is guaranteed to find the optimal path from the start vertex to the goal vertex.

There are two differences between Dijkstra's and A*. The first concerns the key with which the vertices are sorted in the priority queue. In Dijkstra's this key equals the current shortest path distance to the start, which is $g(u)$ for a vertex u. In the A* method a heuristic value $h(u)$ is added to $g(u)$ when computing u's key. This heuristic value must be a lower-bound estimate of the distance from vertex u to the goal vertex, so that the key of a vertex u is a (lower-bound) indication of the length of the shortest path between start and goal to which vertex u contributes. The second difference is that the algorithm stops when the goal vertex is popped from the queue, rather than when the priority queue Q has become empty (see Algorithm 2.2).

Algorithm 2.2 A*

```
1:  for all u ∈ V do
2:      g(u) ← ∞
3:      h(u) ← lower-bound estimate of the distance between u and u_goal
4:  g(u_start) ← 0
5:  Insert u_start into Q
6:  repeat
7:      u ← element from Q with minimal (g(u) + h(u))
8:      Remove u from Q
9:      for all neighbors v of u do
10:         if g(u) + c(u, v) < g(v) then
11:             g(v) ← g(u) + c(u, v)
12:             bp(v) ← u
13:             Insert or update v in Q
14: until u = u_goal or Q = ∅
```

If the heuristic value $h(u)$ is guaranteed to be less than or equal to the actual distance from u to the goal for all u, the heuristic is said to be admissible, and it is guaranteed that the shortest path found to the goal vertex is actually optimal. For instance, for a graph embedded in the plane, where edge costs indicate Euclidean distances between the vertices, an admissible heuristic is the direct Euclidean distance to the goal (any other path to the goal will be longer). For two heuristic functions h_1 and h_2, h_1 is said to be more informed if $h_1(u) > h_2(u)$ for all vertices u. The least informed, yet admissible heuristic is $h(u) = 0$ for all u. This heuristic would make A* equal to Dijkstra's. The better informed a heuristic is, the better the performance of the A* method [117]. In many applications, a well informed heuristic can be used, which increases the performance of the shortest path algorithm considerably, when compared to Dijkstra's. Asymptotically however, the running time of Dijkstra's and A* is equal.

If the heuristic is an overestimate of the actual distance to the goal vertex, it is not guaranteed that the path found to the goal is the actual shortest path. However, if the heuristic is guaranteed to be less than or equal to k times the actual distance to the goal, the found path is guaranteed to be no more than k times longer than the actual shortest path [122]. This

can be used to further increase the performance of finding a path, if it is not so important whether it is actually the shortest.

2.2 Probabilistic Roadmap Methods

The methods described above are able to find a shortest path on a given graph. The issue most path planning methods are dealing with is how to create such a graph, or roadmap. To be useful for path planning applications, the roadmap should represent the connectivity of the free configuration space well, and cover the free configuration space such that any query configuration can easily be connected to the roadmap. See Chapter 1 for various approaches for creating such roadmaps. We will describe the most popular method in some more detail here: the Probabilistic Roadmap Method (PRM). PRM is a probabilistically complete method that is able to solve complicated path planning problems in arbitrarily high-dimensional configuration spaces. It is used throughout this thesis.

The invention of PRM is usually attributed to Kavraki and Latombe [87], and Overmars [120], who worked independently on the method in the early nineties. The first appearance of the idea in literature, however, was by Glavina in 1990 [66].

2.2.1 The Basic PRM Approach

The basic PRM-method is surprisingly simple. It constructs a roadmap by iteratively sampling configurations randomly from the configuration space. Using a collision-checker, it can be determined whether a configuration belongs to the free- or forbidden configuration space. If a configuration is collision-free, it is added as a node to the roadmap. Subsequently, it is attempted to connect this node to other nodes already present in the roadmap. To save time, a connection is only tried to nodes which are close (e.g. to all nodes within a certain distance $d_{\max}$, or to the k nearest neighbors). The set of nodes to which a connection is attempted is called the *neighbor set* of the current node.

Connections between nodes are tried using a *local planner*, which is a simple planner that is allowed to fail on all but the simplest queries. In the basic PRM implementation, the local planner simply tries to connect two configurations by a straight line through the configuration space. The local planner succeeds when the straight-line is collision-free, which is determined by collision-checking intermediate configurations, up to some predefined resolution. If the connection succeeds, an edge between the two associated nodes is added to the roadmap.

If a connection has been tried to all nodes of the neighbor set, a new node is sampled, and the process repeats. This continues until some application-specific stop-criterion is met. Usually this is when some predefined set of query configurations are inter-connected via the roadmap. In Algorithm 2.3, a generic sketch of the algorithm is given.

If a roadmap has been constructed for a particular scene, it can be queried for motion paths between pairs of configurations in the environment. This is done by connecting the two query configurations to the roadmap, as if they were nodes newly inserted into the roadmap in the construction phase. That is, the local planner is used to connect each of the query configurations to nodes already present in the roadmap. If this succeeds, a path

Algorithm 2.3 PRM

```
repeat
    c ← a random configuration in C
    if c ∈ C_free then
        V ← V ∪ {c}
        N ← a subset of V containing nodes neighboring c
        for all c' ∈ N do
            if the local planner connects c and c' then
                E ← E ∪ (c, c')
until some stop-criterion is met
```

is searched for in the roadmap using Dijkstra's algorithm or A*, as we discussed above. The idea of a PRM is that it is a multi-query data structure, which takes some time to build in a preprocessing phase, but allows for very fast answering of as many queries as desired. As paths deduced from a PRM are usually not very visually attractive, they are often smoothed in a post-processing stage (see e.g. [59]).

2.2.2 Details and Variants

The above algorithm describes PRM in its most basic and general form. One important choice has been made already though: we chose the local planner to connect two configurations using a straight line through configuration space, but other types of local planners are possible as well. Also, a number of details have yet to be filled in. For instance, one has to choose a scheme to select potential neighbors for each added node to which a connection is attempted, and one has to choose a sampling scheme to get a desired distribution of samples over the configuration space. We will discuss these one by one.

Local planner: A local planner should be able to find a path between two configurations in 'simple' cases in a 'small' amount of time. Given a configuration and a local planner, one can define the set of configurations in the configuration space to which a local planning attempt will succeed. Let this set be called the visibility region of a node under a certain local planner [60]. The larger this visibility region is, the more powerful the local planner, but often this inherently means that the local planner is more time-consuming. So, one has to find a balance between these quantities. Extensive experimentation has indicated that it is more important for a local planner to be fast, than to be very intelligent (and has a high success rate of connecting samples) [60]. Therefore, in most cases, the local planner simply tries to connect two samples by a straight line though the configuration space. Some other works use a potential field method to connect two configurations [60]. A potential field local planner has a larger visibility region, but it is more expensive such that it does *in general* not outperform the simple straight-line local planner. A global consensus has settled in the PRM community to use the straight-line local planner [58], also because of its ease of implementation and its general applicability.

Neighbor set: When attempting to connect a node to other nodes, one has to choose a set of neighbors that is considered for connection. The most straightforward choice is to try

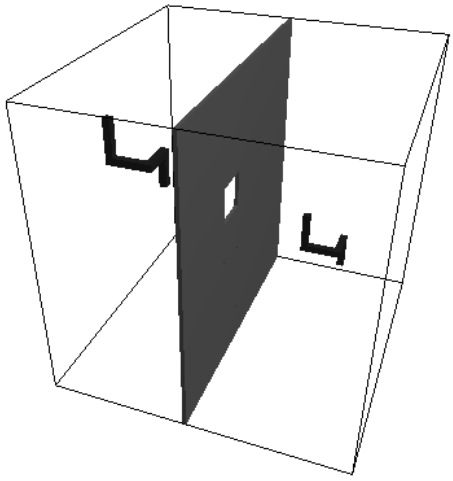


Figure 2.1: A difficult narrow passage problem.

to connect to all nodes present in the roadmap, but this is very time-consuming. Instead, in many cases only the k nearest neighbors are tried, or only neighbors that lie no further than a distance $d_{\max}$ from the current node. The rationale behind this is that the probability that a connection succeeds decreases as the distance between the nodes increases. And one does not want to spend too much time on failed local planner attempts. Another important observation is that it might not be useful to attempt a connection to nodes that are already in the same *connected component* of the roadmap as the current node. In this case a path between the nodes is already available in the roadmap, and hence an attempt to connect the nodes directly will not increase the connectivity of the roadmap. To exploit this optimization fully, it is important to consider the neighbors in order of increasing distance. A drawback is that roadmaps created using this optimization do not contain cycles. Hence, no alternative routes are available in the roadmap. This is remedied in [115], in which a method is presented that does add edges between two nodes from the same connected component, but only when the indirect connection is more than K times longer than the direct connection, for some value of K.

Sampling scheme: A large amount of research on PRM has been devoted to choosing a good distribution of samples over the configuration space. The most straightforward method is to sample configurations *uniform randomly* over the configuration space. However, the standard PRM suffers from the well-known *narrow passage problem*. That is, it is very difficult for PRM to connect two connected components through a narrow passage (see Fig. 2.1). This is because relatively many configurations are (superfluously) sampled in open regions while it is desirable that more samples are picked inside the narrow passage. This has lead to many different sampling strategies that favor the sampling of configurations in narrow passages. As it is difficult to detect what a narrow passage is, most of these works

just pick more samples close to obstacles, such as OBPRM [7, 6] and Gaussian PRM[30]. Other works really try to find out where narrow passages are, such as Bridge Test Sampling [74], Watershed labeling [21], etc. [95, 134]. Yet other works sample far away from the obstacles, on the medial axis of the free configuration space, to increase both the probability of sampling inside narrow passages, and the probability that the local planner succeeds [163, 164, 73]. Work has also been done on local planners to improve the performance of PRM in the presence of narrow passages [78]. Despite the amount of work done on this subject, the various methods only seem to succeed in improving the performance of PRM in very specific academic examples. Schemes that uniformly distribute the samples are still the most popular, as they are the easiest to implement and have the widest applicability.

Many variants on the basic PRM method have been proposed, such as Visibility PRM [142], Fuzzy PRM [112], Lazy PRM [29], Probabilistic Roadmap of Trees (PRT) [125], Probabilistic Cell Decomposition [106], etc. They all rely on the same underlying concepts, and aim at improving specific aspects of PRM.

2.2.3 Analysis

PRM is probabilistically complete (the optimizations described above do not affect this, as long as each ball with positive radius in the configuration space has a positive probability of receiving a random sample). That means, if a path between two query configurations actually exists, the probability that a path is present in the roadmap converges to 1 as the number of sampled nodes approaches infinity. So, if a path exists, it will eventually be found. If a path does not exist, however, the PRM method will never be able to report this with certainty. Although the property of probabilistically completeness is rather theoretical and has little implications in practice, it is still an important property. It shows that the method is generally applicable, and not only works in specific cases. A more important property is the rate with which the probability that a path exists converges to 1 as the number of nodes increases. Under various assumptions, it has been shown that this goes exponentially fast [77, 88, 76], that is, the probability that a path is *not* present in the roadmap decreases with some constant factor every time a new sample is added to the roadmap.

The fast convergence rate does not completely explain the popularity of PRM. The problem with path planning algorithms is that their running time increases exponentially with the dimension of the configuration space. In worst case scenarios this must also hold for PRM, as the path planning problem has been shown to be PSPACE-hard [127]. However, the fact that samples can be connected to each other by the local planner, even if they are far apart (they only need to be inter-visible), makes that PRM works very well in most practical situations. In most theoretical analyses (e.g. [148]), this property is not used. It only assumes that each sample has a ball of positive radius around it containing points it can see. With such an assumption, or in practice when the maximal neighbor distance $d_{\max}$ is chosen too small, it is inevitable that the running time is exponential in the dimension in the worst case, as the minimal number of balls with radius $r < 1$ required to cover the cube $[0,1]^d$ is $O((1/r)^d)$, that is, exponential in the dimension.

Despite the above considerations the actual theoretical running time of PRM is un-

bounded; it is not *guaranteed* that a connection will be found after sampling a number of samples exponential in the dimension. Recently, it is shown in [38] that for configuration spaces with constant dimension, PRM has a *smoothed running time* [144] that is polynomial in the combinatorial complexity of the input.

Another issue that has gotten attention in the literature was whether the randomness involved in the sampling process causes PRM to be efficient [76, 97]. This appears not to be the case. A number of *deterministic* number sequences have been proposed to generate a nice and evenly spread series of samples [31, 98]. These sequences have been proved to perform just as well, if not better, than a straightforward uniform randomly picked sequence of samples [61].

Part I

Planning in Dynamic Environments

Chapter 3

Planning in Known Dynamic Environments

In this chapter a new offline method is presented for path planning in known dynamic environments, that is, finding a collision-free path for a robot in a scene consisting of both static and dynamic, moving obstacles of which the trajectories are known. We follow a practical, decoupled approach consisting of two stages. In the first stage a roadmap is created for the configuration space of the robot that is collision-free with respect to the stationary obstacles in the environment. In the second stage the motion of the robot is coordinated on this roadmap, such that the robot does not collide with any moving obstacle. Our method plans (approximately) time-optimal paths between any given start and goal configuration in the roadmap. These are found by performing a two-level search for a shortest path. On the local level, motions are coordinated along single edges of the roadmap, using a depth-first search in an implicit two-dimensional grid of configuration-times on the edge. On the global level these local paths are coordinated using an A*-search to find a global path to the goal configuration. The approach is applicable to any robot type in configuration spaces with any dimension, and the trajectories of the moving obstacles are unconstrained, as long as they are known beforehand. The approach has been implemented for both free-flying and articulated robots in three-dimensional workspaces. Experiments show that the method achieves interactive performance in complex environments.

This chapter has previously been published as: J. van den Berg and M. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics*, 21(5):885–897, 2005 [20]. An extended abstract has appeared as [18].

3.1 Introduction

Path planning is of great importance, not only in robotics, but also in other fields such as virtual environments, maintenance planning and computer-aided design. Much research

has been done on path planning in static environments and both exact and approximate methods have been devised [96, 97, 41]. A popular approximate method is the probabilistic roadmap planner (PRM) [89, 149]. It is a generic method that creates a roadmap in a preprocessing phase that represents the connectivity of the free configuration space. Individual motion planning problems can then be solved quickly by finding a path in the roadmap. The method has successfully been used in high-dimensional configuration spaces of complex environments.

The extension of the path planning problem to dynamic environments has been extensively studied as well [11, 48, 49, 53, 55, 56, 75, 129], but only a limited number of practical algorithms have been devised that deal genericly with moving obstacles. PRM could be extended to the dynamic motion planning problem by incorporating the absolute notion of time as an additional dimension in the configuration space. However, since the obstacle motions are not assumed to be periodic (cyclic), the configuration space is highly transitory.

As a consequence, building a roadmap during a preprocessing phase is not useful for such configuration spaces. Therefore, single-shot variants of PRM [100] have been the methods of choice for this type of problems [11, 75]. In such methods a roadmap is built incrementally in the form of a directed tree oriented along the time axis for each planning query. Although some promising results have been achieved in real world situations, these methods are less suitable in large scenes in which besides moving obstacles, a large number of stationary obstacles is present. This is because all the effort has to be done in the query phase, which undermines the often required real-time performance of the method.

In this chapter, we propose a new method which is based on a roadmap built in a preprocessing phase. The roadmap is built for the static part of the scene without the moving obstacles and without the additional dimension for time. This can be done using a standard PRM method, but devising a roadmap on the drawing table may suffice just as well. In the query phase then only the moving obstacles need to be dealt with.

Our method searches for a near-time-optimal path in the "roadmap-time" space between a start and a goal configuration in the roadmap, without collisions with the moving obstacles. Previous techniques, for example [53], avoid the moving obstacles by coordinating the robot motion along a pre-planned *path* that is collision-free with respect to the stationary obstacles. We extend this approach to a roadmap, which considerably enlarges the maneuverability of the robot and, hence, the chance that a valid path is found.

We use a two-level approach to find a path. On the local level, paths on single edges of the roadmap are found in an implicit (two-dimensional) grid of configuration-times along the edge. The discretization is similar to the one used in [53], but we use a more efficient *depth-first* search strategy to find a path. On the global level, the local paths are coordinated using an A*-like search to find a near-time-optimal global path in the entire roadmap.

Our method uses principles similar to those of a theoretical method of Fujimura [56], which computes an exact time-optimal path in a roadmap. On the local level however, his method uses visibility graphs in the configuration-time space. This requires the configuration-time space to be constructed explicitly, and therefore his method works only for point robots in two-dimensional environments where the moving obstacles are constrained to piecewise linear motions without rotation. To our knowledge, the method was never implemented or used in practice.

By choosing an approximate approach, we were able to lift these drawbacks. Our method is practical and applicable to any robot type in configuration spaces with any dimension. The only ingredient the method requires is a roadmap for the robot amidst the stationary obstacles. The shape and motions of the moving obstacles are completely free: they may move with any speed following any trajectory, may deform and even jump (warp), as long as the motions are known beforehand. That is, given a position of the robot at a time t we must be able to answer the question whether the robot is collision-free. As in [56] we do not put constraints on the robot's motion, except for an upper bound on its velocity.

The method has been implemented for both free-flying robots and articulated robots with six degrees of freedom, and various experiments show that the method achieves interactive performance in confined dynamic environments. One of the main applications of motion planning in dynamic environments is multi-robot motion planning [47]. This will be discussed in the next chapter of this thesis.

The rest of the chapter is organized as follows. A formal definition of the problem is given in Section 3.2. In Section 3.3 we describe the global approach of our method. The problem is split up in two parts: finding local paths on single edges of the roadmap and finding a global path through the entire roadmap. These will be discussed in Sections 3.4 and 3.5 respectively. In Section 3.6 some extensions and optimizations to the algorithm are discussed and Section 3.7 describes the experimental results.

3.2 Problem Description

3.2.1 The Roadmap

Our method requires a roadmap that is constructed for the static part of the scene, that is, its set of vertices and edges in the configuration space $\mathcal{C}$ must be collision-free with respect to the *static* obstacles (see Fig. 3.1(a)). This means that the roadmap can be constructed in a preprocessing phase. The start and goal configurations are assumed to be present in the roadmap as vertices. If not, they can be connected to the roadmap in the query phase. Our method is applicable to both directed and undirected roadmaps, but we confine ourselves the more general case of undirected roadmaps here.

The idea of using a preprocessed roadmap is that during the query phase, the stationary obstacles do not need to be considered in collision checks, which saves a large amount of time. Moreover, narrow passage problems raised by the stationary obstacles are solved in the preprocessing phase, which substantially relieves the query phase. Actually, in the rest of this chapter we can simply ignore the stationary obstacles.

Also, the search space for feasible paths is substantially reduced; the configuration space is basically brought down to a one-dimensional structure, which makes the problem tractable. If the roadmap given is well covering the free part of the static configuration space, this reduction should hardly affect the chance that a path is found.

The use of a roadmap may have practical advantages as well. In many real-world environments, such as factory floors, sea- and airports, etc., the autonomic robots present are constrained to move along prespecified networks of paths (for instance along lines painted on the floor). They can be modeled perfectly into a roadmap [56].

The quality of the path computed by our method depends directly on the quality of the roadmap. Therefore, using a roadmap containing smooth, natural paths is preferred [113]. The creation of roadmaps is not the topic of study in this chapter. Many techniques exist for this, for example the PRM approach. To have a choice of alternative paths it is though important that the roadmap contains cycles (see e.g. [115]).

3.2.2 The Problem

The problem is formally defined as follows. Let A be a robot with an upper bound $v_{\max}$ on its velocity operating in a two- or three dimensional environment. For the robot A, a roadmap R is constructed that covers the free configuration space $\mathcal{C}$ of A (with respect to the stationary obstacles in the environment). We use the notation $A(x)$ to refer to robot A configured at x, where $x \in R$. Let there further be given a set of moving obstacles O, which state at time t is denoted $O(t)$.

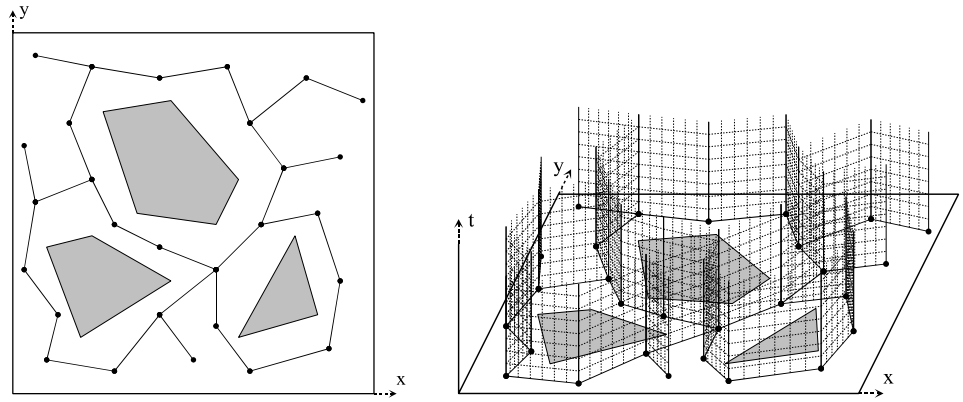
Let $s, g \in R$ be the start and goal configuration of the robot, respectively, and let $t_0 \in \mathbb{R}$ be the start time. Then the task is to compute a path $\pi : [t_0, T] \rightarrow R$, such that $\pi(t_0) = s$ and $\pi(T) = g$, without collisions with the moving obstacles, i.e. $\forall (t \in [t_0, T] :: A(\pi(t)) \cap O(t) = \emptyset)$. (Note that the stationary obstacles are not of concern; the robot moves over a roadmap that is already guaranteed to be collision-free with respect to the stationary obstacles.) Further, the path π should obey the maximum velocity constraint $v_{\max}$ of the robot, i.e. $\forall (t_1, t_2 \in [t_0, T] : t_1 < t_2 : d(\pi(t_1), \pi(t_2)) \leq v_{\max}(t_2 - t_1))$, where $d : R \times R \rightarrow \mathbb{R}$ is the distance between two configurations in the roadmap. T can be considered as the arrival time of robot A at its goal configuration, which should be minimized. That is, we are looking for a path arriving at the goal configuration as soon as possible.

To give maximal flexibility to the moving obstacles, the only way in which the configuration-time space is sensed is by means of a boolean function $cf(c, t)$ that, given a configuration $c \in \mathcal{C}$ and a moment in time t, reports whether the robot configured at c collides with any moving obstacle at time t.

3.2.3 Discretization

We discretize the problem by choosing a small time step Δt, and a principal velocity v_p within the velocity bound $v_{\max}$. The actual velocity v is constrained to be either v_p, 0 or $-v_p$ and may only change at given times $k\Delta t$, where k is an integer. This subdivides each of the edges of the roadmap in small steps of length $v_p \Delta t$. For each edge, we choose v_p to have the largest value smaller than $v_{\max}$ such that $\ell / v_p \Delta t$ is an integer, where ℓ is the length of the edge. This means that each edge is subdivided exactly in an integer number of steps. At each time step the robot may move one step in either direction along the edge, or halt at its current position. If the robot is on a vertex of the roadmap, it can choose among all of the outgoing edges of the vertex. (Such a discretization is common in path planning [53, 99].)

Thus, the discretized search space of the problem (i.e., the "roadmap-time" space) can be visualized as a complex of two-dimensional (configuration-time) grids along each of the edges, which are connected in the vertices of the roadmap (see Fig. 3.1(b)). Each grid point is either collision-free, or inaccessible due to a moving obstacle. The possible steps the robot



(a) (b)

Figure 3.1: (a) An example roadmap that is collision-free with respect to the stationary obstacles. (b) The discretized "roadmap-time" space, which is a complex of two-dimensional grids along each of the edges, forms the search space of the problem.

can take define a directed (acyclic) graph on the complex of grids, in which valid paths can be found.

3.3 Global Approach

A straightforward approach to find a path from s to g, would be to use an A*-search directly on the discretized "roadmap-time" space. However, in this chapter, we will present a method that performs better, by subdividing the search into two levels: on the local level, paths are found on individual edges of the roadmap using an efficient depth-first search, and on the global level these are coordinated using an A*-search to find a global path to the goal.

Let us globally introduce our method using a simple example roadmap (see Fig. 3.2). It consists of three vertices s, u and g, and edges between s and u, and u and g (solid lines in the figure). It takes one unit of time for the robot to traverse each edge. An obstacle (gray disc in the figure) is moving over g at time $t = 1$ to u at $t = 2$ and then moves away from the roadmap (dotted path in the figure). If the robot starts at s at $t = 0$, it is able to reach u at $t = 1$, but then it is not possible to reach g, because the path is blocked by the moving obstacle. If the robot would wait somewhere on the edge (s,u) and arrive at u at $t > 2$, the path to g is free. It would, however, not be useful to arrive even later at u, because the robot would then arrive in the same *free interval* on u. A free interval on a vertex u is defined as follows:

Definition 3.1. *A free interval on a vertex u is a maximal continuous segment in time in which the robot configured at u is collision-free.*

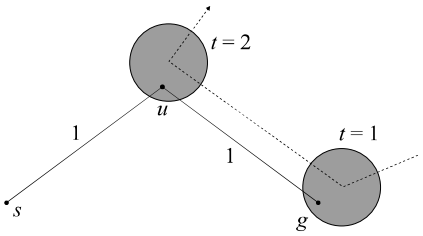


Figure 3.2: A small example roadmap with two edges (solid lines), and a moving obstacle (gray disc) moving over the dotted path. A point robot moving from s to g has to wait until the moving obstacle has cleared u, before advancing to g.

It is easy to see that it is not useful to arrive later in the same free interval. If the robot arrives early at the interval, it can wait on the vertex for the rest of the free interval, so arriving later in the same interval does not extend the possibilities of reaching g. This observation is crucial for the method presented. In fact, the problem can be expressed fully in terms of the free intervals on the vertices by modeling each free interval on a vertex as a *node* in an implicit (directed) tree, which we will call the *interval tree*. The interval tree is rooted at the interval on the start vertex s around the start time t_0, and branches exist in the tree from one node to the other when there is an edge between the corresponding vertices in the roadmap and an appropriate path exists between the associated intervals.[1] A (global) path between the start and goal vertex is contained in the interval tree.

Our approach uses the interval tree to find a path toward the goal vertex. We do not compute the interval tree explicitly, but explore it in a lazy fashion by sending so called *probes* through the roadmap. Probes search for paths between free intervals on neighboring vertices (i.e. branches in the tree). We call such paths *local paths*.

We will describe our algorithm in two stages. The first deals with the behavior of a single probe, i.e. computing a feasible local path on a single edge of the roadmap using a depth-first search. In the second stage we discuss the global probe management, in which we search the interval tree using an A*-like search to find a *global* path to the goal vertex.

In the example of Fig. 3.2 only normal local paths were considered, i.e. paths that originate at one vertex of an edge and advance to the other vertex of the edge. However, a second type of local path has to be taken into account as well: paths returning to a later free interval on the same vertex they originate from. They first move away from the vertex along an edge to make room for a moving obstacle after which they return to the vertex.

So in the search for a global path toward the goal vertex, two types of local paths must be considered; those that move to the other end of the edge and those that return to the same vertex. To distinguish between them they are called *advancing* and *returning* paths respectively.

[1] Throughout this chapter, we will use the terms vertices and edges when we refer to the roadmap, and nodes and branches when we refer to the interval tree.

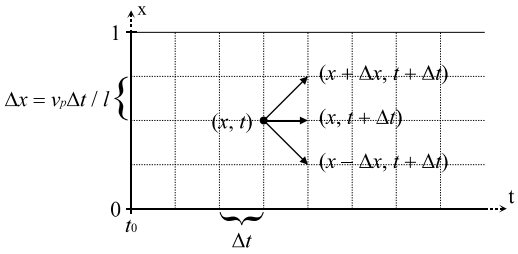


Figure 3.3: A configuration-time grid of a roadmap edge. It shows which neighbors are reachable from a given configuration-time $\langle x, t \rangle$.

3.4 Local Paths

In this section we discuss how an approximately time-optimal local path is computed along a single edge of the roadmap. We assume that the roadmap edges are undirected. For conceptual clarity though, they are considered as two separate 'directed' edges in the rest of this chapter, such that each edge has its own *source* and *destination vertex*.

3.4.1 The Configuration-Time Grid

Since we consider paths along an edge of the roadmap, a configuration of the robot is reduced to a single variable representing the distance traveled along the edge. We denote this variable by x. It ranges from 0 to 1, where $x = 0$ and $x = 1$ respectively correspond to the configurations on the source and destination vertex of the edge. The resulting configuration-time space is two-dimensional and consists of pairs $\langle x, t \rangle \in [0,1] \times [t_0, \infty)$. Obstacles in the configuration-time space may have any shape, since we do not constrain their motions.

The configuration-time space is discretized as explained in Section 3.2.3, which results in a regular two-dimensional grid with spacings Δt along the time axis and $\Delta x = v_p \Delta t / \ell$ along the configuration axis. We divide $v_p \Delta t$ by ℓ here to transform the actual covered distance to the appropriate step Δx on the range $[0,1]$. Recall that v_p has the largest value smaller than $v_{\max}$ such that $1/\Delta x$ is an integer, so that the destination vertex of the edge can be reached exactly in an integer number of steps.

From a given configuration-time $\langle x, t \rangle$ in the grid, three other configuration-times are reachable: $\langle x + \Delta x, t + \Delta t \rangle$, $\langle x, t + \Delta t \rangle$ and $\langle x - \Delta x, t + \Delta t \rangle$. They are associated with the choosing the velocity at v_p, 0 or $-v_p$, respectively (see Fig. 3.3). This defines the connectivity on the configuration-time grid in which the local paths can be found.

Moving obstacles may cause some configuration-times in the grid to be inaccessible, which means that we have to determine whether they are free using a collision-checker. As the grid is infinitely large in the time dimension, we do not do this during preprocessing, but determine the collision-status of a configuration-time only when it appears during the search for a path that this information is required.

3.4.2 Finding a Local Path

The problem of finding a local path is defined as follows. Given an edge and an initial configuration-time $\langle x_s, t_s \rangle$, find a path in the grid to the first reachable free interval on the destination vertex. The destination vertex is reached when $x = 1$. In case of an advancing path, $x_s = 0$, and in case of a returning path $x_s = 1$. To prevent that the algorithm immediately returns success in the latter case, we state that the destination vertex must be reached in an *unvisited* free interval. (How to determine whether an interval has been visited is discussed in Section 3.5.4, as it relates to the global algorithm.)

An approximately time-optimal local path can be found by finding a shortest path from $\langle x_s, t_s \rangle$ to $x = 1$ in the configuration-time grid. In [53] an A*-algorithm is used to find the shortest path, but in our case it can be implemented more efficiently using a *depth-first search*; this requires less collision checks (as we will see shortly) and the elementary operations on the data structure (a stack instead of a priority queue) are cheaper.

The algorithm is initialized with the configuration-time $\langle x_s, t_s \rangle$ on the stack. In every loop the top element $\langle x, t \rangle$ is popped from the stack. If the corresponding configuration-time has not been visited before and if it is collision-free with respect to the moving obstacles, the reachable grid points $\langle x - \Delta x, t + \Delta t \rangle$, $\langle x, t + \Delta t \rangle$ and $\langle x + \Delta x, t + \Delta t \rangle$ are pushed onto the stack *in this particular order* (see Algorithm 3.1). This means that the most promising step (advancing toward the destination vertex), which is pushed last on the stack, is considered first. The algorithm runs until the stack is empty or the configuration $x = 1$ has been reached in an unvisited free interval on the destination vertex (see Algorithm 3.2). Backpointers and information about whether a configuration-time has been visited before, are maintained.

We assume that the time step Δt is chosen small enough such that collision checking each of the neighboring configuration-times is enough to determine whether the traversal between them is collision-free. Such an approximation is common in path planning [149].

Algorithm 3.1 DoStep()

```
1:  ⟨x, t⟩ ← StackPop()
2:  if not ⟨x, t⟩.visited and cf(⟨x, t⟩) then
3:      StackPush(⟨x − Δx, t + Δt⟩)
4:      ⟨x − Δx, t + Δt⟩.backpointer ← ⟨x, t⟩
5:      StackPush(⟨x, t + Δt⟩)
6:      ⟨x, t + Δt⟩.backpointer ← ⟨x, t⟩
7:      StackPush(⟨x + Δx, t + Δt⟩)
8:      ⟨x + Δx, t + Δt⟩.backpointer ← ⟨x, t⟩
9:      ⟨x, t⟩.visited ← true
10:     return ⟨x, t⟩
11: else
12:     ⟨x, t⟩.visited ← true
13:     return NULL
```

Fig. 3.4 shows the working of the depth-first algorithm in an example configuration-time grid. The thin arrows indicate the space explored by the algorithm and the thick arrows form the resulting path. It is guaranteed that the depth-first search method will not explore

Algorithm 3.2 FINDLOCALTRAJECTORY($\langle x_s, t_s \rangle$)

1: STACKPUSH($\langle x_s, t_s \rangle$)
2: **repeat**
3: $\langle x, t \rangle \leftarrow$ DOSTEP()
4: **until** ($x = 1$ **and** the interval on the destination vertex at time t is unvisited) **or** STACKEMPTY()

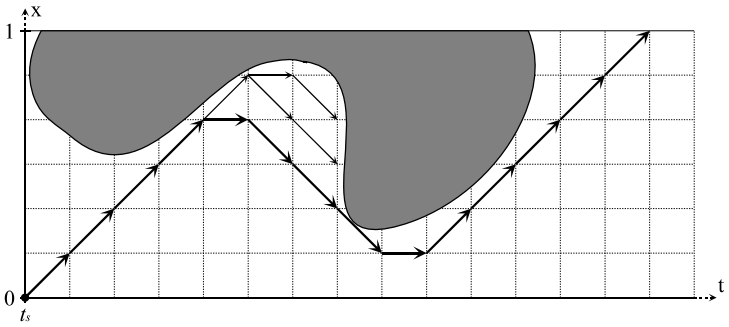


Figure 3.4: Finding a shortest path from $\langle 0, t_s \rangle$ to $x = 1$ using depth-first search. The thin arrows indicate the space explored by the algorithm and the thick arrows form the resulting path. The gray object is a configuration-time obstacle.

parts of the configuration-time space 'beneath' the resulting path. Let us look at what an A*-algorithm would do in the same example (see Fig. 3.5). The A*-method needs a lower bound estimate of the cost (i.e. the amount of time) of going from a configuration-time $\langle x, t \rangle$ to the destination vertex. We used $\frac{1-x}{\Delta x}\Delta t$, which is the optimal estimate. Yet, the A*-method explores a considerably larger part of the configuration-time space, and hence performs many more collision-checks, to eventually find the same path as the depth-first search method.

We can prove that the depth-first algorithm is correct, i.e. it yields a path arriving as early as possible on the destination vertex. Let t_f be the time at which the destination vertex is first reachable from a given initial configuration-time $\langle x_s, t_s \rangle$. Hence, there exists at least one path between $\langle x_s, t_s \rangle$ and $\langle 1, t_f \rangle$ in the grid. Let Π be the set of all valid paths between $\langle x_s, t_s \rangle$ and $\langle 1, t_f \rangle$. We define one of them to be the *highest*, that is the path π_h, which is defined as follows: $\pi_h(t) = \max_{\pi \in \Pi} \pi(t)$, for $t_s \leq t \leq t_f$, where $\pi(t) \in [0,1]$ is the configuration of path π at time t. It is easy to see that π_h is itself an element of Π.

Theorem 3.2. *Given an initial configuration-time $\langle x_s, t_s \rangle$ and an edge of which the destination vertex is earliest reachable at time t_f, the above algorithm finds the highest path π_h between $\langle x_s, t_s \rangle$ and $\langle 1, t_f \rangle$.*

Proof: We will prove that from every configuration-time $\langle x, t \rangle$ on π_h, the algorithm proceeds to the successor of $\langle x, t \rangle$ on π_h. Since the initial configuration-time $\langle x_s, t_s \rangle$ also lies on π_h, the algorithm will then find a path from $\langle x_s, t_s \rangle$ to $\langle 1, t_f \rangle$ that exactly equals π_h.

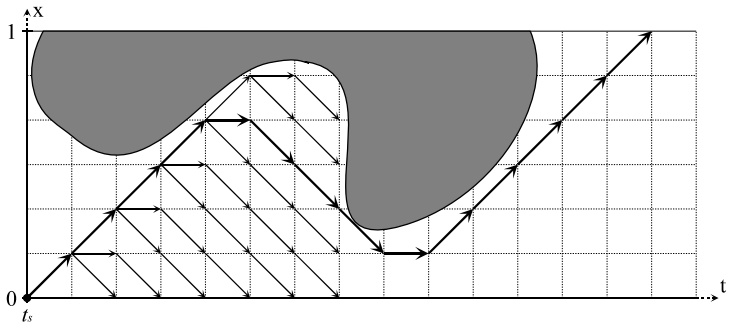


Figure 3.5: Finding a path in the same configuration-time grid as Fig. 3.4 using an A*-search.

Suppose the algorithm has proceeded a number of steps along path π_h up to some configuration-time $\langle x, t\rangle$ on π_h. At this point π_h has three possible successors:

- The successor on π_h is $\langle x+\Delta x, t+\Delta t\rangle$. In this case the algorithm will follow π_h, since it is the most promising step.
- The successor on π_h is $\langle x, t+\Delta t\rangle$. The algorithm at this point first proceeds to configuration-time $\langle x+\Delta x, t+\Delta t\rangle$. Suppose the algorithm finds a path to $x = 1$ from this configuration-time. Then, this path can not reach $x = 1$ before t_f, because t_f is the first time at which $x = 1$ is reachable. So, at some time this path reaches again a configuration-time on π_h, but this is also not possible, because then π_h would not be the highest path to $\langle 1, t_f\rangle$. Hence, no path is found at all from $\langle x+\Delta x, t+\Delta t\rangle$. So, eventually the algorithm returns to configuration-time $\langle x, t\rangle$ and now evaluates $\langle x, t+\Delta t\rangle$, which is also the successor of $\langle x, t\rangle$ on π_h.
- The successor on π_h is $\langle x-\Delta x, t+\Delta t\rangle$. The algorithm now first proceeds to $\langle x+\Delta x, t+\Delta t\rangle$, and then to $\langle x, t+\Delta t\rangle$, but for the same reason as above it will not find a path in either of these cases. So eventually the algorithm evaluates $\langle x-\Delta x, t+\Delta t\rangle$, which is also the successor of $\langle x, t\rangle$ on π_h.

Hence, for every configuration-time on π_h, which includes $\langle x_s, t_s\rangle$, the algorithm follows the path T_h, so the path the algorithm will find exactly equals π_h. ■

In the example of Fig. 3.4 an advancing path is shown, but the method works equally well for returning paths (see Fig. 3.6). The algorithm does not immediately return success, because it starts in an already visited interval. Hence, the algorithm proceeds until an unvisited interval has been reached. How to determine whether an interval has been visited is discussed in Section 3.5.4.

The algorithm described above finds a path to the first reachable free interval on the destination vertex. However, in the example of Section 3.3 we saw that the destination vertex had to be reached in the second reachable free interval. The algorithm is easily adapted to

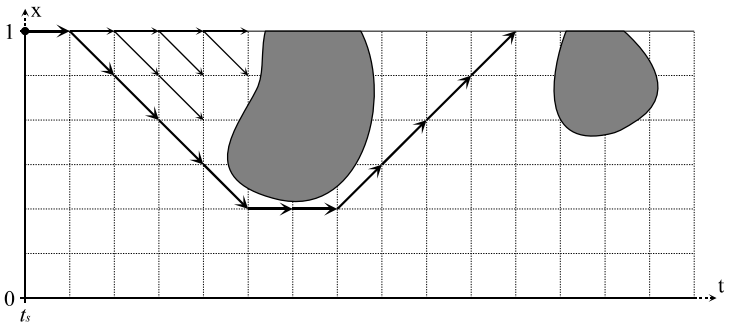


Figure 3.6: Finding a returning path originating at $\langle 1, t_s \rangle$ and arriving at $x = 1$ in an unvisited free interval of the destination vertex.

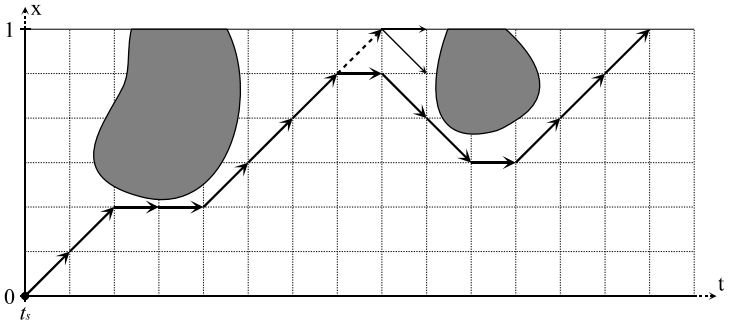


Figure 3.7: Finding a path to the second reachable interval on the destination vertex. The dotted arrow indicates the path to the first reachable interval.

find paths to next intervals as well. If the search is not terminated when the first reachable free interval is found (line 4 of Algorithm 3.2), a path to next intervals will be found as well (see Fig. 3.7).

As proved above, Algorithm 3.2 finds a shortest path in the grid. The associated path is *approximately* optimal when abstracting from the grid, but as the time step Δt approaches zero, the path approaches the continuous time-optimal path. For smaller Δt the algorithm obviously becomes slower, so the choice of Δt gives a trade-off between accuracy and speed.

3.5 Global Paths

In the previous section we discussed how to compute a local path on a single edge. In this section we will show how the global path from a start vertex s to a goal vertex g through the

roadmap is found. As with the local paths, the algorithm will find an approximately time-optimal path.

3.5.1 The Interval Tree

In Section 3.3 we already gave a short introduction to the implicit interval tree. Each node in the interval tree corresponds to a free interval on a vertex in the roadmap. The root of the tree is the interval on the global start vertex s at start time t_0. An important notion is the time at which the interval associated with a node n is *first reachable* for the robot (see Fig. 3.8). We denote this time by $t_f(n)$. The vertex associated with node n is denoted $u(n)$.

A branch exists in the interval tree from a node n_i to a node n_j when all of the following conditions hold:

- Their associated vertices $u(n_i)$ and $u(n_j)$ are either connected by an edge in the roadmap, or $u(n_i) = u(n_j)$.
- There exists a local path starting at time $t_f(n_i)$ on vertex $u(n_i)$ and arriving at time $t_f(n_j)$ on vertex $u(n_j)$.
- No other path has already been found that arrives at time $t_f(n_j)$ on vertex $u(n_j)$.

Consider the example of Fig. 3.8. There will exist a branch from n_1 to n_3, but not from n_2 to n_3 because there is no path between the intervals of n_2 and n_3 obeying the second requirement. Such paths need not be incorporated in the interval tree, because they do not extend the possibilities of reaching the global goal vertex; if the path between the intervals of n_1 and n_3 is extended with two waiting steps it is equivalent with the path between the intervals of n_2 and n_3. In other words, the first path *subsumes* the latter.

The third condition is posed as multiple paths with different sources may reach the interval at the same time t_f. Since these paths have the same possibilities, only one of them needs to be kept as branch in the tree.

In our algorithm, we do not explicitly compute the interval tree, but we lazily evaluate the branches during the search for a global path. To this end, we use the concept of *probes*. Each probe evaluates a branch in the tree (i.e. it tries to find a local path meeting the above three criteria) and when a node has been reached it sends out new probes on the subbranches. So during the search for a path a collection of probes is to be maintained. In principle, each probe is executing Algorithm 3.2, but it is only allowed to proceed one step at a time. The order in which the probes proceed is globally coordinated. For this purpose, we use an A*-like search [96], in which the probe that is most promising to find a path to the global goal vertex is allowed to evaluate and proceed one step. This repeats until the goal vertex has been reached.

3.5.2 Probes

The probe is the main conceptual object of our algorithm. The probes explore the reachable part of the roadmap in a search for a global path from the start to the goal vertex. Each probe p is bounded to one edge, say (u_s, u_d), of the roadmap. A probe is initialized with an initial

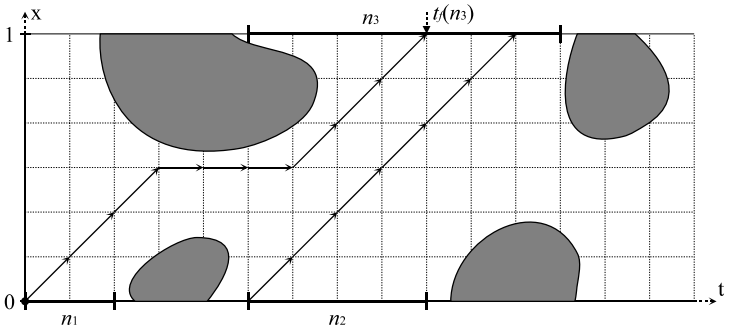


Figure 3.8: A configuration-time grid showing free intervals on the vertices and paths between them. A branch from n_2 to n_3 is not admitted in the interval tree.

configuration-time $\langle x_s, t_s \rangle$ on (u_s, u_d), and is aiming to reach the destination vertex u_d in an unvisited interval. In Section 3.4 we saw how such paths are computed. Each probe carries its own stack of configuration-times.

More than one probe may appear on the same edge, but for now we assume that the probes do not influence each other's behavior. In Section 3.5.5 we discuss how multiple probes are coordinated on an edge.

Since we use an A*-search to coordinate the order in which the probes proceed, we have to define a function $f(p)$ that determines for each probe p how promising it is. $f(p)$ gives an estimate of the cost of the time-optimal global path to which p is contributing. It is computed as follows:

$$f(p) = g(p) + h(p)$$

where $g(p)$ is the cost of the path between the start vertex s and the current configuration-time of p, and $h(p)$ a *lower bound* estimate of the cost of the time-optimal path between the current configuration-time of p and the goal vertex g.

Let $\langle x, t \rangle$ be the top element of p's stack. Then the value of $g(p)$ trivially evaluates to:

$$g(p) = t$$

The value of $h(p)$ is computed as the sum of the estimated amount of time it takes for the probe to reach its destination vertex u_d, and the estimated amount of time it takes to go from u_d to the goal vertex g. For the latter term, we use the number of steps $D(u_d, g)$ in the discretized roadmap between u_d and g. This roadmap distance is available if prior to the query phase a single-source Dijkstra's shortest path algorithm is carried out on the roadmap, with vertex g as its source. The total lower bound estimate for the cost of a time-optimal path from $\langle x, t \rangle$ to g is:

$$h(p) = \frac{1-x}{\Delta x}\Delta t + D(u_d, g)\Delta t$$

It is the amount of time needed to reach the goal vertex if no moving obstacles would stand in the way.

The probe with the highest *priority*, i.e. the most promising probe, is the probe with the lowest value for $f(p)$. If two probes have the same $f(p)$-value, the one with the smallest $h(p)$ is given priority.

3.5.3 Finding a Global Path

Consider a roadmap with start vertex s and goal vertex g and a start time t_0. The root of the interval tree is the interval on s at time t_0. From this interval, there may be branches in the interval tree to intervals on neighboring vertices, so on each outgoing edge (s, u) from s a probe is released with initial configuration-time $\langle 0, t_0 \rangle$ trying to reach unvisited free intervals on the edge's destination vertex u. It is important to note that after a probe arrives at the first reachable interval on its destination vertex, it continues its search for next intervals (see Fig. 3.7). Also the returning paths must be considered, so for this purpose probes have to be launched too. These returning probes are launched on the *incoming* edges (u, s) of s, since s is their destination. Their initial configuration-time is $\langle 1, t_0 \rangle$.

All the probes are stored in a priority queue. In each step of the algorithm, the top element of the priority queue, i.e. the most promising probe with the highest priority, is allowed to proceed one step. This is in principle repeated infinitely. A number of events can occur during the algorithm:

- A probe's stack becomes empty. In this case the probe is deleted and removed from the priority queue. If it was the last probe in the queue, the algorithm terminates and reports that no path exists.
- A probe reaches the global goal vertex. In this case the algorithm is terminated and the near-time-optimal path is read out by following the backpointers stored in the grids.
- A probe p reaches the destination vertex u_d of its edge (u_s, u_d) at time t_p in an *unvisited* interval. (When a visited interval is encountered, nothing happens; the probe just continues its search for an unvisited interval.) In terms of the interval tree, this means that a branch has been established to a new node, so new probes have to be sent out on the incident edges of u_d. Advancing probes are sent out on the outgoing edges (u_d, u) with initial configuration-time $\langle 0, t_p \rangle$ and returning probes are launched on the incoming edges (u, u_d) with initial configuration-time $\langle 1, t_p \rangle$. The probe p itself is *not* deleted; it continues its search for next unvisited free intervals on u_d.

The algorithm terminates when the goal vertex has been found by one of the probes, or when all probes have been deleted. In the latter case there is no path toward the goal vertex. However, it is also possible that no path exists, but that the algorithm is running forever, with probes waiting vainly for the moving obstacles to step aside. Therefore, some upper bound $t_{\max}$ on the time may be set, to make sure that it terminates. If for the most promising probe holds that $f(p) > t_{\max}$, the algorithm stops and reports failure. The pseudo code of the algorithm is given in Algorithm 3.3.

It is easy to prove that this algorithm yields an approximately time-optimal path from the start to the goal vertex. In Section 3.4 we already saw that each local path from interval to interval is near-time-optimal. Since every reachable interval is considered in the algorithm,

Algorithm 3.3 FINDGLOBALTRAJECTORY(s, g, t_0)

1: Initialize advancing probes on all the outgoing edges (s, u) of s with configuration-time $\langle 0, t_0 \rangle$ and returning probes on incoming edges (u, s) of s with configuration-time $\langle 1, t_0 \rangle$, and store them in the priority queue.
2: **while** the priority queue is not empty **do**
3: $p \leftarrow$ top element of the priority queue.
4: $(u_s, u_d) \leftarrow$ edge on which p is active.
5: **if** $f(p) > t_{\max}$ **then**
6: Terminate algorithm. Report failure.
7: $\langle x, t \rangle \leftarrow p$.DOSTEP() {See Algorithm 3.1}
8: **if** $x = 1$ **and** $u_d = g$ **then**
9: Success! Terminate algorithm. The path is read out by following the backpointers.
10: **if** $x = 1$ **and** t is in an unvisited interval on u_d **then**
11: Initialize advancing probes on all the outgoing edges (u_d, u) of u_d with configuration-time $\langle 0, t \rangle$ and returning probes on incoming edges (u, u_d) of u_d with configuration-time $\langle 1, t \rangle$, and append them to the priority queue.
12: **if** p.STACKEMPTY() **then**
13: Delete p and remove it from the priority queue
14: Report that no path exists.

this also holds for the first reachable interval on the goal vertex. Hence, the path found to this interval is near-time-optimal too.

3.5.4 Determining whether an Interval is Unvisited

When a probe reaches a free interval, we have to determine whether it is unvisited. For this purpose we maintain for each vertex in the roadmap at what times it has been visited by a probe. When a probe arrives at the destination vertex u_d at time t and both time t and $t - \Delta t$ on u_d are unvisited, it is sure that an unvisited free interval is reached.

We can prove this as follows. Suppose probe p arrives at time t at an interval on u_d that has been visited before, but that times t and $t - \Delta t$ are unvisited on u_d. Then a probe p' must have visited the interval at a time $< t - \Delta t$. Since a probe reaching its destination vertex is not deleted and goes on with searching for new free intervals on the vertex, probe p' at time $< t - \Delta t$ on u_d had a higher priority than p, so p' is doing steps first. Probe p may arrive at time t on u_d before p' does, but then at least time $t - \Delta t$ on u_d has been visited by p'. This contradicts our assumption, and hence the free interval on u_d is unvisited when a probe reaches it at time t and both time t and $t - \Delta t$ are unvisited.

3.5.5 Coordinating Multiple Probes on an Edge

Up to now we did not let the probes influence each others behavior, but as multiple probes may appear on the same edge, they may explore common parts of the configuration-time space. To prevent this, we allow only one of the probes to proceed in case multiple probes

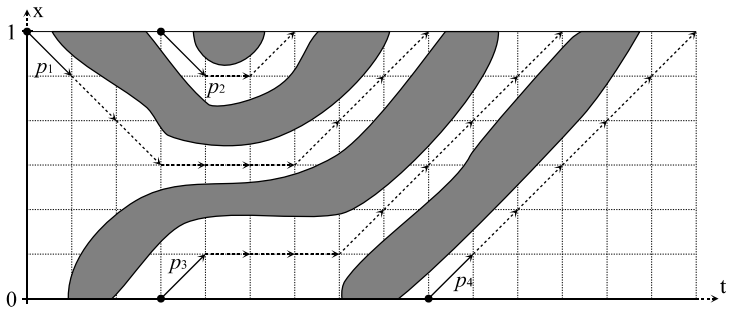


Figure 3.9: Four probes on the same edge, and their projected traces (dotted).

are active on the same edge. To be precise, of each edge at most one probe is represented in the global priority queue that coordinates the order in which the probes proceed.

Consider the example of Fig. 3.9. Two returning probes p_1 and p_2, and two advancing probes p_3 and p_4, originating from different intervals, are active on the same edge (u_s, u_d). All of these probes aim to reach the same destination vertex u_d in the first reachable free interval. Each of these probes has a different *start time*, i.e. the time at which the probe was launched. We now sort the probes in a list according to the following rules:

- Returning probes precede advancing probes
- Returning probes are sorted in *reverse order* of their start times, i.e. the returning probe that started latest is in front of the list.
- Advancing probes are sorted in order of their start times, i.e. the advancing probe that started latest is at the end of the list.

So for the edge in our example, the list would be $\{p_2, p_1, p_3, p_4\}$. We now only allow the probe in front of the list to do steps on the edge, i.e. only this probe is present in the global priority queue as a representative of this edge. The rationale behind this is the following. We can prove that the probe in front of the list will either find a path to the first reachable free interval on the destination vertex, or its stack becomes empty (see Fig. 3.9). In the latter case, the probe is deleted and the second probe in the list becomes the top probe.

None of the probes is allowed to explore parts of the configuration-time space that were visited before by other probes.

3.6 Optimizations

In this section we discuss some optimizations to the algorithm. They were not necessary for the understanding of our approach, but they can give considerable performance improvements.

3.6.1 Launching Probes

In the current algorithm, when a probe reaches the destination vertex of its edge, new probes are launched on all the incident edges of the destination vertex. A few of them however, need not be sent. Suppose probe p on edge (u_s, u_d) reaches it destination vertex u_d at time t_p. Now, no returning probe needs to be sent on (u_s, u_d), because p itself will search at that edge for next intervals on u_d (see Fig. 3.10(a)).

Also no advancing probe needs to be sent on the 'opposite' edge (u_d, u_s), because at the time (an advancing probe) p was launched on (u_s, u_d), a (returning) probe p' was launched on (u_d, u_s) trying to reach u_s as well. This probe p' subsumes a new probe (see Fig. 3.10(b)).

Furthermore, no advancing probes need to be sent on dead ends of the roadmap, i.e. edges leading to vertices with only one incident edge. Arriving at such vertices does not extend the possibilities of reaching the goal vertex. Returning probes though need to be sent on these edges. Only if the dead end leads to the goal vertex itself, an advancing probe has to be sent as well.

3.6.2 Deleting Probes

There are situations in which a probe can be deleted before its stack becomes empty. This saves a lot of unnecessary collision checks.

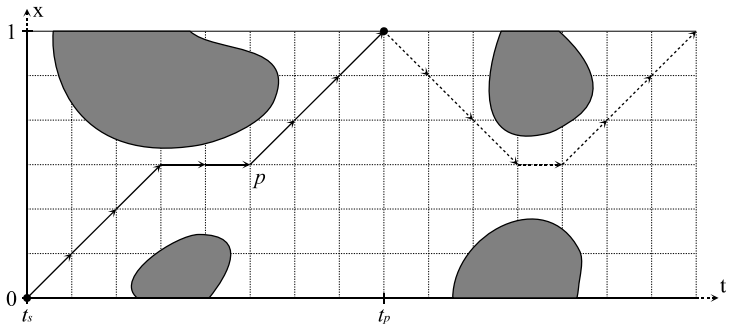
Consider the example of Fig. 3.11. In this case an advancing probe is pushed back to the source vertex u_s of the edge (u_s, u_d) by the configuration-time obstacle. It is clear that once the probe has reached this source vertex, there is no possibility left to reach the destination vertex. Yet, the probe's stack still contains configuration-times. To prevent the probe of unnecessarily exploring the part of configuration-time space bounded by its own trace (light gray area in the figure), it is deleted.

The exact situation in which the probe can be deleted is if it fails to do a step on the source vertex of the edge (thick arrow in the figure). This holds for both advancing and returning probes. In fact, when a probe is deleted for this reason, all probes that are active on the same edge can be deleted, because – given the order in which they are sorted in the list – they would need to cross the trace of the deleted probe to reach the destination vertex.

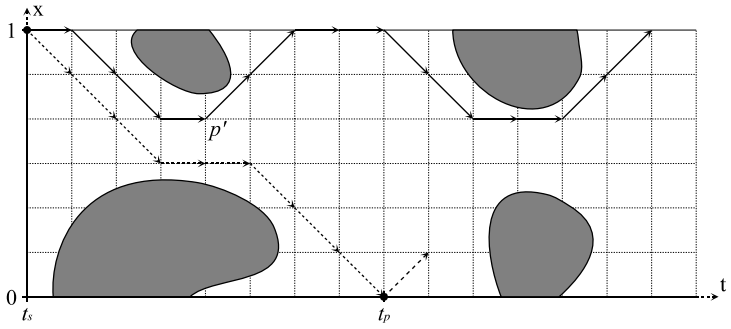
3.6.3 Closing and Opening Vertices

After probes have been launched from some vertex u on the outgoing edges, it is not useful to proceed probes whose destination vertex is u until one of the probes launched from u is deleted. This is because a probe p arriving at u may only contribute to a shortest path to the goal if at least one of the probes launched before from u appears to explore a dead end and is deleted. Suppose we would allow proceeding p, and suppose it arrives at u at some point, then the advancing probes launched at outgoing edges will be enqueued at the back of the queues associated with the concerning edge, and can only come in action if probes already active on the edge are deleted. We can therefore save time in the algorithm by only proceeding probes when they potentially contribute to a shortest path to the goal.

To this end, a vertex u is *closed* after probes have been launched from there. Probes on edges whose destination vertex is closed will not be allowed to proceed until this vertex is



(a)



(b)

Figure 3.10: (a) A probe p active on an edge. Since p is continuing its search after reaching its destination vertex, no returning probe (with initial configuration-time $\langle 1, t_p \rangle$) needs to be sent on this edge. (b) The opposite edge with a mirrored configuration-time space. No advancing probe needs to be sent on this edge from $\langle 0, t_p \rangle$, because it is subsumed by a returning probe p', which was sent at the same time as p. The projection of the path of p on this mirrored configuration-time space is indicated with a dotted line.

open again. Hence, these edges have no representative probe in the global priority queue. A vertex is *opened* again when on one of its outgoing edges the list of active probes becomes empty. Initially all vertices are open.

This extension considerably prevents the algorithm from backtracking to probes that are close to the start vertex, and hence saves a lot of time in the computation.

3.7 Experimental Results

The algorithm has been implemented for both free-flying and articulated robots with six degrees of freedom in a three-dimensional workspace. We performed experiments in different environments and the results indicate that the method achieves interactive performance. In

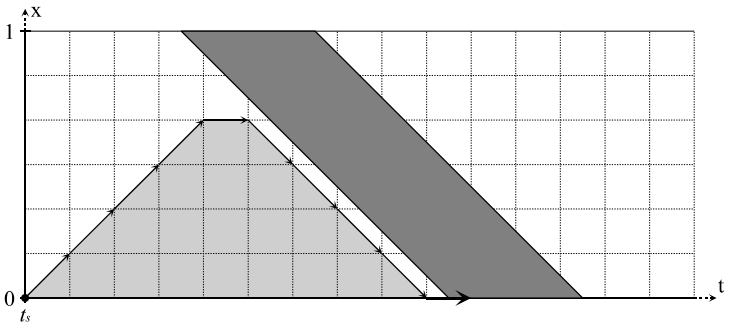
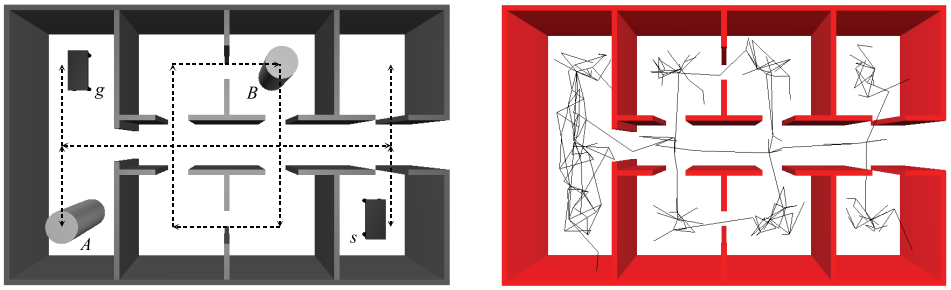


Figure 3.11: A situation in which a probe can be deleted before its stack becomes empty. This prevents the gray area of being explored vainly.



(a) (b)

Figure 3.12: (a) A dynamic environment in which a table has to move from s to g avoiding the moving obstacles A and B (cylinders). The cylinders move cyclically along the dotted lines. (b) A roadmap that is collision-free with respect to the stationary obstacles. The rotational degrees of freedom are not shown in the roadmap.

this section we describe a number of experiments in detail, and compare our method with the straightforward approach as well.

3.7.1 A Preliminary Experiment

Our method was tested in the building floor scene of Fig. 3.12(a). The scene has dimensions of 8 (length) by 5 (width) by 2 (height) units of length. In the scene two moving obstacles A and B are moving. A moves along an H-shaped trajectory and B along a rectangular trajectory. The velocity of both moving obstacles is 1 unit of length per unit of time. The positions of the moving obstacles at the start time are shown in the figure. The motions of both obstacles are cyclic, i.e. they move endlessly.

As the robot we used a free-flying table, which has a radius of 0.5 units of length. It has to move through some narrow passages (having a width of 0.6 units of length) from s in the lower-right room to g in the left room. The distance between two configurations of the robot is measured as the euclidean distance plus the amount of rotation times the robot's radius. Its velocity under this distance measure is bounded by 1 unit of length per unit of time.

For the static part of the scene a roadmap was created using a variant of PRM that allows the formation of cycles in the roadmap [115]. The construction of the roadmap stopped when a predefined set of query configurations was connected by the roadmap. The roadmap is shown in Fig. 3.12(b) and consists of 198 vertices and 478 edges.

The shortest path in the roadmap between the start and the goal configuration is 15.82 units of length. So if no moving obstacles would be present 15.82 units of time are necessary to complete the path. However, moving obstacle A will move through the passageway in the opposite direction of the robot, so the robot must make a detour to avoid this obstacle.

This example problem may look simple, but it certainly is not. The stationary obstacles in the scene strongly confine the maneuverability of the robot, and the environment contains many narrow passages. Moreover, the moving obstacles 'sweep clean' the passageways in the environment, so the robot really has to 'flee' into other rooms to avoid collisions.

The running time of the algorithm is directly dependent on the choice of the value of the principal time step Δt. In this experiment we chose Δt to be 0.07, so that the collision checking resolution with respect to the moving obstacles is exactly corresponding to the resolution in which the roadmap was collision checked with respect to the stationary obstacles.

The algorithm was run on a 3 GHz Pentium IV with 1 GByte of memory. For the problem described above, it returned a path in only 0.46 seconds of computation time. The path takes 35.14 units of time to traverse, and indeed the robot must make quite a detour to avoid the moving obstacles (see Fig. 3.13). It more than once 'flees' into a room at the side of the passageway. Obviously, the path is collision-free with respect to both the static and the moving obstacles.

The computed path is near-time-optimal over the roadmap. Yet, it takes 35.14 units of time to traverse, while this would be 15.82 units of time without obstacles. We call these numbers the *path length* and the *roadmap distance* respectively, and the ratio between them the *delay-factor*. It gives an indication of the 'difficulty' of the problem. For this experiment the delay-factor is 35.14 / 15.82 = 2.22.

3.7.2 Varying the Quantities

In this section we explore the effect on the performance of our method of gradually increasing major quantities, among which the delay-factor of the problem, the size of the roadmap and the value of the time-step Δt. For these experiments, we use the environment and the quantities of the above experiment. Only the quantity under consideration is varied.

Difficulty: The difficulty of a problem is always hard to measure, let alone varying it *ceteris paribus*, i.e. without changing other quantities. We use the delay-factor as a measure for the difficulty of a problem. In the above experiment the delay-factor was 2.22.

We vary the delay-factor over the experiments by tuning the velocity and the behavior of

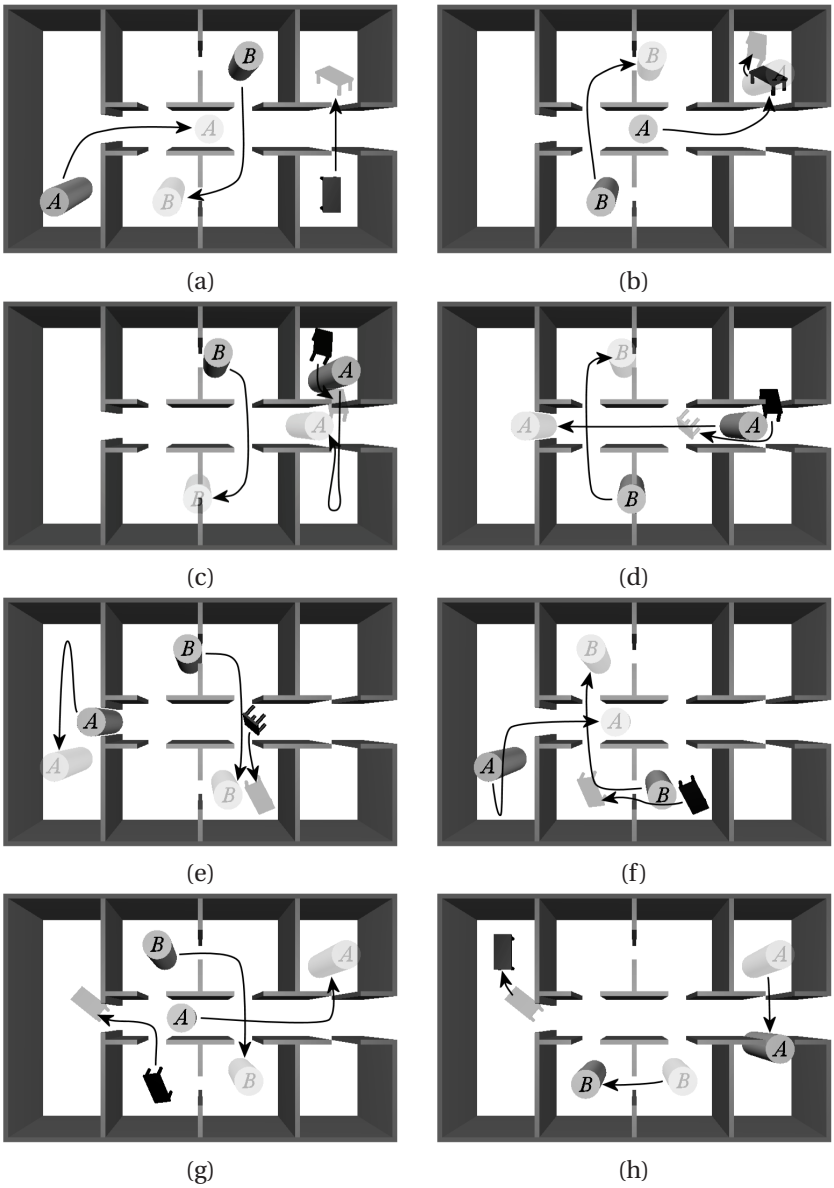


Figure 3.13: Pictures from the path. A table has to move from the lower-right room (a) to the left room (h). The table first moves to the upper-right room to find room for avoiding moving obstacle A (a-b). Then it moves in the slipstream of A through the passageway (c-d). The table then moves to the lower room to find room for avoiding obstacle B (e). After this it moves in the slipstream of B to enter the left room (f). Obstacle A is moving toward the right part of the scene, so the table can safely reach its goal position (g-h).

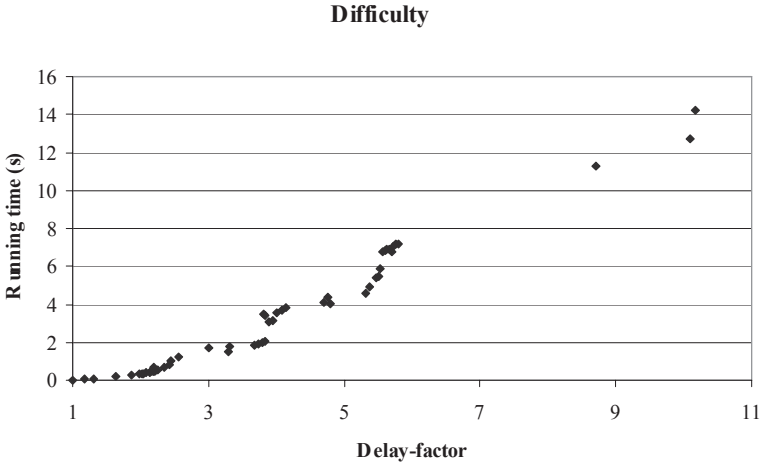


Figure 3.14: A scatter-plot of 55 experiments with different delay-factors.

the moving obstacles. The other quantities are kept equal; we use the same roadmap over the experiments and do not change the value of Δt.

The performance of our method not only depends on the delay-factor, but also on whether an obstacle interferes with the robot in the beginning of the path, when little probes are active, or near the goal configuration, when many probes are active. To cover a large range of possibilities, we performed 55 experiments with different behavior of the moving obstacles.

One would expect that the running time grows exponentially with the delay-factor, because the more time it takes to reach the goal, the wider the interval tree grows. The A*-nature of our algorithm though moderates this effect as it focuses the search toward the goal. Also, for larger delay-factors the space gets more confined. This means that there is less maneuverability for the robot, which has a moderating effect on the width of the interval tree (and hence the running time) as well.

Fig. 3.14 gives the result of the experiments. The scatter-plot indicates that the running time actually grows more or less linear with the delay-factor. It turned out to be quite hard to find problems with a delay factor larger than 6 for which a solution still exists. The few problems we did find however, confirm the apparent linear relationship. Note that when the delay-factor is 1, the running time is nearly zero.

Roadmap size: The size of the roadmap influences the performance of our method. The more edges in the roadmap, the more probes need to be sent and, hence, the interval tree grows wider. On the other hand, more possibilities become available in the roadmap when the roadmap gets larger. This can also have a positive effect on the performance, as this may lower the delay-factor of the problem.

To avoid strong deviations in the running times of the experiments as a result of the ran-

Table 3.1: Results for various roadmap sizes

# vertices	# edges	roadmap distance	path length	delay-factor	running time
100	246	17.15	43.40	2.53	0.67s
200	558	16.52	25.58	1.55	0.33s
300	944	16.52	25.48	1.54	0.62s
400	1376	16.52	25.48	1.54	1.16s
500	1806	14.70	25.48	1.73	1.88s
600	2236	14.70	25.48	1.73	2.13s
700	2698	14.70	25.48	1.73	2.48s
800	3156	14.70	25.48	1.73	2.85s
900	3668	14.21	24.64	1.73	3.17s
1000	4158	14.21	24.64	1.73	3.90s

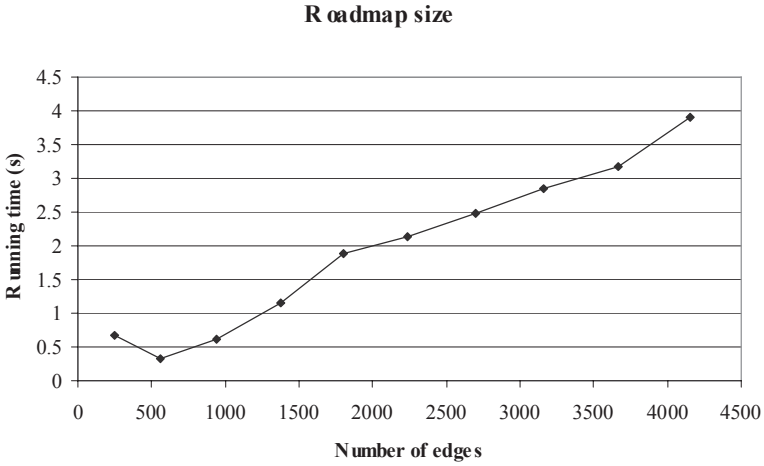


Figure 3.15: The running time for roadmaps of various sizes.

domness involved in creating roadmaps, we extend roadmaps over the experiments. To be precise, in each experiment we add 100 vertices to the roadmap of the previous experiment. The number of edges grows more or less proportionally. The results are shown in Fig. 3.15 and Table 3.1.

The figure shows a linear relation between the roadmap size and the running time. We see that for a too small roadmap the running time actually gets higher. This is because the possibilities of the robot are too much restricted, which results in a high delay-factor of the

problem (see Table 3.1). The minimum running time is found for a roadmap with 200 vertices (and 558 edges). Note that the resulting path is not substantially longer than the paths found in larger roadmaps (see column 'path length').

Time step: The effect of the value of the time step Δt is inversely proportional to the running time of our method. That is, as $1/\Delta t$ increases, the running time linearly grows accordingly. This is confirmed by the results of experiments we performed for several values of Δt.

Other quantities: Other quantities do not directly influence the performance of our method. The number of degrees of freedom of the robot, for instance, or the size of the environment, only influence the performance if they make the problem more difficult, or if they make the environment require a larger roadmap. Also, some quantities, such as the number of moving obstacles or the complexity of the moving obstacles, cause collision-checks to become more expensive. They may as such only have an indirect relation to the performance of our method.

3.7.3 Articulated Robots

Our method is suitable for articulated robots as well. A particular advantage of our method with respect to articulated robots is that no expensive checks for self-collisions have to be done during the algorithm. These are already performed during the preprocessing phase.

We performed an experiment involving an articulated robot with six degrees of freedom. The environment in which the experiment is performed is depicted in Fig. 3.16. The articulated robot stands fixed amidst some static walls. Five moving obstacles (cylinders) hinder the robot in its attempt to go from a start configuration (robot fully bent to the right) to the goal configuration (robot fully bent to the left). Although it is not a very realistic environment, it shows the capabilities of our method.

A roadmap was created for the articulated robot using the variant of PRM that allows cycles in the roadmap. The construction stopped when the start and goal configuration were connected by the roadmap. A roadmap containing only 20 vertices and 49 edges proved to be enough to cover this rather simple static scene. The shortest path in the roadmap takes 3.78 units of time to traverse if there would be no moving obstacles. The path avoiding the moving obstacles computed by our method takes 7.86 units of time to traverse. Hence, the delay-factor for this problem is 2.08. The value of Δt was chosen adequately small. It took only 0.87 seconds to compute the path using our method.

We must note that the collision-checks were more expensive for this environment than for the previous one. On the one side this is because the number of moving obstacles is larger, but more significantly is that it takes much more time to configure and collision-check an articulated robot than a free-flying robot.

3.7.4 Comparison with the Straightforward Approach

To adequately assess the performance of our method, we compared our method to the straightforward approach mentioned in Section 3.3. This method simply performs an A*-search in the space formed by the Cartesian product of the discretized roadmap and the discretized time axis. It finds the same paths as our method, i.e. near-time-optimal ones.

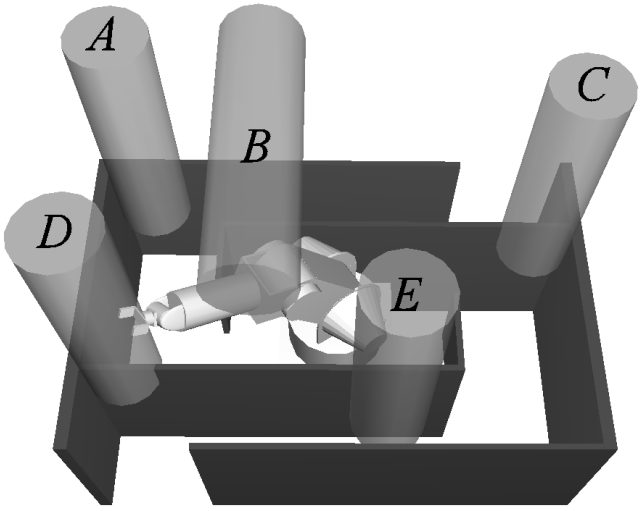


Figure 3.16: A dynamic environment with an articulated robot and five moving obstacles. Obstacles *A*, *B* and *E* move back and forth parallel to the *x*-axis. *C* and *D* move parallel to the *y*-axis.

We ran this method on the examples discussed in this chapter, and the results show that the straightforward method is on average at least a factor 10 slower than our method.

3.8 Discussion and Conclusion

In this chapter we presented a new method for motion planning in dynamic environments. The method finds paths in a given roadmap avoiding collisions with moving obstacles. It is applicable to any robot type with any number of degrees of freedom.

We used an implicit grid in configuration-time space to find near-time-optimal paths. As the principal time step Δt approaches zero, the near-time-optimal path becomes a time-optimal path, so the parameter Δt gives a trade-off between accuracy and speed. We showed in our experiments that our algorithm performs very fast even for small values of Δt in planning problems with high delay-factors.

A great advantage over other methods is that the stationary obstacles are not of concern. Scenes often contain narrow passages through which a path is not easily found. Our method leaves this problem to a preprocessing phase, such that interactive performance can be achieved in the query phase.

The good performance of our method is explained by the two-level strategy of the algorithm. We exploit the advantages of depth-first search on local paths, and use A* to find a global path. Only the potentially interesting parts of the implicit interval tree are evaluated. Note that if no moving obstacles are present, only probes along the path in the roadmap

leading directly to the goal vertex are processed. Since the collision checks done in this case are void (there are no moving obstacles), the path is returned instantly. Also, if all moving obstacles stay away from the optimal path, the path is reported almost instantly.

The method presented in this chapter can directly be applied to path planning for multiple robots. This application will be discussed in the next chapter of this thesis.

Chapter 4

Planning for Multiple Robots

In this chapter we address the problem of path planning for multiple robots. We present a prioritized method, based on the powerful method for path planning in dynamic environments discussed in the previous chapter. Our approach is generically applicable: there is no limitation on the number of degrees of freedom of each of the robots, and robots of various types –for instance free-flying robots and articulated robots– can be used simultaneously. Results show that high-quality paths can be produced in less than a second of computation time, even in confined environments involving many robots. We examine three issues in particular in this chapter: the assignment of priorities to the robots, the performance of prioritized planning versus coordinated planning, and the influence of the extent by which the robot motions are constrained on the performance of the method. Results are reported in terms of both running time and the quality of the paths produced.

This chapter has previously been published as: J. van den Berg and M. Overmars. Prioritized motion planning for multiple robots. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2217–2222, 2006 [19]. Part of this chapter has also appeared in [20].

4.1 Introduction

This chapter addresses the problem of path planning for multiple robots, which is an important topic in the field. The task is to plan trajectories for the robots that bring each robot from some start configuration to some goal configuration without mutual collisions and collisions with stationary obstacles. This problem has been studied extensively.

Most research has focused on *coordinated* approaches. They are often categorized along the spectrum between *centralized* and *decoupled* planning [99]. A centralized planner computes a path in the composite configuration space, which is formed by the Cartesian product of the configuration spaces of the individual robots [136, 139]. In a decoupled approach a path is computed for each robot in its own configuration space, independently of the other robots, and a coordination diagram is used to coordinate the motions of each robot along its path [99, 123, 143]. Approaches that only weakly constrain the motions of the robots be-

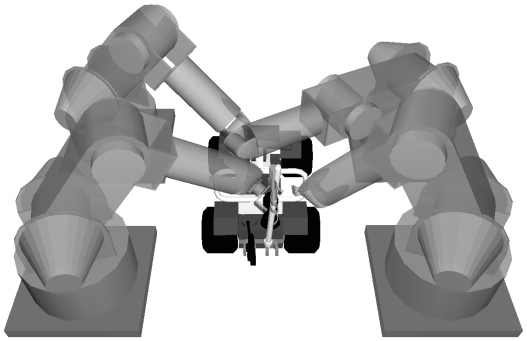


Figure 4.1: An environment with four articulated robots manipulating a car.

fore considering interactions between the robots can be categorized in the middle of the spectrum. They typically use *roadmaps* for each of the robots that cover each of their free configuration spaces well [65, 99, 150].

The various approaches along the spectrum trade off completeness for speed and applicability. Centralized approaches are complete, but are in general computationally demanding, or only applicable to simple robots operating in simple environments. Decoupled approaches are applicable to robots of any kind, but the paths they compute may be far from optimal. Approaches that use a roadmap provide a compromise between the two extremes.

A less studied approach to path planning for multiple robots, which is nevertheless often used in practice, is *prioritized* planning [14, 42, 47]. In a prioritized approach each of the robots is assigned a priority. Then in order of decreasing priority, the robots are picked. For each picked robot a path is planned, avoiding collisions with the stationary obstacles as well as the previously picked robots, which are considered as *moving obstacles*.

This reduces the multi-robot path planning problem to the problem of path planning for a single robot in a known dynamic environment. Also in this case a spectrum can be defined along which the extent is varied by which the motion of the robot is constrained. In [75], the robot motion is not constrained, and in [53] the motion of the robot is constrained to a path that is collision-free with respect to the stationary obstacles. In the previous chapter, we introduced a method that constrains the robot motion to a roadmap (see also [20]).

In this chapter, we present a prioritized method for path planning for multiple robots, based on the method presented in the previous chapter. Each robot is constrained to move over a preprocessed roadmap that is collision-free with respect to the stationary obstacles. Our method is applicable to any number of robots with any number of degrees of freedom in both two- and three-dimensional environments (see e.g. Fig. 4.1). Also, robots of different type can be used simultaneously. Experiments were performed in typical and confined environments involving many robots. Results show that high-quality paths can be produced in less than a second of computation time.

In contrast to coordinated approaches, a prioritized approach is not complete. One can construct example problems (see e.g. Fig. 4.2(a)) for which a prioritized approach will never

find a solution, or will not find one if the wrong *prioritization* is chosen. However, we believe that this does not limit the applicability of prioritized approaches in most practical situations, where this problem is not likely to arise. In this chapter, we only study environments where it is guaranteed to be possible to find a path using any prioritization. Still, the choice of prioritization can have a large effect on the quality of the composite path.

Three issues in particular are addressed in this chapter:

- How are the priorities assigned to each of the robots, and how does this influence the performance of prioritized planning?
- How does our prioritized approach compare to coordinated approaches, and how do the approaches perform when the number of robots increases?
- What is the effect on the performance when we vary the extent by which the robot motions are constrained?

We report results in terms of both the optimality of the paths produced and the running time of the methods.

The rest of this chapter is organized as follows. In the next section we formally define the problem to be solved. In Section 4.3 we introduce our prioritized method, and in Sections 4.4 to 4.6 we address each of the three issues raised above. We conclude the chapter in Section 4.7.

4.2 Problem Definition

4.2.1 Definition

The problem is formally defined as follows. Given are n robots $A_1, \ldots, A_n$ with maximal velocities $v_1, \ldots, v_n$, and a two- or three dimensional environment in which the robots move. For each robot A_i a roadmap R_i is constructed that covers the free configuration space of A_i, and for each robot a start configuration $s_i \in R_i$ and a goal configuration $g_i \in R_i$ is defined. We use the notation $A_i(x)$ to refer to robot A_i configured at x, where $x \in R_i$.

The task is to compute for each robot A_i a path $\pi_i : [0, T_i] \rightarrow R_i$, such that $\pi_i(0) = s_i$ and $\pi_i(T_i) = g_i$, without collisions with other robots, i.e. $\forall (t \in [0, \max_i T_i] :: \forall (i, j : 1 \leq i < j \leq n : A_i(\pi_i(t)) \cap A_j(\pi_j(t)) = \emptyset))$. (Note that the stationary obstacles are not of concern; the robots move over roadmaps that are already guaranteed to be collision-free with respect to the stationary obstacles.) Further, the paths should obey the maximum velocity constraint of the robots, i.e. $\forall (i :: \forall (t_1, t_2 \in [0, T_i] : t_1 < t_2 : d_i(\pi_i(t_1), \pi_i(t_2)) \leq v_i(t_2 - t_1)))$, where $d_i : R_i \times R_i \rightarrow \mathbb{R}$ is the distance between two configurations in the roadmap of A_i. T_i can be considered as the arrival time of robot A_i at its goal configuration. We refer to $(\pi_1, \ldots, \pi_n)$ as the *composite path* of the robot.

4.2.2 Discretization

The problem is discretized by choosing a small time step Δt, and limiting the set of possible velocities of the robots. Each robot may move with a velocity $|v_i|$ or 0 over its roadmap and

may only change it at given times $k\Delta t$, where k is an integer. This subdivides each of the edges of the roadmap in small steps of length $v_i\Delta t$. At each time step the robot may move one step in either direction along the edge, or halt at its current position. If the robot is on a vertex of the roadmap, it can choose among all of the outgoing edges of the vertex. Such a discretization is common in path planning [20, 53, 99].

Hereafter we refer to R_i as the discretized set of states that robot A_i may adopt on its roadmap.

4.2.3 Quality Measure

An important aspect of path planning for multiple robots is to define a quality measure of the composite path, which should be optimized. This measure is usually defined on the vector $(T_1, \ldots, T_n)$ of arrival times of each of the robots. In some previous work, all pareto-optimal solutions are generated [65, 99], but in most cases a single scalar-valued function is optimized, for instance the average arrival time of the robots. The choice of scalarization is quite arbitrary and may depend on the specific application. In this chapter we choose to take the maximum of the arrival times as quality measure, i.e. we optimize the arrival time of the latest robot.

4.3 Prioritized Planning

Prioritized path planning for multiple robots is a simple approach, already introduced by Erdmann and Lozano-Pérez [47] in 1987. It works as follows: Each of the robots is assigned a priority. Next, the robots are picked in order of decreasing priority. For each picked robot a path is planned, avoiding collisions with the stationary obstacles as well as the previously picked robots, which are considered as *moving obstacles*.

The approach requires two important ingredients: a method to plan paths for a single robot in known dynamic environments, and a scheme to prioritize the robots. In this section we discuss how we implemented them in our method.

4.3.1 Path Planning in Dynamic Environments

According to the definition of Section 4.2, we need a method that – given a robot, a roadmap, start and goal configurations s and g in the roadmap, and the scripted motions of the moving obstacles – computes a path for the robot starting at s at $t = 0$, and arriving as soon as possible at g.

A method for path planning in dynamic environments that does exactly the above has been discussed in the previous chapter. We briefly review its properties here:

- It is applicable to *any robot type* in configuration spaces of *any dimension*. The method only requires a roadmap that covers the free configuration space of the robot well, and that is collision-free with respect to the stationary obstacles.
- The shapes and motions of the moving obstacles are *unconstrained*: they may move with any speed following any trajectory, as long as the motions are known beforehand.

That is, given a position of the robot at a time t we must be able to determine whether the robot collides with a moving obstacle [28].

- The roadmap and the time-axis are discretized as described in section 4.2.2. Under these constraints, the method computes a *time-optimal* path on the roadmap that arrives as early as possible at the goal configuration.

- The method achieves *interactive performance* in complicated dynamic environments.

The method finds a path in the discretized roadmap-time space by an efficient multi-layer search. For details, we refer the reader to the previous chapter.

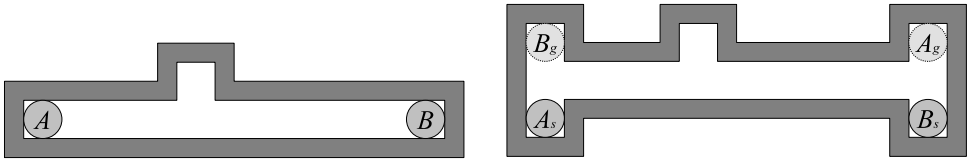
The method is used in our prioritized approach as follows: Let each robot be given a priority (how these priorities are assigned is discussed below). Then iteratively a path is planned for each robot, in order of decreasing priority. If it is the k'th robot's turn, the trajectories of the $k-1$ robots that have previously been planned are considered as moving obstacles. This repeats until a path has been planned for the robot with the least priority. The maximum of the arrival times measures the quality of the planned composite path.

The method has one shortcoming when it is applied to the multi-robot path planning problem: it finds path for the robot arriving at the goal as soon as possible. In principle, this is what we want, but it is very well possible that a robot A_h with a higher priority arrives later at its goal than a robot A_ℓ with a lower priority (let the arrival times be T_h and T_ℓ respectively, where $T_h > T_\ell$). In this case, the path of A_ℓ is guaranteed to be collision-free with respect to A_h between time 0 and time T_ℓ. It is, however, not guaranteed that A_ℓ is collision-free with respect to A_h between time T_ℓ and T_h, during which A_ℓ may be waiting at its goal for A_h to arrive. There may be a collision in this case when A_h passes the goal region of A_ℓ between time T_ℓ and T_h.

It depends on the application whether this really is a problem, but if it is, it could be remedied by changing the method of the previous chapter by letting the robot arrive as soon as possible at its goal, but not before some preferred arrival time t_g. If the robot is able to arrive at the goal before t_g, it must wait at the goal until time t_g. If this waiting appears not to be collision-free, it should evade the particular moving obstacle and return to the goal as soon as possible. Now, for each robot, t_g is set at the maximum of the arrival times of the previously planned higher priority robots.

This change in the algorithm is in principle easily carried through, by simply adding a condition to the stop criterion of the algorithm (i.e., changing line 8 of Algorithm 3.3 to "**if** $x = 1$ **and** $u_d = g$ **and** $t \geq t_g$ **then**"). However, most of the efficiency of the algorithm will be lost in this case, as the A*-algorithm will wrongly focus its search. The algorithm will backtrack many times, as many paths may have an estimated arrival time at the goal smaller than t_g. All these paths will be explored before some probe at the goal reaches the preferred arrival time.

Even though this problem is already reduced by the extension we discussed in the previous chapter of opening and closing vertices (see Section 3.6.3), which considerably prevents the algorithm from backtracking, the loss of efficiency is still significant. The problem can be remedied further by inflating the heuristic in the A*-algorithm (i.e. the $h(p)$ function, which is a lower bound estimate of the time needed to reach the goal). If this heuristic is inflated



(a) (b)

Figure 4.2: (a) An example scene where robots A and B have to exchange positions. A solution does not exist if robot A is given priority over robot B (and both robots have the same maximal velocity). (b) A similar example where a solution will always exist. Yet, the prioritization has a significant influence on the result. If robot A has priority over robot B, robot B can only start moving after A has reached its goal. If robot B has priority, A can use the cavity in the passageway to let robot B pass, giving a much better result.

by some factor $E > 1$, probes closer to the goal are heavily favored over probes close to the start, which further reduces the backtracking.

Inflating the heuristic comes at some cost. It is no longer guaranteed that the time-optimal path to the goal is found, but only a path that is no more than E times longer than the optimal path. In the case where a preferred arrival time t_g is given, and t_g is larger than the optimal arrival time t_o, the loss in path length will be minimal in practice, as any path arriving at t_g is suboptimal anyway. For example, an inflation factor of $E = t_g / t_o$ will still give an "optimal" solution.

In order to have fair experiments in this chapter, we will adopt the so called garage box assumption [5]. That is, we assume that each robot is guaranteed to be collision-free at both its start and its goal configuration. This solves the problem identified above, but more importantly, it ensures that a solution exists to any planning problem, by simply executing the trajectories of the each of the robots one after the other.

4.3.2 Prioritization

An important question is how to assign the different priorities to the robots. There can of course be natural reasons to assign different priorities, for instance based on the importance of the tasks or on different starting times of the robots, but we will not assume that here.

If we have n robots, there are $n!$ different priority schedules, so in general we cannot try them all and select the best one. Yet, the order in which the trajectories are planned can have a significant influence on how optimal the resulting path is (see Fig. 4.2(b)). To still find a near-optimal prioritization in only a few iterations, a randomized search with hill-climbing is applied in [14]. However, this still means that the multi-robot path planning problem is solved multiple times, which costs valuable CPU-time.

We propose a simple heuristic to assign priorities to the robots. Let $D_i(s_i, g_i)$ be the number of steps in the discretized roadmap R_i of robot A_i on the shortest path between its start configuration s_i and its goal configuration g_i. Then, if the other robots do not stand in the way, robot A_i can reach its goal in $D_i(s_i, g_i)\Delta t$ time. We call this number the *query*

distance.

In our heuristic the priority of a robot equals its query distance. The rationale behind this heuristic is – keeping in mind that we aim to minimize the maximum of the arrival times – that robots that have to traverse long distances (and hence need much time) should be able to do this relatively unhindered, while robots that have to traverse short distances can afford to spend time on avoiding robots with higher priority.

The query distances are straightforwardly computed by performing Dijkstra's shortest path algorithm [96] on each of the roadmaps.

4.3.3 An Example

As an example, we applied our method in the environment of Fig. 4.3. All the robots are cylinders that can translate in the plane. Since they are all the same, they can use the same roadmap (note that this is not a requirement of our method). We have constructed a roadmap using a probabilistic roadmap (PRM) method [89]. To have a choice of alternative routes we used a variant that allows the creation of cycles in the roadmap [115]. It consists of 750 vertices and 1004 edges (see Fig. 4.3(a)).

In the roadmap we have defined 12 query configurations (a to l – see Fig. 4.3(a)). Our experiment involves 12 robots with the queries $a \rightarrow g$, $b \rightarrow h$, $c \rightarrow i$, $d \rightarrow j$, $e \rightarrow k$, $f \rightarrow l$, $g \rightarrow b$, $h \rightarrow f$, $i \rightarrow e$, $j \rightarrow a$, $k \rightarrow c$ and $l \rightarrow d$. The start and goal configurations are shown in Fig. 4.3(b) and Fig. 4.3(i), respectively. The environment and the queries are chosen such that a clash might arise in the center of the environment. All the shortest paths between the query configurations go through this center region. There are detours possible that avoid the center, but they are much longer.

To compute a multi-robot path, we ordered the queries of the robots according to their roadmap distance, and executed the method for path planning in dynamic environments from the previous chapter on each of the robots in this order. The dynamic environment in each iteration consists of the trajectories previously computed for the robots with a higher rank. For the robot with the highest rank, the dynamic environment is empty, so its path is returned instantly.

The results are shown in Table 4.1. The multi-robot path was computed in only 1.94s and the arrival time of the last robot is only 13.51, where in any case 11.48 units of time would have been needed to complete a path of the multi-robot. The trajectory is visualized in Fig. 4.3.

An interesting result from Table 4.1 is that the delay-factors are not very high, despite the fact that the environment gets quite crowded with 12 robots. We expect that this is exemplary for path planning in dynamic environments, and that in practical situations the delay-factor will rarely exceed a value of 2. Yet, as we saw in the previous chapter, our method for path planning in dynamic environments works fine even for high delay-factors.

4.4 Analyzing the Prioritization

The scene of Fig. 4.2 was carefully chosen to thwart the prioritized method, but one would typically not encounter such a scene in practice. In this section we examine the influence

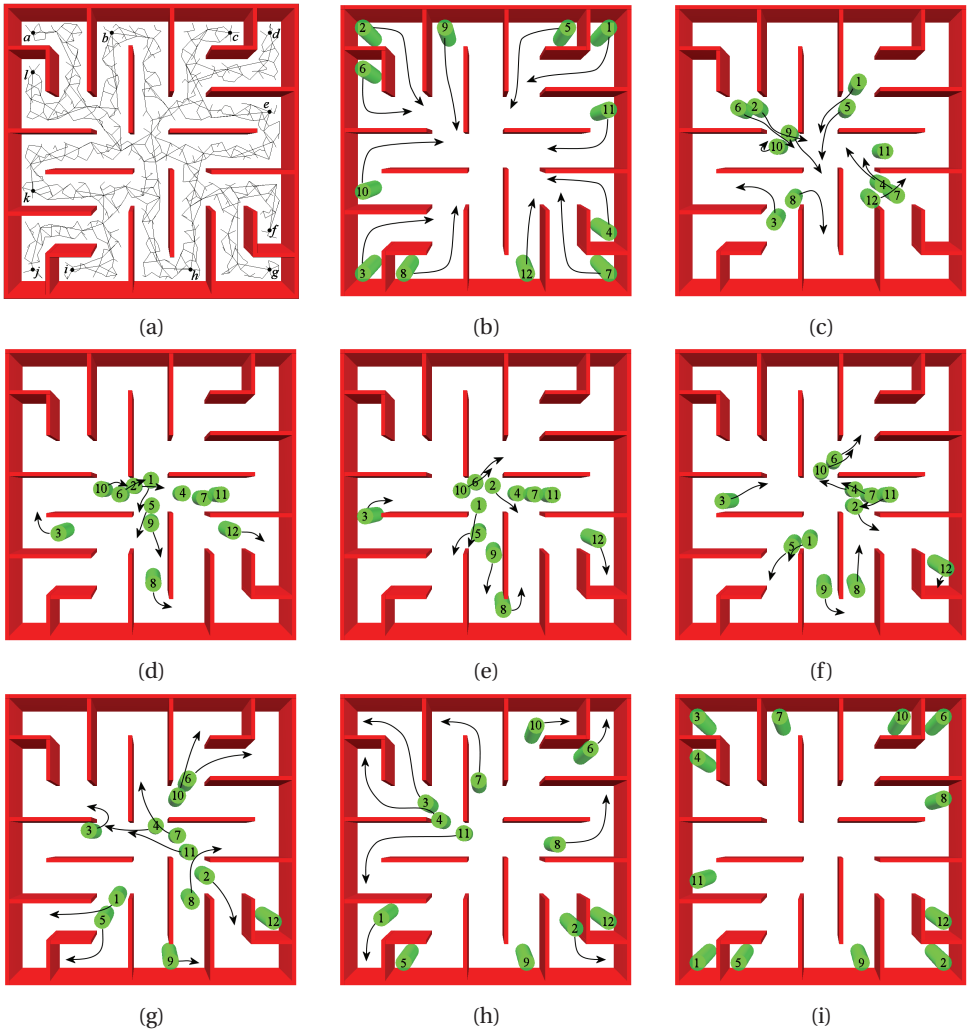


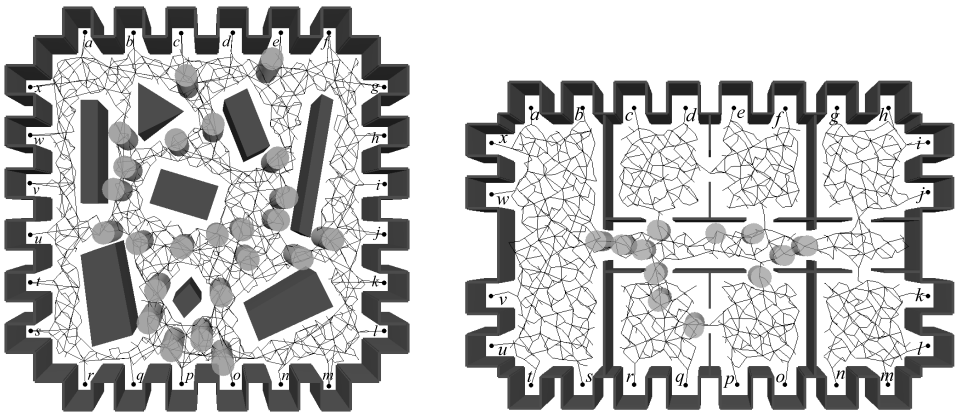
Figure 4.3: Pictures from the multi-robot path. Each robot is labeled according to its rank. The start and goal configurations are shown in (b) and (i), respectively. The robots clash in the center (d). Only robot 3 and 8 choose to take a detour and avoid the bustle.

of the choice of prioritization in more typical environments, and we assess the performance of our prioritization heuristic. We report results in terms of the quality (i.e. the latest of the arrival times) of the produced paths.

We experimented with our method in two environments: the 'clutter' scene and the 'office' scene (see Fig. 4.4). All the robots are cylinders that can translate in the plane, and for both scenes all robots use the same roadmap, which was created using a variant of the PRM method allowing cycles in the roadmap.

Table 4.1: Results for the experiment of Fig. 4.3

robot rank	query	query distance	arrival time	delay-factor	running time
1	$d \rightarrow j$	11.48	11.48	1.00	0.01s
2	$a \rightarrow g$	11.13	11.55	1.04	0.01s
3	$j \rightarrow a$	10.78	13.23	1.23	0.11s
4	$f \rightarrow l$	10.36	13.37	1.29	0.17s
5	$c \rightarrow i$	9.94	9.94	1.00	0.01s
6	$l \rightarrow d$	9.80	11.20	1.14	0.06s
7	$g \rightarrow b$	9.66	12.67	1.31	0.24s
8	$i \rightarrow e$	9.45	12.46	1.32	0.30s
9	$b \rightarrow h$	8.33	8.54	1.03	0.02s
10	$k \rightarrow c$	8.26	10.99	1.33	0.27s
11	$e \rightarrow k$	7.91	13.51	1.71	0.61s
12	$h \rightarrow f$	5.18	7.21	1.39	0.13s
total		11.48	13.51		1.94s



(a) (b)

Figure 4.4: Two environments and their roadmaps with 24 query configurations (a to x). (a) The 'clutter' scene. 24 robots (cylinders) are shown in a configuration along a composite path. The roadmap contains 1500 vertices. (b) The 'office' scene. Here 12 robots are shown. The roadmap contains 1218 vertices.

In both roadmaps we have defined 24 query configurations (a to x – see Fig. 4.4). Our experiments involve 12 robots with the queries $a \rightarrow n$, $c \rightarrow p$, $e \rightarrow r$, $g \rightarrow t$, $i \rightarrow v$, $k \rightarrow x$, $m \rightarrow b$, $o \rightarrow d$, $q \rightarrow f$, $s \rightarrow h$, $u \rightarrow j$ and $w \rightarrow l$. We performed the prioritized method for 500

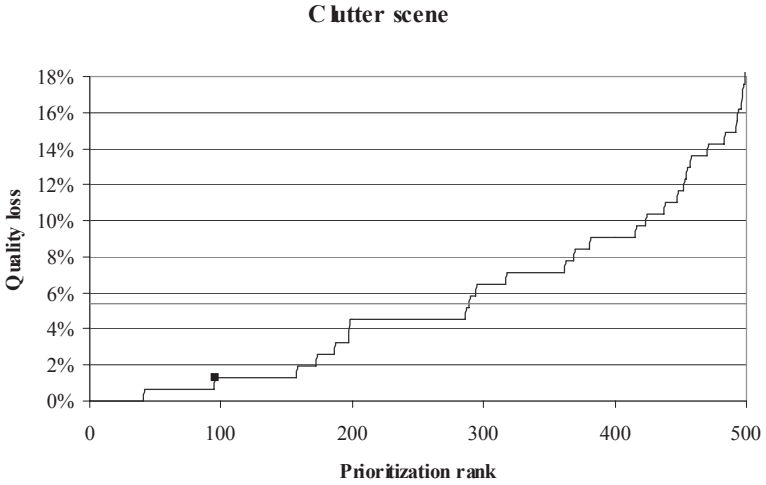


Figure 4.5: Results for 500 random prioritizations in the clutter scene. The prioritizations are shown in order of decreasing quality relative to the optimal path quality. The dotted line indicates the average quality and the quality of our heuristic is indicated with a square.

randomly picked priority schedules. The results are given in Figs. 4.5 and 4.6. In the charts we plotted the priority schedules in order of decreasing quality relative to the optimal path quality (i.e. the maximum of the query distances, which gives a lower bound on the achievable arrival time). This gives a good indication of how the choice of prioritization influences the quality of the resulting path. The quality of our prioritization heuristic is indicated with a square.

From the results we can see that for the clutter scene the choice of prioritization has a moderate influence on the quality of the resulting path. The average path (dotted line) is slightly more than 5% longer than the optimal path. Some prioritizations actually achieve this optimum. Our prioritization heuristic performs well; it performs less than 2% worse than the optimal prioritization and far better than the average.

The office scene is more confined, and here we see that the theoretical optimum cannot be achieved. The average prioritization produces paths that are about 30% longer. A remarkable result for this scene is that the vast majority of the prioritizations produce paths of more or less equal quality and better than the average. A considerable minority however, produces paths of poor quality. Our heuristic again performs well below average; for both scenes approximately 20% of the prioritizations perform better than our heuristic, and 80% worse.

Overall we can say that the choice of prioritization can have a significant influence on the resulting path quality. Yet, our heuristic provides an efficient way to generate high-quality paths, even in confined environments.

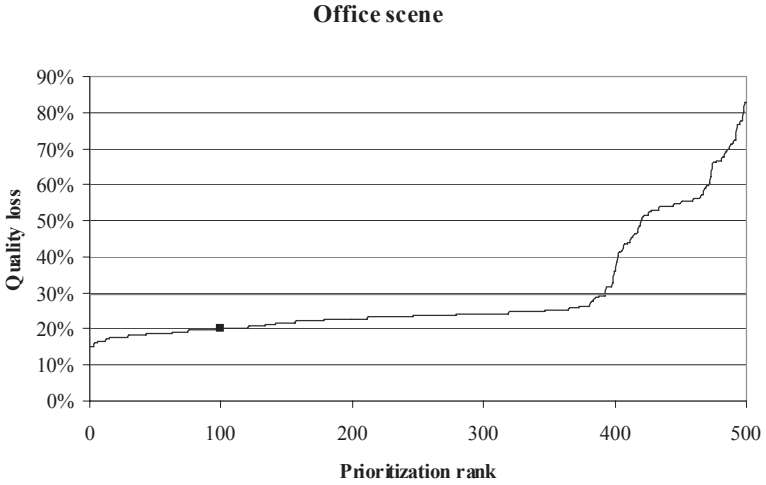


Figure 4.6: Results for 500 random prioritizations in the office scene (b). The prioritizations are shown in order of decreasing quality relative to the optimal path quality. The dotted line indicates the average quality and the quality of our heuristic is indicated with a square.

Our prioritized method is very fast: In the above experiment, the running time to produce a composite path for 12 robots was – averaged over all 500 prioritizations – only 0.76 seconds for the clutter scene, and 0.95 seconds for the office scene. In the environment of Fig. 4.1 we computed a path for four articulated robots between their upright positions and the manipulating configurations as shown in the figure. Again, it took only 0.89 seconds. The experiments were performed on a Pentium IV 3.0GHz with 1 GByte of memory.

4.5 Comparison with a Coordinated Approach

We assess the performance of our prioritized approach, in terms of both the quality of the paths produced and the running time, by comparing it to a roadmap coordination approach introduced in [99]. The method gives the optimal robot coordination, and therefore paths with optimal quality.

4.5.1 Optimal Roadmap Coordination

Given n robots $A_1,\ldots,A_n$, discretized roadmaps $R_1,\ldots R_n$, and start and goal configurations s_i and g_i for each of the robots, the method of [99] finds an optimal composite path in the *roadmap coordination space* $\mathcal{R}$, formed by the Cartesian product $R_1 \times \cdots \times R_n$ of

the roadmaps of the individual robots. The task is to find a path in $\mathcal{R}$ from $(s_1,\ldots,s_n)$ to $(g_1,\ldots g_n)$.

In each time step, the robots may either move one step over the roadmap, or halt at the current position, according to the discretization described in section 4.2.2. This defines a neighborhood in the roadmap coordination space.

The composite path with the minimal latest arrival time is searched using the A*-algorithm [96]. Each state that is encountered during the search is checked for mutual collisions of the robots.

The complexity of the search space is exponential in the number of robots, but the algorithm exploits the *cylindrical* nature of the obstacles in the roadmap coordination space, allowing the number of collision-checks to grow only quadratic with the number of robots.

4.5.2 Experimental Results

We implemented the above algorithm and performed some experiments in the office scene of Fig. 4.4(b). The coordinated approach turned out to be impractical for more than 3 robots. This is mainly caused by the exponential growth of the search space in the number of robots, and the large configuration space that each of the robots have. Somewhat surprisingly, the chosen optimization criterion also plays a major role: in the A*-method, the multi-robot state in the queue with the minimal estimated arrival time of the latest robot is expanded. If for multiple states this value is equal, the estimated arrival time of the second latest robot is considered, etc.

The problem with this optimization criterion is that the move costs of robots with a small query distance are hardly counted. This causes the A*-algorithm to first explore the entire reachable space of the robot with a small query distance (as long as it does not become the robot with the latest estimated arrival time) before considering a (waiting) step of another robot that *does* affect the latest estimated arrival time, even though this step may be crucial for finding an optimal coordination.

This results in large parts of the roadmap coordination space being (unnecessarily) explored. Yet, if we want to find the coordination with the optimal latest arrival time, there is no other option. Other optimization criteria, for instance the sum of the arrival times of the robots, do not have this specific problem, but may perform poorly in other situations. In general, a drawback of the A*-method is that it continuously backtracks to states whose rank in the open-list relies heavily on too optimistic arrival time estimates.

Nevertheless, we performed some experiments with the coordinated method for two optimization criteria. We optimized the maximum and the sum of the arrival times. The experiments involve two and three robots in the office scene. We compare the running time and the path quality to the prioritized approach, for which the prioritization was chosen according to the heuristic we introduced in the previous section. To have a fair comparison between the two methods, both of them use the same roadmaps. The results are given in Table 4.2. The path quality is reported relative to the coordinated approach, which computes optimal paths.

From the results it is clear that the prioritized approach is much faster than the coordinated approach, at the expense of only a small increase in path length. Due to the problems

Table 4.2: Prioritized vs. Coordinated Planning

query set		Coord.	Coord. (sum)		Prioritized	
start	goal	**run. time**	**run. time**	**qual. loss**	**run. time**	**qual. loss**
(w, j)	(k, v)	1.81s	1.17s	0.0%	0.03s	5.6%
(k, v)	(w, j)	1.53s	0.84s	0.0%	0.03s	7.8%
(e, d)	(c, f)	0.70s	0.30s	1.3%	0.01s	2.5%
(a, k, v)	(n, b, j)	6.44m	31.6m	0.5%	0.06s	2.1%
(s, l, j)	(g, v, b)	7.12m	2.27m	0.0%	0.13s	5.1%
(v, c, f)	(k, o, r)	0.01s	0.02s	0.0%	0.02s	0.0%
(k, v, o)	(w, j, r)	>60m	12.53s	0.0%	0.03s	7.8%
(n, g, k)	(h, m, j)	1.65m	21.45s	0.0%	0.05s	0.0%

described above, the coordinated method could not solve the seventh experiment, even though it is a rather simple problem. When we optimize the sum of the arrival times instead of the latest arrival time, this problem is solved relatively fast. In the fourth experiment however, this optimization criterion performs poorly. In general we can conclude that the running times of the coordinated methods too heavily depend on the optimization criterion and on the specific query set to be solved. In practice, it cannot be applied to problems involving 4 or more robots.

In contrast, the performance of the prioritized method scales well with the number of robots. In the clutter scene we performed an experiment involving up to 24 robots. The results are given in Fig. 4.7. Even for 24 robots (the scene gets very crowded then – see Fig. 4.4(a)) the prioritized method returns a path within reasonable running time.

Theoretically, the running time of the prioritized method grows quadratically with the number of robots: If the number of robots is n, then n times a path is planned avoiding $O(n)$ other robots. In the figure we see that the running time grows quickly for more than 20 robots. This is explained by the fact that the space gets more confined when the number of robots is high, giving more difficult problems to solve.

4.6 Analyzing the Spectrum

As stated in the introduction, most approaches to path planning for multiple robots can be categorized along a spectrum indicating how much the motions of the robots are constrained. In the one extreme the motions are constrained to a path, and in the other extreme the motions are not constrained at all. In this section we examine the effect of the extent by which the motions are constrained on the performance of our prioritized method. Again, we report results in terms of both path quality and running times.

Our method was developed to constrain the robot motions to a roadmap, and it can as

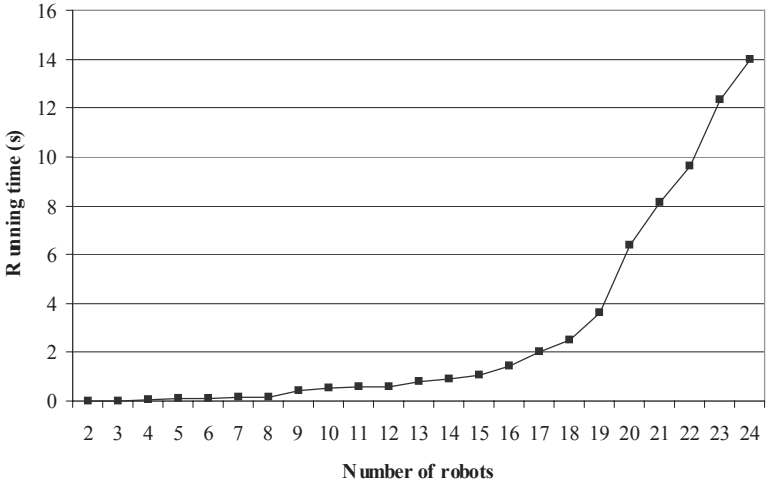


Figure 4.7: Running times of the prioritized method in the clutter scene for an increasing number of robots. To each experiment one extra query is added in this order: $a \rightarrow n$, $m \rightarrow b$, $c \rightarrow p$, $o \rightarrow d$, $e \rightarrow r$, $q \rightarrow f$, $g \rightarrow t$, $s \rightarrow h$, $i \rightarrow v$, $u \rightarrow j$, $k \rightarrow x$, $w \rightarrow l$, $n \rightarrow a$, $b \rightarrow m$, $p \rightarrow c$, $d \rightarrow o$, $r \rightarrow e$, $f \rightarrow q$, $t \rightarrow g$, $h \rightarrow s$, $v \rightarrow i$, $j \rightarrow u$, $x \rightarrow k$ and $l \rightarrow w$. In each experiment the queries are prioritized according to our heuristic.

such be categorized in the middle of the spectrum. It is also applicable when the robots are constrained to paths, which are special instances of roadmaps. Our method is not applicable to unconstrained configuration spaces, but we approximate this situation by increasing the density of the roadmap. As such, we slide along the spectrum between the two extremes.

We experimented in the clutter scene and the office scene with 12 robots having the same queries as in section 4.4. We constructed collision-free paths for the robots by taking the shortest paths from the roadmaps of Fig. 4.4. The density of the roadmaps was varied by tuning the number of vertices. To avoid strong deviations in the results as a consequence of the randomness involved in creating roadmaps, we *extend* roadmaps over the experiments: In each experiment we add a number of vertices to the roadmap of the previous experiment. The number of states in the roadmap grows more or less proportionally. The results are shown in Table 4.3. The path quality is represented here as the absolute arrival time of the latest robot.

From the results we can see that constraining the motions to individual paths indeed gives low running times, but also yields composite paths of poor quality. This is caused by robots that move in opposite directions and share large portions of the same path. This means that one robot has to wait a long time until the other robot has cleared the path, more or less similar to the situation of Fig. 4.2(b).

When we constrain the motions to roadmaps, we see that the path quality improves

Table 4.3: Sliding along the Spectrum

	Clutter scene			Office scene		
roadm. #vertices	**roadm.** #states	**run. time**	**path length**	**roadm.** #states	**run. time**	**path length**
path	~140	0.18s	20.44	~150	0.17s	26.32
200	1980	0.12s	11.06	2187	0.30s	19.11
500	3728	0.35s	10.64	4194	0.59s	19.88
1000	5730	0.35s	10.36	6696	0.56s	14.49
1500	7305	1.09s	10.78	8479	0.70s	14.42
2000	8846	0.97s	10.01	9978	0.86s	14.42
3000	12033	1.33s	10.01	13100	1.76s	14.35
5000	17215	1.82s	10.01	19204	1.80s	14.35
7500	22501	2.38s	10.01	25014	2.15s	14.35

as the roadmaps more densely cover the free configuration space of the individual robots. Moreover, the optimal quality is approached very quickly. After some rather low threshold value (about 2000 vertices) it seems that adding more states to the roadmap hardly helps anymore. This leads us to the conclusion that planning in unconstrained spaces is not useful; constraining motions to roadmaps is much cheaper [99] and approximates the continuous situation very well.

As can be deduced from the results, the running time of our prioritized method only grows linearly with the number of states in the discretized roadmap.

4.7 Conclusion

In this chapter we introduced a generally applicable prioritized approach to path planning for multiple robots. The method is fast; even problems involving as many as 24 robots in confined environments can be solved in mere seconds of computation time. Choosing a good prioritization is a crucial ingredient for a successful planner. Our heuristic appeared to perform well in practice.

In this chapter we assumed that the robots are collision-free on their query configurations, so that we did not have to deal with failures in the experiments. Our method, however, is also applicable without this assumption. In the runs of Fig. 4.7 where more than 12 robots were involved, and in the experiment of Section 4.3.3, for instance, the assumption was violated. It is possible though, that without this assumption the prioritized method will not find a solution, even if one exists. This is, however, unlikely to happen in most environments.

Coordinated approaches do not have this disadvantage (they compute optimal paths), but appeared not to be generally applicable to 4 or more robots. The prioritized approach, in contrast, scales well with the number of robots, at the expense of only a small increase in path length in typical scenes. Prioritized planning has more advantages. It is easily extended

to path planning for multiple robots in dynamic environments. Also, it is easily applicable in situations where multiple robots have different start times and continuously share a common environment.

Finally, we have shown that constraining the robot motions to roadmaps hardly affects the quality of the produced paths: A roadmap of only moderate density already well approximates the continuous, unconstrained configuration space. An additional advantage of using roadmaps is that they are created in a preprocessing phase. This substantially relieves the query phase (e.g. of narrow-passage problems).

Chapter 5

Planning in Partially-Known Dynamic Environments

In the previous two chapters, we discussed an offline planner for completely known dynamic environments. In this chapter we discuss a different domain of the dynamic path planning problem, namely *online* planning in *partially* known dynamic environments. 'Online' means that during planning, the moving obstacles keep on moving, i.e. the time is not "paused" for the planner. 'Partially known' means that we do not know precisely what the future behavior of the moving obstacles is, but have to rely on estimates of future obstacle trajectories to plan a sensible path for the robot. When these estimates are no longer in accordance with the actual obstacle trajectories, we have to *replan* our current solution path. As the obstacles keep on moving during replanning, this has to be done in very timely manner, to be able to react on changing obstacle trajectories in real-time. One goal of this work is to reuse information from previous plans to be able to quickly plan a new path (previous works replan from scratch). To this end, we present an *anytime* planner which is based on the D*-algorithm that is able to quickly replan in *static* environments. We extend this approach such that it is applicable in dynamic environments as well. A concept equivalent to the previous chapters is that we first construct a roadmap for the static environment, and then plan a path avoiding the moving obstacles on this roadmap, in order to make the problem tractable. Results are shown indicating that the approach works in confined environments with many moving obstacles.

This chapter has previously been published as: J. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2366–2371, 2006 [16].

5.1 Introduction

A common task in robotics is to plan a path from an initial robot location to a desired goal location. Depending on the nature of the problem, we may be interested in *any* collision-free path, or one that provides the minimum (or close to minimum) overall cost, where the cost of a path may be a function of several factors including time for traversal, traversal risk, stealth, and visibility.

Several approaches exist for generating such paths. A popular technique is to uniformly discretize the configuration space and then plan an optimal path through this discretized representation using Dijkstra's algorithm or A* (see Chapter 2). However, for more complex configuration spaces, such as those involving robots with several degrees of freedom, using a uniform discretization of the configuration space is intractable in terms of both memory and computation time. As a result, randomized approaches such as Rapidly-exploring Random Trees (RRTs) and Probabilistic Roadmaps (PRMs) have been widely-used in these domains [100, 89]. These approaches work by randomly sampling points in the continuous configuration space and then growing the current solution out towards these points.

Extensions have also been developed to the above approaches that efficiently repair the current solution when changes are made to the configuration space [146, 92, 103, 79, 81, 17]. These considerations are particularly important when we are dealing with a sensor-equipped robot moving through a partially-known environment, as the robot will be continually updating its information through its sensors.

When there are moving obstacles in the environment, the planning problem is more challenging. There are again several approaches that can be taken. First, we can assume the environment is static and use any of the approaches above, then when changes are observed (due to the moving obstacles) we can replan. This can be efficient, but does not take into account any information the robot may have concerning the moving obstacles (for instance, their velocities and directions of movement) and hence may produce highly suboptimal results.

There is a qualitative difference between planning in dynamic environments and planning in static environments that these approaches do not address. For instance, imagine an robot trying to cross a road on which cars are driving. If the robot was to take a snapshot of the environment with its sensors and assume fixed positions for each obstacle, then it would be in serious risk of getting runover if it attempted to cross the road. In order to successfully accomplish this task, the cars really need to be modeled as moving obstacles so that it can be anticipated where they will be at future times.

A second set of approaches does just this, by planning in the joint configuration-time state space. In such a situation, the trajectories of the moving obstacles can be modelled explicitly and taken into account when performing initial planning. Time-optimal or near time-optimal approaches for computing paths through configuration-time space have been developed [96], however these algorithms are typically limited to low-dimensional configuration spaces and/or require significant computation time. In [75, 124], probabilistic approaches are used for computing paths in configuration-time space. These planners incrementally build a tree of explored configurations for each planning query.

More recently, researchers have looked at reducing the complexity of the planning

problem by first constructing a path [53] or a PRM (see Chapter 3) based on the stationary obstacles in the environment, and subsequently planning a collision-free path on this path/roadmap that takes into account the moving obstacles.

The combination of probabilistic sampling and deterministic planning has been found to be particularly useful in generating time-optimal paths through roadmaps with known moving obstacles (see Chapter 3). However, none of the approaches mentioned above efficiently solves the general case, where (i) perfect information, (ii) imperfect information, *and/or* (iii) no information may be available regarding the moving obstacles in the environment, and where the robot's path needs to be quickly repaired when new information is received concerning *either* the moving *or* stationary obstacles of the environment. Further, most of these approaches assume that the only factor influencing the overall quality of a solution is the time taken for the robot to traverse the resulting path. In fact, we are often concerned with minimizing a more general cost associated with the path that may incorporate, for example, traversal risk, stealth, and visibility, as well as time of traverse.

In this chapter, our goal is to combine some of the positive characteristics of several previous algorithms with new ideas to generate an approach that provides an effective solution to the general problem of planning low-cost paths in dynamic environments. Our approach uses probabilistic sampling to create a robust roadmap encoding the planning space, then plans and replans over this graph in an anytime, deterministic fashion. The resulting algorithm is able to generate and repair solutions very efficiently, and improve the quality of these solutions as deliberation time allows.

We begin by describing the problem we are trying to solve. In Section 5.3 we introduce our new algorithm and describe how it relates to current approaches. We present a number of results in Section 5.4. We conclude with extensions and discussion.

5.2 Problem Description

Consider a mobile robot navigating a complex, outdoor environment in which adversaries or other agents exist (such as the one presented in Fig. 5.1). Imagine that this environment contains terrain of varying degrees of traversability, as well as desirable and undesirable areas based on other metrics (such as proximity to adversaries or friendly agents, communication access, or resources). Imagine further that this robot must reach some goal configuration in the environment, while incurring the minimum possible combined cost according to all of the above metrics, as well as (perhaps) time. Specifically, imagine that the cost of the robot's path should be minimized according to some function C, defined as:

$$C(\text{path}) = w_t \cdot t_{\text{path}} + w_c \cdot c_{\text{path}},$$

where t_{path} is the time needed to traverse the path, c_{path} is the cost of the path based on all relevant metrics other than time, and w_t and w_c are the weight factors specifying the extent to which each of these values contributes to the overall cost of the path ($w_t + w_c = 1$).

In a real world scenario it is likely that perfect information concerning the environment will not be available, and that there may be moving obstacles in the environment with autonomous behavior. It is important that it is possible to plan a path for the robot given

Figure 5.1: Data acquired from Fort Benning and used for testing.

various degrees of uncertainty regarding both the stationary and moving obstacles in the environment.

Further, as the robot navigates through this environment, it receives updated information through onboard sensors concerning both the environment itself and the moving obstacles within. Thus, the planning problem is continually changing and it is important that the robot's previous path can be repaired or that a new one can be generated to account for the latest information. This planning must be performed in a very timely manner, as the best solutions may require immediate action and so that solutions are not out-of-date when they are finally generated.

Our aim in this work is to do just this by combining the efficient replanning ability of deterministic incremental planning algorithms with the efficient representations produced by probabilistic sampling approaches. Further, we would like our resulting approach to be anytime, so that solutions can be generated very quickly when deliberation time is limited, and these solutions can be improved while deliberation time allows.

5.3 Approach

Our approach consists of three separate stages. In the first, a roadmap is built representing the static portion of the planning space. Next, an initial path is planned over this roadmap in configuration-time space, taking into account any known moving obstacles. This path is planned in an anytime fashion and is continually improved until the time for deliberation is exhausted. Further, while the robot executes its traverse, its path continues to be improved upon. Finally, when changes are observed, either to the stationary obstacles of the environment (for instance, when a map of the static environment appears to be incorrect) or to the moving obstacles of the environment (for instance, when a moving obstacle changes its course), the current path is repaired to reflect these changes. Again, this is done in an anytime fashion, so that at any time a solution can be extracted.

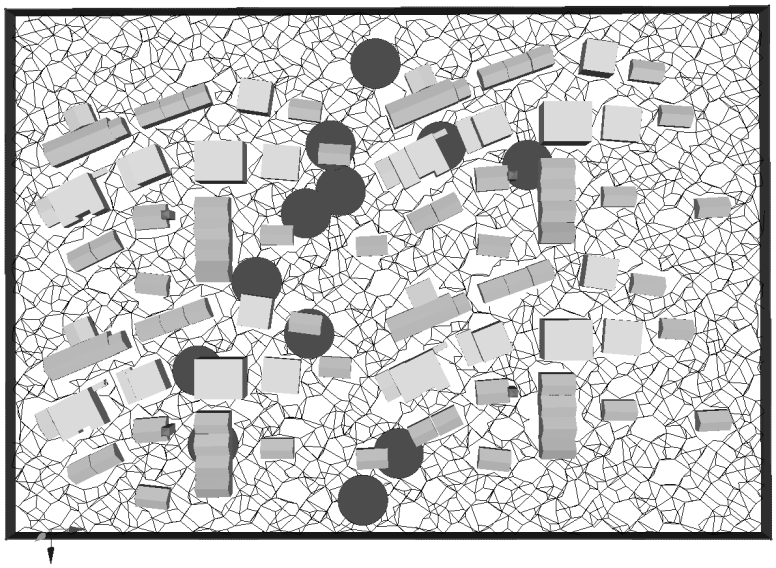


Figure 5.2: A roadmap overlaid onto the Fort Benning data, along with 12 moving obstacles (in red).

5.3.1 Constructing the Roadmap

As in other recent approaches dealing with dynamic environments (e.g. [79]), we begin by creating a PRM taking into account the static portion of the environment (see Chapter 3). This PRM encodes any internal constraints placed on the robot's motion (such as degrees of freedom, kinematic limitations, etc) and takes into account the known costs associated with traversing different areas of the environment. It also includes cycles to allow for many alternative routes to the goal [115]. The objective in this initial phase is to reduce the continuous planning space into a discrete graph that is compact enough to be planned over while still being extensive enough to provide low-cost paths. Fig. 5.2 illustrates a PRM constructed from the outdoor environment presented in Fig. 5.1.

5.3.2 Planning over the Roadmap

We then plan a path over this PRM from the robot's start vertex to its goal vertex, taking into account any known moving obstacles. To do this, we add the time dimension to our search space, so that each *state s* consists of a vertex on the PRM u and a time t. This allows us to represent trajectories of moving obstacles. We discretize the time-axis into small steps of Δt, and allow transitions from state $\langle u, t\rangle$ to state $\langle u, t+\Delta t\rangle$ and to states $\langle u', t+c_t(u,u')\rangle$, where u' is a successor of u in the roadmap and $c_t(u,u')$ is the time it takes to traverse the edge between them. This allows the robot to wait at a particular vertex of the roadmap as well as to transition to an adjacent roadmap vertex. The total cost of transitioning between state

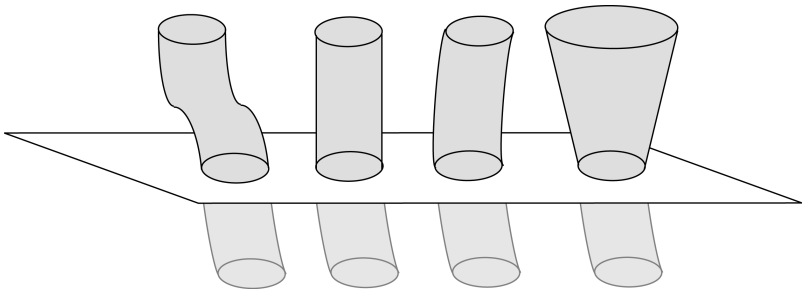


Figure 5.3: Different configuration-time obstacles based on different kinds of information. On the far left is a known trajectory, on the center-left an assumed-stationary obstacle, on the center-right an extrapolated trajectory based on previous motion, and on the far right a worst-case trajectory based on current position and maximum velocity.

$s = \langle u, t \rangle$ and some successor $s' = \langle u', t' \rangle$ is defined as:

$$c(s, s') = w_t \cdot (t' - t) + w_c \cdot c_r(u, u'),$$

where $c_r(u, u')$ is the cost of traversing the edge between u and u' in the roadmap.

Because perfect information concerning the moving obstacles in the environment may not be available, it is important that partial information is adequately coped with. There are a number of existing methods for dealing with this scenario [154, 124]. In particular, we can estimate future trajectories of the obstacles based on current behavior, or we can assume worst-case trajectories. Whichever of these we choose, we end up with some trajectory or set of trajectories that we can represent as 3D objects in our configuration-time space (see Fig. 5.3). We can then avoid these objects as we plan a path for the robot.

Planning a least-cost path through this space can be computationally expensive, and although there may be time to generate the robot's initial path, if the robot is continuously receiving new information as it moves, replanning least-cost paths over and over again from scratch may be infeasible.

Instead, it would be much better if the robot's previous solution could be repaired incrementally, e.g. as in D* and D* Lite [146, 92]. In order to do this, we need to plan in a backwards direction from the goal to the start, so that when the robot moves, the stored paths and path costs of all the states in the search space that have already been computed are not affected[1]. Since we don't know in advance at what time the goal will be reached, we seed the search with multiple goal states. For our implementation,

$$\begin{aligned} GOALS = [&\langle u_{\text{goal}}, h_t(u_{\text{start}}, u_{\text{goal}}) \rangle, \\ &\langle u_{\text{goal}}, h_t(u_{\text{start}}, u_{\text{goal}}) + \Delta t \rangle, \\ &\qquad \vdots \\ &\langle u_{\text{goal}}, \textit{max-arrival-time} \rangle], \end{aligned}$$

[1] If we store a cost-to-goal (as in D*) rather than a cost-to-start (as in A*), then when the robot state (start) changes, the costs are still valid, as the goal to which the costs refer remains unchanged.

where u_{goal} is the goal vertex in the PRM, *max-arrival-time* is the maximum time allowed for traveling to the goal, and h_t is described below.

To improve the efficiency of the search, we use two important heuristic values. First, we compute the minimum possible *time* $h_t(u_{\text{start}}, u)$ for traversing from the current start vertex u_{start} to any particular vertex u in the PRM. Second, we compute the minimum possible *cost* $h_c(u_{\text{start}}, u)$ from the current start vertex to any particular vertex u on the PRM. We then use the time heuristic to prune states added to the search and the cost heuristic to focus the search.

Specifically, if we are searching backwards from the goals and come across some state $s = \langle u_s, t_s \rangle$, then if $t_s - t_{\text{start}} < h_t(u_{\text{start}}, u_s)$ we know that it is not possible to plan a path from the start vertex and start time to this vertex by time t_s, so this state cannot be part of a solution and can be ignored. If the state passes this test, then we insert it into our search queue with a priority based on a heuristic estimate of the overall cost of a path through this state:

$$\text{key}(s) = rhs(s) + w_t \cdot (t_s - t_{\text{start}}) + w_c \cdot h_c(u_{\text{start}}, u_s),$$

where $rhs(s)$ is the current cost-to-goal value of state s. This overall heuristic estimate serves the same purpose as the f-value in classic A* search: it focuses the search towards the most relevant areas of the search space.

However, as already mentioned, there may not be enough time to plan and replan optimal paths across the PRM. Instead, we need to be satisfied with the best solutions that can be generated in the time available. To this end, we make use of a recently developed algorithm, *Anytime D** (AD*), that incrementally plans and replans in an anytime fashion [104].

AD* begins by quickly generating an initial, highly-suboptimal solution, by inflating the heuristic value for each state by some $\epsilon > 1$. It then works on efficiently improving this solution while deliberation time allows, by decreasing ϵ. At any point during its processing, the current solution can be extracted and traversed. Then, as the robot begins its execution, AD* is able to continually improve the solution.

We have included pseudocode of our approach, along with AD*, in Algorithms 5.1 through 5.5.

Algorithm 5.1 KEY(s)

```
1: if g(s) ≥ rhs(s) then
2:     return [rhs(s) + ε · (w_t · (t_s − t_start) + w_c · h_c(n_start, n_s)); rhs(s)]
3: else
4:     return [g(s) + w_t · (t_s − t_start) + w_c · h_c(n_start, n_s); g(s)]
```

5.3.3 Repairing the Plan

While the robot is traveling through the environment, it will be receiving updated information regarding its surroundings through its onboard sensors. As a result, its current solution path may be invalidated due to this new information. However, it would be prohibitively expensive to replan a new path from scratch every time new information arrives.

Algorithm 5.2 UPDATESETMEMBERSHIP(s)

```
1: if g(s) ≠ rhs(s) then
2:    if s ∉ CLOSED then insert/update s in OPEN with key(s)
3:    else if s ∉ INCONS then insert s into INCONS
4: else
5:    if s ∈ OPEN then remove s from OPEN
6:    else if s ∈ INCONS then remove s from INCONS
```

Algorithm 5.3 COMPUTEPATH()

```
1: while min_{s∈OPEN}(key(s)) < key(s_start) or rhs(s_start) > g(s_start) do
2:    remove state s with the smallest key(s) from OPEN
3:    if g(s) > rhs(s) then
4:       g(s) ← rhs(s); CLOSED ← CLOSED ∪ {s}
5:       for each predecessor s' of s do
6:          if s' was not visited before then
7:             g(s') ← rhs(s') ← ∞; bp(s') ← NULL
8:          if rhs(s') > g(s) + c(s', s) then
9:             bp(s') ← s
10:            rhs(s') ← g(s) + c(s', s); UpdateSetMembership(s')
11:   else
12:      g(s) ← ∞; UpdateSetMembership(s)
13:      for each predecessor s' of s do
14:         if s' was not visited before then
15:            g(s') ← rhs(s') ← ∞; bp(s') ← NULL
16:         if bp(s') = s then
17:            bp(s') ← argmin_{s''∈Succ(s')}(g(s'') + c(s', s''))
18:            rhs(s') ← g(bp(s')) + c(s', bp(s')); UpdateSetMembership(s')
```

Instead, our approach is able to repair the previous solution, in the same way as incremental replanning algorithms such as D* and D* Lite. However, as with initial planning, it is also able to do this repair in an anytime fashion. Thus, solutions can be improved and repaired at the same time, allowing for true interleaving of planning, execution, and observation.

To do this, it first updates the heuristic values of states on the roadmap based on the current configuration of the robot. This can be done very quickly as it only concerns the static environment's roadmap. It then finds the states in the search tree that have been affected by the new information and updates these states. As we discuss in the following section, this can also be performed efficiently.

Algorithm 5.4 MAIN()

```
1: construct PRM of static portion of environment
2: use PRM and Dijkstra's to extract predecessor and successor functions, heuristic functions h_c and h_t, and goal list GOALS
3: g(s_start) ← rhs(s_start) ← ∞
4: OPEN ← CLOSED ← INCONS ← ∅; ε ← ε_0
5: for each s_goal in GOALS do
6:    g(s_goal) ← ∞; rhs(s_goal) ← 0; bp(s_goal) ← NULL
7:    insert s_goal into OPEN with key(s_goal)
8: fork MoveAgent()
9: while s_start ∉ GOALS do
10:   ComputePath()
11:   publish ε-suboptimal solution path
12:   if ε = 1 then wait for changes in search tree edge costs
13:   for all directed search tree edges (s, s') with changed edge costs do
14:      c_old ← c(s, s'); update the search tree edge cost c(s, s')
15:      if s ∉ GOALS then
16:         if s was not visited before then
17:            g(s) ← rhs(s) ← ∞; bp(s) ← NULL
18:         if c(s, s') > c_old and bp(s) = s' then
19:            bp(s) ← argmin_{s''∈Succ(s)} g(s'') + c(s, s'')
20:            rhs(s) ← g(bp(s)) + c(s, bp(s)); UpdateSetMembership(s)
21:         else if rhs(s) > g(s) + c(s, s') then
22:            bp(s) ← s'
23:            rhs(s) = g(s') + c(s, s'); UpdateSetMembership(s)
24:   if significant search tree edge cost changes were observed then
25:      increase ε or re-plan from scratch
26:   else if ε > 1 then decrease ε
27:   move states from INCONS into OPEN
28:   update the priorities for all s ∈ OPEN according to key(s)
29:   CLOSED ← ∅
```

5.4 Experiments and Results

5.4.1 Implementation Details

As said in the previous section, when the robot observes changes regarding the trajectory of the moving obstacles, we should first find all edges (i.e. transitions between states) in the search tree in configuration-time space considered so far whose collision status must be updated. (We will refer to these edges as *search tree edges* to distinguish them from roadmap edges.) This can potentially be rather expensive, but if we model both the robot and the moving obstacles as discs moving in the plane, we can do this very efficiently. First, we add the radius of the robot to the radii of each obstacle, so that we can treat the robot as a point. Next, we mathematically describe the volumes in the configuration-time space carved out

Algorithm 5.5 MOVEAGENT()

```
1:  while s_start ∉ GOALS do
2:    update s_start to be successor of s_start in current solution path
3:    use Dijkstra's to recompute h_c and h_t given the new value of s_start
4:    while robot is not at s_start do
5:      move robot towards s_start
6:      if new information is received concerning the static environment then
7:        update the affected edges in the PRM
8:        update the successor and predecessor functions
9:        use Dijkstra's to recompute h_c and h_t
10:       mark the affected search tree edges in the AD* search as changed (so that these
          are triggered in Algorithm 5.4, line 13)
11:     if new information is received concerning the moving obstacles then
12:       mark the affected search tree edges in the AD* search as changed (so that these
          are triggered in Algorithm 5.4, line 13)
```

by each moving obstacle. For instance, if the estimated trajectory of a moving obstacle is an extrapolation of its current velocity, this volume becomes a slanted cylinder, which can be described as

$$((x - x_0) - (t - t_0)\,v_x)^2 + ((y - y_0) - (t - t_0)\,v_y)^2 = r^2,$$

where (x_0, y_0) is the current position of the obstacle, (v_x, v_y) its current velocity, r its radius, and t_0 the current time. If we assume the obstacles to be stationary, this volume becomes a vertical cylinder, and if we model worst-case trajectories, this volume becomes a cone (provided that the maximal velocity of the obstacle is given). In any case, it is easily checked whether or not a line segment in configuration-time space (e.g. a search tree edge) intersects these volumes. Experiments show that it is also very fast; over 50000 search tree edges can be checked within 0.01 seconds.

As the environment is dynamic, these estimated future trajectories may change over time. For instance, if we use the extrapolation method, whenever an obstacle changes its velocity we should re-check all search tree edges against its new volume. However, given an indication of the frequency of trajectory changes (the dynamicity of the environment) we can set some *horizon* on the validity of the estimated trajectory such that only search tree edges in the near future are collision-checked. Search tree edges in the far future are simply considered to be collision-free. As time goes by, these search tree edges are eventually checked as well. This facilitates replanning as collision-checks become faster and less configuration-time space becomes (unnecessarily) inaccessible when searching for a path.

5.4.2 Experimental Setup

We experimented with our method in the environment of Figs. 5.1 and 5.2. The robot (a point) has to move from the lower-rightmost vertex in the roadmap to the upper-leftmost vertex. The roadmap consists of 4000 vertices and 6877 edges. Twelve disc-shaped moving obstacles (see Fig. 5.2) start in the centre of the scene and spread out in random directions

with random velocities. During the traversal of the path by the robot, the obstacles may change their course. These changes are randomly generated and are not known beforehand and can not be taken into account by the planner. In our experiments the total number of course changes varies around 100. Moreover, when an obstacle hits the boundary of the scene, it is bounced backwards, which accounts for additional course changes (these could be anticipated by the planner, but in our implementation they are not). The obstacles heavily impede the robot in its attempt to reach the goal vertex. For the sake of experimentation, the cost values for traversing edges in the roadmap were chosen to be random variations of their length. The time axis was discretized into intervals of 0.1 seconds.

The initial value of ϵ is 10, and this is gradually decreased to 1 as deliberation time allows. After every 0.1 seconds, we check whether the collision status of any edge of the search tree in configuration-time space encountered thus far has changed with respect to the obstacles. Each time this occurs, ϵ is reset to 10 to quickly repair the path. Meanwhile, the position of the robot along its path is updated every 0.1 seconds according to the best available path. This continues until the robot has reached the goal. The experiments were performed on a Pentium IV 3.0GHz with 1 Gbyte of memory.

In our experiments we used two models for estimating the future course of the obstacles: the extrapolation method and the assumed-stationary method (see Fig. 5.3). In the extrapolation method we assume that the current course of an obstacle (a linear motion) is also its future course. This gives a slanted cylindrical obstacle in configuration-time space, which is avoided during planning. Fig. 5.4 shows an example path planned through configuration-time space, along with the extrapolated trajectories of the moving obstacles. In the assumed-stationary case, we assume the obstacles to be stationary (analogous to several previous approaches), giving vertical cylindrical configuration-time obstacles. In each iteration of the algorithm the position of the obstacle is updated according to its actual trajectory. For both models the horizon was set to 10 seconds. We also implemented the worst-case model (assuming a maximum velocity for each obstacle), but this approach was not useful for this problem, as the obstacles could move so quickly that their worst-case conical volumes quickly grew so large that no feasible paths existed. The advantage of this model is that the planned paths are inherently safe, whereas for the other models this is not the case (as we will see below). In the next chapter, we discuss a method designed especially for the worst-case model with conical configuration-time obstacles.

5.4.3 Results

In our experiments we compared the PRM-based Anytime D* method (in which ϵ is regularly reset to 10) to a PRM-based D* Lite method (in which ϵ is always 1). The Anytime D* method was run in real-time, so that the robot was moved along its current path regardless of whether it had finished repairing or improving the path. The D* Lite method was not run in real-time and as much time as needed for planning in between robot movements was allowed (in other words, time was 'paused' for the planner). We included this latter approach simply to demonstrate the relative efficiency and solution quality of Anytime D* compared to an optimal planner.

For each run (in which the path is improved and repaired many times) we measured

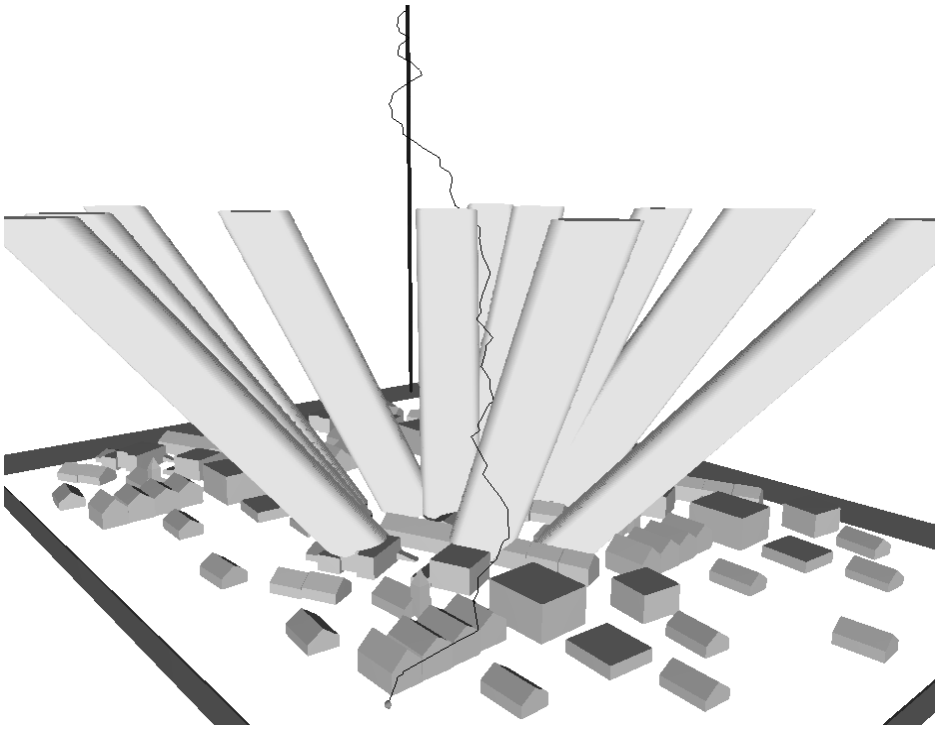


Figure 5.4: An example path (in black) planned through configuration-time space from the robot's start configuration (in green) to the goal (shown as a vertical blue line extending from the goal configuration upwards through time). Also shown are the extrapolated trajectories of the moving obstacles (in yellow). The underlying PRM has been left out for clarity but can be seen in Fig. 5.2.

the maximum time needed to improve or repair the path. As the obstacle trajectories are randomly generated, we performed 50 runs using each method with the same random sequence, and averaged the maximal planning times.

In addition, we measured the average overall cost of the path, and the number of times the generated path was not safe (i.e. in which there was a collision between the robot and the obstacles). Results are reported for both the extrapolation model and the stationary model in Table 5.1.

From the results we can see that for both models the maximal amount of time needed to replan the path is on average three to four times less for Anytime D* than for D* Lite. The path quality did not suffer much from the anytime-characteristic: for both Anytime D* and D* Lite the average path costs are about the same.

If we compare the extrapolation model (in which we use information of the current velocity of the obstacles) to the assumed-stationary case, we see that the assumed stationary case is not very safe. In approximately half of the cases, the robot hits an obstacle. In the

Table 5.1: Results (averaged over 50 runs)

Obstacle Model	Approach	Cost	Max.time	Invalid
Extrapolation	Anytime D*	81.07	0.19s	22%
Extrapolation	D* Lite	80.20	0.74s	18%
Assumed-Stationary	Anytime D*	85.09	0.22s	52%
Assumed-Stationary	D* Lite	81.08	0.67s	58%

extrapolation case about one fifth of the runs result in a collision. These collisions are explained by the radical course changes the obstacles can make; if the robot moves alongside an obstacle, and suddenly the obstacle decides to take a sharp turn, there may not be enough time for the robot to escape a collision (also because the velocity of the obstacle is unbounded). We expect more realistic obstacle behavior, more conservative trajectory estimation, as well as penalizing being close to a moving obstacle in the costs of the edges in the search tree, will remedy this.

5.4.4 Extensions

It may be possible to make this approach even more efficient by using an expanding horizon for the moving obstacles. Specifically, when producing an initial plan we set the horizon for each obstacle to zero, so that a path is produced very quickly. Then, as deliberation time allows, we improve the accuracy and robustness of this path by gradually increasing the horizon. This modification may significantly reduce the time required to generate an initial path.

5.5 Discussion

We have presented an approach for anytime path planning and replanning in partially-known, dynamic environments. Our approach handles the case where (i) perfect information, (ii) imperfect information, *and/or* (iii) no information may be available regarding the moving obstacles of the environment, and where the robot's path needs to be quickly repaired when new information is received concerning *either* the moving *or* stationary obstacles of the environment. We have shown it to be capable of solving large instances of the navigation problem in dynamic, non-uniform cost environments.

Our approach combines research in deterministic and probabilistic path planning. We are unaware of any approaches to date that combine the strong body of work on deterministic replanning algorithms with compact, probabilistically-generated representations of the environment, and yet we believe the union of these two areas of research can lead to very effective solutions to a wide range of problems. As such, we are currently looking at how some of these ideas can be applied to other robotics domains.

Chapter 6

Planning in Unpredictable Dynamic Environments

In the previous chapter, we presented a general framework for path planning in partially known dynamic environments, where the future behavior of the obstacles is estimated based on current sensor information. In Section 5.3.2, we mentioned three ways to model the future obstacle motions into configuration-time obstacles, one of which was called the worst-case model, in which the region where an obstacle can be is a disc that grows over time with a rate corresponding to the maximum velocity of the obstacles. This gives conical obstacles in the configuration-time space. One of the advantages of this model is that the planned paths are inherently safe, as the obstacles will always stay within the growing regions, no matter what they do (assuming that the obey their maximum velocity). Hence, there is no need for precise motion predictions. The method presented in the previous chapter was not good enough to deal with these kind of obstacles, as the available free space is quickly consumed by fast growing obstacle regions. In this chapter we will specifically focus on this model, and discuss the problem of planning safe paths amidst unpredictably moving obstacles in the plane. We present an approach to compute the *shortest path* between two points in the plane that avoids the growing discs associated with the regions in which the moving obstacles can be. The generated paths are thus guaranteed to be collision-free with respect to the moving obstacles while being executed. We created a fast implementation that is capable of planning paths amidst many growing discs within milliseconds.

This chapter has previously been published as: J. van den Berg and M. Overmars. Planning the shortest safe path amidst unpredictably moving obstacles. In *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006 [24]. An extended abstract has appeared as [22].

6.1 Introduction

An important challenge in robotics is path planning in dynamic environments. That is, planning a path for a robot from a start location to a goal location that avoids collisions with the moving obstacles. In many cases the motions of the moving obstacles are not known beforehand, so often their future trajectories are estimated by extrapolating current velocities (acquired by sensors) in order to plan a path [16, 49, 154]. This path may become invalid when some obstacle changes its velocity (say at time t), so then a new path should be planned (see also the previous chapter). However, there is actually no time for planning; as the world is continuously changing, the computation would already be outdated even before it is finished.

To overcome this problem, often a fixed amount of time, say τ, is reserved for planning [75, 124]. The planner then takes the expected situation of the world at time $t+\tau$ as initial world state, and the plan is executed when the time $t+\tau$ has come. This scheme carries two problems:

- The predicted situation of the world at time $t+\tau$ may differ from the actual situation when some obstacles change their velocities again during planning. This may result in invalid paths.

- The path the robot will follow between time t and time $t+\tau$ is not guaranteed to be collision-free, because this path was computed based on the old velocities of the obstacles.

In this chapter we take a first step to overcome these problems. We present an approach to compute a path from a start location to a goal location that is guaranteed to be collision-free, no matter how often the obstacles change their velocities in the future. Replanning might still be necessary from time to time, to generate trajectories with more appealing global characteristics, but the two problems identified above do not occur in our case. The first problem is solved by incorporating all the possible situations of the world at time $t+\tau$ in the world model. The second problem is solved as the computed paths are guaranteed to be collision-free regardless of what the moving obstacles do.

We assume that all obstacles and the robot are modeled as discs in the plane, and that the robot and each of the obstacles have a (known) maximum velocity. The maximum velocity of the obstacles should not exceed the maximum velocity of the robot. The problem is solved in the configuration space, that is, the radius of the robot is added to the radii of the obstacles, so that we can treat the robot as a point.

Given the initial positions of each of the obstacles, the regions of the space that are possibly not collision-free are modeled by discs that grow over time with rates corresponding to the maximal velocities of the obstacles. Our goal is to compute a *shortest path* (a minimum time path) from a start to a goal configuration that avoids these growing discs (see Fig. 6.1).

Although computing shortest paths is a well studied topic in computational geometry (see [110] for a survey), the problem we study in this chapter is new. In fact, it is a three-dimensional shortest path problem, as the time accounts for an additional dimension. Such problems are NP-hard in general, yet we present an $O(n^3 \log n)$ algorithm (n being the num-

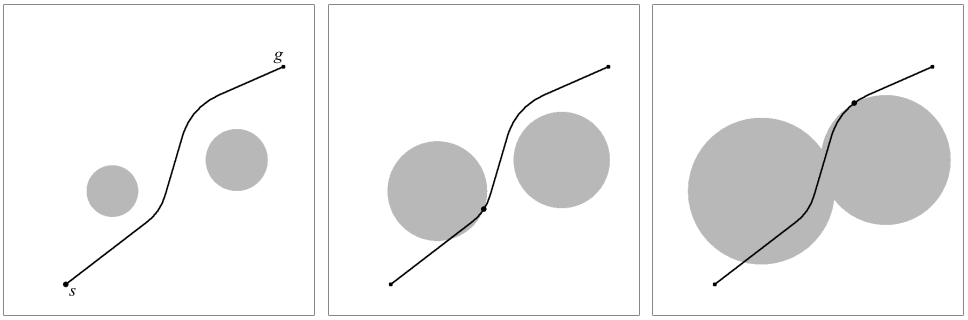


Figure 6.1: An environment with two moving obstacles and a shortest path. The pictures depict the growing discs at $t = 0$, $t = 1$ and $t = 2$, respectively. A small dot indicates the position along the path.

ber of discs) for our problem in the restricted case that all discs have the same growth rate.[1] In case the growth rates are different, we cannot give a time bound expressed in n. Instead, we present an efficient heuristic algorithm for this general case. We implemented this algorithm, and experimental results show that we are able to generate shortest paths amidst many growing discs within only milliseconds of computation time.

The rest of the chapter is organized as follows. We formally define the problem in Section 6.2. In Section 6.3 we examine the structure of shortest paths amidst growing discs. We sketch our global approach in Section 6.4, and in Section 6.5 we present efficient algorithms for the restricted and general case. Experimental results are given in Section 6.6, and Section 6.7 concludes the chapter.

6.2 Problem Definition

The problem is formally defined as follows. Given are n moving obstacles $O_1, \ldots, O_n$ which are discs in the plane. The centers of the discs (i.e. the positions of the obstacles) at time $t = 0$ are given by the coordinates $p_1, \ldots, p_n \in \mathbb{R}^2$, and the radii of the discs by $r_1, \ldots, r_n \in \mathbb{R}^+$. All of the obstacles have a maximal velocity, given by $v_1, \ldots, v_n \in \mathbb{R}^+$. The robot is a point (if it is a disc, it can be treated as a point when its radius is added to the radii of the obstacles), for which a path should be found between a start configuration $s \in \mathbb{R}^2$ and a goal configuration $g \in \mathbb{R}^2$. The robot has a maximal velocity $V \in \mathbb{R}^+$ which should be larger than each of the maximal velocities of the obstacles, i.e. $\forall (i :: V > v_i)$.

As we do not assume any knowledge of the velocities and directions of motion of the moving obstacles, other than that they have a maximal velocity, the region that is guaranteed to contain all the moving obstacles at some point in time t is bounded by $\bigcup_i B(p_i, r_i + v_i t)$, where $B(p, r) \subset \mathbb{R}^2$ is an open disc centered at p with radius r. In other words, each of the

[1] Note that the special case of discs with zero growth rate gives a two-dimensional shortest path problem, which can be solved in $O(n^2 \log n)$ time (see e.g. [37]).

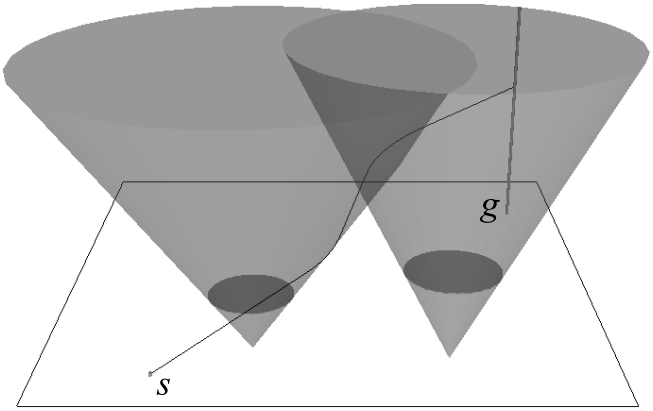


Figure 6.2: The three-dimensional space of the same environment as Fig. 6.1.

moving obstacles is conservatively modeled by a disc that grows over time with a rate corresponding to its maximal velocity (see Fig. 6.1 for an example environment).

Definition 6.1. *A point* $p \in \mathbb{R}^2$ *is* collision-free *at time* $t \in \mathbb{R}^+$ *if* $p \notin \bigcup_i B(p_i, r_i + v_i t)$.

The task is to compute the shortest possible path $\pi : [0, t_g] \to \mathbb{R}^2$, such that $\pi(0) = s$ and $\pi(t_g) = g$ (i.e., a minimal time path with minimal t_g) that is collision-free with respect to the growing discs, i.e. $\forall (t \in [0, t_g] :: \pi(t) \notin \bigcup_i B(p_i, r_i + v_i t))$. Further, the path π should obey the maximum velocity constraint V of the robot, i.e. $\forall (t_1, t_2 \in [0, t_g] : t_1 < t_2 : d(\pi(t_1), \pi(t_2)) \leq V(t_2 - t_1))$, where $d : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ is the Euclidean distance between two points in the plane.

6.3 Properties of Shortest Paths

In this section we deduce some elementary properties of shortest paths amidst growing discs. We first show that we are actually dealing with a three-dimensional path planning problem: As the discs grow over time, we can see the obstacles as cones in a three-dimensional space (see Fig. 6.2), where the third dimension represents the time. Each obstacle O_i transforms into a cone C_i, whose central axis is parallel to the time-axis of the coordinate frame, and intersects the xy-plane at point p_i. The maximal velocity v_i determines the opening angle of the cone, and together with the initial radius r_i, it determines the (negative) time-coordinate of the apex. The equation of cone C_i is given by:

$$C_i : (x - p_{ix})^2 + (y - p_{iy})^2 = (v_i t + r_i)^2. \tag{6.1}$$

The goal configuration g is transformed into a line parallel to the time-axis, where we want to arrive as soon as possible (i.e., for the lowest value of t). In the three-dimensional space it is easier to reason about the properties of shortest paths.

6.3.1 Maximal Velocity

We will first show that a shortest path is always traversed at the maximal velocity V, and hence a shortest path makes a constant angle $\arctan(1/V)$ with the xy-plane.

Lemma 6.2. *A point $p \in \mathbb{R}^2$ that is collision-free at time $t = t'$, is collision-free for all $t :: 0 \leq t \leq t'$.*

Proof: If $t_1 \leq t_2$, we know that $\bigcup_i B(p_i, r_i + v_i t_1) \subseteq \bigcup_i B(p_i, r_i + v_i t_2)$. Thus if a point p is collision-free at time t_2, i.e. $p \notin \bigcup_i B(p_i, r_i + v_i t_2)$, it is certainly not in $\bigcup_i B(p_i, r_i + v_i t_1)$. Hence point p is collision-free at time t_1 as well. ∎

Theorem 6.3. *The velocity $\frac{\|(\delta x, \delta y)\|}{\delta t}$ of a shortest path is constant and equal to the maximal velocity V.*

Proof: Suppose π is a path to g, of which a sub-path has a velocity smaller than V. Then this sub-path could have been traversed at maximal velocity, so that points further along the path would be reached at an earlier time. Lemma 6.2 proves that these points are then collision-free as well, so also g could have been reached sooner, and hence π is not a shortest path. ∎

6.3.2 Straight-Line Segments and Spiral Segments

Next, we prove that that a shortest path can only consist of straight-line motions, and motions that stay in contact with the growing discs. These latter motions are parts of a spiral, as the path 'orbits' a disc while it grows. In three-dimensional terms, a spiral segment of a shortest path lies on the surface of a cone.

Theorem 6.4. *A shortest path solely consists of straight-line segments, and spiral segments on the boundary of a growing disc.*

Proof: Theorem 6.3 implies that the time it takes to traverse a path is proportional to its length. Hence, parts of the path in 'open' space can always be shortcut by a straight-line segment. Only when the path stays in contact with a growing disc, it is not possible to shortcut. These segments are part of a spiral. ∎

We next show that in fact, as both the velocity of the path and the growth rate of the discs are constant, spiral segments on the boundary of a disc are supported by a *logarithmic spiral*.

Without loss of generality, we assume that the disc has radius 0 at $t = 0$, that the disc is centered at the origin, and that the disc grows with velocity 1 (other discs can be transformed such that these conditions hold). Let the velocity of the path be V. We express the equations of the spiral in polar coordinates $(r(t), \theta(t))$, parametrized by the time t. The radius $r(t)$ of the spiral at time t is equal to the radius of the disc at time t, thus:

$$r(t) = t. \tag{6.2}$$

The angle $\theta(t)$ is not trivially deduced, but we know that

$$\sqrt{x'(t)^2 + y'(t)^2} = V, \tag{6.3}$$

as the velocity along the spiral is constantly equal to V. From this equation, we deduce a closed form for $\theta(t)$:

$$\begin{aligned}
\{x(t) &= r(t)\cos\theta(t), \\
y(t) &= r(t)\sin\theta(t)\}, \\
\{x'(t) &= r'(t)\cos\theta(t) - r(t)\theta'(t)\sin\theta(t), \\
y'(t) &= r'(t)\sin\theta(t) + r(t)\theta'(t)\cos\theta(t)\}, \\
x'(t)^2 + y'(t)^2 &= r'(t)^2 + r(t)^2\theta'(t)^2 = 1 + t^2\theta'(t)^2.
\end{aligned} \tag{6.4}$$

Combining Equation (6.4) with (6.3), and solving for $\theta(t)$ gives:

$$\begin{aligned}
\sqrt{1 + t^2\theta'(t)^2} &= V, \\
1 + t^2\theta'(t)^2 &= V^2, \\
\theta'(t) &= \pm\frac{\sqrt{V^2 - 1}}{t}, \\
\theta(t) &= \pm\sqrt{V^2 - 1}\log t + \theta_0.
\end{aligned} \tag{6.5}$$

The $\pm$ indicates whether the spiral revolves counterclockwise (+), or clockwise (−) about the growing disc. The term θ_0 gives the starting angle of the spiral. Equations (6.2) and (6.5) together define the spiral. This form is well known as the *logarithmic spiral* [162].

6.3.3 Path Smoothness

Theorem 6.5. *A shortest path is C^1-smooth.*

Proof: Suppose path π is not smooth and contains sharp turns. Then these turns could be shortcut by a straight-line segment. Hence π is not a shortest path. ■

This theorem implies that in a (general) shortest path the straight-line segments and spiral segments alternate each other, and that the straight-line segments must be *tangent* to the supporting spirals of the spiral segments. In terms of the three-dimensional space this means that the straight-line segments (which “connect" two spiral segments), are *bitangent* to the cones on which the spirals lie.

6.3.4 Departure Curves

There are four ways in which a straight-line segment can be bitangent to a pair of cones (say C_i and C_j): left-left, right-right, left-right and right-left. In each of these cases, there is an infinite number of possible segments (whose slope corresponds to the maximal velocity V) that are tangent to both C_i and C_j. However, the possible tangency points at the surface of C_i form a continuous curve on that surface. We call such curves *departure curves*. They play a major role in our algorithm to compute a shortest path.

Definition 6.6. *For two cones C_i and C_j, the set $DC(C_i, C_j)$ is defined as the collection of points on the surface of C_i, for which the straight-line of slope $1/V$ that is tangent to C_i in that point is also tangent to C_j. The set $DC(C_i, C_j)$ consists of four continuous curves, each associated with one of the tangency cases. We call them* departure curves. *They are denoted $DC_{ll}(C_i, C_j)$, $DC_{lr}(C_i, C_j)$, $DC_{rl}(C_i, C_j)$ and $DC_{rr}(C_i, C_j)$, respectively.*

The set $DC(C_i, g)$ to the goal configuration g is defined similar, but then the tangent line segment should go through the goal configuration g. In this case the departure curves $DC_r(C_i, g)$ and $DC_l(C_i, g)$ are distinguished.

We now show how we can deduce equations for the departure curves on the surface of a cone C. Again, without loss of generality, we assume that the disc associated with the cone has radius 0 at $t = 0$, that the disc is centered at the origin, and that the disc grows with velocity 1. Let the velocity of the path be V. The surface of C can be parametrized by two variables, time T and angle Θ:

$$C : (T, \Theta) \rightarrow \{T\cos\Theta, T\sin\Theta, T\}.$$

Let us consider the counterclockwise spirals about this cone. Each of them is uniquely defined by the initial angle θ_0 (see Equation (6.5)). Each point (T, Θ) on the surface of the cone has a unique spiral that goes through that point. This spiral can be found by solving $\theta(T) = \Theta$ for θ_0:

$$\theta_0 \quad = \quad -\sqrt{V^2-1}\log T + \Theta. \tag{6.6}$$

Hence, the spiral though (T, Θ) is described in Euclidean coordinates as:

$$\begin{aligned} \{x(t) &= t\cos\left(\sqrt{V^2-1}\log t - \sqrt{V^2-1}\log T + \Theta\right), \\ y(t) &= t\sin\left(\sqrt{V^2-1}\log t - \sqrt{V^2-1}\log T + \Theta\right)\}. \end{aligned}$$

If we walk along this spiral, we can depart for another cone if the straight-line segment tangent to the spiral is tangent to another cone as well. The straight-line segment ℓ tangent to the spiral at point (T, Θ) is represented by:

$$\begin{aligned} \ell(t) &= \left\{x(T) + (t-T)x'(T),\ y(T) + (t-T)y'(T)\right\} = \\ &= \left\{t\cos\Theta - (t-T)\sqrt{V^2-1}\sin\Theta,\ t\sin\Theta + (t-T)\sqrt{V^2-1}\cos\Theta\right\}. \end{aligned} \tag{6.7}$$

This segment must be tangent to another cone, say C_i with position p_i, initial radius r_i and velocity v_i, in order for point (T, Θ) to be on a departure curve of $DC(C, C_i)$. The surface of C_i is given by Equation (6.1). If we fill in line ℓ in (6.1), by substituting $x = \ell_x(t)$ and $y = \ell_y(t)$, and solve for t, we get a solution of the following form:

$$t_{1,2} = A(T, \Theta) \pm \sqrt{D(T, \Theta)}. \tag{6.8}$$

Here, $D(T, \Theta)$ is the discriminant whose sign indicates whether or not line ℓ intersects C_i. When $D(T, \Theta) = 0$, ℓ is tangent to C_i, hence $D(T, \Theta) = 0$ is an implicit equation for the set

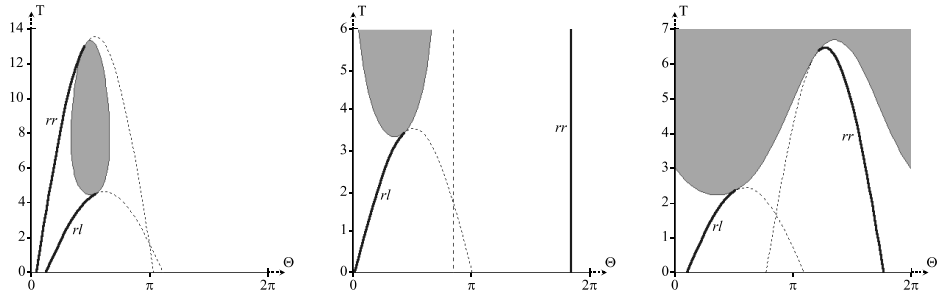


Figure 6.3: Departure curves $DC_r(C,C_i)$ on the surface of C parametrized by angle Θ and time T for different values of v_i. (a) $v_i < 1$. (b) $v_i = 1$. (c) $v_i > 1$. The dashed curves are improper departure curves. The gray area, given by the inequality $(T\cos\Theta - p_{ix})^2 + (T\sin\Theta - p_{iy})^2 < (r_i + v_i T)^2$, is the region on the surface of C that is penetrated by C_i, i.e. these points are not collision-free.

$DC_r(C,C_i)$. We can make this explicit by solving $D(T,\Theta) = 0$ for T. In Fig. 6.3 this function is plotted for various values of v_i (note that the function has a period of 2π). In each of these cases we see two sine-like curves (for $v_i = 1$, it is degenerate). They correspond with $DC_{rl}(C,C_i)$ and $DC_{rr}(C,C_i)$, respectively. The other departure curves $DC_{lr}(C,C_i)$ and $DC_{ll}(C,C_i)$ can be found when considering clockwise spirals.

Given a position (T,Θ) on the surface of cone C for which $D(T,\Theta) = 0$, the arrival time of the straight-line segment at cone C_i is given by $A(T,\Theta)$. The departure time of the segment is given by T. For some points along the departure curve $A(T,\Theta)$ is smaller than T. They correspond with bitangencies in the negative direction, i.e. the arrival time on C_i is smaller than the departure time at C. In the plots this is indicated by dashed curves. In the remainder of this chapter these improper curves are ignored when we refer to departure curves.

We also have to take into account departure curves of $DC(C,g)$ associated with segments tangent to C and leading to the goal configuration g. In this case, we have to solve the system of equations $[\ell(t) = g]$ for T, to get a closed form for the departure curve.

6.4 A Naive Algorithm

With the notions introduced so far, we can devise a first, rather naive algorithm to find a shortest path amidst growing discs from some start configuration s to some goal configuration g. Our approach grows a tree of possible shortest paths that is rooted in the start configuration at time $t = 0$. A leaf is *expanded* if the length of its path from the start configuration is minimal among all leafs of the tree. To this end, each leaf is maintained in a *priority queue*, with a *key value* equal to its time coordinate (which equals the length of its path from s). The priority queue is initialized with the initial motions from the start configuration s that possibly belong to a shortest path. These are straight-line segments with slope $1/V$ leading either directly to the goal configuration, or to a tangency point on the surface

of one of the cones. Some of these segments may intersect other cones, which would make them invalid, so only the collision-free segments are considered. The endpoint of each valid segment is put into the priority queue with a key corresponding to its t-value.

Now, the algorithm proceeds by handling the point with the lowest t-value in the queue (the front element of the queue). This point is either the goal configuration, in which case the shortest path has been found, or a point on the surface of a cone. In this latter, more general case we proceed by walking along a spiral about the cone. This spiral *either* runs into an obstacle (another cone), in which case there is no valid continuation of the path, *or* it encounters a departure curve on the surface of the cone. In this case there are two outgoing branches: (1) continuing along the spiral on the surface of the cone to find a next departure curve, and (2) departing for the other cone by a straight-line segment. If this latter segment is collision-free, its endpoint is inserted into the queue. Also for the first option an entry is enqueued.

This procedure is repeated until the goal configuration is popped from the priority queue. In this case the shortest path has been found, and can be read out if backpointers have been maintained during the algorithm. If the priority queue becomes empty, or if the front element of the queue has a time-value for which the goal configuration is not collision-free anymore (it is occupied by one of the growing discs), no valid path exists. In Algorithm 6.1, the algorithm is given in pseudocode.

Algorithm 6.1 SHORTESTPATHNAIVE(s, g)

```
1:  Initialize priority queue Q with endpoints of all valid outgoing segments from s.
2:  while Q is not empty do
3:      Pop the front element ⟨q, t⟩ from the queue.
4:      if the goal configuration is not collision-free anymore at time t then
5:          Path does not exist. Terminate.
6:      else if q = g then
7:          Shortest path found! Terminate.
8:      else
9:          q is on the surface of a cone, say C_i, so proceed along the spiral about C_i until it
            runs into another cone, or encounters a departure curve.
10:         if the spiral encounters a departure curve, say DC(C_i, C_j), then
11:             ⟨q', t'⟩ ← the intersection point of the spiral and the departure curve.
12:             ⟨q'', t''⟩ ← arrival point of the bitangent segment on the surface of C_j.
13:             Insert ⟨q', t'⟩ into Q.
14:             if segment ⟨q', t'⟩, ⟨q'', t''⟩ is collision-free then
15:                 Insert ⟨q'', t''⟩ into Q.
16: Path does not exist.
```

In the above algorithm, we have to identify the spiral we are on (let us assume that it is a counterclockwise spiral), given a point on the surface of the cone (line 9). Let q be a point on the surface of some cone, say C_i, given in Euclidean coordinates (x, y, t). Then the corresponding coordinates (T, Θ) on the surface of C_i are given by $(T, \Theta) = (t, \arctan \frac{y-p_{iy}}{x-p_{ix}})$.

The spiral on the surface of C_i going through (T, Θ) is given by θ_0 as computed in Equa-

tion (6.6). Equation (6.5) then gives a function for the angle $\theta(t)$ along the spiral through (T,Θ). In line 10 of Algorithm 6.1, we wish to know whether the spiral encounters any departure curves. To this end, we should find the intersections of the spiral and the departure curves on the surface of C_i. Recall that we can deduce an implicit equation $D(T,\Theta) = 0$ for the departure curves of any pair of cones (see Equation (6.8)). The intersections are thus found by solving $D(t,\theta(t)) = 0$ for t.

When we have found an intersection for some value $t = T$ of the spiral and a departure curve of, say, $DC(C_i, C_j)$, we wish to know what kind of departure curve we have encountered. The arrival time at cone C_j when departed from time T is $A(T,\theta(T))$ (see Equation (6.8)). If this arrival time is smaller than T, the intersection can be ignored. If it is larger, we like to know whether the tangent straight-line segment arrives on the left side of C_j (and should be succeeded by a clockwise spiral on C_j), or on the right side of C_j (and should be succeeded by a counterclockwise spiral). This is determined by the derivative of $D(t,\theta(t))$ to t. If this derivative is negative at point T, we have arrived on the left side. If it is positive, we have arrived on the right side. The exact arrival location on the surface of cone C_j is given by $\ell(A(T,\theta(T))$ (see Equation (6.7)). From this information we can deduce the parameters defining the spiral on C_j on which we have arrived.

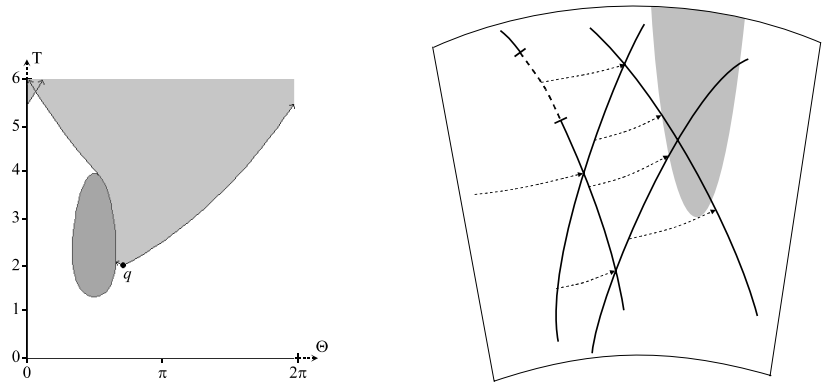
6.5 An Efficient Algorithm

The algorithm described above will indeed find a shortest path to the goal within a finite amount of time. However, in order to have a bound on the running time we must define *nodes* that can provably be visited only once in a shortest path, such that we can do *relaxation* on them as in Dijkstra's algorithm [97]. We will show that this is easy to achieve in the restricted case where all discs have equal growth rates, and present an $O(n^3 \log n)$ algorithm (n being the number of discs). In the general case we could not do this, but we will present an algorithm that is very fast in practice, by pruning large parts of the search tree.

6.5.1 Discs with Equal Growth Rates

If a point $q = (T,\Theta)$ on the surface of a cone has been visited during the search for a shortest path to the goal, all points on the cone that are reachable from q by following some collision-free path with a velocity less than the maximal velocity V (i.e., $\frac{\|(\delta x,\delta y)\|}{\delta t} < V$), can never lie on a shortest path from the start to the goal (this follows directly from Theorem 6.3). These points are contained in the wedge formed by the clockwise and counterclockwise spiral going through q (see Fig. 6.4(a)), as we know that on the spirals $\frac{\|(\delta x,\delta y)\|}{\delta t} = V$. We call this region the *wedge region* of q.

Let us consider the *arrangement* [26] on the surface of a cone containing all (proper) departure curves and all obstacle regions (other cones penetrating the surface) on that cone (see Fig. 6.4(b) for an impression). Note that the departure curves may be subdivided into a number of *collision-free intervals* by the obstacle regions. In case all discs have the same growth rate, say $v_i = v < V$ for all i, these intervals satisfy an interesting property (see Fig. 6.3(b)): let (T,Θ) be a point on some departure curve interval, then all points (T',Θ') on the same interval for which $T' > T$ are within the wedge region of (T,Θ). To prove this, we must



(a) (b)

Figure 6.4: (a) The region (light grey) on the surface of a cone that is reachable from point q by paths with $\frac{\|(\delta x,\delta y)\|}{\delta t} \leq V$. The dark grey area is an obstacle. (b) An impression of an arrangement on the surface of the cone. The thick lines are the departure curves, of which one has a shadow interval (dashed). The thin dashed lines are spiral segments that delimit trapezoidal regions that have the same next departure curve or collision (only the counter-clockwise spirals are shown). The gray area depicts an obstacle area of another cone penetrating the surface, and cutting several departure curves into two intervals.

show that for the departure curves hold that $\frac{\|(\delta x,\delta y)\|}{\delta t} \leq V$. As the proof is rather technical, we omit it here.

This means that these departure curve intervals can serve as *nodes* in our Dijkstra-algorithm. Only the path arriving earliest in an interval can contribute to a shortest path. Paths arriving later in the interval cannot be part of the shortest path, because the path arriving earliest in the interval can be extended with a traversal along the interval to end up at the same position (and time) as the path arriving later in the interval.

Each node (an interval) has two outgoing *edges*. Let the interval be a segment of a departure curve of $DC(C_i, C_j)$, then the first edge is a spiral segment to the next departure curve on the surface of C_i, and the second edge consists of a bitangent straight-line segment and a spiral segment and arrives in the first departure curve encountered on the surface of C_j. For the first edge, which stays on the cone, we have to determine the next departure curve that is encountered if we proceed by moving along the spiral about the cone. This can be done efficiently using the arrangement, if we have computed its *trapezoidal map* [26], where the sides of the trapezoids are spiral segments.

For the second edge, which traverses to another cone, we have to determine what the first departure curve is we will encounter there. This can be done efficiently using the arrangement we have computed on that cone. Using a point-location query, we can determine in what cell of the arrangement the straight-line segment has arrived, and using the trapezoidal map we know what the first departure curve is we will encounter if we proceed from there.

Finally, we must ascertain that each edge is collision-free with respect to the other cones.

Spiral segments may collide with other cones if these penetrate the spiral's cone surface. Since obstacle areas are incorporated into the arrangement, such collisions are easily detected. Straight-line segments may collide with any cone, so for each departure curve and each cone, we calculate the *shadow interval* this cone casts on the departure curve, in which a departure will result in collision. These shadow intervals are stored in the arrangement as well. In Fig. 6.4(b), an impression is given of how such an arrangement might look.

Theorem 6.7. *The algorithm to compute a shortest path amidst n growing discs with equal growth rates runs in $O(n^3 \log n)$ time.*

Proof: For each pair of cones there are $O(1)$ departure curves. Since there are $O(n^2)$ pairs of cones, there are $O(n^2)$ departure curves in total. Each of the departure curves can be segmented into at most $O(n)$ intervals, as there are at most $O(n)$ cones intersecting the departure curve (each cone can split the departure curve into at most two segments). Hence, there are $O(n^3)$ departure curve intervals. Each departure curve interval has $O(1)$ outgoing edges, making a total of $O(n^3)$ edges.

The complexity of Dijkstra's algorithm is known to be $O(N \log N + E)$ where N is the number of nodes, and E the number of edges. Each edge requires some additional work. Firstly, we have to find the departure curve interval in which it will arrive, by doing a point-location query in the trapezoidal map of one of the arrangements. This takes $O(\log n)$ time. Further, we must determine whether an edge is collision-free. Using the shadow intervals stored at the departure curves, this can be done in $O(\log n)$ time as well. Thus, as both N and E are $O(n^3)$, Dijkstra's algorithm will run in $O(n^3 \log n)$ time in total.

Computing the arrangements and their trapezoidal maps takes $O(n^2)$ time per cone, as there are $O(n)$ departure curves on each cone, and $O(n)$ intersection areas of other cones. As there are $O(n)$ cones, this step takes $O(n^3)$ time in total. All the shadow intervals can be computed in $O(n^3)$ time as well, as there are $O(n^2)$ departure curves and $O(n)$ cones.

Overall, we can conclude that our algorithm runs in $O(n^3 \log n)$ time. ■

6.5.2 General Case: Discs Have Different Growth Rates

In the general case, where the discs may have different growth rates, the problem becomes much harder. We can follow the same approach as above, but let us look at what happens to the slope of the departure curves in this case (see Figs. 6.3(a) and (c)). In the case where the arrival cone has a slower growth rate, the departure curves (provably) satisfy $\frac{\|(\delta x, \delta y)\|}{\delta t} \leq V$ (see Fig. 6.3(a)). However, in the case where the arrival cone has a faster growth rate (Fig. 6.3(c)), it is clear that this is not the case. The departure curve DC_{rr} is horizontal at some point, meaning that $\frac{\|(\delta x, \delta y)\|}{\delta t} = \infty$. Hence, we cannot define intervals on these departure curves that serve as nodes in the search process.

We can still use Algorithm 6.1 for the general case, but a problem is that this algorithm considers many branches in the search tree of which we know that they will not lead to a shortest path. For instance, it lets the spirals circle around the cones forever, thereby encountering many departure curves, which in turn generate other spirals on other cones. Hence, it lets the size of the search tree blow up quickly.

In order to have an algorithm that runs fast in practice, we need to prune these useless branches of the search tree. The key observation we use for this is that a point (T,Θ) on the surface of cone C_i cannot be part of shortest path if we have visited (T',Θ) already (where $T' < T$) *and* the vertical line segment on the surface of the cone between (T',Θ) and (T,Θ) is collision-free. This is because (T,Θ) is then in the wedge region of (T',Θ) (note that the velocity $\frac{\|(\delta x,\delta y)\|}{\delta t}$ along the vertical line segment equals $v_i < V$). Hence a spiral encountering (T,Θ) need not be expanded any further.

To implement this practically, we only do this test for a constant number of Θ's. To this end, we augment Algorithm 6.1 by choosing a small constant ϵ, and drawing $\frac{2\pi}{\epsilon}$ evenly distributed vertical lines on the surface of each cone. These vertical lines are segmented into collision-free intervals by obstacle regions on the surface. Now, these intervals will serve as *nodes* in our practical algorithm on which we perform *relaxation*.

This means that if we walk along a spiral on the surface of a cone, and the spiral crosses a vertical line, we have to check whether this spiral is the first to arrive in the particular interval. If not, this spiral can never be part of a shortest path, for the same reasons as above. Thus, this branch of the search tree can be pruned. The smaller ϵ is chosen, the sooner the spirals can be pruned, and hence the smaller the size of the search tree will be. Even though this algorithm has no running time that can be expressed in the number of discs (n), it turns out to be very fast in practice, as we will see next.

6.5.3 Implementation Details

We created a fast implementation of Algorithm 6.1, augmented with the pruning heuristic presented above. We did not create an arrangement of all vertical lines and all obstacle regions on each cone. This would take too much time. Instead, we maintain for each vertical line m one time-value at which it was last visited, say t_m. Given the order in which the points are considered in the priority queue, we know that when a point q is popped from the queue, it has a higher time value than any point previously considered. So, if point q lies on vertical line m, its time value q_t is larger than the time-value t_m of the point previously considered on that line. If the line segment between t_m and q_t on m is collision-free, q is in a previously visited interval, and hence this point is not expanded. However, if the vertical line segment between these two points is not collision-free, point q is the first to arrive in a new interval, and its outgoing edges must be inserted into the priority queue. From this moment on, q_t is set as the time value attached to the vertical line m, as we know that no point below q_t will be considered anymore.

Outgoing edges of a point q on a vertical line segment are a spiral segment to the next vertical line, and –in case this spiral segment crosses one or more departure curves– segments to vertical lines on other cones. In our implementation, the intersection between spiral segments and departure curves is found using a combination of two approximate root-finding algorithms [33]. We must note that these may fail if between two consecutive vertical lines the spiral intersects the *same* departure curve twice. In this case only one root will be found. However, as we assume that ϵ, the radial distance between two consecutive vertical lines, is small, the probability that this occurs is negligible. In our experiments, we never encountered this anomaly.

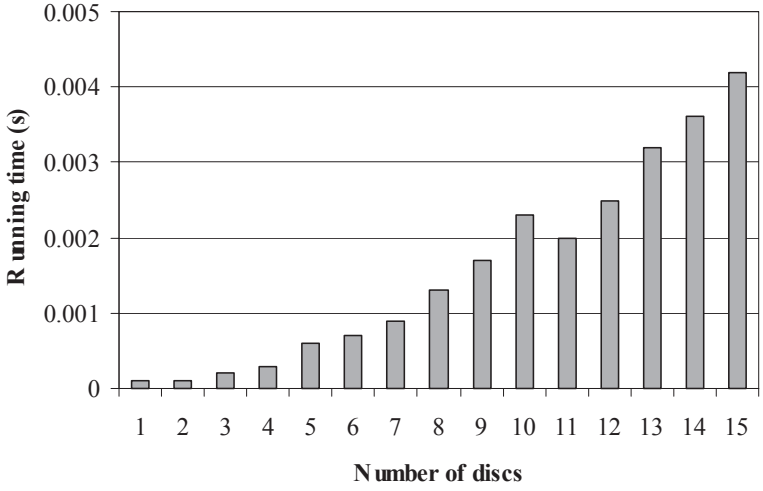


Figure 6.5: Results of our experiments.

Collision-checking straight-line segments is done by testing them for intersections with all cones, except the ones they are tangent to. As ϵ is small, we approximate a spiral segment between two consecutive vertical lines by a straight-line segment, and collision-check it in the same way.

Finally, the Dijkstra paradigm was replaced by an equally suited A*-method [97], that is faster in practice as it focusses the search to the goal. It adds a lower bound estimate of the distance to the goal to the key-value of each point in the priority queue. In our implementation, the lower bound estimate is simply the Euclidean distance divided by the maximal velocity.

6.6 Experimental Results

We created an interactive application for planning paths amidst growing discs. The properties of the growing discs (position, size, growth rate) can be changed by the user, and on-the-fly a new path is computed. From this application we report results. Experiments were run on a Pentium IV 3.0GHz with 1 GByte of memory. The value of ϵ was chosen sufficiently small at $\frac{2\pi}{40}$.

We report the running times of the algorithm for a varying number of discs. As the running time of the algorithm does not only depend on the number of obstacles, but also on the exact configuration of the discs, and how well the A* method manages to focus the search, etc., we averaged the running times over various positions of the start configuration for each experiment. In Fig. 6.5 the results are given.

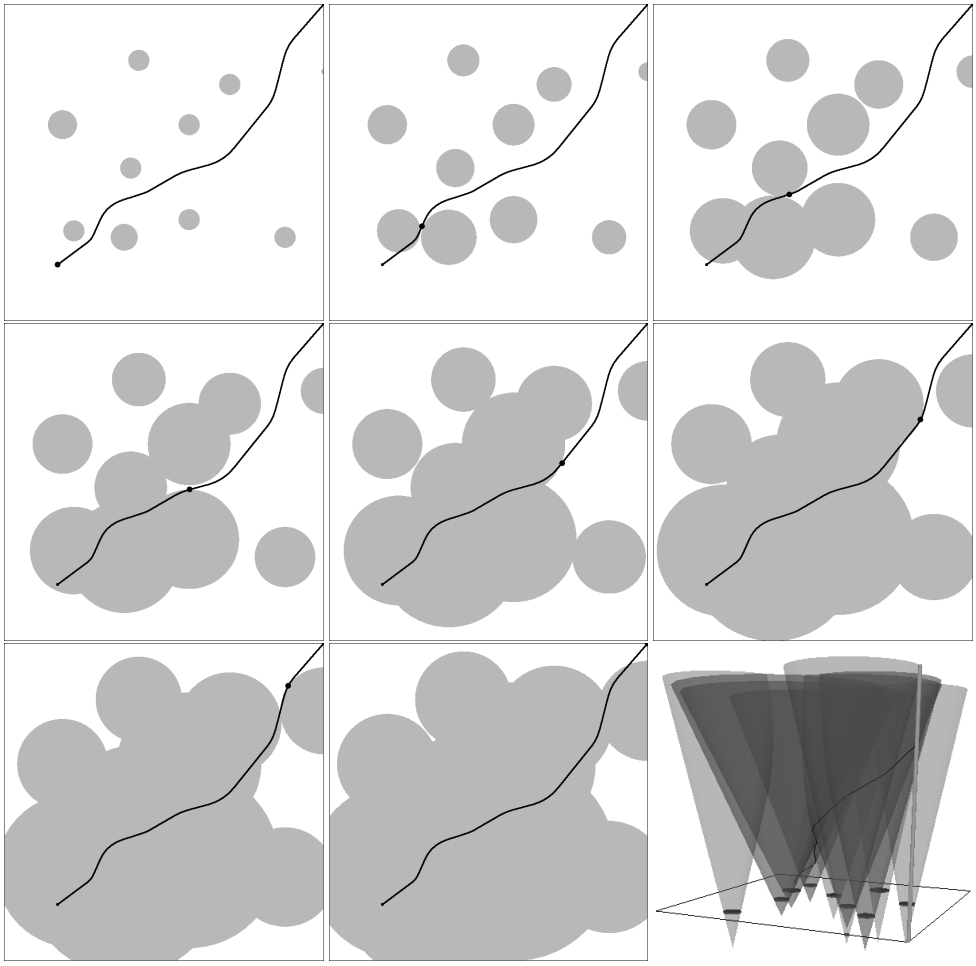


Figure 6.6: A shortest path amidst 10 growing discs. A small dot indicates the position along the path at $t = 0, 1, \ldots, 7$. The lower-right picture gives the 3-D view. The pictures were generated by our application.

What first of all can be seen from the results is that our implementation is very fast. Even for 15 growing discs, the running time is only 0.0042 seconds, well within real-time requirements. We did not show results for more than 15 discs, as it appeared to be difficult to find sensible setups with this many discs that still contain a valid path to the goal. From the figure it seems that the running time is more or less quadratically related to the number of discs. This is what we expected based on the implementation. In Fig. 6.6, snapshots are shown of a shortest path amidst 10 growing discs.

6.7 Conclusion

In this chapter we presented an algorithm for computing shortest paths (minimum time paths) amidst discs that grow over time. A growing disc can model the region that is guaranteed to contain a moving obstacle of which the maximal velocity is given. Hence, using our algorithm, paths can be found that are guaranteed to be collision-free in the future, regardless of the behavior of the moving obstacles. As the regions grow fast over time, a new path should be planned from time to time –based on newly acquired sensor data– to generate paths with more appealing global characteristics. Our implementation shows that such paths can be generated very quickly. A great advantage over other methods is that this replanning can be done safely. The old path that is still used *during* replanning is guaranteed to be collision-free. A requirement though, is that the robot has a higher maximal velocity than any of the moving obstacles.

A drawback of the method we presented is that a path to the goal often does not exist. This occurs when the goal is covered by a growing disc before it can be reached. A solution to this problem would be to find the path that comes closest to the goal. It seems that this can easily be incorporated into our algorithm. Other possible extensions include allowing obstacles with different shapes (other than discs), and fixed obstacles in the environment, but they are still subject of ongoing research.

Chapter 7

Planning in Repetitive Dynamic Environments

In this chapter we discuss path planning in a special class of dynamic environments, namely *repetitive environments*, in which the motions of the moving obstacles are periodic (i.e. repetitive). It is not directly related to previous chapters, but we show that for these kind of environments, many problems inherent to planning in dynamic environments (as mentioned in the introduction of this thesis) disappear. For repetitive environments it is possible to generate a roadmap in a preprocessing stage, which can be used multiple times to quickly solve individual planning queries. This is in contrast to general dynamic environments, whose *transitoriness* invalidates the premise that a preprocessed roadmap can be used multiple times. Our approach is suitable for any robot with any number of degrees of freedom in two- and three dimensional workspaces.

This chapter has previously been published as: J. van den Berg and M. Overmars. Path planning in repetitive environments. In *Proc. IEEE Int. Conf. on Methods and Models in Automation and Robotics*, pages 657–662, 2006 [23].

7.1 Introduction

Path planning is of great importance in various fields such as robotics and virtual environments. The problem is generally formulated as finding a path for a robot between two query configurations that avoids collisions with the obstacles in the environment. Most research has focused on the problem of path planning in *static* environments, where the geometry of the environment is known in advance and the obstacles do not move. Many solutions to this problem have been presented, most of them being *probabilistic* [96, 97]. They can roughly be subdivided into two classes: single-shot and multiple-shot. Single-shot approaches aim to solve a single planning query as fast as possible. The most popular method is the Rapidly-exploring Random Tree (RRT)-approach [93], where a random tree of possible

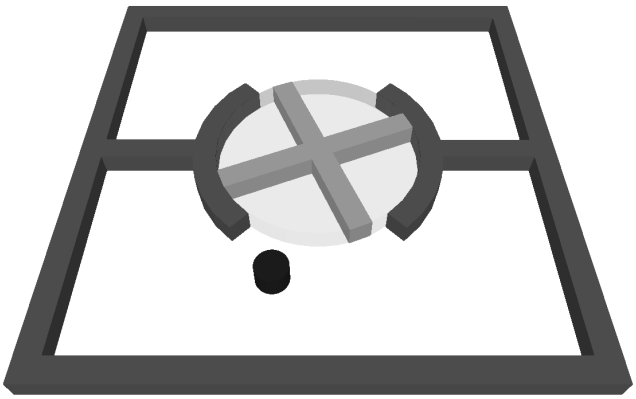


Figure 7.1: A repetitive environment with an automatically revolving door.

paths is grown outward from the start until it can be connected to the goal configuration. Multiple-shot approaches do some (relatively expensive) preprocessing on the environment, such that (multiple) individual path planning queries can be answered quickly. They are useful when expectedly many queries will be done in the same environment. The most well-known approach is the Probabilistic Roadmap (PRM)-method [89, 149]. In the preprocessing stage, it generates a roadmap that represents the connectivity of the free space by connecting randomly sampled configurations with straight-line paths. Individual queries are then answered by connecting the start and goal to the roadmap, and searching for a shortest path (using e.g. Dijkstra's). Both PRM and RRT are proven to be *probabilistically complete*, and have successfully been used in complex, high-dimensional configuration spaces.

The extension of the path planning problem to dynamic environments, where some of the obstacles are moving, has been extensively studied as well [55, 75, 124, 20]. The static-environment approaches can be applied to dynamic environments when the absolute notion of time is incorporated in the configuration space as an additional dimension. Moving obstacles in the workspace then transform to stationary obstacles in the configuration-time space. However, the nature of configuration-time spaces differs considerably from the nature of regular configuration spaces (without time). Firstly, configuration-time spaces are *transitory*, as the notion of time in these spaces corresponds to the real-world time. Because of this, multiple-shot approaches are *not* applicable here: it is possible to construct a roadmap, but it can only be used once – for a query of which the real-world time corresponds to the time interval modeled in the configuration-time space. Therefore, single-shot approaches have been the methods of choice in these spaces [75, 124].

A second problem of the inherence of the real-world time and the time modeled in the configuration-time space, is that planning a path takes time itself, and must be finished before the path is executed. So, when a path planning query is formulated in a dynamic environment, one must attach a time value to the start configuration. This value should not be chosen too close to the current real-world time, because that would leave too little time

for planning the path. On the other hand, the start time should not be chosen too far from the current real-world time, because this would result in a latency that is unacceptable for most applications. Ideally, the start time is chosen equal to the current real-world time plus the time needed for planning, where the planning time is preferably as short as possible. However, an RRT-approach (like in [75]) does not give any guarantee on the amount of time needed to plan a path, which makes it hard to pick a suitable start time. Also, the planning may take quite long, particularly in cluttered spaces with many narrow passages [20].

The above problems can be overcome if the moving obstacles have *periodic motions*. That is, after some time π the obstacles return to their initial configuration and execute the same motion again. Such *repetitive environments* do not have the problem of transitoriness, so we *can* apply a multiple-shot approach and build a roadmap in a preprocessing stage. We exploit this roadmap to tackle the second problem, as finding a path in the query stage can be done very fast and within an accurately estimated amount of time.

Repetitive environments are often observed in practice, particularly in virtual environments. Examples are automated revolving doors (see Fig. 7.1), automated sliding doors, regular bus services, etc. Environments in which a subset of the free space has a periodic motion also fall within the definition, such as automated platforms, elevators, etc. Yet, we are unaware of any approaches to date dealing specifically with this class of dynamic environments.

In this chapter we present a framework for path planning in repetitive environments. Our method is suitable for any robot with any number of degrees of freedom, both in two- and three dimensional workspaces. Once a roadmap has been constructed, queries can be answered within real-time requirements. The rest of the chapter is organized as follows. In Section 7.2 we will describe a naive approach to the problem. Section 7.3 describes a more advanced approach which tackles the problems raised by the naive approach, and in Section 7.4 we will discuss experiments performed with our method proving the claims we make.

7.2 Naive Approach

7.2.1 Problem Specification

The static path planning problem is generally formulated in terms of the configuration space $\mathcal{C}$, the set of all possible configurations of the robot. The dimension of the configuration space corresponds to the number of the robot's degrees of freedom. To extend the definition to path planning in dynamic environments, time is added as a dimension. Let $\mathcal{T} \subset \mathbb{R}$ be the time interval of interest, then the *configuration-time space* $\mathcal{CT}$ is formed as $\mathcal{C} \times \mathcal{T}$. It consists of pairs $\langle c, t \rangle$, where c is an element of $\mathcal{C}$ denoting the robot's configuration, and t a scalar in $\mathcal{T}$ denoting the time. The robot is represented by a point in the configuration-time space, and both stationary and moving obstacles in the workspace transform to stationary obstacles in $\mathcal{CT}$.

For repetitive environments, let λ be the period of the motions of the moving obstacles. Then the time interval $\mathcal{T}$ is defined as $[0, \lambda)$. Further, it is combined with the modulo-λ operator, making configuration-time $\langle c, \lambda \rangle$ equal $\langle c, 0 \rangle$, or in general: $\langle c, t \rangle = \langle c, t \bmod \lambda \rangle$. Hence, the space is periodically connected through the boundaries $t = \lambda$ and $t = 0$.

Given a start configuration $c_s \in \mathcal{C}$ and a start time $t_s \in \mathbb{R}$, and a goal configuration $c_g \in \mathcal{C}$, the problem to solve is to find a valid path $\pi : [t_s, t_g] \to \mathcal{C}$, such that $\pi(t_s) = c_s$, and $\pi(t_g) = c_g$, for some arrival time $t_g \in \mathbb{R}$. Hence, we must find a collision-free path in the configuration-time space from $\langle c_s, t_s \bmod \lambda \rangle$ to $\langle c_g, t_g \bmod \lambda \rangle$. However, this path should obey some additional constraints posed by the time dimension. Firstly, time marches forward, so all paths should be monotonic in the time-dimension. Further, we would like the paths satisfy some maximum velocity $v_{\max}$ of the robot. So, the slope of valid paths in the configuration-time space is bounded: $\forall (t_1, t_2 \in [t_s, t_g] : t_2 > t_1 : d_{\mathcal{C}}(\pi(t_1), \pi(t_2)) \leq v_{\max}(t_2 - t_1))$, where $d_{\mathcal{C}} : \mathcal{C}^2 \to \mathbb{R}$ is the distance measure on the configuration space $\mathcal{C}$.

Given the above constraints, we can define a distance measure $d : \mathcal{CT}^2 \to \mathbb{R}$ on the configuration-time space $\mathcal{CT}$:

$$d(\langle c_1, t_1 \rangle, \langle c_2, t_2 \rangle) = \left\lceil \frac{t_1 + d_{\mathcal{C}}(c_1, c_2) / v_{\max} - t_2}{\lambda} \right\rceil \lambda + t_2 - t_1.$$

It measures the *time* needed to go from one configuration-time to the other without violating the maximum velocity constraint. Note that since paths in $\mathcal{CT}$ are monotonic in the time dimension, this distance measure is *asymmetric*.

7.2.2 PRM for Static Environments

A PRM-approach aims to create a roadmap that represents the connectivity of the free configuration space, so that it can be used effectively for path planning. The roadmap is constructed by sampling configurations randomly from the configuration space. If a configuration is collision-free (determined by a collision-checker [28]), it is added as a node to the roadmap. Subsequently, it is attempted to connect the node to other nodes already present in the roadmap. If such a connection succeeds –that is when the straight-line segment in the configuration space between the two nodes is collision-free– it is added as an (undirected) edge to the roadmap. This procedure is repeated until some stop-criterion is met, usually that is when some predefined set of configurations is connected via the roadmap.

Since collision-checking is expensive, connections are only attempted to a specific set of *potential neighbors* for which there is a high probability that a connection succeeds. Usually, this set is formed by the nodes that are closer than some preset maximum neighbor distance $d_{\max}$. Further, connections to nodes in the same *connected component* of the roadmap are omitted. This is because they would not extend the connectivity of the roadmap. To fully exploit the latter rule, the neighboring nodes are considered in the order of increasing distance to the newly added node.

7.2.3 PRM for Periodic Configuration-Time Spaces

To implement a PRM for periodic configuration-time spaces, we can fairly straightforwardly follow the PRM-approach for static environments. We start with sampling configuration-times from $\mathcal{CT}$, i.e. a configuration c is randomly picked from $\mathcal{C}$ and a moment in time t is chosen randomly out of the interval $[0, \lambda)$ to form a random configuration-time $\langle c, t \rangle$. If this configuration-time is collision-free, i.e. the robot configured at c at time t does not collide with any stationary or moving obstacle, it is added as a node to the roadmap.

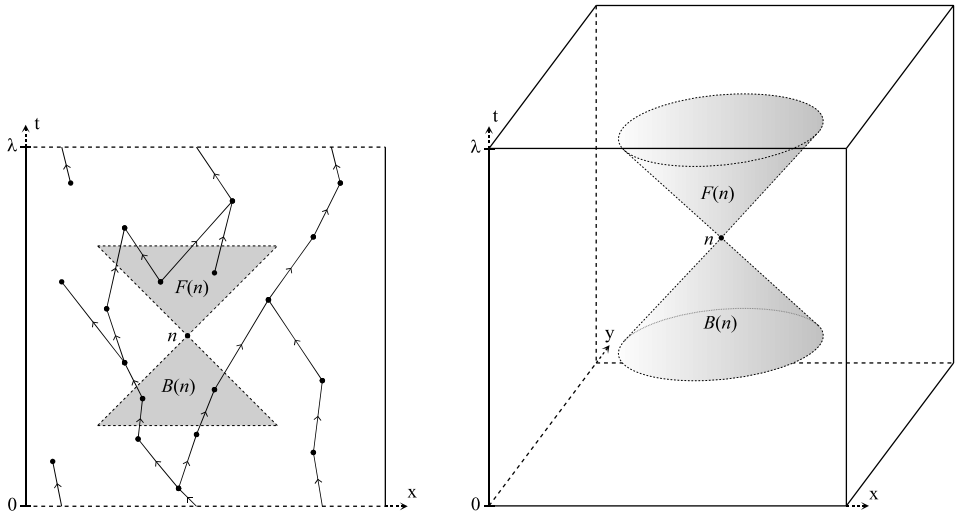


Figure 7.2: The neighbor region of a node n in a 2-D (a) and a 3-D (b) $\mathcal{CT}$-space. The horizontal boundaries are induced by the maximum neighbor distance $d_{\max}$ and the diagonal boundaries are induced by the maximum velocity $v_{\max}$.

Since the distance measure on $\mathcal{CT}$ is asymmetric, the roadmap must be *directed*. Therefore, a new node now has two sets of potential neighbors: potential forward neighbors (for outgoing edges) and potential backward neighbors (for incoming edges). In both cases we choose a maximum neighbor distance $d_{\max}$. So, if the newly added node is denoted by n, its sets $F(n)$ and $B(n)$ of potential forward and backward neighbors, respectively, are defined as:

$$\begin{aligned} F(n) &= \{n' \in N - \{n\} \mid d(n, n') \leq d_{\max}\}, \\ B(n) &= \{n' \in N - \{n\} \mid d(n', n) \leq d_{\max}\}, \end{aligned}$$

where N denotes the set of nodes in the roadmap. The region in the configuration-time space containing the potential neighbors is hourglass-shaped (see Fig. 7.2).

Having determined the set of potential neighbors of a newly added node n, we try to connect to them in the order of *increasing* distance. Just as in the static case, we like to avoid redundant connections to nodes to which a connection already exists. However, in directed roadmaps there is no such thing as connected components. Therefore, we maintain for the newly added node n two auxiliary sets, FC and BC, containing the nodes to which n is *transitively forward connected* and to which it is *transitively backward connected*, respectively. Initially these sets are empty, but each time an outgoing edge from n to another node n' is added to the roadmap, we add n' to the set of forward connected nodes FC, and do this recursively for the outgoing edges of n'. The recursion stops when a node is already in FC, or

when the node is not contained in the potential forward neighbor set $F(n)$. Now, each time we consider an edge (n, n') for addition to the roadmap, we first check whether n' is already in FC. If so, we can omit the edge. For backward neighbors we proceed similarly. When all the potential neighbors have been considered, a new node is sampled and the above procedure is repeated. The complete algorithm is given in Algorithm 7.1.

Algorithm 7.1 CONSTRUCTPERIODICPRMNAIVE

```
1: repeat
2:    Sample a configuration-time ⟨c, t⟩ ∈ CT
3:    if configuration-time ⟨c, t⟩ is collision-free then
4:       Add ⟨c, t⟩ as a node n to the roadmap
5:       FC ← ∅; BC ← ∅
6:       for all n' ∈ F(n) in order of increasing d(n, n') do
7:          if n' ∉ FC and the straight-line between n and n' in CT is collision-free then
8:             Add edge (n, n') to the roadmap
9:             Update FC
10:      for all n' ∈ B(n) in order of increasing d(n', n) do
11:         if n' ∉ BC and the straight-line between n' and n in CT is collision-free then
12:            Add edge (n', n) to the roadmap
13:            Update BC
14: until some stop-criterion is met
```

We consider the nodes in the potential neighbor sets in the order of increasing distance to n. This may look trivial, but actually we can only do this because of the property that all edges are oriented in the same direction along the time axis. For general directed roadmaps the definition of potential neighbor sets is much more complicated (as is the evaluation whether or not a node is a potential neighbor), and the nodes in the potential neighbor set are considered in the order of *decreasing* distance [149].

We applied the above algorithm on the scene of Fig. 7.1, where the period λ covers one quarter-revolution of the door. The construction of the roadmap stopped when two specific configurations on either side of the door were mutually forward connected. To this end, we maintained the forward connected set of these two nodes during the entire algorithm. The maximum neighbor distance $d_{\max}$ was set to $\lambda/4$, and the maximum velocity $v_{\max}$ of the robot was set in accordance with the angular velocity of the door.

Averaged over 100 runs, it took 1.52 seconds to create a roadmap for this scene on a Pentium IV 3.0GHz with 1 GByte of memory. The average roadmap contained 2276 nodes (see Fig. 7.3 for an example).

7.2.4 Query Stage

A roadmap created as shown above can be used to quickly answer queries for a path in the environment. Suppose we want to find a path from some configuration c_s to some configuration c_g, and suppose that the query is posed at real-world time t_w. Then, we should find a path from $\langle c_s, (t_w + \Delta t) \bmod \lambda \rangle$ to $\langle c_g, t_g \bmod \lambda \rangle$ in $\mathcal{CT}$, where t_g is some preferred arrival

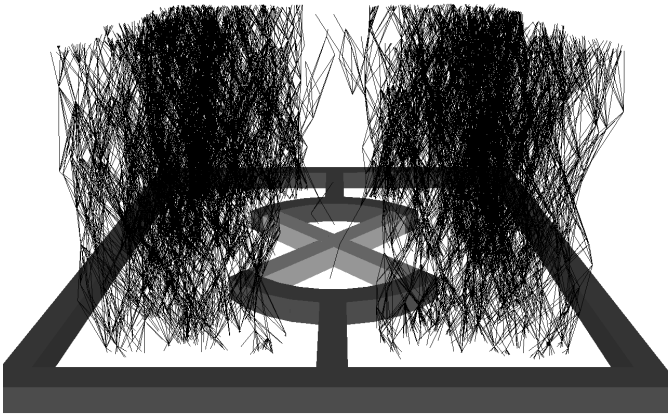


Figure 7.3: An example roadmap in the configuration-time space of the revolving door scene of Fig. 7.1. The third dimension is the time axis.

time. Δt is preferably an as small as possible amount of time in which we should (1) connect both query configuration-times to the roadmap as if they were newly added nodes, (2) find a path in the roadmap from the start to the goal, for instance using Dijkstra's algorithm, and (3) do some optional post-processing on the path, like smoothing. The value of Δt should be chosen before step 1 is carried out, and the path can only be executed after step 2 is finished. Hence, we must have an accurate estimate of how long these steps take to choose a good value for Δt.

Experiments on the above scene indicate that connecting the query configurations takes on average 0.006 seconds and performing a shortest path query (using Dijkstra's in our case) takes 0.003 seconds. So, within only 0.01 seconds we can successfully answer a query. Step 3, smoothing the path, is not necessary for answering the query, but it makes the path look nicer and more natural. An appealing property of most smoothing algorithms is that they are incremental [59], so we can spend as much time as we like on smoothing the path. So if, for instance, we choose Δt to be 0.1 seconds, we have enough time to perform the necessary steps 1 and 2, and can spend the remaining time on step 3.

7.3 Advanced Approach

Although we showed some promising results in the previous section, the method described so far has some clear drawbacks:

- Large parts of the environment are usually time-invariant. Yet, samples that are picked from these regions include a time-value, while they could have been treated as samples in a static environment. This could substantially lower the number of nodes in the roadmap (see e.g. Fig. 7.3).

- If there is more than one periodic obstacle, the overall period λ is calculated as the *least common multiple* of the periods of the individual obstacles. This may cause λ, and thus the number of nodes necessary to adequately cover the $\mathcal{CT}$-space, to become very large.

- If a query is posed on the repetitive environment, one has to attach some time-value to the goal configuration (t_g in the previous section). Usually we do not want this, but rather reach the goal as soon as possible.

In this section we propose a more advanced approach in which we resolve the above issues. The main idea of the advanced approach is that local periodicities are considered instead of one global period.

7.3.1 Advanced PRM for Repetitive Environments

In the advanced method we define each configuration $c \in \mathcal{C}$ to have its own *local period* $\lambda(c)$. If the configuration c lies in a (time-invariant) part of the space where it cannot possibly be intersected by a periodic obstacle, we define $\lambda(c)$ to be zero. If c does possibly intersect a periodic obstacle, the local period $\lambda(c)$ of the configuration equals the period of this obstacle. It may also be possible that a configuration intersects more than one obstacle, but for now we assume that this will not happen.

To determine the local period of a configuration c, we check of which periodic obstacles the *sweep volumes* in workspace intersect with the robot configured at c. To this end, we assume that the sweep volumes of the periodic obstacles are modeled as objects in the collision-checker (see for instance Fig. 7.1; the sweep volume is indicated as a transparent disc), along with information about the period of the obstacle.

Sampling is done as follows. First, a random configuration $c \in \mathcal{C}$ is picked. Then we determine the local period $\lambda(c)$ of c. If $\lambda(c)$ is zero, we can treat the sample as a node in a static environment and need not attach a time value to it. Otherwise, we should randomly pick a time-value for the sample from the range $[0, \lambda(c))$. However, this yields relatively little samples in time-variant parts of the space. Therefore, we let the number of time-values m that are attached to a configuration c be a function of the local period: $m(c) = \max(1, \lceil \nu\lambda(c) \rceil)$, where ν is a parameter controlling the ratio between the number of samples in time-variant and time-invariant parts of the space.

As there are different local periods, the distance measure becomes somewhat more complicated:

If $\lambda(c_1) = 0 \vee \lambda(c_2) = 0$, then:

$$d(\langle c_1, t_1 \rangle, \langle c_2, t_2 \rangle) = d_{\mathcal{C}}(c_1, c_2) / v_{\max}$$

else if $\lambda(c_1) = \lambda(c_2)$ then:

$$d(\langle c_1, t_1 \rangle, \langle c_2, t_2 \rangle) = \left\lceil \frac{t_1 + d_{\mathcal{C}}(c_1, c_2)/v_{\max} - t_2}{\lambda(c_2)} \right\rceil \lambda(c_2) + t_2 - t_1$$

else the distance is undefined.

The above distance measure implies that we can connect two nodes when (1) they are both in a time-invariant part of the space, (2) one node is time-variant and the other is not, or (3) they are both time-variant but have the same local period. Nodes that are both time-variant but have different local periods cannot be connected, but this is not a problem as indirect connections are possible via time-invariant parts of the space. In general, we do not allow edges that cross more than one border between regions of the space with different local periods. So, for example an edge between two nodes in the time-invariant part of the space that goes through a time-variant part is not admitted.

The definitions of the potential neighbor sets $F(n)$ and $B(n)$ remain the same under the new distance measure. However, the three types of edges identified above differ in nature, affecting the way in which we can discard edges for which a connection already exists. For instance, if we consider an edge between two time-variant nodes (with local period λ_ℓ) for addition to the roadmap, we know that if some other path through the roadmap (with length $< \lambda_\ell$) already exists between these nodes, the amount of time it takes to travel from one node to the other is not decreased when we add the new edge. For an edge between two time-invariant nodes on the other hand, this is not the case, as such edges are always traversed at maximal velocity and the nodes do not have a time-value. Hence, a direct connection between two nodes in time-invariant parts of the space always shortcuts possible existing paths in the roadmap (in terms of time). Yet, we do not want all of these edges to be added, for a slightly longer detour may also be acceptable. The same holds (more or less) for edges connecting one time-variant node and one time-invariant node. (Note that such edges can only be traversed at times in accordance with the time-value attached to the time-variant node of the edge.)

Therefore, we adopt a variant of the method of [115] for deciding whether or not the edge is added in these cases: An edge is added between two nodes *unless* there already exists a path in the roadmap that is shorter than k times the length of the direct connection between the two nodes, for some parameter $k \geq 1$. To this end, we maintain for a newly added node a *partial* shortest path tree up to some distance horizon. This horizon equals k times the length of the edge that is considered for addition. Since we consider possible edges in order of increasing length, the shortest path tree is incrementally grown each time a new edge is considered.

The complete advanced algorithm is given in Algorithm 7.2. The function PARTIAL-INCRDIJKSTRA(n,h) grows the current shortest path tree of n up to horizon h. The graph distance between two nodes n and n' is denoted $G(n,n')$. It is defined to be ∞ if n' is not in the partial shortest path tree of n.

7.3.2 Query Stage

Suppose we want to find a path from a configuration c_s to a configuration c_g in a repetitive environment using the advanced method. If c_g is time-invariant, i.e. $\lambda(c_g) = 0$, we do not need to attach a time value to c_g, and thus do not need to predetermine some 'preferred arrival time' like in the previous section. Also, if c_s is time-invariant as well, we can connect both query configurations to the roadmap *before* determining the start time of the query. Let t_w be the real-world time after we have connected the query configurations to the roadmap,

Algorithm 7.2 CONSTRUCTPERIODICPRMADVANCED

```
1: repeat
2:    Sample a configuration c ∈ C
3:    for max(1, ⌈vλ(c)⌉) times do
4:       Sample a time t ∈ [0, λ(c)).
5:       if configuration-time ⟨c, t⟩ is collision-free then
6:          Add ⟨c, t⟩ as node n to the roadmap
7:          for all n' ∈ F(n) in order of incr. d(n, n') do
8:             PARTIALINCRDIJKSTRA(n, k · d(n, n'))
9:             if k · d(n, n') < G(n, n') and the edge (n, n') is collision-free in CT then
10:               Add edge (n, n') to the roadmap
11:         Do the same for backward neighbors in B(n)
12: until some stop-criterion is met
```

then the start time t_0 of the query will equal $t_w + \Delta t$, for an as small as possible Δt. The only thing we have to do in this time lapse Δt is finding a path in the roadmap starting from c_s at time t_0 and arriving at c_g as quickly as possible (and optionally some path smoothing afterwards).

For this, we slightly adapt Dijkstra's algorithm, as we should be careful if we encounter edges from a time-invariant node n to a time-variant node $n' = \langle c', t' \rangle$. Such an edge is only valid if we leave n at times t_ℓ for which: $t_\ell \bmod \lambda(c') = (t' - d(n, n')) \bmod \lambda(c')$. Hence, if we arrive at n at time t, we must wait at n for $(t' - d(n, n') - t) \bmod \lambda(c')$ time, before we can proceed to n'.

7.3.3 Overlapping Moving Obstacles

In the above advanced method we have not dealt with cases in which the sweep volumes of the repetitive obstacles overlap in the *configuration space*. That is, at some configurations the robot intersects more than one sweep volume in *workspace*. Let us first discuss the case that a *pair* of obstacles intersect. To create a roadmap in such environments, we should treat these two obstacles as one obstacle with a period equal to the *least common multiple* of the periods of the individual obstacles. Having defined two intersecting obstacles as one obstacle, we can thus handle the case in which three or more obstacles (either directly or indirectly) intersect as well. Eventually, each *connected component* of intersecting obstacles is treated as one obstacle with a period equal to the least common multiple of the periods of the constituting obstacles.

To show that we cannot do any better than this, consider Fig. 7.4. Suppose that we want to find a path from s to g through the series of sliders (the white squares are free space that slide back and forth along the arrows). Obviously, if the robot has area, the sweep volumes of the three obstacles (indirectly) intersect. Now, if each slider has a different period, and the motions of the sliders are out of phase, the robot has to stay in the horizontal slider for a couple of periods (i.e. moving back and forth multiple times) until it can switch to the right vertical slider. Even though the periods of the individual sliders may be small, the combined

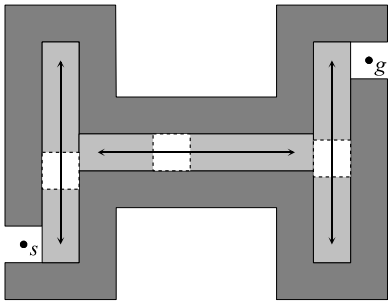


Figure 7.4: A repetitive environment with three sliders. The white squares are free space that slide back and forth along the arrows.

period of the sliders can be large, and the path that is found through these sliders can indeed be of the complexity of this combined period.

7.4 Experiments

We implemented both the naive and the advanced method for creating a roadmap in a repetitive environment. The advanced method has two parameters, ν and k, which the naive method does not have. Note that if we define the local period $\lambda(c)$ to equal the global period λ for all configurations c, and we set $\nu = 0$ and $k = 1$, the advanced method has become equivalent to the naive approach.

In this section we compare the performance of both approaches on the environment of Fig. 7.5 for various periods of the revolving doors. That is, we measure the time needed for creating the roadmap, the time needed to attach the query configurations to the roadmap, and the time needed to perform Dijkstra's shortest path algorithm. Since randomness in involved in creating a roadmap, these figures are averaged over 100 runs. Parameters such as maximal velocity $v_{\max}$ of the robot, and maximal neighbor distance $d_{\max}$ are set equal for both methods. Throughout the experiments, we fix $\nu = 1$ and $k = 3$ for the advanced method. In all cases, the stopping criteria is that both query configurations are mutually forward connected. The experiments were performed on a Pentium IV 3GHz with 1 GByte of memory. The results are summarized in Table 7.1. The first column gives the periods of both doors and the global period, i.e. the least common multiple of both.

The results clearly show that the time it takes for the naive approach to generate a roadmap grows more or less proportional with the global period of the environment. This relation is not strict however, as is indicated by the second experiment. Although the global period is smaller than in the third experiment, it still takes more time to generate a roadmap. This is because the individual periods of the doors are larger in the second experiment, giving less opportunities to pass the revolving doors. The running times of the advanced approach are not affected by the global period. Only the local periods of the doors play a role, which is indicated by experiments 2 and 4, which feature the largest local periods and hence

Table 7.1: Results for various periods of the revolving doors (see Fig. 7.5)

	Periods		Naive approach				
	local	**global**	**#vertex**	**#edge**	**roadmap**	**add query**	**dijkstra**
1	1, 1	1	665	9079	4.84s	0.017s	0.001s
2	5, 5	5	4010	59058	28.23s	0.031s	0.010s
3	2, 2.5	10	3198	30403	16.66s	0.018s	0.005s
4	2.5, 4	20	6975	69186	39.27s	0.037s	0.012s
5	1.3, 3.5	45.5	11404	95235	64.02s	0.059s	0.016s
6	2.9, 3.7	107.3	22177	167791	140.44s	0.135s	0.032s

	Periods		Advanced approach				
	local	**global**	**#vertex**	**#edge**	**roadmap**	**add query**	**dijkstra**
1	1, 1	1	711	2193	1.57s	0.031s	0.001s
2	5, 5	5	1499	6021	7.07s	0.053s	0.002s
3	2, 2.5	10	897	3110	2.55s	0.034s	0.001s
4	2.5, 4	20	1099	4073	3.91s	0.042s	0.001s
5	1.3, 3.5	45.5	899	3230	2.90s	0.038s	0.001s
6	2.9, 3.7	107.3	1146	4237	4.03s	0.041s	0.001s

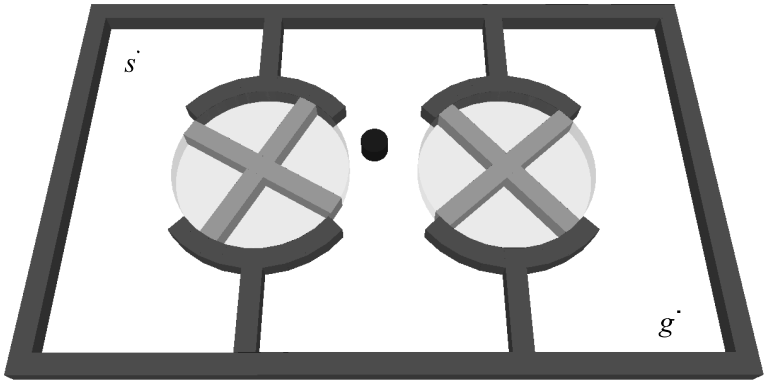


Figure 7.5: A repetitive environment with two revolving doors.

require the most time for a roadmap to be generated.

Adding the query configurations to the roadmap takes more time (relative to the roadmap size) in the advanced method than in the naive approach, because the technique used to determine whether or not an edge should be added between two nodes is more expensive in this case. However, for the advanced method this step is not part of the planning

time that needs to be pre-estimated. Only the Dijkstra-step is, and because the roadmaps stay small in the advanced method, this can be performed within insignificant amounts of time. Also, the amount of time needed can be estimated accurately; it scales more or less linearly with the number of edges in the roadmap (see the results of the naive approach). We can conclude that after a query is posed, the path can be executed with negligible latency, well within real-time constraints.

7.5 Conclusion

In this chapter we presented a probabilistically complete approach for path planning in repetitive environments. We showed that these environments need a different treatment than general dynamic environments, as they are not transitory. This allows for a multiple-shot approach in which a roadmap is created in a preprocessing phase. Experiments presented in this chapter prove that, using such a roadmap, path planning queries can be answered within minimal and predictable amounts of time. Hence, our approach can be used effectively in real-time applications.

Part II

Corridor Planning

Chapter 8

The Visibility-Voronoi Complex

In this chapter, we introduce a new type of diagram called the $VV^{(c)}$-diagram (the Visibility–Voronoi diagram for clearance c), which is a hybrid between the visibility graph and the Voronoi diagram of polygons in the plane. It evolves from the visibility graph to the Voronoi diagram as the parameter c grows from 0 to ∞. This diagram can be used for planning backbone paths for corridors. A good corridor backbone path, is short, smooth, and keeps — where possible — an amount of clearance c from the obstacles. The $VV^{(c)}$-diagram contains such paths. We also propose an algorithm that is capable of preprocessing a scene of configuration-space polygonal obstacles and constructs a data structure called the VV-complex. The VV-complex can be used to efficiently plan corridor backbone paths for any start and goal configuration and *any clearance value c*, without having to explicitly construct the $VV^{(c)}$-diagram for that c-value. The preprocessing time is $O(n^2 \log n)$, where n is the total number of obstacle vertices, and the data structure can be queried directly for any c-value by merely performing a Dijkstra search. We have implemented a CGAL-based software package for computing the $VV^{(c)}$-diagram in an *exact* manner for a given clearance value and used it to plan natural-looking paths in various applications.

This chapter has previously been published as: R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications*, 36:66–87, 2007 [160]. Extended abstracts have appeared as [156, 157].

8.1 Introduction

We study the problem of planning backbone paths for collision-free corridors in the plane among polygonal obstacles. A corridor may be used to provide the global direction of motion for some entity moving in an dynamic environment, yet providing flexibility to avoid local hazards. For a good corridor: (a) the (backbone) path should be *short* — that is, it should not contain long detours when significantly shorter routes are possible; (b) it should have a guaranteed amount of *clearance* — that is, the distance of any point on the path to the closest obstacle should not be lower than some prescribed value; and (c) it should be *smooth*, not

containing any sharp turns. Requirements (b) and (c) may conflict with requirement (a) in case it is possible to considerably shorten the path by taking a shortcut through a narrow passage. In such cases we may prefer a path with less clearance (and perhaps containing sharp turns).

Planning a path in a two dimensional environment is equivalent to the path planning problem for a robot with two degrees of freedom (a disc robot, or a polygonal robot that can only translate — and not rotate — in the plane) moving amidst polygonal obstacles. This problem can be efficiently solved by computing a complete representation of the free configuration space, as suggested by Kedem *et al.* [90]. This approach was simplified, by decomposing the configuration space into pseudo-trapezoidal cells and constructing a roadmap of the free cells, and was robustly implemented for a polygonal robot [4] and for a disc robot [72]. However, the trapezoidal-map approach yields paths that are piecewise linear (hence not smooth) and that are often not the shortest paths. Another popular approach for solving path planning problems is using Probabilistic Roadmaps (PRMs — see, e.g., [89]) — but the output paths in this case are also piecewise linear and may be far from the shortest possible paths. Indeed, in both cases it is possible to perform path smoothing as a post-processing stage and produce a more natural-looking path (see [59] for a summary of applicable smoothing techniques), but as there is no guarantee that the initial path is in the same homotopy class as the best path possible, the smoothed path may be different from the optimal corridor backbone path.

The *visibility graph* is a well-known data structure for computing the shortest collision-free path between a start and a goal configuration (see, e.g., [26, Chapter 15]). However, shortest paths are in general tangent to obstacles, so a path computed from a visibility graph usually contains semi-free configurations (the robot is in contact with an obstacle, but their interiors do not intersect) and therefore does not have any clearance. This is unacceptable for corridors, and for many other path planning applications as well. On the other hand, planning motion paths using the *Voronoi diagram* of the obstacles [119] yields a path with maximal clearance, but this path may be significantly longer than the shortest path possible and may also contain sharp turns.

We suggest a hybrid of these two latter approaches, called the $\mathrm{VV}^{(c)}$*-diagram* (the Visibility–Voronoi diagram for clearance c), yielding good corridor backbone paths, meeting all three criteria mentioned above (with the reservation mentioned above regarding narrow passages). It evolves from the visibility graph to the Voronoi diagram as c grows from 0 to ∞, where c is the preferred amount of clearance. The $\mathrm{VV}^{(c)}$-diagram contains the visibility graph of the obstacles dilated with a disc of radius c. The dilated obstacle vertices become circular arcs in this case, and the visibility edges are bitangent to these arcs. This guarantees that the paths in the diagram are not only *short* but also *smooth*. However, due to this obstacle inflation, narrow passages in the scene may disappear, which implies that it is not possible to pass through these narrow passages keeping a distance of at least c from the obstacles. As we still want to keep the option of traversing these narrow passages (for example when a pass through a narrow passage is significantly shorter than any alternative path), we integrate into the diagram paths with the maximal possible clearance in regions where the preferred clearance c cannot be obtained. It is easy to see that these paths are portions of the Voronoi diagram of the original obstacles.

Besides the straightforward algorithm for constructing the $VV^{(c)}$-diagram for a given clearance value c, we also propose an algorithm for preprocessing a scene of configuration-space polygonal obstacles and constructing a data structure called the VV-*complex*.[1] The VV-complex can be used to efficiently plan motion paths for any start and goal configuration and any given clearance value c, without having to explicitly construct the $VV^{(c)}$-diagram for that c-value, by performing a Dijkstra search on an implicitly constructed graph encoded by the VV-complex. The preprocessing time is $O(n^2 \log n)$, where n is the total number of obstacle vertices, and the query takes $O(n \log n + \ell)$ time, where ℓ is the number of edges encountered during the search and is bounded by the number of diagram edges. Furthermore, we reduce the number of costly geometric operations in the query stage and perform the most time-consuming computations in the preprocessing stage.

8.1.1 Applications

Other than planning corridor backbone paths, a direct application of our construct is planning natural motion paths for a polygonal robot among polygonal obstacles. We can compute the Minkowski sum of each obstacle with the robot rotated by 180° to obtain a set of configuration-space obstacles, which are also polygonal. After this initial step we may assume that the robot is a point. Constructing the $VV^{(c)}$-diagram of these configuration-space obstacles and giving adequate weights to the diagram edges (see the discussion in Section 8.3) yield more natural motion paths, compared, for example, to the implementation of [4].

A direct application of using corridors is motion planning for a group of entities in a two-dimensional environment cluttered with polygonal obstacles. Kamphuis and Overmars [82] solve this problem by planning a collision-free path for a single entity, then "inflating" this backbone path up to a diameter of a preferred group width w, wherever possible, and governing the motions of the individual entities inside this inflated path using a potential field. They use a PRM with cycles [115] to compute the initial path, then apply smoothing techniques on it to achieve the final backbone path. While the smoothing procedure outputs more natural paths, it has several drawbacks. First, it is an expensive operation, so to obtain real-time performance it can only be done for the final selected path, and not for other candidate paths that can be computed using the PRM. Note that as smoothing may considerably deform the original path, it is not guaranteed that we get the shortest path possible if we smooth the shortest path obtained from the PRM. Also, as the PRM method is not complete, it is not guaranteed that the PRM contains all possible paths between the start and goal locations. It is even possible that the two locations cannot be connected using the PRM although there exists a path connecting them. In contrast, the $VV^{(c)}$-diagram of the environment for $c = \frac{w}{2}$ contains all paths between a start and a goal configuration and is ideal for computing the backbone path, especially if the weight given to the diagram edges is proportional to the *time* it takes the group to traverse each edge. Furthermore, the $VV^{(c)}$-diagram does not require any smoothing step, which saves a precious amount of time and enables real-time performance.

[1] Despite the similarity in names, our structure is different from the *visibility complex* introduced by Pocchiola and Vegter [126] for efficiently computing the visibility among disjoint convex objects in the plane.

The principles of our construction may also be applied to sensor-based coverage using a robot with a limited sensor radius. Acar *et al.* [3] devised an algorithm for a disc robot of radius r, carrying a detector with a range $R > r$, to detect all points in an unknown environment. They decompose the free space into *vast* cells, where the robot traverses the boundary of the obstacles dilated by radius R, and *narrow* cells, where the obstacles are within the detector range and the robot has to follow the Voronoi diagram. It is possible to use the $\mathrm{VV}^{(c)}$-diagram in this case for traversing the narrow cells, as it naturally connects the relevant portions of the Voronoi diagram to the vast cells.

We have implemented our algorithm for constructing the $\mathrm{VV}^{(c)}$-diagram for a given clearance value and applied it to the problem of motion planning for coherent groups of entities. The paths contained in the $\mathrm{VV}^{(c)}$-diagram yield convincing group motions, and the approach we propose has several advantages over methods used so far to generate group paths. We note that the robust construction of the $\mathrm{VV}^{(c)}$-diagram involves many non-trivial geometric procedures and requires careful algebraic computations, which we also discuss in this chapter.

8.1.2 Outline

The rest of this chapter is organized as follows: In Section 8.2 we give a short review of the geometric data structures we use for constructing the $\mathrm{VV}^{(c)}$-diagram. In Section 8.3 we present the $\mathrm{VV}^{(c)}$-diagram in more detail and explain how to construct it, given a scene with obstacles and a preferred clearance value c. In Section 8.4 we introduce the VV-complex, show how to efficiently construct it and explain how to query it. In Section 8.5 we review the software we have developed to robustly compute the $\mathrm{VV}^{(c)}$-diagram of a set of obstacles and a given c-value. We conclude with some experimental results in Section 8.6 and closing remarks in Section 8.7.

8.2 Preliminaries

8.2.1 Visibility Graphs

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of simple pairwise interior-disjoint polygons having n vertices in total. The *visibility graph* of $\mathcal{P}$ is an undirected graph defined on the set of polygon vertices, whose set of edges consists of those pairs of vertices that are mutually visible. Two vertices are *mutually visible* if the straight line segment connecting them does not intersect the *interior* of any of the polygons in $\mathcal{P}$ — in this case, we call this segment a *visibility edge*.

The visibility graph can be used to compute shortest paths amidst configuration-space polygonal obstacles, where the polygons are considered as open sets. Each edge is given a weight equal to the Euclidean distance between its two end-vertices. To find a shortest path between a start and a goal configuration, one simply needs to connect them to the visibility graph and execute Dijkstra's algorithm starting from the vertex representing the start configuration. In fact, it is sufficient to consider only the edges that are bitangent to the polygons they connect, namely edges that can be infinitesimally extended in both directions

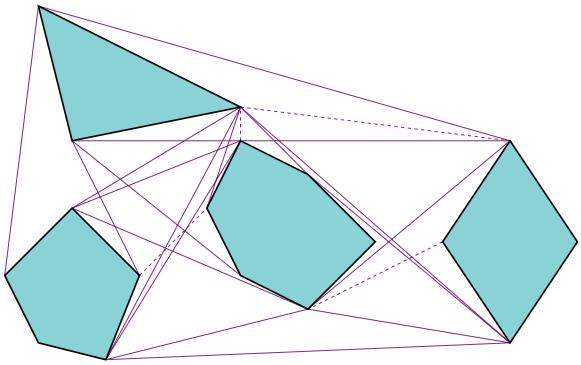


Figure 8.1: The (partial) visibility graph of a set of four convex polygons. The valid visibility edges are drawn with solid lines, while some invalid edges are also shown, drawn with dashed lines. Notice that all obstacle edges are also valid visibility edges.

without penetrating any polygon. Such bitangent edges are called *valid* visibility edges (see Fig. 8.1 for an illustration).

The visibility graph can be computed in $O(n^2 \log n)$ time, performing a straightforward radial sweep around each of the polygon vertices (see, e.g., [26, Chapter 15]). Ghosh and Mount [64] were the first to give an output-sensitive algorithm for computing the visibility graph in optimal $O(n \log n + k)$ time, where k is the number of visibility edges in the output visibility graph. For more information on shortest paths, see [110].

8.2.2 Voronoi Diagrams of Polygons

Given a set S of geometric entities in $\mathbb{R}^d$ and a distance metric $\|\cdot\|$, the *Voronoi diagram* of S, denoted $\mathrm{Vor}(S)$, is the subdivision of $\mathbb{R}^d$ into maximal connected cells, such that the points in each *Voronoi cell* are closer to a specific entity of S than to all other entities of S.

There are many variants of Voronoi diagrams (see [9, 51] for extensive reviews). Here we focus on the Voronoi diagram of a set of pairwise interior-disjoint polygons in $\mathbb{R}^2$ under the Euclidean distance metric, which can be regarded as a special case of a Voronoi diagram of line segments [102]. For each point $p \in \mathbb{R}^2$ we consider the polygon feature (a polygon *feature* is either a vertex or an edge) closest to p. The *Voronoi vertices* in this case are points equidistant to closest features of three (or more) different polygons. The vertices are connected by continuous *chains* of *Voronoi arcs*. An arc may be equidistant to two closest vertices or to two closest polygon edges — in which case it is a straight line segment, or to a polygon vertex and a (non-incident) polygon edge — in which case it is a segment of a parabola (parabolic arc). Each arc has two *endpoints*, which either connect it to the next arc in the chain or to a Voronoi vertex.

For any point p in the plane, let the *clearance value* of p be the distance from the point to the closest polygon. If we examine the clearance value along a Voronoi chain, we notice that in most cases the minimum clearance value is obtained in the interior of a vertex–vertex

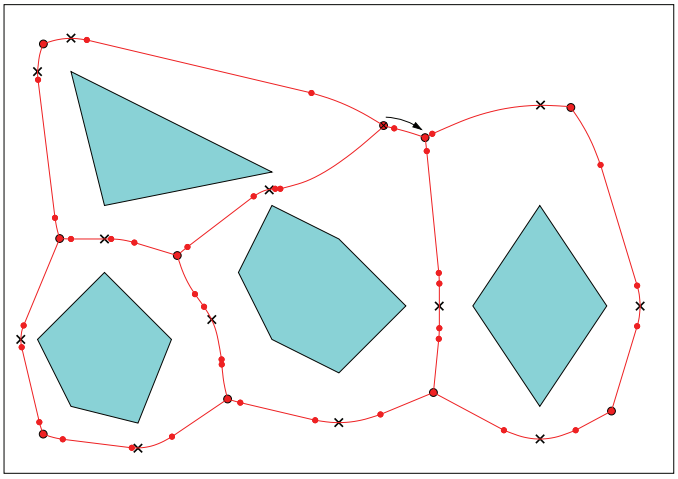


Figure 8.2: The Voronoi diagram of four convex polygons contained inside a rectangle. Small dots mark the endpoints of each Voronoi arc, while the Voronoi vertices are marked by larger dots. The point of minimum clearance along each chain is marked by ×. Notice that the chain marked by an arrow is a monotone chain and obtains its minimal clearance on its left Voronoi vertex — so when we traverse it in the arrow's direction, the clearance only increases.

or a vertex–edge arc inside the chain (note that the interior of an edge–edge arc will never contain a clearance minimum). In such cases, the clearance value increases as we move from this minimum point toward either of the chain's end-vertices. However, for some chains the minimum clearance value is obtained at one of their end-vertices and grows as we move along the chain toward its other end. We call such a chain a *monotone Voronoi chain* (see Fig. 8.2 for an illustration).

The Voronoi diagram can be used to compute paths with maximal amounts of clearance from the obstacles. It can be shown that the total complexity of the Voronoi diagram is $O(n)$, where n is the total number of polygon vertices, and that it can be constructed in $O(n \log n)$ time (see, e.g, [9, 102]). For more details on the connection between Voronoi diagrams and path planning see [118, 119, 132].

8.2.3 Minkowski Sums

The *Minkowski sum* of two given sets $A, B \in \mathbb{R}^d$, denoted $A \oplus B$, is defined as:

$$A \oplus B = \{a + b \mid a \in A,\ b \in B\} \ .$$

In particular, if we are given a polygon P, the set of points whose distance from P is less than ρ is the Minkowski sum $P \oplus B_\rho$, where B_ρ is a disc of radius ρ. This set can be computed in time linear in the size of the polygon P. The Minkowski sum of a set of simple polygons $\mathcal{P}$ and a disc has $O(n)$ complexity and can be computed in $O(n \log^2 n)$ time using a divide-and-conquer algorithm [90], where n is the total number of polygon vertices. It is also possible

to use an incremental randomized algorithm that achieves a running time of $O(n \log n)$ (see, e.g., [27]). See [26, Chapter 13] and [72] for further discussions and more references.

8.3 The VV$^{(c)}$-Diagram

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of simple pairwise interior-disjoint polygons in the plane, having n vertices in total, representing two-dimensional configuration-space obstacles. Let $c > 0$ be the preferred distance we wish to keep from these obstacles. Our goal is to preprocess $\mathcal{P}$, so that given a start configuration s and a goal configuration g, we can efficiently compute a shortest path between s and g, keeping a clearance of at least c from the obstacles where possible, but allowing to get closer to the obstacles in narrow passages when it is possible to make considerable shortcuts.

8.3.1 Constructing the VV$^{(c)}$-Diagram

We begin by dilating each obstacle by c — that is, computing the Minkowski sum of each polygon with a disc of radius c. The visibility graph of the dilated obstacles contains all shortest paths with a clearance of at least c from the obstacles. Note that the dilated polygon edges are also valid visibility edges. Moreover, as each convex polygon vertex becomes a circular arc of radius c, the valid visibility edges are bitangents to two circular arcs. This guarantees that a shortest path extracted from such a visibility graph is $\mathcal{C}_1$-smooth and contains no sharp turns. The only disadvantage in this approach is that narrow, yet collision-free, passages can be blocked when we dilate the obstacles (for example, in Fig. 8.3 there exists such a narrow passage between P_1 and P_3). It is clearly not possible to pass through such passages with a clearance of at least c, but we still wish to allow a path with the maximal clearance possible in this region. To do this, we compute the portions of the free configuration space that are contained in at least two dilated obstacles and add their intersection with the Voronoi diagram of the original polygons to our diagram. The resulting structure is called the VV$^{(c)}$-diagram.

Formally, given a collection of disjoint convex obstacles $P_1, \ldots, P_m$ (we will later discuss non-convex obstacles as well) and a preferred clearance value c, we perform the following steps:

1. We construct the Minkowski sum $M_i^{(c)} = P_i \oplus B_c$ for every obstacle P_i, where B_c is a disc with radius c. Note that the inflated obstacles $M_i^{(c)}$ may no longer be disjoint.

2. We compute the union $\mathcal{M}^{(c)}$ of all $M_i^{(c)}$. The boundary of $\mathcal{M}^{(c)}$ consists of circular arcs and straight line segments. Reflex vertices may appear on the boundary of $\mathcal{M}^{(c)}$, which are the intersection of the boundary arcs of two dilated obstacles, and we refer to them as *chain points*, as they lie on Voronoi chains, since their distance from both relevant polygons is exactly c.

3. We compute the modified visibility graph $\mathcal{G}^{(c)}$ of $\mathcal{M}^{(c)}$. This graph consists of every free[2] bitangent of two circular arcs of the boundary of $\mathcal{M}^{(c)}$ (the edges that form the

[2] A line segment is *free* if its interior is not contained in the interior of any dilated obstacle.

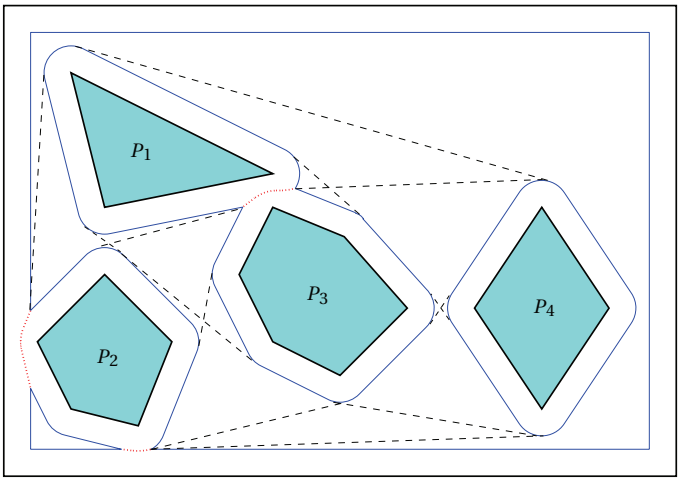


Figure 8.3: The $\mathrm{VV}^{(c)}$-diagram for four convex obstacles located in a rectangular room. The boundary of the union of the dilated obstacles is drawn in a solid line, the relevant portion of the Voronoi diagram is dotted. The visibility edges are drawn using a dashed line. Notice that an endpoint of a visibility edge may either lie on a circular arc or on the intersection of two dilated obstacle boundaries (a chain point).

boundary of $\mathcal{M}^{(c)}$ are also regarded as bitangents to two neighboring dilated vertices), every free line segment between two chain points, and every free line segment from a chain point tangent to a circular arc.

4. We construct $\mathcal{V}$, the Voronoi diagram of the original set of polygons and compute the intersection $\mathcal{V} \cap \mathcal{M}^{(c)}$, namely the portion of the Voronoi diagram that is contained within the union of the dilated obstacles. We combine the corresponding Voronoi arcs (and sub-arcs) with $\mathcal{G}^{(c)}$ to connect the chain points via narrow passages and form the final $\mathrm{VV}^{(c)}$-diagram.

As mentioned in Section 8.2.3, step 1 can be carried out in linear time while step 2 takes $O(n \log^2 n)$ time (or $O(n \log n)$ expected time using a randomized algorithm). In step 4 we construct the Voronoi diagram in $O(n \log n)$ time, while computing the intersection $\mathcal{V} \cap \mathcal{M}^{(c)}$ can be carried out in linear time. Thus, step 3, which can easily be performed in $O(n^2 \log n)$ time, clearly dominates the running time of the $\mathrm{VV}^{(c)}$-diagram construction process. We conclude that it takes $O(n^2 \log n)$ time to construct the $\mathrm{VV}^{(c)}$-diagram of an input set $\mathcal{P}$ of pairwise interior-disjoint polygons for a given c-value if we use a straightforward approach. We note that it might also be possible to improve the running time to be $O(n \log n + k)$, where k is the number of visibility edges, by constructing the visibility complex of the dilated polygons [126].[3]

[3]The main difficulty here is that we handle *dilated* obstacles, which may *not* be disjoint. Moreover, the obstacles (and of course the dilated obstacle) are not of constant complexity.

In case our polygons are not convex, we decompose them to obtain a set of convex polygons and compute $\mathcal{M}^{(c)}$ for this set. Note that in this case not every reflex vertex of $\mathcal{M}^{(c)}$ is now a chain point, since reflex vertices can also be induced by reflex vertices of the original polygons. However, these reflex vertices of $\mathcal{M}^{(c)}$ can be easily identified and are not taken into account in the $\mathrm{VV}^{(c)}$-diagram (namely the diagram does not contain visibility edges emanating from these vertices).

8.3.2 Querying the $\mathrm{VV}^{(c)}$-Diagram

Having constructed the $\mathrm{VV}^{(c)}$-diagram, once we are given a start configuration s and a goal configuration g we just have to connect them to our diagram and compute the shortest path between s and g using Dijkstra's algorithm. It takes $O(n\log n)$ time to connect s and g to the diagram, by performing a radial sweep from each configuration. The execution of Dijkstra's algorithm takes $O(n\log n+\ell)$, where ℓ is the number of diagram edges we encounter during the search (which is at most k).

As mentioned before, we may compromise on the amount of clearance our motion path keeps from the obstacles if we can make a shortcut by traversing through a narrow passage. It should be noted that if a path contains a portion of the Voronoi diagram it may not be smooth any more (this is however acceptable, as we consider making sharp turns inside narrow passages to be natural). In order to balance between the length and the clearance of the selected path we have to associate the appropriate weight with each diagram edge, so the Dijkstra algorithm outputs the path which is most suitable for our application. The weight of a visibility edge can simply be equal to its length (the lengths of the circular arcs we traverse must also be taken into consideration), while for Voronoi edges we may add some penalty to the edge length, taking into account their clearance values, which are below the preferred c-value. For example, if the minimal clearance of a Voronoi arc is $c' < c$, we can give it the weight of its length multiplied by $\left(\frac{c}{c'}\right)^{\kappa}$, where $\kappa > 0$ is a parameter controlling the amount of extra weight given to Voronoi arcs.

Another option of weighting the edges, especially suitable for the application of coherent group motion (see Section 8.1) is to estimate the time it takes the group to traverse each edge: For edges with a clearance of at least $c = \frac{w}{2}$, where w is the preferred group width, this time is clearly proportional to the edge length. On the other hand, for Voronoi edges the actual clearance of the edge would also be taken into account, as the moving entities will have to traverse this edge in a long row. The resulting path will therefore be the one enabling the group to reach its goal as quickly as possible.

8.4 The VV-Complex

The construction of the $\mathrm{VV}^{(c)}$-diagram for a given c-value is straightforward, yet it requires some non-trivial geometric and algebraic operations that should be computed in a robust manner — see Section 8.5 for more details. Moreover, if we wish to plan motion paths for different c-values and select the best one (according to some criterion), we must construct the $\mathrm{VV}^{(c)}$-diagram for each c-value from scratch. In this section we explain how to efficiently

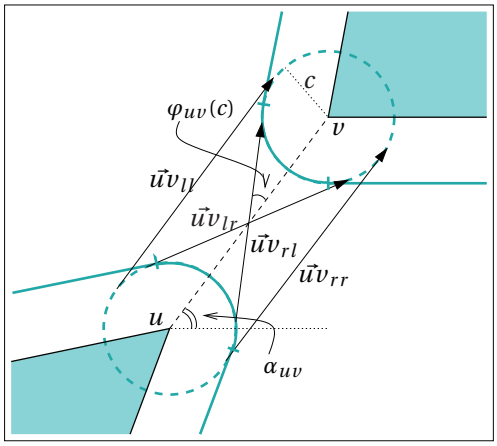


Figure 8.4: The four possible bitangents to the circles $B_c(u)$ and $B_c(v)$ of radius c centered at two obstacle vertices u and v. Notice that in this specific scenario only the bitangent $\vec{uv}_{rl}$ is a valid visibility edge.

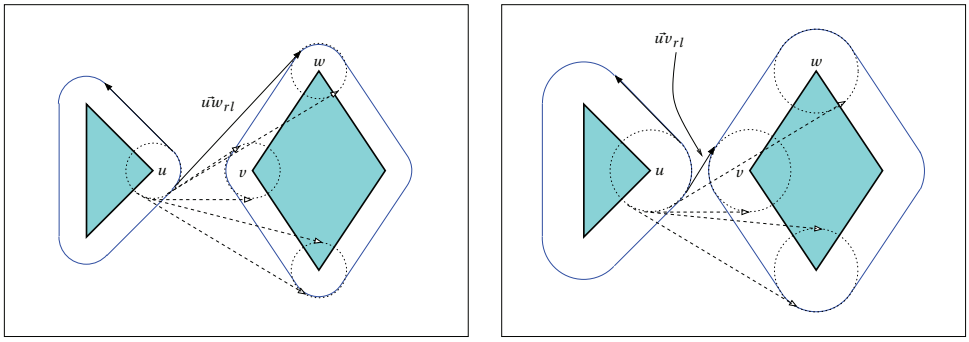
preprocess an input set of polygonal obstacles and construct a data structure called the VV-complex, which can be queried to produce a natural-looking path for every start and goal configuration and for *any* preferred clearance value c.

Let us examine what happens to the $\mathrm{VV}^{(c)}$-diagram as c continuously changes from zero to infinity. For simplicity, we consider only convex obstacles in this section. As we mentioned before, $\mathrm{VV}^{(0)}$ is the visibility graph of the original obstacles, while $\mathrm{VV}^{(\infty)}$ is their Voronoi diagram, so as c grows visibility edges disappear from $\mathrm{VV}^{(c)}$ and make way to Voronoi chains. We start with a set of visibility edges containing all pairs of the polygonal obstacle vertices that are mutually visible, regardless whether these edges are bitangents of the obstacles.[4] We also include the original obstacle edges in this set, as they can be viewed as visibility edges between two adjacent polygon vertices. Furthermore, we treat our visibility edges as directed, such that if the vertex u "sees" the vertex v, we will have two directed visibility edges in our structure, $\vec{uv}$ and $\vec{vu}$.

As c grows larger than zero, each of the *original* visibility edges potentially spawns as many as four bitangent visibility edges. These edges are the bitangents to the circles $B_c(u)$ and $B_c(v)$ (where $B_r(p)$ denotes a circle centered at p whose radius is r) that we name $\vec{uv}_{ll}$, $\vec{uv}_{lr}$, $\vec{uv}_{rl}$ and $\vec{uv}_{rr}$, according to the relative position (left or right) of the bitangent with respect to u and to v (see Fig. 8.4).[5] Let α_{uv} be the angle between the vector $\vec{uv}$ and the

[4] Visibility edges are only *valid* when they are bitangents, otherwise they do not contribute to shortest paths in the visibility graph. However, as c grows larger the invalid edges may become bitangents, as shown in Fig. 8.6(b), so we need them in our data structure.

[5] Recall that edges in the visibility graph are *undirected*, thus our *directed* visibility edges come in pairs. According to our notation, $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ are equivalent to the opposite edges $\vec{vu}_{rr}$ and $\vec{vu}_{ll}$, respectively, while $\vec{uv}_{lr}$ and $\vec{uv}_{rl}$ are equivalent to $\vec{vu}_{lr}$ and $\vec{vu}_{rl}$, respectively. A pair of opposite edges always become valid or invalid simultaneously.



(a) (b)

Figure 8.5: The circular list $\mathcal{L}_r(u)$ of "right" visibility edges associated with an obstacle vertex u. Valid visibility edges are drawn as solid arrows while invalid edges are drawn as dashed arrows (for clarity, some edges are omitted). Note that in (a) $\vec{uw}_{rl}$ is a valid visibility edge, but as we increase c in (b), it is blocked by $\vec{uv}_{rl}$ (which becomes a valid edge) and removed from the list.

x-axis, and $d(u,v)$ the Euclidean distance between u and v, then it is easy to see that the two bitangents $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ retain the same slope α_{uv} for increasing c-values. The slope of the other two bitangents changes as c grows: $\vec{uv}_{rl}$ rotates counterclockwise and $\vec{uv}_{lr}$ rotates clockwise by the same amount, both around the midpoint $\frac{1}{2}(u+v)$ of the original edge, so their slopes become $\alpha_{uv}+\varphi_{uv}(c)$ and $\alpha_{uv}-\varphi_{uv}(c)$, respectively, where $\varphi_{uv}(c) = \arcsin(\frac{2c}{d(u,v)})$. For $c > \frac{1}{2}d(u,v)$ the two edges $\vec{uv}_{rl}$ and $\vec{uv}_{lr}$ disappear.

Note that for a given c-value, it is impossible that all four edges are valid (as we consider only obstacles with non-empty interiors, line segments do not qualify, and therefore each circular arc on a boundary of a dilated obstacle is of angle less than 180° — thus, at most three can be valid, and the edges $\vec{uv}_{ll}$ and $\vec{uv}_{rr}$ can never be valid simultaneously). Our goal is to compute a *validity range* $R(e) = [c_{\min}(e), c_{\max}(e)]$ for each edge e, such that e is part of the $\mathrm{VV}^{(c)}$-diagram for each $c \in R(e)$.[6] If an edge is valid, then it must be tangent to both circular arcs associated with its end-vertices. There are several reasons for an edge to change its validity status:

- The tangency point of e to either $B_c(u)$ or to $B_c(v)$ leaves one of the respective circular arcs.
- The tangency point of e to either $B_c(u)$ or to $B_c(v)$ enters one of the respective circular arcs.
- The visibility edge becomes blocked by the interior of a dilated obstacle.

[6] Liu and Arimoto [107] use a similar notion to construct a structure that answers shortest-path queries for disc robots, where the radius of the robot is given in the query. They do not, however, incorporate portions of the Voronoi diagram in their construct.

The important observation is that at the moment that a visibility edge $\vec{uv}$ gets blocked, it becomes tangent to another dilated obstacle vertex w, so essentially one of the edges associated with $\vec{uv}$ becomes equally sloped with one of the edges associated with $\vec{uw}$ (see Fig. 8.6(a)). The first two cases mentioned above can also be realized as events of the same nature, as they occur when one of the $\vec{uv}$ edges becomes equally sloped with $\vec{uw}_{lr}$ (or $\vec{uw}_{rl}$), when v and w are adjacent vertices in a polygonal obstacle — see Fig. 8.6(b).

This observation stands at the basis of the algorithm we devise for constructing the VV-complex: We sweep through increasing c-values, stopping at critical *visibility events*, which occur when two edges become equally sloped.[7] We note that the edge $\vec{uv}_{ll}$ (or $\vec{uv}_{lr}$) can only be involved in visibility events with arcs of the form $\vec{uw}_{ll}$ or $\vec{uw}_{lr}$, while the edge $\vec{uv}_{rl}$ (or $\vec{uv}_{rr}$) can only have events with arcs of the form $\vec{uw}_{rl}$ or $\vec{uw}_{rr}$. Hence, we can associate two circular lists $\mathcal{L}_l(u)$ and $\mathcal{L}_r(u)$ of the left and right edges of the vertex u, respectively, both sorted by the slopes of the edges. Two edges participate in an event at some c-value only if they are neighbors in one of these lists for infinitesimally smaller c (see Fig. 8.5 for an illustration). At these event points, we should update the validity range of the edges involved and also update the adjacencies in their appropriate lists, resulting in new events.

As mentioned in Section 8.3, an endpoint of a visibility edge in the $\mathrm{VV}^{(c)}$-diagram may also be a chain point, so we must consider chain points in our algorithm as well. As a Voronoi chain is either monotone or has a single point with minimal clearance, we need to associate at most two chain points with every Voronoi chain. Our algorithm will also have to compute the validity ranges of edges connecting a chain point with a dilated vertex or with another chain point. For that purpose, we will have a list $\mathcal{L}(p)$ of the outgoing edges of each chain point p, sorted by their slopes (notice that we do not have to separate the "left" edges from the "right" edges in this case).

In the next subsection we review the algorithmic details of the preprocessing stage for constructing the VV-complex, and describe how to query this data structure in Section 8.4.2. We continue the presentation of the algorithm by a proof of correctness in Section 8.4.3 and a complexity analysis in Section 8.4.4. We finally explain how the algorithm can be generalized for non-convex polygons in Section 8.4.5.

8.4.1 The Preprocessing Stage

Initialization

Given an input set $P_1,\ldots,P_m$ of convex interior-disjoint polygonal obstacles, we start by computing their visibility graph and classifying the visibility edges as valid (bitangent) or invalid. We examine each bitangent visibility edge uv: For an infinitesimally small c only one of the four $\vec{uv}$ edges it spawns is valid — we assign 0 to be the minimal value of the validity range of this edge (and of the opposite $\vec{vu}$ edge).

As our algorithm is event-driven, we initialize an empty event queue $\mathcal{Q}$, storing events by increasing c-order.

We proceed by constructing the circular lists $\mathcal{L}_l(u)$ and $\mathcal{L}_r(u)$ for each obstacle vertex u, based on the visibility edges we have just computed. We examine each pair of adjacent

[7] Our visibility events are reminiscent of the *merge events* and *split events* that occur in the algorithm for drawing "fat" planar edges, as suggested by Duncan *et al.* [46].

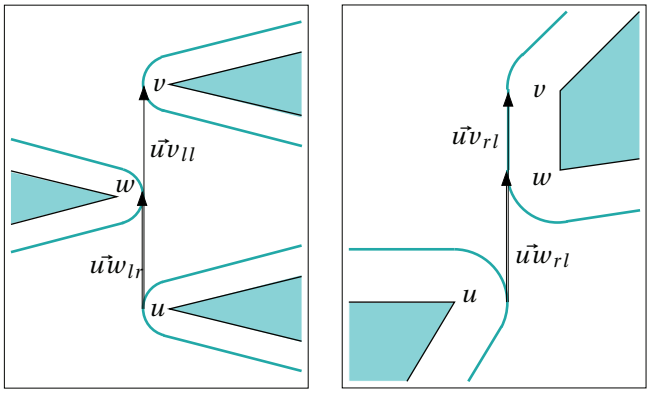


Figure 8.6: Visibility events involving u, v and w: (a) The dilated vertex w blocks the visibility of u and v. (b) As $\vec{uw}_{rl}$ becomes equally sloped with $\vec{uv}_{rl}$ (where vw is an obstacle edge), it becomes a valid visibility edge.

edges e_1, e_2 in $\mathcal{L}_l(u)$ (and in $\mathcal{L}_r(u)$), compute the c-value at which e_1 and e_2 become equally sloped — if one exists — and insert the *visibility event* $\langle c, e_1, e_2 \rangle$ into the event queue. In a visibility event some edges become blocked and reach the end of their validity range, while some new edges may become valid.

As our VV-complex also contains Voronoi chains, we have to compute the Voronoi diagram of the polygonal obstacles. For each *non-monotone* Voronoi chain we locate the arc a that contains the minimal clearance value $c_{\min}$ of the chain in its interior and insert the *chain event* $\langle c_{\min}, a \rangle$ into $\mathcal{Q}$. A chain event occurs when a Voronoi chain starts contributing to the $\mathrm{VV}^{(c)}$-diagram, namely when we sweep through its minimal clearance value. For now, we do not need to worry about monotone chains — in the next section we explain how we can correctly handle them without associating chain events with them.

Event Handling

While the event queue is not empty, we proceed by extracting the event in the front of $\mathcal{Q}$, associated with minimal c-value, and handle it according to its type.

Visibility event: Visibility events always come in pairs — that is, if $\vec{uv}$ becomes equally sloped with $\vec{uw}$,[8] we will either have an event for the opposite edges $\vec{vu}$ and $\vec{vw}$, or for the opposite edges $\vec{wu}$ and $\vec{wv}$. We therefore handle a pair of visibility events as a single event. Let us assume that the edges $\vec{uv}$ and $\vec{uw}$ become equally sloped for a clearance value c', and at the same time the edges $\vec{vu}$ and $\vec{vw}$ become equally sloped (see Fig. 8.6).

[8] In the rest of this section, we use the notation $\vec{uv}$ to represent any of the four edges $\vec{uv}_{ll}$, $\vec{uv}_{lr}$, $\vec{uv}_{rl}$ or $\vec{uv}_{rr}$. We also use $\mathcal{L}(u)$ to denote either $\mathcal{L}_l(u)$ or $\mathcal{L}_r(u)$ (whether we choose the "left" or the "right" list depends on the type of edge involved).

As the edges $\vec{uv}$ and $\vec{vu}$ now become blocked, we assign c' to be the maximal c-value of the validity range of these edges. We also remove the other event, if any, involving $\vec{uv}$ (based on its other adjacency in $\mathcal{L}(u)$) from $\mathcal{Q}$, and delete this edge from $\mathcal{L}(u)$. We examine the new adjacency created in $\mathcal{L}(u)$ and insert its visibility event into the event queue $\mathcal{Q}$. We repeat this procedure for the opposite edge $\vec{vu}$.

If the edge $\vec{uv}$ was valid before it was deleted and the edge $\vec{uw}$ (or $\vec{vw}$) does not have a minimal validity value yet, we assign c' to it, because this edge has become bitangent for this c-value (see Fig. 8.6(b) for an illustration).

Chain event: The value c equals the minimal clearance of a Voronoi chain χ_a, obtained on the arc a, which is equidistant from an obstacle vertex u and another obstacle feature (see Fig. 8.7(b)).[9] Let z_1 and z_2 be a's endpoints. We initiate two chain points $p_1(\chi_a)$ and $p_2(\chi_a)$ associated with the Voronoi chain χ_a. As c grows, $p_1(\chi_a)$ moves toward z_1 and $p_2(\chi_a)$ moves toward z_2 (see Fig. 8.7(c) for an illustration).

As we increase c, larger portions of χ_a will enter the $\mathrm{VV}^{(c)}$-diagram and visibility edges will become incident to its chain points, rather than to dilated vertices. We therefore have to examine all edges e incident to u, compute the minimum c-value c' for which e becomes incident to one of the chain points $p_i(\chi_a)$, and insert the *tangency event* $\langle c', e, p_i(\chi_a)\rangle$ into the event queue. If a is equidistant from u and from another obstacle vertex v (i.e., a is a vertex–vertex Voronoi arc), we do the same for the edges incident to v.

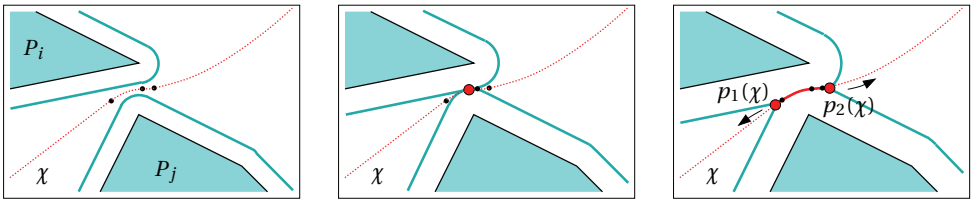
Finally, we create two *endpoint events*, $\langle c_1, p_1(\chi_a), z_1\rangle$ and $\langle c_2, p_2(\chi_a), z_2\rangle$, associated with the clearance values c_1 and c_2 obtained at z_1 and z_2, respectively.

When dealing with a chain event, we introduced two additional types of events, used to handle chain points: tangency events and endpoint events. For a small enough c value (smaller than the clearance value of any point on the Voronoi diagram) the endpoints of all visibility edges lie on dilated obstacle vertices, but as c grows these endpoints gradually become chain points. A *tangency event* occurs when a visibility edge becomes incident to a chain point. The *endpoint events* are used to transfer the chain points along Voronoi chains. We next explain how we deal with these events.

Tangency event: A visibility edge $e = \vec{ux}$ (the endpoint x may either represent a dilated vertex or a chain point) becomes tangent to $B_{c'}(u)$ at a chain point $p(\chi_a)$ associated with the Voronoi arc a (see Fig. 8.8 for an illustration for the case when $x = v$ is a dilated obstacle vertex). In this case we have to replace e by the visibility edge $p(\vec{\chi_a})x$ associated with the chain point $p(\chi_a)$. We assign c' to be the maximal validity value of the edge e, and remove it from $\mathcal{L}(u)$. We now insert a reincarnate of e to $\mathcal{L}(p(\chi_a))$, and assign c' as its minimal validity value. We examine the new adjacency in $\mathcal{L}(p(\chi_a))$ and insert, if necessary, a new visibility event into $\mathcal{Q}$.[10] Finally, we replace the edge $\vec{xu}$ in $\mathcal{L}(x)$

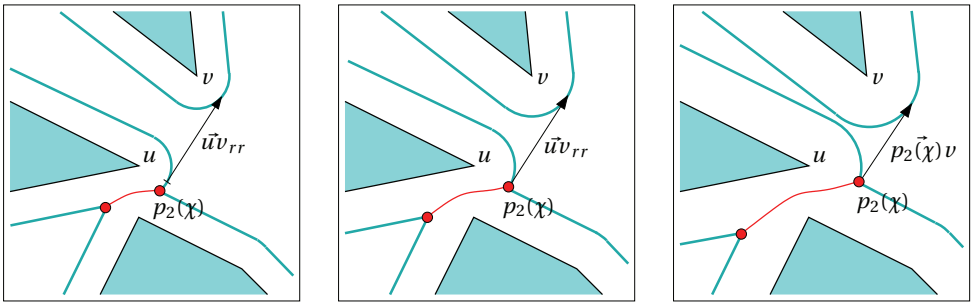
[9] Recall that a Voronoi arc equidistant to two polygon edges is always monotone with respect to the clearance and can never contain a chain minimum in its interior.

[10] For a given c-value, let $\vec{\omega}$ be the direction of the tangent to the Voronoi chain χ_a, such that when we infinitesimally increase c, the chain point $p(\chi_a)$ moves in this direction. Note that even though $\mathcal{L}(p(\chi_a))$ is represented as a circular list, the vector -$\vec{\omega}$ splits it into a linear list. We note that a tangency event always results in the insertion of a new edge at one of the list ends, so only one true adjacency is created.



(a) (b) (c)

Figure 8.7: A chain event associated with the Voronoi chain χ (dotted) induced by the two obstacles P_i and P_j. The endpoints of the arcs forming χ are drawn as small black dots. (a) The clearance value c is less than the minimal clearance of the chain χ, so this chain does not contribute to the $\mathrm{VV}^{(c)}$-diagram. (b) c equals the minimal clearance of the chain χ and a chain event occurs. Note that the two dilated obstacles now begin to intersect. (c) When c grows the two chain points $p_1(\chi)$ and $p_2(\chi)$, that define the portion of χ lying inside the $\mathrm{VV}^{(c)}$-diagram (drawn in a solid line) move along the arcs of the chain χ toward its end-vertices (not shown in this figure).



(a) (b) (c)

Figure 8.8: A tangency event: (a) The chain point $p_2(\chi)$, whose creation is depicted in Fig. 8.7, lies on the supporting circle of the dilated vertex u. (b) The visibility edge $\vec{uv}_{rr}$ becomes tangent to $B_c(u)$ exactly at $p_2(\chi)$, so a tangency event occurs. (c) The reincarnated visibility edge $p_2(\vec{\chi})v$ replaces $\vec{uv}_{rr}$ as c grows. Note that this edge is not tangent to $B_c(u)$ any more.

by $xp(\vec{\chi}_a)$, recompute the critical c-values of the visibility events of this edge with its neighbors (notice that the slope of $x\vec{p}(a)$ becomes a different function of c from now on) and modify the corresponding visibility events in $\mathcal{Q}$.

In case x is a dilated obstacle vertex, we may have another tangency event in the queue, associated with $\vec{xu}$, which was computed under the (false) assumption that the tangency point of the edge on x coincides with a chain point before the one on u does. In this case, we have to locate the tangency event in $\mathcal{Q}$ that is associated with $\vec{xu}$ and recompute the c-value associated with it.

Endpoint event: A chain point $p(\chi_a)$ reaches the endpoint z of the Voronoi arc a. We should

consider the following cases here:

- The endpoint z is incident only to two Voronoi arcs a and a' belonging to the same chain (i.e., $\chi_a = \chi_{a'}$). In this case the chain point $p(\chi_a)$ is transferred from a to a', and we only have to examine the adjacencies in $\mathcal{L}(p(\chi_{a'}))$ and modify the corresponding visibility events in the queue (as the slopes of these arcs become a different function of c from now on). We also have to handle the opposite edges, as we did in the tangency-event procedure. Moreover, if there are tangency events associated with the opposite edges we should modify them as well.

  As the chain point $p(\chi_a)$ now moves on the Voronoi arc a', we have to take care of tangency events that occur in the range of this new arc. Thus, if one of the polygon features associated with a' is a vertex u, we iterate over all edges incident to u and check whether each edge has a tangency event in the range of the new Voronoi arc a' — if so, we insert the appropriate tangency event into the event queue.[11] In case a' is a vertex–vertex arc, associated with two vertices u and v, we repeat this procedure for v as well.
- If z is a Voronoi vertex *and* a local maximum of the clearance function, there are multiple endpoint events associated with it. In non-degenerate cases, the edge lists of all chain points coinciding with z are already empty. Only in degenerate cases may chain points involved in an endpoint-event at z still have incident edges, and in this case we just assign a maximal validity value to these edges and empty the edge lists associated with these chain points.
- Otherwise, z is the endpoint of the chain χ_a (i.e., a Voronoi vertex) and it is *not* a local maximum of the clearance function. In this case we may have several chains $\chi_1, \chi_2, \ldots$ ending at z, having a simultaneous endpoint event, and a single monotone chain $\hat{\chi}$ beginning at z (see for example the left Voronoi vertex of the marked chain in Fig. 8.2). We therefore create a new chain point $p(\hat{\chi})$ associated with the monotone chain, assign a maximal validity value c' to each edge in $\mathcal{L}(p(\chi_1)), \mathcal{L}(p(\chi_2)), \ldots$, where c' is the clearance value at z. We remove all visibility events associated with these edges from $\mathcal{Q}$ and insert their reincarnates into $\mathcal{L}(p(\hat{\chi}))$. We examine all adjacencies in $\mathcal{L}(p(\hat{\chi}))$ and add the appropriate visibility event to $\mathcal{Q}$. We also have to deal with the opposite edges and modify any tangency events they are involved in.

We note that in order to avoid duplicate work, when we have several events occurring at the same c-value, we deal with endpoint events first, to make sure that edges are associated with the correct chain. We can then handle the visibility events, chain events and finally the tangency events.

8.4.2 Querying the VV-Complex

The result of the preprocessing stage is the VV-complex $\langle \mathcal{V}, \mathcal{T} \rangle$, where:

[11] Note that edges that had a tangency event in the range of the previous Voronoi arc a have already been deleted from the incident-edge list of the vertex at the moment this endpoint event occurs.

- $\mathcal{V}$ is the Voronoi diagram of the polygonal obstacles. We also store the clearance value $c(z)$ of each vertex z in the Voronoi diagram, and for each non-monotone chain χ we store its minimal clearance value $c_{\min}(\chi)$.

- $\mathcal{T}$ is a set of interval trees: For each obstacle vertex u, $\mathcal{T}_u \in \mathcal{T}$ contains, for each edge incident to u, its validity range (namely the intervals are the valid c-ranges of the edges incident to u). For each Voronoi chain χ, $\mathcal{T}_{\chi,i} \in \mathcal{T}$ is an interval tree storing edges and incident Voronoi arcs incident to the ith chain point ($i \in \{1,2\}$) of the chain χ, along with their validity ranges.

A query on the VV-complex is defined by a triple $\langle s, g, \hat{c} \rangle$, where s and g are the start and goal configurations, respectively, and $\hat{c}$ is the preferred clearance value (one could apply standard techniques for testing if s and g have sufficient clearance). We assume that s and g themselves have a clearance larger than $\hat{c}$. Given a query, we start by computing the relevant portion of the Voronoi diagram: For each Voronoi chain we can examine the clearance values of its end-vertices, as well as the chain minimum, and determine which portion of the chain (if at all) we should consider. This way we also obtain all the chain points for the given c-value $\hat{c}$.

Next we need to find the incident edges of s and g. This means that we should obtain two lists $\mathcal{L}(s)$ and $\mathcal{L}(g)$ containing the visibility edges emanating from s and g (respectively) to every visible circular arc and chain point (or to original obstacle vertices if $c = 0$). This can be done using a radial sweep-line algorithm. We can now start searching the implicitly constructed $\mathrm{VV}^{(\hat{c})}$-diagram using a Dijkstra-like search to find the "shortest" path between s and g.

When we reach a vertex x (a dilated polygon vertex or a chain point) during the Dijkstra search we query $\mathcal{T}_x$ with the given c-value $\hat{c}$ to obtain the valid edges incident to x, as we do not have an explicit representation of the graph. In addition, we add g to the list of x's neighbors if $x \in \mathcal{L}(g)$ (that is, if the goal is visible from x). If x is an obstacle vertex, we should keep in mind to add the length of the portion of the corresponding circular arc to the distance.[12] We proceed until the goal configuration g is reached.

The way we select the weights associated with the graph edges may depend on the path-planning strategy we employ. All visibility edges (and portions of the circular arcs which need to be traversed) have a clearance of at least $\hat{c}$, so their distance measure depends only on their length. For the portions of the Voronoi diagram, the limited amount of clearance may add extra weight (see the discussion in Section 8.3 about the weight we give the graph edges). Since the graph edges are implicitly represented, we have to dynamically compute their associated weights, but this can be done in $O(1)$ time per edge and does not incur a significant computational load.

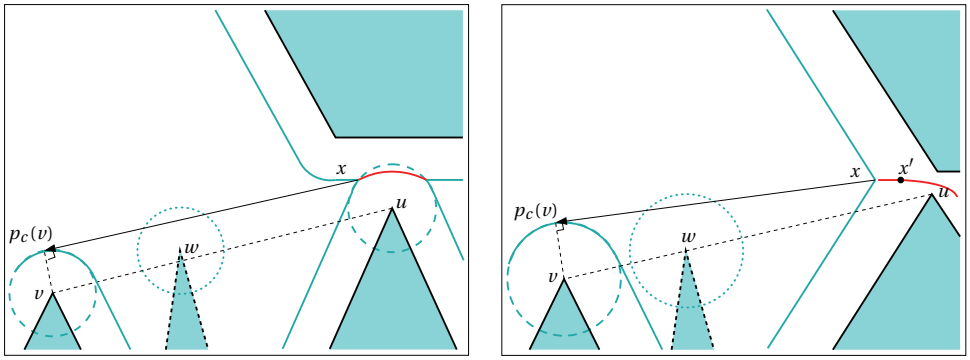


Figure 8.9: Visible dilated vertices from a chain point x. (a) If a vertex v is visible from a vertex–edge Voronoi arc, it must be also visible from the vertex u inducing this arc, and cannot be blocked by the dashed polygon. (b) In case of an edge–edge Voronoi arc, we consider the vertex u, which lies closest to the arc endpoint with the minimum clearance x', as the "inducing vertex".

8.4.3 Proof of Correctness

We begin by stating a lemma that proves that the manner in which we move visibility events from dilated vertices to chain points when handling tangency events is indeed correct — i.e., that a chain point cannot start "seeing" an object (a dilated vertex or another chain point) all of a sudden, unless this object is visible from one of the vertices inducing the Voronoi arcs along the chain.

Lemma 8.1. *If a dilated obstacle vertex $B_c(v)$ is visible from a chain point on a Voronoi arc, then the original vertex v is visible from the vertices inducing this arc. In case of an edge–edge Voronoi arc, we consider the arc endpoint with the minimum clearance value, and refer to the obstacle vertex that lies closest to this point as the "inducing vertex".*

Proof: Consider the example depicted in Fig. 8.9(a), where the dilated vertex $B_c(v)$ is visible from the chain point x, which lies on a vertex–edge Voronoi arc. Let u be the obstacle vertex inducing this arc. Assume u and v are not mutually visible, then there must exist some polygon blocking the straight line segment uv — let w be an extreme vertex of this polygon. Let $p_c(v)$ be the tangency point of the visibility edge emanating from x toward $B_c(v)$. It is clear that the distance of v from the line supporting $(x, p_c(v))$ is exactly c, but the distance of u from this line is *less* than c, as it cannot be tangent to $B_c(u)$ and penetrates the interior of this circle. We conclude that the distance of w from this line must also be less than c, thus

[12] In some cases we will have fictitious visibility edges of length 0, for example when we have a chain point y that lies on a vertex–vertex or a vertex–edge Voronoi edge (see Fig. 8.8(a) for an illustration). In this case, y is connected to the polygon vertices that induce this Voronoi edge with visibility edges of distance 0, and when we examine a path through the relevant Voronoi edge and involving a visibility edge incident to one of the vertices inducing y, we should only consider the length of the circular arcs between y and the endpoint of the visibility edge.

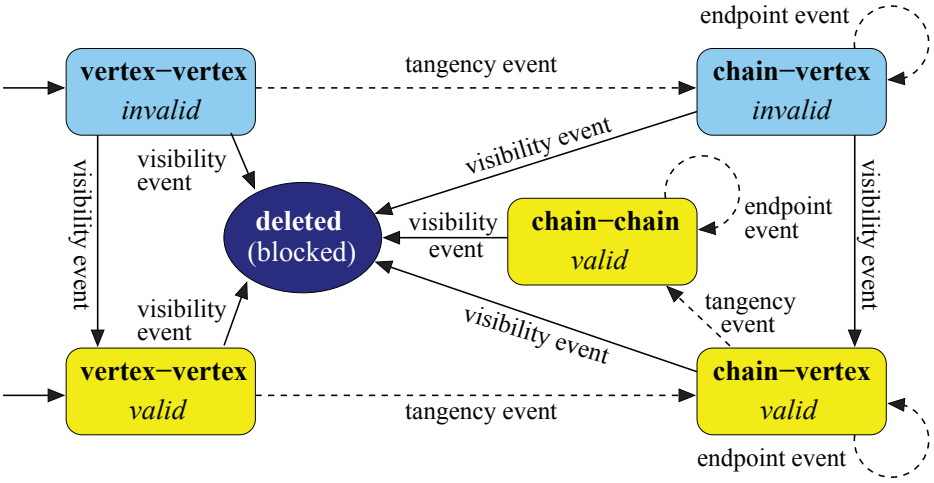


Figure 8.10: The schematic "life-cycle" of a visibility edge during the execution of the preprocessing stage. The rounded-corner rectangles denote possible visibility edges by the type of their endpoints. The solid arrows denote a change in the validity of the edge while the dashed arrows denote a reincarnation of the edge. For $c = 0$, all visibility edges, valid or invalid, are incident to two vertices (represented by the two boxes on the left). As c grows and parts of the Voronoi diagram are included in the $\mathrm{VV}^{(c)}$-diagram, an endpoint of such an edge may become incident to a Voronoi chain — namely a tangency event occurs and the edge is reincarnated as a vertex–chain edge (and later on as a chain–chain edge). Such edges are affected by endpoint events that occur along the Voronoi chain, but their validity status remains unchanged. Visibility events can turn invalid edge to valid ones, or to block visibility edges. In the latter case, the blocked edge is deleted.

$B_c(w)$ blocks the visibility of $B_c(v)$ from the chain point x. We have reached a contradiction, so we conclude that the original vertices u and v are mutually visible.

The same arguments hold for a chain point located on a vertex–vertex Voronoi arc, and we conclude that v is visible from *both* vertices inducing the arc. The case of a chain point which lies on an edge–edge Voronoi arc is depicted in Fig. 8.9(b). Note that in this case the two dilated polygon edges incident to x define the portion of the plane it can "see". Once again, assume w blocks the visibility edge of u and v (recall that u is the vertex inducing the Voronoi arc). Since the distance of u from the supporting line of $(x, p_c(v))$ must be less than c (notice that also in this case this line intersects the interior of $B_c(u)$), the distance of w from this line is also less than c. Again, we have reached a contradiction, as $B_c(w)$ blocks the segment $(x, p_c(v))$. ∎

Theorem 8.2. *Every visibility edge has only one continuous range* $[c_{\min}, c_{\max}]$ *of* c*-values for which it is valid. Thus, once it has been deleted it will not become valid again for a higher* c*-value.*

Proof: Consider Fig. 8.10, which describes the schematic "life-cycle" of a visibility edge along

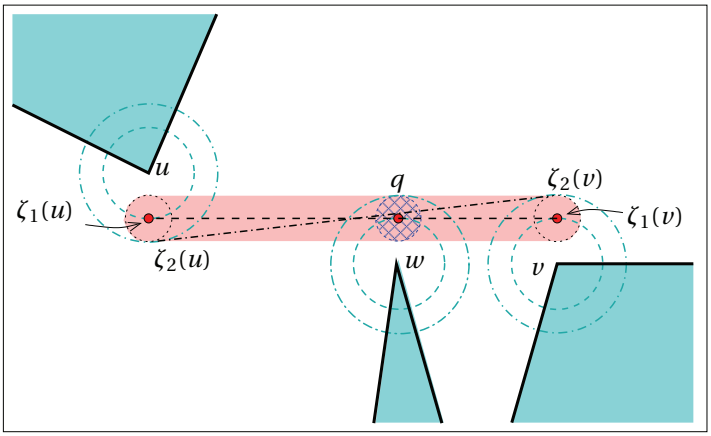


Figure 8.11: The visibility edges $\vec{uv}_{rl}$ and $\vec{vu}_{rl}$, realized as the segment $(\zeta_1(u),\zeta_1(v))$ (the dashed black line), are blocked at q by the dilated vertex $B_{c_1}(w)$. For $c_2 > c_1$, $(\zeta_2(u),\zeta_2(v))$ (the dash-dotted line segment) is contained in the region $(\zeta_1(u),\zeta_1(v)) \oplus B_{c_2-c_1}$ (lightly shaded), which is divided into two by the disc $B_{c_2-c_1}(q)$.

the preprocessing step described in Section 8.4.1. When we construct the VV-complex by gradually increasing the c-value, edges can only be deleted when a visibility event occurs and they become blocked by some dilated vertex: It is clear that just before a dilated vertex w starts blocking the visibility of x and y (x and y may be dilated vertices or chain points), it must lie on the line segment connecting x and y, so a visibility event must occur and no visibility edge can "disappear" as c grows without being involved in a visibility event. Note that an edge can also reincarnate as a different edge (see Fig. 8.8), but in this case we can treat the validity range of its reincarnate as a direct continuation of the range of the original edge.[13] Here we show that once an edge becomes blocked, it does not become unblocked again for a higher c-value.

Consider a visibility edge $\vec{uv}$ (it may either be invalid or valid) tangent to the supporting circles of the dilated vertices u and v for some clearance value $c_1 > 0$. Let $\zeta_1(u)$ and $\zeta_1(v)$ be the two endpoints of this edge, lying on $B_{c_1}(u)$ and $B_{c_1}(v)$, respectively. As illustrated in Fig. 8.11, for a clearance value $c_2 > c_1$, the edge $(\zeta_2(u),\zeta_2(v))$ between u and v for clearance c_2 is contained in the Minkowski sum $(\zeta_1(u),\zeta_1(v)) \oplus B_{c_2-c_1}$, as the distance of both $\zeta_2(u)$ and $\zeta_2(v)$ from the line segment $(\zeta_1(u),\zeta_1(v))$ is clearly less than $c_2 - c_1$.

Let us assume that for the clearance value c_1 the visibility edge $\vec{uv}$ becomes blocked by a dilated obstacle vertex w, which touches $(\zeta_1(u),\zeta_1(v))$ at some point q — then for each $c_2 > c_1$ the disc $B_{c_2-c_1}(q)$ of radius $c_2 - c_1$ centered at q is fully contained in a dilated obstacle, and no visibility edges can cross it. It is clear that this disc divides the region $(\zeta_1(u),\zeta_1(v)) \oplus B_{c_2-c_1}$

[13] When presenting the algorithm we created a new validity range for reincarnated visibility edges instead of treating the validity ranges as a single continuum, as we do in this theorem. This representation simplifies the algorithm without incurring any asymptotic run-time penalty. Our theorem is therefore slightly stronger than what we need for proving the correctness of our algorithm.

into two, making it impossible for the edge $(\zeta_2(u), \zeta_2(v))$ to be valid.

It is therefore clear that once a visibility edge between two dilated vertices becomes blocked, it can never become unblocked again.[14] Moreover, similar arguments apply if one of the endpoints of the visibility edge (or both its endpoints) is a chain point lying on a Voronoi arc. We begin by showing that the chain point for the clearance value c_2 lies inside the cigar-shaped region obtained by taking the Minkowski sum of the original visibility edge with $B_{c_2-c_1}$:

- The endpoint ζ_1 of a visibility edge for a clearance value c_1 lies on a vertex–vertex Voronoi arc (see Fig. 8.12(a) for an illustration). Without loss of generality, let us assume that the two vertices u and v inducing this Voronoi arc are located at $(0,-\delta)$ and $(0,\delta)$, where $2\delta < c_1$ is the distance between the vertices. In this case the Voronoi arc is supported by the line $y = 0$ and the two chain points for c_i $(i = 1,2)$ are given by $\zeta_i = (\sqrt{c_i^2 - \delta^2}, 0)$.

  Let us consider the extremal case where the visibility edge is tangent to $B_{c_1}(0,\delta)$ — that is, it is tangent to one of the dilated obstacles and if its slope is increased by $\varepsilon > 0$ it will penetrate this dilated obstacle and become blocked. In this case, the lower part of the "cigar" intersects $y = 0$ at $\tilde{\zeta}$, where:

$$x_{\tilde{\zeta}} = x_{\zeta_1} + \frac{c_2 - c_1}{\sin\theta} = \sqrt{c_1^2 - \delta^2} + \frac{c_1(c_2 - c_1)}{\sqrt{c_1^2 - \delta^2}} = \frac{c_1 c_2 - \delta^2}{\sqrt{c_1^2 - \delta^2}}$$

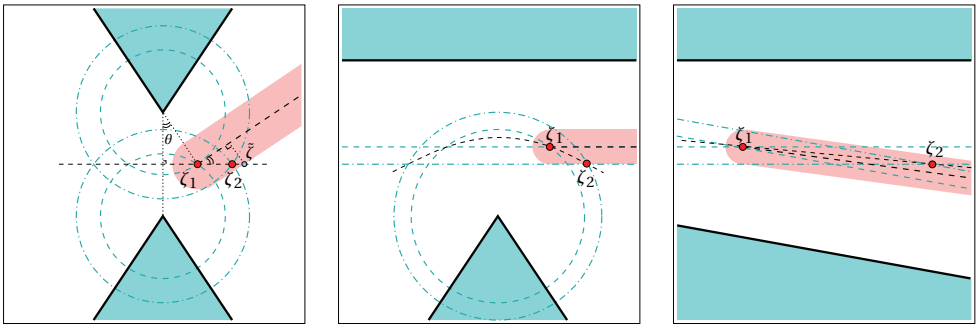
  It is straightforward to show that $x_{\tilde{\zeta}} > x_{\zeta_2}$, hence ζ_2 is contained in the "cigar".

- The endpoint ζ_1 lies on a vertex–edge Voronoi arc. Without loss of generality, we assume that the obstacle edge inducing the arc is supported by the line $y = \delta$ and the obstacle vertex is given by $(0,-\delta)$ (again, we have $2\delta < c_1$). It is clear that the slope of a visibility edge emanating from ζ_1 is non-positive. In the extremal case, depicted in Fig. 8.12(b), it is a horizontal segment, and since $|y_{\zeta_2} - y_{\zeta_1}| = c_2 - c_1$ then ζ_2 is located on the boundary of the cigar-shaped region around the horizontal visibility edge. It is also clear that in other cases, when the slope of the original visibility is negative, then ζ_2 is located in the interior of the "cigar" around this edge.

- The same arguments also apply if ζ_1 and ζ_2 lie on an edge–edge Voronoi arc. Note that in this case we should consider the slopes of both obstacle edge involved: indeed, $\|\zeta_1 - \zeta_2\|$ may be significantly larger that $c_2 - c_1$, as shown in Fig. 8.12(c), but since the slope of the visibility edge is bounded by the slope of the obstacle edges, it follows that ζ_2 must be contained in the "cigar".

We have showed that a visibility edge $\bar{e}_2$ for c_2 is always contained in the cigar-shaped region, which is the Minkowski sum of the visibility edge $\bar{e}_1$ for $c_1 < c_2$ with $B_{c_2-c_1}$. According

[14] In this case, there is also a simple algebraic proof for this fact: The bitangent to $B_{c_1}(u)$ and $B_{c_1}(v)$ is also tangent to $B_{c_1}(w)$ only when c_1 equals half the distance between u and the line connecting v and w. For the edge to become unblocked at some $c_2 > c_1$, the three circles $B_{c_2}(u)$, $B_{c_2}(v)$ and $B_{c_2}(w)$ must have another common tangent, but this is of course impossible.



(a) (b) (c)

Figure 8.12: The chain points ζ_1 and ζ_2, at clearance values c_1 and c_2, respectively ($c_2 > c_1$). The relevant Voronoi arcs are drawn as thin dashed lines, were the light dashed (dash-dotted) segments and circles correspond to clearance c_1 (c_2, respectively) from the obstacle features inducing these arcs. The visibility edges emanating from ζ_1 are drawn in a thick dashed line, with the Minkowski sum of the edge with $B_{c_2-c_1}$ is lightly shaded. (a) An extreme case where the visibility edge from a chain point lying on a vertex–vertex arc is tangent to one of the dilated obstacles. (b) Another extreme case where the visibility edge from a chain point lying on a vertex–edge arc is parallel to the edge. (c) The case of chain points lying on an edge–edge arc.

to our assumption, $\bar{e}_1$ is blocked at some point q, so $\bar{e}_1 \oplus B_{c_2-c_1}$ is divided into two by the disc $B_{c_2-c_1}(q)$. We argue that each part contains exactly one endpoint of $\bar{e}_2$, which can be easily verified by examining the various cases in Fig. 8.12. If this is not the case, then q must lie between ζ_1 and the projection of ζ_2 onto $\bar{e}_1$ — this is of course impossible, as it implies that there exists another obstacle on the way, other than the ones defining the Voronoi arc. As a consequence, the visibility edge $\bar{e}_2$ must also be blocked.

We conclude that once a visibility edge has been blocked, it will never become valid again. Note that what we have shown so far is that we can associate a single validity range with a visibility edge one of whose endpoints lie on a Voronoi *arc*, while our edges are actually associated with chain points that move along Voronoi *chains*. However, when a chain point is created, there are no visibility edges associated with it. By Lemma 8.1, visibility edges can be associated with a chain point only when it is involved in tangency events, as it traverses a vertex–vertex or a vertex–edge Voronoi arc, and it cannot "see" any object not visible from the relevant vertex. As the chain point moves along the chain, these visibility edges are eventually blocked (the chain point can never move from an edge–edge arc to another edge–edge arc, as there should always be a vertex on the way). We conclude that the association of a single validity range with each visibility edge (and with its reincarnates) is indeed correct. ■

8.4.4 Complexity Analysis

Theorem 8.3. *Constructing the* VV*-complex takes* $O(n^2 \log n)$ *in total, where* n *is the total number of obstacle vertices.*

Proof: In the initialization of the preprocessing stage we first have to compute the visibility graph, which can be performed in $O(n^2 \log n)$ time — this also accounts for the time needed to construct the initial edge lists $\mathcal{L}(u)$ for each obstacle vertex u (we need $O(n \log n)$ time to construct each of the $2n$ edge lists) and label the valid visibility edges. The construction of the Voronoi diagram can be performed in $O(n \log n)$, and the complexity of the diagram (the number of arcs) is linear in n.

After the initialization, the priority queue $\mathcal{Q}$ contains $O(1)$ events per visibility edge, of which there are $O(n^2)$ in total, and in addition $O(n)$ chain events. Any operation on the event queue thus takes $O(\log n)$. The initialization takes $O(n^2 \log n)$ time in total.

As the preprocessing algorithm proceeds, it starts handling events: In total, Theorem 8.2 implies that we have $O(n^2)$ visibility events:[15] Every vertex can be involved at most once in a visibility event with another vertex, where a visibility edge between the two vertices (or their dilated version) is created. Each of the visibility events can be handled in $O(\log n)$ time as it involves a constant number of operations on the queue and on the edge lists. There are $O(n)$ chain events, each of them can be handled in $O(n \log n)$ time. Each chain event spawns $O(n)$ tangency events, so in total there are $O(n^2)$ tangency events, each of them can be handled in $O(\log n)$ time. Finally, there are $O(n)$ endpoint events, and we need $O(n \log n)$ time to handle each of these events.[16] ∎

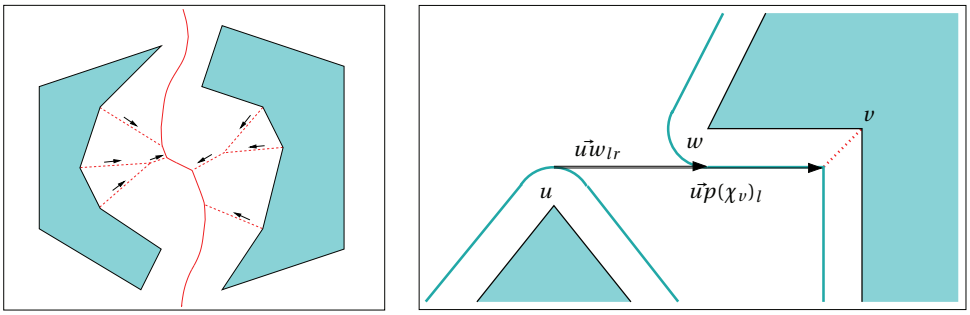
The query phase starts with a stage that takes $O(n \log n)$ time, which is spent on calculating the valid visibility edges emanating from s and g. Calculating the relevant portions of the Voronoi diagram takes $O(n)$ time (note that the Voronoi diagram itself has already been constructed in the preprocessing phase).

The rest of the query phase consists of executing Dijkstra's algorithm, or an equally suited A*-algorithm. The worst-case running-time of these algorithms is $O(n \log n + \ell)$ where $\ell = O(k)$ is the number of edges encountered during the search (recall that k is the number of visibility edges). In practice, Dijkstra's algorithm turns out to be very fast, because hardly any geometric operations have to be performed anymore. In particular the A*-variant of Dijkstra may be the method of choice here, as it biases the search toward the goal configuration, which keeps the number ℓ low.

As we noted in Section 8.3, the VV$^{(c)}$-diagram for a fixed c-value may be constructed in $O(n \log n + k)$ time, so it may seem we do not need any preprocessing stage, and it is better to construct the VV$^{(c)}$-diagram from scratch whenever we are given a preferred clearance value. However, this algorithm involves the construction of the planar arrangement

[15] We consider all potential events in our analysis. In practice, some of these events were computed under false assumptions (see Section 8.4.1) and will be eventually discarded.

[16] It is in fact possible to construct the visibility graph of the input polygons in $O(n \log n + k)$ time, where k is the number of visibility edges in this graph (valid and invalid ones), construct the initial edge lists in $O(k \log n)$ time and then charge each of the $O(k)$ directed visibility edges with $O(\log n)$ operations, to account for all visibility events, chain events and tangency events. Unfortunately, the entire preprocessing stage cannot be completed in $O(k \log n)$ time even if $k = o(n^2)$, as there are cases where $\Theta(n^2 \log n)$ operations are needed to handle the endpoint events.



(a) (b)

Figure 8.13: (a) A portion of the Voronoi diagram of two non-convex polygons. The Voronoi chain separating the two obstacles is drawn with a solid line, while the Voronoi chains induced by features of the same polygon are drawn with a dashed line. (b) The edge $\vec{uw}_{lr}$ becomes valid after being involved in a visibility event with a visibility edge to the chain point $p(\chi_v)$ that is associated with the reflex obstacle vertex v.

of line segments, circular arcs and parabolic arcs, which is very complicated when carried out in a *robust* manner (see the next section). Such an approach will require longer running times than the query stage of the second algorithm. We note that Dijkstra's algorithm, whose running time theoretically dominates the query phase, is in practice very fast if after preprocessing our set of input obstacles in an exact manner, we switch to machine-precision floating-point arithmetic in the query stage.[17]

8.4.5 Handling Non-Convex Obstacles

So far we described the algorithm for constructing a VV-complex for a set of convex polygonal obstacles. Our algorithm can however be easily adapted to work with non-convex obstacles as well. The only thing that is changed is the way in which the Voronoi diagram is constructed.

Due to the non-convexity of the obstacles, some obstacles may contain reflex vertices. These reflex vertices are treated as normal vertices in the initial construction (for $c = 0$) of the visibility graph. Note that the visibility edges emanating from reflex vertices will never be part of a shortest path, but we still need to keep track of these edges, as they may induce visibility events that give other valid edges the correct c-values of their validity ranges (see Fig. 8.13(b) for an illustration).

As c grows, the reflex vertices will be treated as chain points. These chain points move over monotone Voronoi chains originating in the reflex vertices themselves (see Fig. 8.13(a)). To this end, the definition of the Voronoi diagram should be adapted such that Voronoi arcs can be equidistant to two edges of the same polygon as well. Still, this new Voronoi dia-

[17] Indeed, we lose some accuracy here, but as our constructed diagram is topologically correct, the worst thing that can happen is that we may compute a path that is only slightly longer than the shortest possible path.

gram is an instance of the Voronoi diagram of line segments, so this change is easily carried through.

The rest of the algorithm remains unchanged. Also, the complexity analysis is still valid, since the construction time and the complexity of both the visibility graph and the Voronoi diagram are not affected by the non-convexity of the input obstacles. We should mention that when we query the VV-complex we do not compute the chain points along Voronoi chains induced by reflex vertices, and therefore do not account for these "reflex" chains, as these chains lead to a dead-end (a reflex vertex) and can never be used for making shortcuts in the motion path.

8.5 Implementation Details

CGAL, the Computational Geometry Algorithms' Library [1], offers the infrastructure we need for developing a robust software for computing the $\mathrm{VV}^{(c)}$-diagram. We use the software components developed by Hirsch and Leiserowitz [72] for constructing the union of the Minkowski sums of the polygonal obstacles with a disc of radius c.

The Voronoi diagram of the polygons is computed using a recently implemented CGAL package by Karavelas [86] for computing Voronoi diagrams of line segments: We simply add a label to each segment (polygon edge) and each segment endpoint (polygon vertex) that identifies the source polygon and the feature index within this polygon. We then can conveniently disregard Voronoi chains induced by features of the same polygon.[18] The Minkowski-sum computation is carried out by decomposing each non-convex obstacle to convex polygons, dilating them by the preferred clearance value and computing the union of the sums, so it is very convenient to compute the diagram of these convex sub-polygons as well, instead of using the non-convex input obstacle. In this case we label each polygon feature with both the convex polygon and the input (non-convex) polygonal obstacle from which it originated. This labeling helps us to determine which Voronoi arcs should be ignored.

The intersection among the dilated obstacles and between the boundary of the union of the dilated obstacles and the Voronoi arcs is robustly computed using the conic-arc traits [155] of CGAL's arrangement package [50, 161]. We exploit the fact that our polygonal obstacles are given as sequences of points with *rational* coordinates, so that the supporting curves of each dilated obstacle boundary and each Voronoi arc can be represented as algebraic curves of degree 2 with rational coefficients if the squared clearance value is also rational (see below), to robustly maintain the arrangement of such curves. The endpoints of the line segments, the circular arcs and the parabolic arcs that form our arrangement are in general algebraic numbers of degree 4.

In the rest of this section we give a constructive proof of a lemma that enables us to robustly construct the skeleton of the $\mathrm{VV}^{(c)}$-diagram for rational inputs, based on robust computations with the conic-arc arrangement traits:

Lemma 8.4. *Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of pairwise interior-disjoint simple polygons, such*

[18] The complete Voronoi diagram of the polygon edges contains also the *medial axis* of each polygon, which is redundant in our case.

that all polygon vertices have rational *coordinates. Then all Voronoi arcs have supporting algebraic curves of degree 2 at most with rational coefficients and all chain minima are also points with rational coordinates. Moreover, for a clearance value c such that c^2 is rational, the dilated obstacle boundaries are also supported by algebraic curves of degree 2 with rational coefficients.*

8.5.1 Voronoi Arcs

An arc a of the Voronoi diagram corresponds to the locus of all points equidistant from two polygon features, and the following cases are possible:

Vertex–vertex arc: The arc is equidistant from two polygon vertices u and v. The equation of its supporting curve, a line in this case, is simply given by (throughout this section we use the squared distance, in order to avoid the square-root operation):

$$\begin{aligned} (x-x_u)^2+(y-y_u)^2 &= (x-x_v)^2+(y-y_v)^2 \\ 2(x_v-x_u)x+2(y_v-y_u)y &= x_v^2+y_v^2-(x_u^2+y_u^2)\,. \end{aligned} \tag{8.1}$$

This line is perpendicular to the line segment connecting u an v and bisects it. The point with minimal clearance on the arc is therefore the midpoint between u and v, $z_{\min}=\frac{1}{2}(x_u+x_v, y_u+y_v)$, and its clearance is of course $c_{\min}=\frac{1}{2}d(u,v)$.

Vertex–edge arc: The arc is equidistant from a polygon vertex u and a polygon edge vw, whose supporting line will be denoted $\ell:\ Ax+By+C=0$, where A, B and C are rational (since the vertices have rational coordinates). The equation of its supporting curve, a parabola in this case, is thus given by:

$$\frac{(Ax+By+C)^2}{A^2+B^2}=(x-x_u)^2+(y-y_u)^2\,. \tag{8.2}$$

In this case, to find the point with minimal clearance on the arc we compute a line perpendicular to ℓ that passes through u. The equation of this line is $\ell^\perp:\ By-Ax+(Ay_u-Bx_u)=0$, and the point with minimal clearance is the midpoint between u and the intersection point of ℓ and $\ell^\perp$:

$$z_{\min}=\frac{1}{2}\left(x_u+\frac{B^2x_u-A(By_u+C)}{A^2+B^2}, y_u+\frac{A^2y_u-B(Ax_u+C)}{A^2+B^2}\right)\,. \tag{8.3}$$

The minimal clearance value, obtained at $z_{\min}$ is half the distance between u and the line ℓ.

Edge–edge arc: The arc is equidistant from two polygon edges, whose supporting lines are denoted $\ell_1:\ A_1x+B_1y+C_1=0$ and $\ell_2:\ A_2x+B_2y+C_2=0$, respectively. The supporting curve of this edge is a line bisecting the angle formed between ℓ_1 and ℓ_2, but in general this line cannot be represented as a linear curve with rational coefficients.[19] Instead,

[19] For example, if $\ell_1: y=0$ and $\ell_2:\ y=x$, the slope of the line bisecting the angle between ℓ_1 and ℓ_2 is $\tan 22.5^\circ=\frac{1}{1+\sqrt{2}}$, and this line ($y=\frac{1}{1+\sqrt{2}}x$) cannot be represented using rational coefficients. Note however that the perpendicular line $y=\frac{1}{1-\sqrt{2}}x$ is also an angle bisector in this case, and angle bisector therefore consists of a *pair* of perpendicular lines.

we represent the edge as a segment of a pair of perpendicular lines (naturally, only one line in this pair supports the relevant segment), which form the two angle bisectors of ℓ_1 and ℓ_2:

$$\frac{(A_1x + B_1y + C_1)^2}{A_1^2 + B_1^2} = \frac{(A_2x + B_2y + C_2)^2}{A_2^2 + B_2^2} \ . \tag{8.4}$$

Using this representation, it is possible to represent the Voronoi arc as a segment of a curve of degree 2 with rational coefficients. As we mentioned before, such an arc is always monotone — that is, as we traverse it from the endpoint with smaller clearance value to the other endpoint, we get further away from the obstacles.

8.5.2 Dilated Obstacle Boundaries

Dilated vertex: Each convex polygon vertex u induces a circular arc, which is a segment of the circle $B_c(u)$, given by the equation:

$$(x - x_u)^2 + (y - y_u)^2 = c^2 \ . \tag{8.5}$$

Since x_u, y_u and c^2 are all rational, $B_c(u)$ has rational coefficients.

Dilated edge: The edges of the dilated obstacles are formed by offsetting the polygon edges parallel to themselves. However, in general it is impossible to represent a dilated edge as a linear curve with rational coefficients.[20] Instead, we treat it as a segment of a pair of parallel lines, representing the locus of all points whose distance from the line $\ell: \ Ax + By + C = 0$ supporting the original polygon edge equals c:

$$\frac{(Ax + By + C)^2}{A^2 + B^2} = c^2 \ . \tag{8.6}$$

The two endpoints of the segment lie of course on one of the two lines given by the equation above, and not on the other.

8.6 Experimental Results

Our software is implemented using CGAL 3.1, relying on the exact number types supplied by CORE 1.7 [2]. In particular, the `CORE::Expr` number-type class is capable of performing exact computations with polynomial roots. As we wish to obtain an exact representation of the $\mathrm{VV}^{(c)}$-diagram, we may spend some time on the diagram construction, especially if it contains chain points, which are algebraically more difficult to handle. For example, the construction of the $\mathrm{VV}^{(c)}$-diagram depicted in Fig. 8.3 (the `four_shapes` scene) takes about 10 seconds (running on a Pentium IV 2 GHz machine with 512 MB of RAM), but if we choose a smaller clearance value for the same scene, such that no chain points appear in the diagram,

[20] For example, if we seek a line lying at a distance 1 from $\ell: \ y = x$, we find the line $y = x + \sqrt{2}$, that cannot be represented using rational coefficients. However, the line $y = x - \sqrt{2}$ is also parallel to ℓ and lies at a distance 1 from it, so the locus of points at a given distance from ℓ is really comprised of a *pair* of parallel lines.

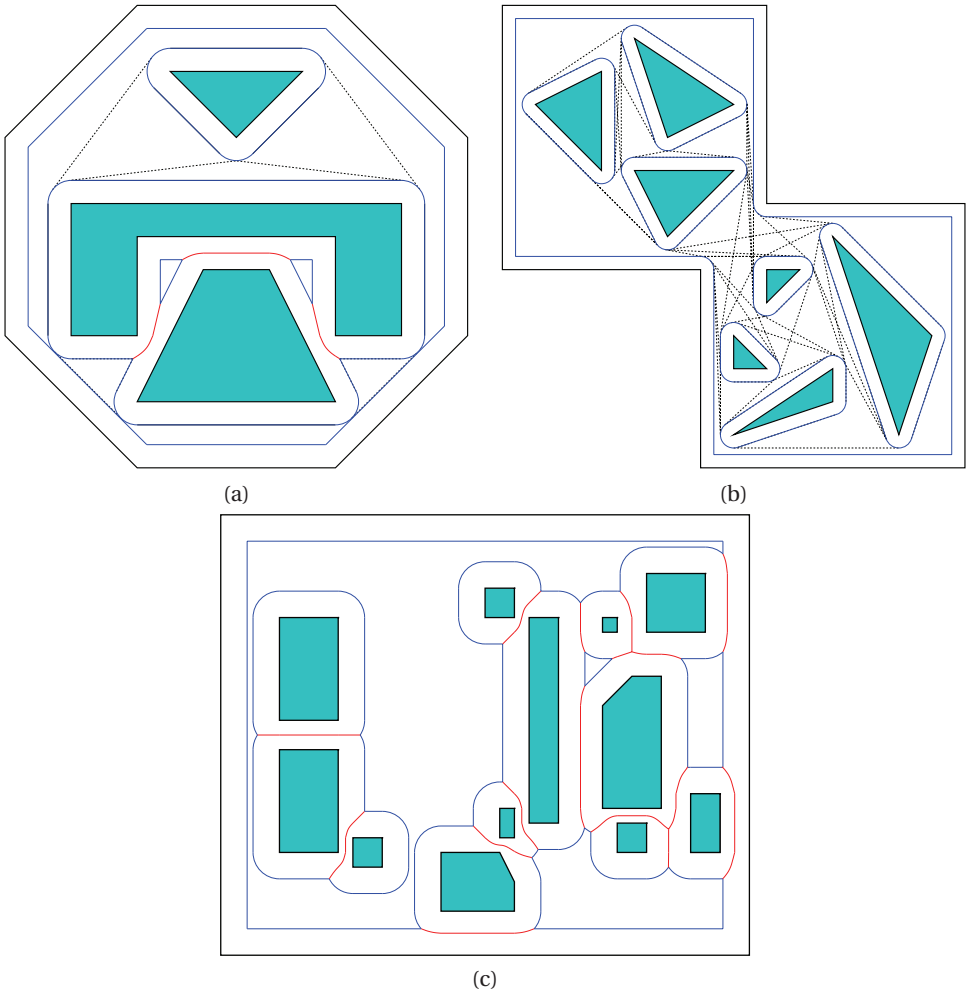


Figure 8.14: The $VV^{(c)}$-diagrams constructed for several input files and c-values: (a) `octagon` with $c = \frac{7}{10}$, (b) `two_rooms` with $c = \frac{2}{5}$, and (c) `rectangles` with $c = \frac{9}{10}$ (visibility edges are not shown in this case).

the construction time drops to 2.3 seconds (see Table 8.1). In more involved scenes, the construction of the diagram may take 15–20 seconds (see Fig. 8.14 and Table 8.1).

However, once the $VV^{(c)}$-diagram is constructed, it is possible to use a floating-point approximation of the edge lengths to speed up the time needed for answering path planning queries, so that the average query time is only a few milliseconds.

We also used the $VV^{(c)}$-diagram to generate convincing group motions in a more complex scene, as the one depicted in Fig. 8.15. The construction of such diagrams takes about 40–60

Table 8.1: The construction time of the $VV^{(c)}$-diagram for several input scenes and different c-values. The query times where averaged over five manually entered queries for each scene.

Input file	Bounding-box dimensions	c	Construction time (sec.)	Average query time (sec.)
four_shapes	10×7	$1/5$	2.3	0.01
four_shapes	10×7	$2/5$	9.7	0.01
octagon	14×14	$3/10$	4.9	0.01
octagon	14×14	$7/10$	15.2	0.01
two_rooms	14×14	$2/5$	2.8	0.02
rectangles	18×15	$9/10$	15.4	0.02

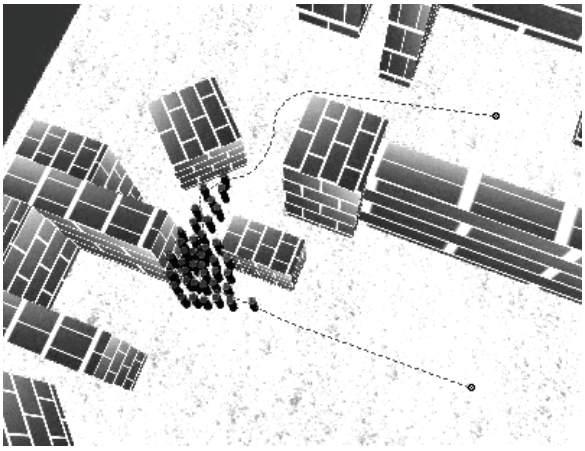


Figure 8.15: A group of 40 entities moving in a virtual scene along a backbone path, drawn with a dashed line. (Courtesy of Arno Kamphuis.)

seconds (for clearance values that induce chain points), but the average query time was only a few milliseconds. This is a considerable improvement over previous techniques, which require smoothing operations in the query stage, taking about one second on average.

8.7 Conclusion

We introduced a simple, yet powerful, data structure — the $VV^{(c)}$-diagram — which contains all backbone paths for a corridor in a planar environment of configuration-space obstacles, given a preferred clearance value and that allows for a trade-off between path length and clearance in the presence of narrow passages. We have implemented a robust software package that maintains this data structure and used it to plan natural-looking paths for co-

herent groups of moving entities in the plane. Our method, which requires some preprocessing for constructing the diagram but can answer queries very efficiently without the need for smoothing or additional post-processing, is especially suitable to real-time applications, such as computer games.

We have also introduced the VV-complex, a data structure that efficiently encodes all $\mathrm{VV}^{(c)}$-diagrams for all possible clearance values. We showed how to efficiently construct the VV-complex for a given set of obstacles and how to query it given start and goal configurations and a preferred clearance value.

The backbone paths contained in the VV-complex are natural, but they do not optimize a sensible mathematical quality criterion. In the next chapter we will define a quality criterion, and investigate properties of corridors that are optimal under this criterion.

Chapter 9

Planning Optimal Corridors

Planning corridors among obstacles has arisen as a new approach to path planning. Instead of devising a one-dimensional motion path for a moving entity, it is possible to let it move in a corridor, where the exact motion path is determined by a local planner. In this chapter we introduce a quantitative measure for the quality of such corridors. We analyze the structure of optimal corridors amidst point obstacles and polygonal obstacles in the plane, and propose an algorithm to compute approximations for optimal corridors according to our measure.

This chapter has previously been published as: R. Wein, J. van den Berg, and D. Halperin. Planning near-optimal corridors amidst obstacles. In *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006 [158].

9.1 Introduction

The task of planning a natural path for a moving entity that avoids obstacles plays an important role in robotics, as well as in game design. The problem is often solved by constructing a graph that discretizes the environment, and extracting a collision-free path from this graph. The nodes of such a graph may be the cells of a uniform grid (see, e.g., [133]), or — according to Probabilistic Roadmap (PRM) paradigm [41, 89] — free configurations that are randomly chosen, attempting to capture the connectivity of the free configuration space.

A common drawback of the above methods is that they output a fixed path in response to a query. This is often not the ideal solution for path planning, as it lacks flexibility to avoid local hazards (such as small obstacles, other moving entities, etc.) that are encountered during the motion. It also leads to predictable, and possibly unrealistic motions, which are not suitable for some applications, such as computer games. One approach for tackling these problems is a potential-field planner, in which the moving entity is attracted to its goal configuration, and repelled by obstacles, or other moving entities (see, e.g., [91]). However, this approach is prone to get stuck in local minima of the potential field; while there are methods that help in resolving such situations (see, e.g., [96]), they may still not yield valid

motions at all.

We would therefore like to indicate the global direction of movement for the moving entity, while leaving enough flexibility for some *local planner* to avoid local hazards. An ideal solution for this is to use *corridors*, which have recently been introduced in the game design field [121]. Corridors are defined as a union of balls whose center points lie along a backbone path. The radius of the balls is determined by the *clearance* (i.e., the distance to the nearest obstacle) along the backbone path. The more restricted task of locally planning the motion around the backbone path can be successfully performed by potential-field methods. In order to guarantee that the local planner operates on a restricted environment, the radii of the balls are upper bounded by some predetermined value.[1] As a result, rather than moving along a fixed path, the moving entity moves within a corridor around the backbone path. This gives a strict global direction of movement, yet provides the local flexibility we look for.

Planning within corridors has many applications. It has been used to plan motions for coherent groups of entities, where the backbone path provides the global motion of the group [83]. The interactions between entities of the group are locally controlled by a social potential-field method [130]. Corridors have also been used to plan the motion of a camera that follows a moving character (a *guide*) [114]. If the guide moves along the backbone path, the corridor gives the flexibility for the camera to swerve if necessary. Another advantage of corridors is that they allow for non-holonomic and kinodynamic planning, if the motion of a single entity (or multiple entities) is planned using a potential field method within the corridor [84]. This is very difficult to achieve and incorporate into a fixed path. A common property of the applications of corridors is that the moving entity is small compared to the scale of the environment. In many fields (open field robotic navigation, games, etc.) this is indeed the case.

The problem we consider in this chapter is how to plan a good corridor. A good corridor is short, avoiding unnecessary detours, and at the same time it should be wide (up to some prescribed maximum) to provide local maneuvering space. These requirements often contradict. Given start and goal configurations and a set of obstacles, the shortest collision-free path is contained in the *visibility graph* of the obstacles; see, e.g., [110]. However, such a path is incident to obstacle boundaries and cannot serve as a backbone path of a valid corridor. If one is only concerned with clearance, allowing paths that are as long as needed, then such paths are easily found using the *Voronoi diagram* of the given obstacles [119]. It is also possible to consider interpolates of these two structures, named *visibility–Voronoi diagrams*, as suggested in the previous chapter. Indeed, a good corridor makes a good trade-off between length and clearance.

In this chapter we introduce a measure for the quality of corridors, and present methods to plan corridors that are (nearly) optimal with respect to this measure amidst point obstacles or polygonal obstacles in the plane.

The rest of this chapter is organized as follows. In Section 9.2 we formally define corridors and introduce the quality measure. Section 9.3 discusses properties of optimal corridors amidst point obstacles in the plane, and in Section 9.4 we generalize our results to polygonal

[1] The fact that the radii of the balls are bounded is also a major difference between a corridor and the *medial axis transform* of the free workspace.

obstacles. We give some concluding remarks and future-work directions in Section 9.5.

9.2 Measuring Corridors

A *corridor* $C = \langle \gamma(t), w(t), w_{\max} \rangle$ in a d-dimensional workspace (typically $d = 2$ or $d = 3$) is defined as the union of a set of d-dimensional balls whose center points lie along the *backbone path* of the corridor, which is given by the continuous function $\gamma : [0, L] \longrightarrow \mathbb{R}^d$, where L is the length of γ. The radii of the balls along the backbone path are given by the function $w : [0, L] \longrightarrow (0, w_{\max}]$. Both γ and w are parameterized by the length of the backbone path. In the following, we will refer to $w(t)$ as the *width* of the corridor at point t. The width is positive at any point along the corridor, and does not exceed $w_{\max}$, a prescribed *desired width* of the corridor.

Given a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ of length L in $\mathbb{R}^d$, the interior of the corridor is thus defined by $\bigcup_{t \in [0,L]} B\big(\gamma(t); w(t)\big)$, where $B(p; r)$ is an open d-dimensional ball with radius r that is centered at p. In typical motion-planning applications we are given a set of obstacles $\mathcal{O}$ that the moving entities should avoid. The interior of the corridor should be disjoint from the interior of the given obstacles, otherwise it is an *invalid* corridor. In this chapter we study the problem of computing *valid* corridors amidst obstacles in the plane.

9.2.1 The Weighted Length Measure

As we have already indicated, a good corridor must be short — namely its backbone path should avoid unnecessarily long detours — and its width should be as wide as some predefined maximum in order to allow maximal flexibility for the motion within the corridor. The corridor should contain narrow passages only if they allow considerable shortcuts.

If we examine the intersection of the corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ with an orthogonal $(d-1)$-dimensional hyperplane at $\gamma(t)$, the volume of the cut is proportional to $w^{d-1}(t)$. Thus, in order to combine the two desired properties of the corridor as discussed above, we define the *weighted length* $L^*(C)$ of a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ to be:

$$L^*(C) = \int_\gamma \left(\frac{w_{\max}}{w(t)} \right)^{d-1} dt \,. \tag{9.1}$$

We wish to minimize the weighted length by either shortening the backbone path or by extending the corridor's width (up to $w_{\max}$). Given a start position $s \in \mathbb{R}^d$ and a goal position $g \in \mathbb{R}^d$, a corridor $C = \langle \gamma(t), w(t), w_{\max} \rangle$ satisfying $\gamma(0) = s$ and $\gamma(L) = g$ is *optimal* if for any other valid corridor C' connecting the two endpoints we have $L^*(C) \leq L^*(C')$.

Our weighting scheme can be directly applied for extracting backbone paths from PRMs that contain cycles [113, 115], where instead of considering the Euclidean length we try to minimize the weighted length of the backbone path we compute, in order to obtain a better corridor. However, for some sets of obstacles we can actually devise a complete scheme for computing an optimal corridor, as we show in the rest of this chapter.

9.2.2 Properties of an Optimal Corridor

Observation 9.1. *If for some portion of the backbone path γ of a corridor C, we have $w(t) < \min\{c(\gamma(t)), w_{\max}\}$ for $t \in [t_0, t_0 + \tau]$ ($\tau > 0$), where $c(p)$ is the* clearance *of the point p, namely its distance to the nearest obstacle, we can improve the quality of the corridor by letting $w(t) \longleftarrow \min\{c(\gamma(t)), w_{\max}\}$ for each $t \in [t_0, t_0 + \tau]$.*

Given a set of obstacles and a $w_{\max}$ value, we can associate the *bounded clearance* measure $\hat{c}(p)$ with each point $p \in \mathbb{R}^d$, where $\hat{c}(p) = \min\{c(p), w_{\max}\}$. Using the observation above, it is clear that the width function of an optimal path $C = \langle \gamma(t), w(t), w_{\max} \rangle$ is simply $w(t) = \hat{c}(\gamma(t))$. Note that $\hat{c}(\gamma(t))$ is a continuous function along any path γ.

Lemma 9.2. *Given a set of obstacles and $w_{\max}$, the backbone path of the optimal corridor connecting any given start position s with any goal position g is* smooth.

Proof: We have already observed that the width function of the optimal corridor connecting s and g is the bounded clearance function of the backbone path and it is a continuous function. Assume that γ contains a sharp turn (a $\mathcal{C}_1$-discontinuity). Let us shortcut the sharp turn using a circular arc of radius r (as r approaches 0 the approximation is tighter). Let ℓ_1 be the length of the original path segment we shortcut and let ℓ_2 be the length of the circular arc. It is easy to show that there exist $\hat{r} > 0$ and some constants $A_1 > A_2 > 0$ such that for each $0 < r < \hat{r}$ we have $\ell_1 \geq A_1 r$ and $\ell_2 = A_2 r$. If the maximal width w^* along the original path segment is obtained at some point p^*, then as the distance of any point p along the circular arc from p^* is bounded by Kr, where K is some constant, and as the weight function is continuous, we can write $w^* - w(p) < Mr$ for some positive constant M. Let L_1^* be the weighted length of the original path segment and let L_2^* be the weighted length of the circular arc. We can write:

$$\frac{L_1^*}{L_2^*} \geq \frac{\frac{w_{\max}}{w^*}\ell_1}{\frac{w_{\max}}{w^*-Mr}\ell_2} = \frac{w^* - Mr}{w^*} \cdot \frac{A_1}{A_2}.$$

As $A_1 > A_2$, we can choose $0 < r < \min\left\{\frac{w^*}{M}\left(1 - \frac{A_2}{A_1}\right), \hat{r}\right\}$ such that the entire expression above is greater than 1. We thus have $L_1^* > L_2^*$, and we managed to decrease the weighted length of the corridor, in contradiction to its optimality. We conclude that $\gamma(t)$ must be a smooth function. ∎

At several places in this chapter we apply infinitesimal analysis, where we assume that the bounded clearance measure (hence the width function) is not continuous. Assume that we have some hyperplane $\mathcal{H}$ in $\mathbb{R}^d$ that separates two regions, such that in one region the bounded clearance is w_1 and in the other it is w_2. Minimizing the weighted length between two endpoints that are separated by $\mathcal{H}$ is equivalent to applying Fermat's principle, stating that the actual path between two points taken by a beam of light is the one which is traversed in the least time. The optimal backbone thus crosses the separating hyperplane once, such that the angles α_1 and α_2 it forms with the normal to $\mathcal{H}$ obey Snell's Law of refraction,[2] with

[2] See, e.g., `http://scienceworld.wolfram.com/physics/SnellsLaw.html` for the details and for a detailed proof. See also Mitchell and Papadimitriou [111], who used this observation in a similar setting of the problem.

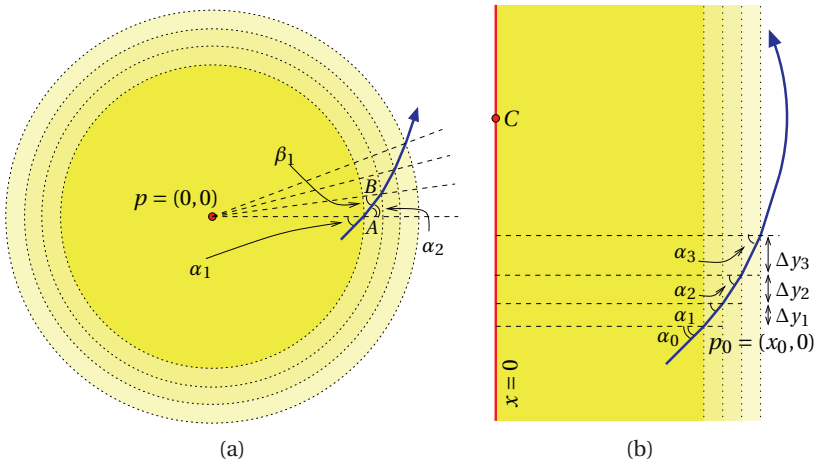


Figure 9.1: Analysis of the optimal backbone path in the vicinity of a single obstacle: (a) near a point obstacle $p = (0,0)$, (b) near a line segment supported by $x = 0$.

w_1 and w_2 playing the role of the "speed of light" in the respective regions:

$$w_2 \sin\alpha_1 = w_1 \sin\alpha_2 \,. \tag{9.2}$$

9.3 Optimal Corridors amidst Point Obstacles

In this section we consider planar environments cluttered with point obstacles $p_1, \ldots, p_n \in \mathbb{R}^2$ and a preferred corridor width $w_{\max}$. Given two endpoints $s, g \in \mathbb{R}^2$, we show how to compute a (near-)optimal corridor that connects s and g.

9.3.1 A Single Point Obstacle

Let us assume we have a single point obstacle p. Without loss of generality we assume p is located at the origin. We start with computing an optimal corridor between two endpoints whose distance from p is smaller than or equal to $w_{\max}$. Note that the width of such a corridor at $\gamma(t)$ along its backbone is $\|\gamma(t)\|$.

We first approximate the optimal backbone by a polyline: for any $\Delta r > 0$, if we look at the circles of radii $\Delta r, 2\Delta r, 3\Delta r, \ldots$ that are centered at the origin, each two neighboring circles define an annulus; since Δr is small we assume that the distance from p of all points in the kth annulus is constant and equals $k\Delta r$. Consider the scenario depicted in Fig. 9.1(a), where γ enters one of the annuli at some point A, where $\|A\| = r_1$, and leaves this annulus at B, where $\|B\| = r_2 = r_1 + \Delta r$. The angles that the backbone path forms with pA and pB are α_1 and β_1, respectively. When entering the annulus we have $w_1 = r_1$ and $w_2 = r_2$, so applying Equation (9.2) we can express the refracted angle α_2, using $\sin\alpha_2 = \frac{r_2}{r_1}\sin\alpha_1$. By applying the

Law of Sines on the triangle $\triangle pAB$, we get $\frac{r_2}{\sin(\pi-\alpha_2)} = \frac{r_1}{\sin\beta_1}$, therefore:

$$\sin\beta_1 = \frac{r_1}{r_2}\sin(\pi-\alpha_2) = \frac{r_1}{r_2}\sin\alpha_2 = \sin\alpha_1 .$$

Thus $\beta_1 = \alpha_1$. Taking $\Delta r \longrightarrow 0$, we obtain a smooth curve γ, such that the angle that $\nabla\gamma(t)$ forms with $\overrightarrow{p\gamma(t)}$ is a constant ψ. It is possible to show that a curve that has this property must be segment of a *logarithmic spiral* (also named an *equiangular spiral*)[3] whose polar equation is given by $r(t) = ae^{b\theta(t)}$, where a is a constant and $b = \cot\psi$. See, e.g., [68] for a proof of this latter fact.

Proposition 9.3. *Given a single point obstacle located at the origin, a start position $s = r_s e^{i\theta_s}$ and a goal position $g = r_g e^{i\theta_g}$ (in polar coordinates), where $r_s, r_g \leq w_{\max}$, the backbone of the optimal corridor connecting s and g is a spiral arc supported by the logarithmic spiral $r = a^* e^{b^*\theta}$. Since both s and g lie on this spiral, we have (assuming $\theta_s \neq \theta_g$, otherwise the optimal backbone path is simply a line segment):*

$$a^* = r_g^{\frac{\theta_s}{\theta_s-\theta_g}} \cdot r_s^{-\frac{\theta_g}{\theta_s-\theta_g}} , \qquad b^* = \frac{1}{\theta_g-\theta_s}\cdot\log\frac{r_g}{r_s} . \tag{9.3}$$

We now consider the case where the clearance of the two endpoints exceeds $w_{\max}$, namely the two endpoints of our path lie outside the closure of the disc $B(p; w_{\max})$. There are two possible scenarios: (i) The straight line segment $\overline{sg}$ does not intersect $B(p; w_{\max})$; in this case, this segment is the backbone of the optimal corridor. (ii) $\overline{sg}$ intersects $B(p; w_{\max})$. In this latter case the optimal backbone path is a bit more involved. Consider some backbone path γ connecting s and g. It is clear that the intersection of γ with $B(p; w_{\max})$ comprises a single component, so we denote the point where the path enters the disc by s' and the point where it leaves the disc by g' (see Fig. 9.2(a)). As s' and g' lie on the disc boundary, their polar representation is $s' = w_{\max}e^{i\theta_{s'}}$ and $g' = w_{\max}e^{i\theta_{g'}}$, so we use Equation (9.3) and obtain $a^* = w_{\max}$ and $b^* = 0$. The optimal path between s' and g' therefore lies on the degenerate spiral $r = w_{\max}$, namely the circle that forms the boundary of $B(p; w_{\max})$. We conclude that the optimal backbone path between s and g must contain a circular arc on the boundary of $B(p; w_{\max})$. As according to Lemma 9.2 this path must be smooth, it should comprise two line segments ss^* and g^*g that are tangent to the disc and a circular arc that connects the two tangency points s^* and g^* (see the dashed path in Fig. 9.2(a)). Note that as there are two possible smooth paths from s to g we select the shortest one.

9.3.2 Multiple Well-Separated Point Obstacles

Let us now go back to our original setting, where we are given a set of point obstacles $\mathcal{O} = \{p_1, \ldots, p_n\}$, along with a preferred width $w_{\max}$, and wish to compute the optimal corridor from s to g, where we assume that $c(s) = \min_i \|s - p_i\| \geq w_{\max}$ and $c(g) = \min_i \|g - p_i\| \geq w_{\max}$.

In case the points are well separated — that is, for each $i \neq j$ the discs $B(p_i; w_{\max})$ and $B(p_j; w_{\max})$ are disjoint in their interiors (implying that $\|p_i - p_j\| \geq 2w_{\max}$), we can follow

[3] http://www-groups.dcs.st-and.ac.uk/~history/Curves/Equiangular.html

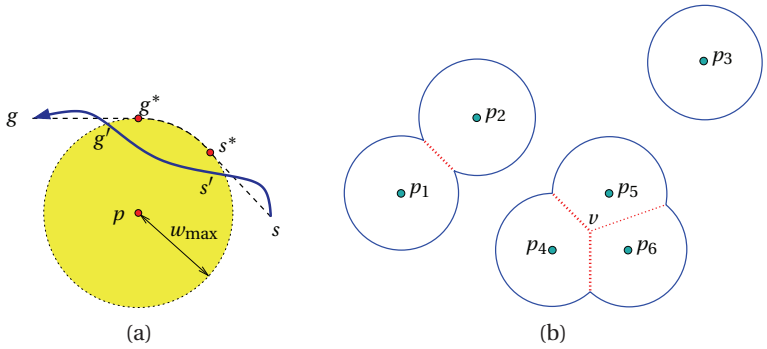


Figure 9.2: (a) The optimal backbone path in the presence of a single point obstacle. (b) The bounded Voronoi diagram of six points; the boundary of $\mathcal{M}$ is drawn is solid lines and the Voronoi edges are dotted.

the same arguments we used above for a single obstacle and conclude that the optimal backbone is either the straight line segment sg (in case it is *free*, namely its interior does not intersect the interior of any of the discs), or it comprises circular arcs and line segments that connect them.

We can therefore construct the visibility graph of the dilated obstacles and use it to construct optimal paths. The vertices of this graph are the endpoints of the free bitangents to two dilated obstacles, which in turn are represented as graph edges. In addition, each two neighboring tangency points on a disc $B(p_i; w_{\max})$ are connected by a circular arc. Given a path-planning query, namely two endpoints s and g, we treat s and g as vertices and add all free tangents from s and from g to the discs as graph edges. If the segment sg is free, we add it to the graph as well. We then perform Dijkstra's algorithm from s to find the shortest path to g in the resulting graph. The weight $\omega(e)$ given to each graph edge e is its weighted length, which simply equals its length in this case.

Proposition 9.4. *Given a set $\mathcal{O}$ of n point obstacles in the plane that are well-separated with respect to $w_{\max}$, and two endpoints s and g with clearance at least $w_{\max}$, it is possible to compute the optimal corridor connecting s and g in $O(E \log n)$ time using the visibility graph of the dilated obstacles, where E is the number of visibility edges in this graph.*

9.3.3 Corridors amidst Point Obstacles: The General Case

We now consider the general case where the endpoints s and g have arbitrary clearance, and where the dilated obstacles $B(p_1; w_{\max}), \ldots, B(p_n; w_{\max})$ are not necessarily pairwise disjoint in their interiors. The boundary of $\mathcal{M} = \bigcup_{i=1}^{n} B(p_i; w_{\max})$ comprises whole circles and circular arcs, such that a common endpoint of two arcs is a reflex vertex. We now construct $\mathcal{V}$, the Voronoi diagram of the points, and compute the intersection $\mathcal{V} \cap \mathcal{M}$, namely the portions of the Voronoi edges contained within the union of the dilated obstacles. Note that reflex vertices are equidistant to two point obstacles, so they serve as the connection points be-

tween the Voronoi edges and the boundary arcs of $\mathcal{M}$. We will refer to the Voronoi edges in $\mathcal{V} \cap \mathcal{M}$, together with the circular arcs that form the boundary of $\mathcal{M}$, as the *bounded Voronoi diagram* of the point set $\mathcal{O} = \{p_1, \ldots, p_n\}$, which we denote $\hat{\mathcal{V}}(\mathcal{O})$ (see Fig. 9.2(b)).

Note that $\hat{\mathcal{V}}(\mathcal{O})$ partitions the plane into two-dimensional cells of two types: Voronoi regions of the point obstacles, and regions where the clearance is larger than $w_{\max}$. Given two points s' and g' that belong to the same cell κ, we know that:

- If κ is a cell whose clearance is greater than $w_{\max}$, the optimal backbone path between $s' = (x_1, y_1)$ and $g' = (x_2, y_2)$ is the straight line segment σ that connects them, provided that σ does not intersect any feature of $\hat{\mathcal{V}}(\mathcal{O})$. The weighted length of this segment simply equals the Euclidean distance $\|g' - s'\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

- If κ is a Voronoi cell of a point obstacle p_i, the optimal backbone path between s' and g' is a spiral arc σ centered at p_i, provided that σ does not intersect any feature of $\hat{\mathcal{V}}(\mathcal{O})$. If $s' = r_1 e^{i\theta_1}$ and $g' = r_2 e^{i\theta_2}$ are the polar coordinates of the endpoints with respect to p_i, the weighted length of σ is given by (recall that from Equation (9.3) we have $b = \frac{1}{\theta_2 - \theta_1} \cdot \log \frac{r_2}{r_1}$):

$$\begin{aligned} L^*(\sigma) &= \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{r(\theta)} \sqrt{r^2(\theta) + \left(\frac{dr}{d\theta}\right)^2(\theta)}\, d\theta = \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{ae^{b\theta}} \sqrt{1+b^2}\, ae^{b\theta}\, d\theta = \\ &= \int_{\theta_1}^{\theta_2} w_{\max} \sqrt{1+b^2}\, d\theta = w_{\max} \sqrt{1+b^2} (\theta_2 - \theta_1) = \\ &= w_{\max} \sqrt{(\theta_2 - \theta_1)^2 + (\log r_2 - \log r_1)^2}\,. \end{aligned}$$

In addition, the features of $\hat{\mathcal{V}}(\mathcal{O})$ are also *locally optimal*, namely they can serve as backbone paths of optimal corridors (see Fig. 9.3(a)). We already know that portions of the circular arcs that form the boundary of $\mathcal{M}$ are locally optimal, and that the weighted length of such a circular arc simply equals its length. The Voronoi edges are also locally optimal: given s' and g' on the same Voronoi edge, the optimal backbone path that connects them is simply the straight line segment $s'g'$ which coincides with the Voronoi edge.

Following the construction of the visibility graph of the dilated point obstacles (Section 9.3.2), it is possible to add *visibility edges* to the bounded Voronoi diagram, namely to consider every free bitangent of two circular arcs, every free line segment from a reflex vertex tangent to a circular arc and every free line segment between two reflex vertices.[4] However, a path extracted from such a graph may pass through Voronoi vertices and reflex vertices, thus it may contain sharp turns. According to Lemma 9.2, such a path cannot serve as a backbone to an optimal corridor. We can try and rectify this problem by introducing a *shortcut edge* between each pair of Voronoi edges that are incident to a common Voronoi vertex (see Fig. 9.3(a) for an illustration), and between each pair consisting of a Voronoi edge and a visibility edge that are both incident to a common reflex vertex. However, this is not sufficient. We can show that it is sometimes possible to shortcut two Voronoi vertices v_1 and v_2 at once by connecting two Voronoi edges that are separated by another edge using a single curve. This

[4] The resulting construct is the visibility–Voronoi diagram of the obstacles; see [157] for more details.

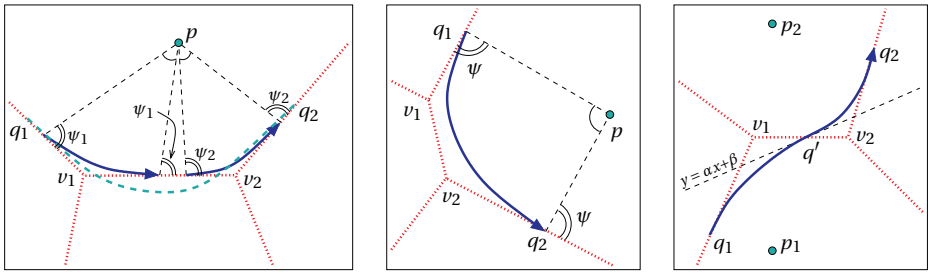


Figure 9.3: (a) The spiral arc connecting q_1 and q_2 (dashed) crosses the Voronoi edge v_1v_2; the optimal backbone path between q_1 and q_2 is therefore comprised of two spiral arcs that shortcut v_1 and v_2 (solid arrows) and portions of Voronoi edges. (b) Shortcutting two adjacent Voronoi vertices v_1 and v_2 by a single spiral arc. (c) Shortcutting two Voronoi vertices by a cross-cell curve comprised from a smooth concatenation of two spiral arcs. Both arcs have a common tangent $y = \alpha x + b$, which crosses the Voronoi edge v_1v_2 at q'.

curve may be contained in a single Voronoi cell, as in the example depicted in Fig. 9.3(b), or it may cross the Voronoi edge v_1v_2 at some point q' (see Fig. 9.3(c)). We should continue and examine the possibility of shortcutting $k > 2$ Voronoi vertices by considering sequences of $(k+1)$ contiguous Voronoi edges and trying to locate an endpoint q_1 on the first edge and q_2 on the last edge that are connected by a smooth curve comprising spiral arcs. This operation is not trivial, and requires solving a system of low-degree polynomial equations with $2(N_c+1)$ unknowns, where N_c is the number of crossings between the shortcut curve and the Voronoi diagram. In some scenarios it may be possible to construct shortcuts to $\Theta(n)$ Voronoi vertices by considering sequences of $\Theta(n)$ contiguous Voronoi edges, thus the size of the augmented diagram may blow up exponentially.

We therefore devise an approximation algorithm based on the structure of the bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{O})$ and the planar partition it induces. Given $\varepsilon > 0$, we subdivide the line segments and the circular arcs that form the features of $\hat{\mathcal{V}}(\mathcal{O})$ into small intervals of length $\frac{c(I)}{w_{\max}}\varepsilon$ (as ε is small, we consider the clearance of an interval I to be constant and denote it $c(I)$). Notice that the intervals are shorter in regions where the clearance is smaller, and that each interval has weighted length ε. Hence, if Λ is the total weighted length of the features of $\hat{\mathcal{V}}(\mathcal{O})$, then there are $\frac{\Lambda}{\varepsilon}$ intervals in total. Let us now define a graph $\mathcal{D}$ whose set of nodes equals the set of intervals $\mathcal{I}$. Each interval is incident to two of the cells defined by the bounded Voronoi diagram, and we connect $I_1, I_2 \in \mathcal{I}$ by an edge if and only if they are incident to a common cell. This edge is a line segment in a cell where the clearance is larger than $w_{\max}$, a spiral segment in a Voronoi region of one of the point obstacles, a circular arc on the boundary of a dilated obstacle, or a straight line segment on a Voronoi edge. In addition, an edge should not cross any of the features of $\hat{\mathcal{V}}(\mathcal{O})$. Using a brute-force algorithm that checks each candidate edge versus the $O(n)$ diagram features, $\mathcal{D}$ can be constructed in $O\left(\frac{\Lambda^2}{\varepsilon^2}n\right)$ time.

Given two endpoints s and g, we can connect them to the graph and use Dijkstra's algorithm to compute a near-optimal backbone connecting s and g in $O\left(\frac{\Lambda^2}{\varepsilon^2}\right)$ time. Let γ^* be the backbone path of the optimal corridor between s and g, which comprises $k = O(n)$ segments $\gamma_1, \ldots, \gamma_k$ (a path segment may be a straight line segment, a spiral arc, a portion of a circular arc or a portion of a Voronoi edge). We next show that each such segment is approximated by an edge in the graph $\mathcal{D}$ we have constructed.

Lemma 9.5. *For each segment γ_i of the optimal backbone path γ^*, there exists an edge e in $\mathcal{D}$ such that $L^*(e) < L^*(\gamma_i) + 2\sqrt{2}\varepsilon$.*

Proof: Let us denote the endpoints of the path segment γ_i by q_1 and q_2, and let I_1 and I_2 be the intervals that contain these endpoints, respectively.

In case γ_i is a straight line segment in a cell κ whose clearance is greater than $w_{\max}$, then its weighted length simply equals $\|q_2 - q_1\|$, the Euclidean distance between its endpoints. In the graph $\mathcal{D}$ there exists an edge connecting I_1 and I_2, and we denote its endpoints by $\tilde{q}_1$ and $\tilde{q}_2$. By the construction of the intervals, we know that $\|q_j - \tilde{q}_j\| \leq \frac{c(I_j)}{w_{\max}}\varepsilon = \varepsilon$ (for $j = 1,2$), hence:

$$\|\tilde{q}_2 - \tilde{q}_1\| < \|q_2 - q_1\| + 2\varepsilon \ .$$

Similar arguments hold when γ_i is a circular arc with clearance $w_{\max}$.

In case γ_i is a segment on a Voronoi edge, the graph $\mathcal{D}$ contains a segment $\tilde{q}_1\tilde{q}_2$ that in the worst case extends $\frac{c(q_1)}{w_{\max}}\varepsilon$ to one side of q_1 and $\frac{c(q_2)}{w_{\max}}\varepsilon$ to the other side of q_2. Since the contribution of each of these extensions is $\frac{w_{\max}}{c(q_j)}$ times its length (for $j = 1,2$), the weighted length of $\tilde{q}_1\tilde{q}_2$ is at most 2ε more than $L^*(\gamma_i)$.

The case where γ_i is a spiral arc contained in a Voronoi cell of a point obstacle p_i is a bit more involved. Let $q_1 = r_1 e^{i\theta_1}$ and $q_2 = r_2 e^{i\theta_2}$ be the polar coordinates of γ_i's endpoints with respect to p_i, then we have $L^*(\gamma_i) = w_{\max}\sqrt{(\theta_2 - \theta_1)^2 + (\log r_2 - \log r_1)^2}$. $\mathcal{D}$ contains a spiral arc connecting I_1 and I_2, and we denote its endpoints by $\tilde{q}_j = \tilde{r}_j e^{i\tilde{\theta}_j} \in I_j$ (for $j = 1,2$). As $c(q_j) = r_j$, we know that the length of each of these two intervals is $\|I_j\| = \frac{r_j}{w_{\max}}\varepsilon$. If we denote $\Delta\theta_j = \theta_j - \tilde{\theta}_j$, we can write:

$$\sin\left(\frac{\Delta\theta_j}{2}\right) < \frac{\frac{1}{2}\|I_j\|}{r_j} = \frac{\varepsilon}{2w_{\max}} \ .$$

As for small angles $\sin\phi \approx \phi$, we conclude that $|\Delta\theta_j| < \frac{\varepsilon}{w_{\max}}$. At the same time, $|\Delta r_j| = |r_j - \tilde{r}_j| < \frac{\varepsilon r_j}{w_{\max}}$, thus we have:

$$\left|\log\tilde{r}_j - \log r_j\right| < \left|\log\left(r_j\left(1 + \frac{\varepsilon}{w_{\max}}\right)\right) - \log r_j\right| = \log\left(1 + \frac{\varepsilon}{w_{\max}}\right).$$

As $\log(1+x) \approx x$ for small x values, we conclude that $|\log\tilde{r}_j - \log r_j| < \frac{\varepsilon}{w_{\max}}$. The length of the approximated spiral arc contained in $\mathcal{D}$ can therefore be at most $L^*(\gamma_i) + 2\sqrt{2}\varepsilon$. ∎

Corollary 9.6. *For each two endpoints s and g, it is possible to use the graph $\mathcal{D}$ and compute a near-optimal backbone path $\tilde{\gamma}$ connecting s and g in $O\left(\frac{\Lambda^2}{\varepsilon^2}\right)$ time, such that $L^*(\tilde{\gamma}) < L^*(\gamma^*) + O(n)\varepsilon$.*

9.4 Optimal Corridors amidst Polygonal Obstacles

In this section we generalize the data structures introduced in Section 9.3 to compute optimal corridors amidst polygonal obstacles. As we did in case of point obstacles, we first examine how an optimal backbone path looks like in the vicinity of a single obstacle. Note that the polygon P can be viewed as a collection of points (vertices) and line segments (edges), such that the distance of a point $q \in \mathbb{R}^2$ to P is attained on a polygon vertex or in the interior of an edge. We can thus subdivide the plane into regions, such that the identity of the closest polygon feature is the same for all points in any of the regions. Using the analysis we performed in Section 9.3.1 we already know that the optimal backbone path in a region closest to a polygon vertex is an arc of a logarithmic spiral. We now study the case of two points that lie in a region closest to a polygon edge.

Without loss of generality, we shall assume that the polygon edge we consider is an arbitrarily long segment of the vertical line $x = 0$, and analyze the optimal backbone path γ between two points s and g, whose distance from this line is less than $w_{\max}$ (see Fig. 9.1(b) for an illustration). Note that the width of the corridor at $\gamma(t) = (x(t), y(t))$ simply equals $|x(t)|$.

We begin by approximating the backbone path by a polyline. Assume that $\gamma(t)$ passes through a point $p_0 = (x_0, 0)$ and forms an angle α_0 with the line $y = 0$ perpendicular to the obstacle. For any $\Delta x > 0$ we can define the lines $x = x_0, x = x_0 + \Delta x, x = x_0 + 2\Delta x, \ldots$, where each two neighboring lines define a vertical slab; since Δx is small we assume that the distance of all points in the slab from the obstacle is constant and equals $x_0 + k\Delta x$. We can now use Equation (9.2) and write: $\sin\alpha_1 = \frac{x_0+\Delta x}{x_0}\sin\alpha_0$, $\sin\alpha_2 = \frac{x_0+2\Delta x}{x_0+\Delta x}\sin\alpha_1 = \frac{x_0+2\Delta x}{x_0}\sin\alpha_0$, …, $\sin\alpha_k = \frac{x_0+k\Delta x}{x_0}\sin\alpha_0$. If we examine the kth slab we can write $x = x_0 + k\Delta x$, so we have:

$$\Delta y_k = \Delta x \tan\alpha_k = \Delta x \cdot \frac{\sin\alpha_k}{\sqrt{1-\sin^2\alpha_k}} = \Delta x \cdot \frac{x\sin\alpha_0}{\sqrt{x_0^2 - x^2\sin^2\alpha_0}}\,. \tag{9.4}$$

Letting Δx tend to zero we obtain a smooth curve. We can use Equation (9.4) to express the derivative of the curve and we obtain:

$$y'(x) = \lim_{\Delta x \longrightarrow 0} \frac{\Delta y_k}{\Delta x} = \frac{x\sin\alpha_0}{\sqrt{x_0^2 - x^2\sin^2\alpha_0}}\,, \tag{9.5}$$

$$y(x) = -\frac{1}{\sin\alpha_0}\sqrt{x_0^2 - x^2\sin^2\alpha_0} + K\,. \tag{9.6}$$

As the point $(x_0, 0)$ lies on the curve, it is easy to see that the constant K equals $x_0\cot\alpha_0$.

Observe that $y(x)$ is defined only for $x < \frac{x_0}{\sin\alpha_0}$. When $x = \frac{x_0}{\sin\alpha_0}$ the path is reflected from the vertical wall and starts approaching the obstacle. We note that squaring and rearranging Equation (9.6) we obtain that $x^2 + (y - x_0\cot\alpha_0)^2 = \left(\frac{x_0}{\sin\alpha_0}\right)^2$, thus we conclude that γ is a circular arc, whose supporting circle is centered at $C = (0, x_0\cot\alpha_0)$ and its radius is $\frac{x_0}{\sin\alpha_0}$.

Proposition 9.7. *Given a start position $s = (x_s, y_s)$ and a goal position $g = (x_g, y_g)$ in the vicinity of a segment supported by $x = 0$ and with $0 < x_s, x_g \leq w_{\max}$, the backbone of the optimal corridor between these two endpoints is a circular arc supported by a circle of radius r^* that is centered at $(0, y^*)$, where (we assume that $y_s \neq y_g$, otherwise the optimal backbone path is simply the line segment sg):*

$$y^* = \frac{y_s + y_g}{2} + \frac{x_g^2 - x_s^2}{2(y_g - y_s)}\,, \tag{9.7}$$

$$r^* = \sqrt{\frac{1}{2}(x_s^2 + x_g^2) + \frac{1}{4}(y_g - y_s)^2 + \frac{(x_g^2 - x_s^2)^2}{4(y_g - y_s)^2}}\,. \tag{9.8}$$

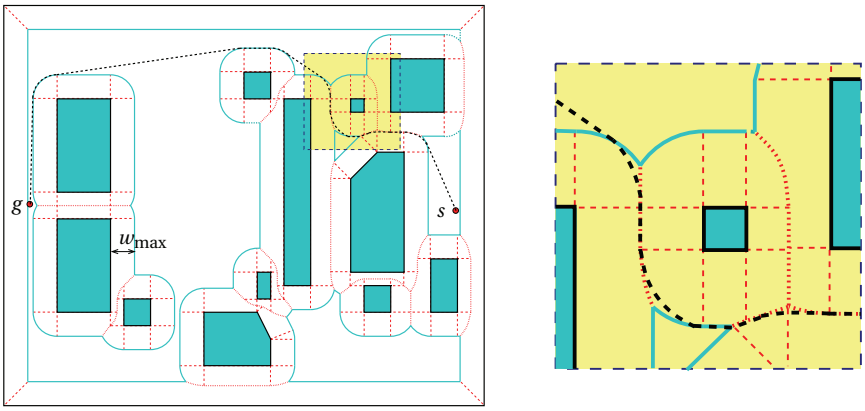
9.4.1 Moving amidst Multiple Polygons

We are given a set $\mathcal{P} = \{P_1, \ldots, P_k\}$ of polygonal obstacles having n vertices in total, along with a preferred corridor width $w_{\max}$.

We first mention that if the polygons are well-separated, namely the distance between each P_i and P_j $(1 \leq i < j \leq k)$ is more than $2w_{\max}$, we can use the visibility graph of the dilated polygons to plan optimal backbone paths. The dilated obstacles in this case are Minkowski sums of the polygonal obstacles with a disc of radius $w_{\max}$ and their boundary comprises line segments, which correspond to dilated polygon edges, and circular arcs, which correspond to dilated vertices. Visibility edges in this case correspond to line segments tangent to two circular arcs. Proving that the visibility graph indeed contains optimal backbone paths is done exactly the same as we did in Section 9.3.2 for point obstacles.

In case there exist narrow passages between the obstacles, we generalize the construction detailed in Section 9.3.3 to polygons, and introduce the bounded Voronoi diagram of the set of polygons $\mathcal{P}$. Note that in this case we have *Voronoi chains* that are sequences of Voronoi edges. A Voronoi edge may be induced by two polygon vertices or by two polygon edges, in which case it is a line segment, or by a polygon vertex and an edge of another polygon, in which case it is a parabolic arc. Thus, the Voronoi chains are smooth curves that are piecewise linear or piecewise parabolic and are equidistant to two nearest polygons; see, e.g., [102] for more details. The bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{P})$ also contains edges that separate the Voronoi cells of adjacent polygon features, namely a polygon edge and a vertex incident to this edge. These edges are line segments perpendicular to the obstacles (see Fig. 9.4 for an illustration).

Observe that if we are given two points on the same Voronoi chain, then the locally optimal backbone path between them is simply the segment of the chain they define. This is clear in case of point obstacles, as the edges are straight line segments. In case of chains that separate Voronoi cells of polygons and may contain parabolic arcs this fact is less obvious. However, we are able to prove that parabolic arcs are also locally optimal — namely, it is not possible to shortcut such an arc by choosing a shorter route that is closer to one of the polygons, as such a route always has a larger weighted length. This proof is rather technical and we refer the reader to [159] for its details.



(a) (b)

Figure 9.4: (a) A near-optimal backbone path (dashed) amidst polygonal obstacles, overlayed on top of the the bounded Voronoi diagram of the obstacles. Boundary edges are drawn in light solid lines, Voronoi chains between polygons are dotted, and Voronoi edges that separate cells of adjacent polygon features are drawn in a light dashed line. The bounded Voronoi diagram was computed using the software described in [157]. The backbone path was computed using an A* algorithm on a fine grid discretizing the environment. (b) Zooming on a portion of the path; note the shortcuts the path takes.

$\hat{\mathcal{V}}(\mathcal{P})$ subdivides the plane into cells of three types: regions where the clearance is larger than $w_{\max}$, Voronoi cells of polygon vertices, and Voronoi cells of polygon edges. We have already encountered cells of the first two types in the bounded Voronoi diagram of a set of points (Section 9.3.3). We also know from Proposition 9.7 that if we have two points in the Voronoi cell of a polygon edge, the optimal backbone path connecting them is a circular arc whose center lies on this edge. Assume, without loss of generality, that the obstacle edge lies on the line $y = 0$ and that the center of the circular arc a is the origin, and let $r^* e^{i\theta_1}$ and $r^* e^{i\theta_2}$ be the arc endpoints. The weighted length of the circular arc is therefore given by (note that $r(\theta) = r^*$):

$$\begin{aligned} L^*(a) &= \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{r^* \sin\theta} \sqrt{r^2(\theta) + \left(\frac{dr}{d\theta}\right)^2(\theta)}\, d\theta = \int_{\theta_1}^{\theta_2} \frac{w_{\max}}{\sin\theta}\, d\theta = \\ &= w_{\max} \left(\log \frac{1-\cos\theta}{\sin\theta} \right) \Bigg|_{\theta_1}^{\theta_2} = w_{\max} \left(\log\tan\frac{\theta_2}{2} - \log\tan\frac{\theta_1}{2} \right). \end{aligned}$$

The approximation algorithm given in Section 9.3.3 can also be extended to handle polygonal obstacles. In this case we also consider intervals that lie on Voronoi edges that separate the Voronoi cell of each polygon into simple regions — thus, each region is induced by a polygon vertex, a polygon edge, or correspond to regions where the clearance is above $w_{\max}$. We can show that Lemma 9.5 also applies for the circular arcs inside a Voronoi cell of a polygon edge: Let γ_i be such a circular arc and let I_1 and I_2 be the intervals containing its endpoints q_1 and q_2, receptively. $\mathcal{D}$ contains a circular arc σ connecting I_1 and I_2, and

we denote its endpoints $\tilde{q}_j = \tilde{r}_j e^{i\tilde{\theta}_j} \in I_j$ (for $j = 1,2$). As $c(q_j) = r^* \sin\theta_j$, we know that the length of each interval is $\|I_j\| = \frac{r^* \sin\theta_j}{w_{\max}}\varepsilon$. If we denote $\Delta\theta_j = \theta_j - \tilde{\theta}_j$, we can write:

$$\sin\left(\frac{\Delta\theta_j}{2}\right) < \frac{\frac{1}{2}\|I_j\|}{r^*} = \frac{\varepsilon}{2w_{\max}}\sin\theta_j \,.$$

As for small angles $\sin\phi \approx \phi$, we conclude that $|\Delta\theta_j| < \frac{\varepsilon}{w_{\max}}\sin\theta$. If we use the fact that $f(x+\Delta x) \approx f(x) + f'(x)\Delta x$ (for small Δx) with $f(x) = \log\tan\frac{x}{2}$, we can bound the weighted length of σ (recall that $f'(x) = \frac{1}{\sin x}$ in our case):

$$\begin{aligned} L^*(\sigma) &= w_{\max}\left(\log\tan\frac{\theta_2+\Delta\theta_2}{2} - \log\tan\frac{\theta_1+\Delta\theta_1}{2}\right) < \\ &< w_{\max}\left(\log\tan\frac{\theta_2}{2} + \frac{\varepsilon\sin\theta_2}{w_{\max}}\cdot\frac{1}{\sin\theta_2} - \log\tan\frac{\theta_1}{2} + \frac{\varepsilon\sin\theta_1}{w_{\max}}\cdot\frac{1}{\sin\theta_1}\right) = \\ &= L^*(\gamma_i) + 2\varepsilon \,. \end{aligned}$$

Corollary 9.8. *Given a set of polygonal obstacles $\mathcal{P}$ having n vertices in total, let Λ be the total weighted length of the bounded Voronoi diagram $\hat{\mathcal{V}}(\mathcal{P})$ with respect to a given $w_{\max}$ value. Given $\varepsilon > 0$, we can construct a graph $\mathcal{D}$ over the intervals of $\hat{\mathcal{V}}(\mathcal{P})$ in $O\left(\frac{\Lambda^2}{\varepsilon^2}n\right)$ time, such that for each two endpoints s and g it is possible to use $\mathcal{D}$ and compute a near-optimal backbone of a corridor C connecting s and g. $L^*(C)$ is at most $O(n)\varepsilon$ more than the weighted length of the optimal corridor connecting s and g.*

9.5 Conclusions and Future Work

In this chapter we have introduced a measure for the quality of corridors and studied the structure of optimal corridors amidst point obstacles and polygonal obstacles in the plane. We have devised an approximation algorithm for computing near-optimal corridors amidst obstacles. We are also investigating methods to speed up our approximation algorithm, as well as design simple practical methods to compute good corridors. We are interested in extending our result to corridors in three dimensions as well.

In some applications having a winding backbone path decreases the quality of the corridor. We can therefore augment the weighted length function by considering the curvature of the backbone path γ as follows:

$$L^*_\mu(C) = \int_\gamma \left(\frac{w_{\max}}{w(t)}\right)^{d-1} dt + \mu\int_\gamma w(t)\kappa(t)\,dt \,, \tag{9.9}$$

where $\kappa(t)$ is the curvature of $\gamma(t)$, and $0 < \mu \le 1$ is the weight we give to the curvature measure. We are able to show that in case of well-separated obstacles, optimal corridors under the L^*_μ measure are still contained in the visibility graph of the obstacles dilated by $w_{\max}$. We are still exploring methods of computing optimal corridors in the case of denser scenes.

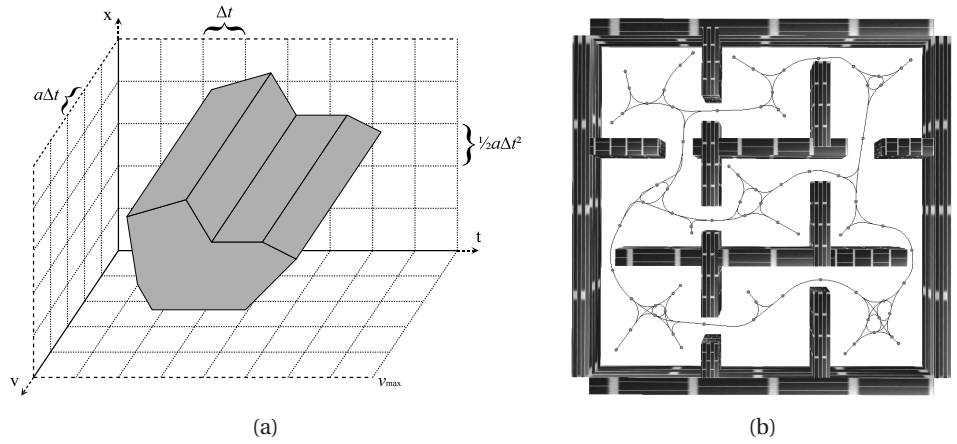
Chapter 10

Conclusion and Future Work

In this thesis, we studied the problem of path planning in dynamic environments, and we introduced new approaches to several variants of the problem. Roughly, the approaches can be subdivided into the themes: offline planning, online planning and corridors. We will discuss each of them in this conclusion, and provide some outlook for future work on these topics.

10.1 Offline planning

In Chapter 3, we introduced a new *offline* method for planning in dynamic environments in which the moving obstacles have trajectories that are known beforehand. As the configuration-time space is transitory, preprocessing a roadmap directly in the configuration-time space is not useful (although we have presented a special class of dynamic environments for which it is useful in Chapter 7). Therefore, we proposed an approach that preprocesses a roadmap for the static configuration space, that is, the roadmap is collision-free with respect to the stationary obstacles of the scene. In the second stage, we query efficiently for a time-optimal path avoiding the moving obstacles in the 'roadmap-time' space, which is a complex of two-dimensional configuration-time grids. This decoupling is a novel idea to the field of path planning for dynamic environments (it has been used for the multi-robot path planning problem, though [150, 99, 40, 65]). We have shown that this approach is very successful. Paths in complicated environments (having narrow passages and requiring true temporal coordination to avoid the obstacles) can be computed in tenths of seconds of computation time. Also, a path is guaranteed to be found by our method whenever one exists in the discretized 'roadmap-time' space. Moreover, the method we presented is very generally applicable, that is, it can be used for any robot type in configuration spaces of arbitrary dimension. To be precise, the method is applicable whenever PRM is applicable for the robot type and configuration space. Also, there are no constraints or assumptions about the behavior of the moving obstacles; they may move with any speed along any trajectory, they may even deform, as long as the motions are known beforehand. As such, we do not believe that given the current state of the art, there will be much room



(a) (b)

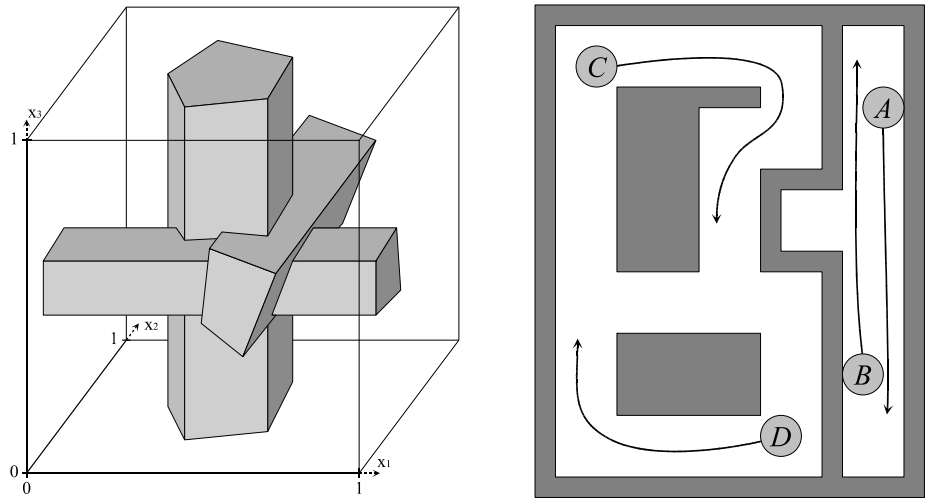
Figure 10.1: (a) A three-dimensional state-time grid. (b) A smooth roadmap (Courtesy of Dennis Nieuwenhuisen).

left for improvement as far as this particular problem is concerned.

An aspect that is not considered in this work, though, is acceleration constraints on the robot. Interesting future work would be to extend our approach such that acceleration constraints can be taken into account. Then, instead of choosing a velocity v, we will choose an *acceleration* a from the set $\{-a_{\max}, 0, a_{\max}\}$. Given the time step Δt, the possible velocities v, as well as the possible positions x along an edge, are discretized [53]. Hence, we will plan in a three-dimensional *state-time* grid (see Fig. 10.1(a)). Further, to have convincing motions, a *smooth* roadmap without C^1-discontinuities is required. A normal PRM-roadmap would force an acceleration-constrained robot to have zero velocity at each of its vertices, in order for the robot to change its direction. The approach of [113] looks promising in this direction (see Fig. 10.1(b)).

A further future challenge is to make this planner having real-time performance. As it is not known beforehand how long the planning will take, it is hard to choose the time value of the start configuration-time for the planner in a real-time setting (see the discussion in Chapter 7). Therefore, it is desirable that the planner is given some time τ to initialize, and then explores the search tree further while the robot is executing its path.

To show the strength of the above method, we applied it to the problem of multi-robot path planning in Chapter 4. We presented a prioritized approach, in which each of the robots is given a priority according to the length of its query. In this order, paths for the robots are planned avoiding the previously planned robots with higher priority, that are considered as moving obstacles. Our method inherits the general applicability from the method for planning in dynamic environments, that is, it is applicable to robots of any kind, and robots of different types can be used simultaneously. Experimental results have shown that paths for as many as 24 robots can be planned in confined and crowded workspaces within seconds of computation time. In contrast, coordinated approaches have been shown not to perform



(a) (b)

Figure 10.2: (a) A three-dimensional coordination space with cylindrical obstacles. (b) An environment in which the motions of robots A and B should be coordinated without taking into account robots C and D.

well for 4 or more robots.

As we chose a prioritized approach, the method is not complete in terms of the problem formulation. That is, scenarios exist in which a multi-robot path exists, but will not be found by our method. However, in most environments one would encounter in practice our method works very fine. Further, the prioritized approach is a natural approach; consider an environment in which multiple robots are active (e.g. airplanes in the airspace around an airport). If at some point a new robot (an arriving airplane) appears in the environment, it is sensible to let this new robot avoid the previously planned robots, and give priority to robots that were already present in the environment.

As above, an interesting extension of our method is to take the acceleration constraints on the robot into account. Other future work concerns coordinated approaches; they are very interesting from an algorithmic perspective. An interesting open problem is whether the *cylindrical* nature of the obstacles in the multi-robot's coordination space (which are caused by robot-robot collisions) can be exploited intrinsically to create an efficient algorithm (see Fig. 10.2(a)). In current works, the cylindrical nature is only used to save robot-robot collision-checks. A further challenge is to create a planner that only coordinates the motion of a pair of robots when they are actually obstructing each other. Other robots, moving in completely different parts of the environment and not impeding each other, then need not be taken into account in the coordination (see Fig. 10.2(b)). This keeps the dimension of the search space low, and makes the problem tractable. Some work in this direction has been done in [150, 71], but improvements may be made on the practicality of the approach.

10.2 Online planning

In Chapters 5 and 6 we explored ideas on *online* planning amidst moving obstacles whose future trajectories are not known, and can only be estimated based on sensory data. At several places in this thesis, we have identified a number of fundamental difficulties that are inherent to online planning in partially known dynamic environments:

1. In a continuous cycle of sensing and planning, only a limited amount of time is reserved for planning in each cycle (let this amount be τ). Most planning algorithms, however, are not guaranteed to finish planning within this amount of time.

2. When planning starts at time t_0, and τ time is reserved for planning, the time value of the start configuration-time in the planner is $t_0+\tau$. The planner relies on the estimates of the future trajectories of the obstacles, based on sensory data acquired at time t_0. In the next iteration, planning starts at time $t_0+\tau$, and the start configuration-time of the planner is $t_0+2\tau$. Hence, if the plan from the previous cycle is not safe between time $t_0+\tau$ and $t_0+2\tau$, e.g. due to incorrect obstacle trajectory estimates, it may fail.

3. If τ is small, there may not be enough time to plan a path with enough look-ahead to bring the robot in a safe situation. Also, it is hard to bias the search towards the goal if only limited time is available.

4. If τ is large, the planner cannot react quickly on new sensory data, which may jeopardize its safety. Also, there may be too much look-ahead. That is, the planner plans a path based on estimates of the obstacle trajectories too far in the future, which are very likely to be inaccurate. Hence, invalid information about the far future may influence (important) decisions in the near future. Also, much precious time is vainly spent on (portions of) paths that need be replanned anyway.

In Chapter 5, we presented a planner that solves difficulty 1 mentioned above. We chose an *anytime* approach, which means that an initial path (all the way to the goal) of possibly poor quality is planned very quickly, and as long as time (τ) has not run out, the quality of the path is improved. This guarantees that a path is available when one is needed. Also, the path will be directed towards the goal. Another objective of this chapter was to use information from the previous planning cycle to speed up the search when replanning. (In other works, this potentially powerful source of information has so far largely been neglected, and new paths are planned from scratch.) To this end, we applied the D*-algorithm to dynamic environments. D* is a powerful method to quickly adapt a plan in *static* environments, when during the execution of the path some change in the environment is detected. In our method, we used the same principles for dynamic environments; whenever a moving obstacle changes its course, we quickly adapt the previous plan, using all its available information. Our method is in principle applicable to any robot type in any configuration space, although we only implemented it for discs moving in the plane. We have shown that our approach works, and that it contains promising ideas for future research on online path planning in dynamic environments.

We did not address the fundamental issue of safety (see difficulty 2 above) in the above work. In Chapter 6, we focused on creating a planner that is inherently safe, regardless of

what the obstacles do (we assumed that the robot and the obstacles are discs in the plane). To this end, we did not rely on inaccurate trajectory estimates, but only assumed knowledge about the maximal velocities of the moving obstacles. If the positions of the moving obstacles are known at time t_0, the regions in which the obstacles are guaranteed to be at times $t > t_0$ are discs growing over time with a rate corresponding to the maximal velocities of the obstacles. In the configuration-time space, this gives conical obstacles that should be avoided when planning a path (this is only possible when the robot moves faster than any of the obstacles). We presented an efficient algorithm to compute the *shortest* path avoiding the growing discs, that is the shortest inherently safe path. Its running time is $O(n^3 \log n)$ where n is the number of moving obstacles. Experimental results have shown that paths can be computed in *milliseconds* of computation time, which is small enough to finish within a sensibly preset amount of time τ (see difficulty 1 above).

In the broader perspective of the difficulties shown above, the two works presented are only first steps toward full autonomous and safe path planning in highly dynamic environments. Yet, they contain some interesting principles to be used in future work. A major challenge is to develop a fundamental framework for online path planning in which all of the above difficulties are addressed, and also takes constraints on the acceleration of the robot into account. Below, we will sketch how such a framework might look.

Let us assume that when we acquire data from a sensor about the positions and velocities of the obstacles at time t_0, we can estimate the *probability* that configuration-time $\langle c, t \rangle$ is collision-free, for $t > t_0$, and let this probability be denoted $P_{t_0}(c, t)$. How these probability distributions look for various values of t depends on the assumptions we make on the obstacle behavior.

Now, we use a continuous cycle of interleaved sensing and planning to navigate through the environment. Let the duration of each planning cycle be τ. In this amount of time, we use a *partial planner* to explore different paths incrementally until time has run out (it is simple to incorporate acceleration constraints into a partial planner [124]). There may not be enough time to plan all the way to the goal configuration, but we can select the best path among the ones explored: For each (partial) path, we can compute its probability of being collision-free (by integrating the function P_{t_0} over the partial path) and we know how far it has advanced towards the goal (by computing its endpoint's distance to the goal). Based on these two quantities, we define a local optimization criterion (let a parameter ω determine the weight balance between the two quantities), and select the optimal partial path among the ones explored. Then, we continue with the next planning cycle, in which new information has become available via the sensors.

The amount of time we can plan ahead each cycle, denoted by η, is obviously a monotonically increasing function of τ. That is, when τ is larger, we have more look-ahead in our paths. More look-ahead results in a path that has better global characteristics, yet it requires a larger τ, which is less safe if quick reaction is required. Hence, there is some optimum in the choice for τ, depending on the nature of the environment.

Given a choice for the parameters ω and τ, and a start and a goal configuration in an environment containing moving obstacles, the planner as described above, will either find a path to the goal (say with length T time on average), or result in a collision with an obstacle (say with probability Q). Hence, T and Q are functions of ω and τ. Now, given a global

optimization criterion based on T and Q, we can compute the optimal ω and τ.

Interesting questions arising from this framework is how we can compute the probability distributions $P_{t_0}(c,t)$ (e.g. assuming Brownian motion of the obstacles), whether we can make it such that inherent safety is guaranteed between time $t_0+\tau$ and time $t_0+2\tau$ (e.g. along the same lines as in Chapter 6), how we integrate over the distribution to find the accumulated probability that a path is collision-free, and how we determine the optimal value of τ. In all other works about online path planning, the value of τ is chosen manually, as is often the look-ahead horizon η and other parameters. Resolving these issues will tackle all four difficulties mentioned above.

10.3 Corridors

Corridors have recently appeared as a new approach to path planning. The central idea of corridors is to decouple the global path planning (planning towards the goal) from the local path planning (avoiding collisions with obstacles). Corridors provide a global direction of motion towards the goal, yet leaves the flexibility to avoid local hazards, e.g., small moving obstacles. First, a corridor is planned between a global start configuration and a goal configuration, and then a local planner, for instance a potential field method, operates within the environment of the corridor. As the corridor provides a convex environment, i.e. a union of discs, the risk of getting trapped in local minima of the potential field disappears. As such, the notion of corridors is a promising approach to enable planning in highly dynamic environments, yet assuring that the motion will globally leads towards the goal.

In Chapter 8 we presented an efficient algorithm to construct a fully covering roadmap of corridor backbone paths, which we call the $\mathrm{VV}^{(c)}$-diagram. The corridors can directly be extracted from the roadmap, as opposed to the method of [82], for instance, that requires smoothing of a PRM-path in a post-processing stage. This enables application in real-time settings. The $\mathrm{VV}^{(c)}$-diagram is an interpolate between the visibility graph, which contains the shortest paths in the environment, and the Voronoi diagram, which contains paths with maximal clearance. Indeed, a good corridor backbone path trades off length and clearance. Although the $\mathrm{VV}^{(c)}$-diagram contains sensible corridors, they do not optimize a reasonable mathematical quality criterion. In Chapter 9, we have defined such a criterion, and provided an approximation algorithm to compute corridors that are optimal according to this criterion.

Summarizing, we laid down a solid theoretical foundation for the notion of corridors. Interesting future work would be to actually implement corridors for planning in dynamic environments. As the corridor already indicates a global direction of motion, we may be able to implement an online path planner with a very small value for τ to avoid obstacles. This would enable application in very highly dynamic environments. Reactive local planners, such as [145] already provides some promising ideas in this direction.

Bibliography

[1] The CGAL project homepage. http://www.cgal.org/.

[2] The CORE library homepage. http://www.cs.nyu.edu/exact/core/.

[3] E. Acar, H. Choset, and P. Atkar. Complete sensor-based coverage with extended-range detectors: A hierarchical decomposition in terms of critical points and Voronoi diagrams. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1305–1311, 2001.

[4] P. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[5] S. Akella and J. Peng. Time-scaled coordination of multiple manipulators. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3337–3344, 2004.

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

[7] N. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 113–120, 1996.

[8] B. Aronov and M. Sharir. On translational motion planning of a convex polyhedron in 3-space. *SIAM Journal on Computing*, 26(6):1785–1803, 1997.

[9] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter V, pages 201–290. Elsevier Science Publishers, Amsterdam, The Netherlands, 2000.

[10] F. Avnaim, J. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1656–1661, 1988.

[11] B. Baginski. The Z^3-method for fast path planning in dynamic environments. In *Proc. IASTED Conf. on Applications of Control and Robotics*, pages 47–52, 1996.

[12] J. Barraquand, L. Kavraki, J. Latombe, T. Li, R. Motwani, and P. Raghavan. A random sampling scheme for path planning. *International Journal of Robotics Research*, 16(6):759–774, 1997.

[13] J. Barraquand and J. Latombe. Robot motion planning: a distributed representation approach. *International Journal of Robotics Research*, 10(6):628–649, 1991.

[14] M. Bennewitz, W. Burgard, and S. Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 41(2):89–99, 2002.

[15] M. Bennewitz, W. Burgard, and S. Thrun. Learning motion patterns of persons for mobile service robots. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3601–3606, 2002.

[16] J. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2366–2371, 2006.

[17] J. van den Berg, D. Nieuwenhuisen, L. Jaillet, and M. Overmars. Creating robust roadmaps for motion planning in changing environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2415–2421, 2005.

[18] J. van den Berg and M. Overmars. Roadmap-based motion planning in dynamic environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1598–1605, 2004.

[19] J. van den Berg and M. Overmars. Prioritized motion planning for multiple robots. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2217–2222, 2005.

[20] J. van den Berg and M. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics*, 21(5):885–897, 2005.

[21] J. van den Berg and M. Overmars. Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. *International Journal of Robotics Research*, 24(12):1055–1072, 2005.

[22] J. van den Berg and M. Overmars. Computing shortest paths amidst growing discs in the plane. In *Proc. European Workshop on Computational Geometry*, pages 59–62, 2006.

[23] J. van den Berg and M. Overmars. Path planning in repetitive environments. In *Proc. IEEE Int. Conf. on Methods and Models in Automation and Robotics*, pages 657–662, 2006.

[24] J. van den Berg and M. Overmars. Planning the shortest safe path amidst unpredictably moving obstacles. In *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006.

[25] M. de Berg, L. Guibas, and D. Halperin. Vertical decompositions for triangles in 3-space. *Discrete Computational Geometry*, 15(1):35–61, 1996.

[26] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 2nd edition, 2000.

[27] M. de Berg, J. Matoušek, and O. Schwarzkopf. Piecewise linear paths among convex obstacles. *Discrete and Computational Geometry*, 14:9–29, 1995.

[28] G. van den Bergen. *Collision detection in interactive 3D environments*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[29] R. Bohlin and L. Kavraki. Path planning using Lazy PRM. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 521–528, 2000.

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

[31] M. Branicky, S. LaValle, K. Olson, and L. Yang. Quasi-randomized path planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1481–1487, 2001.

[32] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 341–346, 1999.

[33] R. Burden and J. Faires. *Numerical analysis*. Brooks/Cole, Pacific Grove, CA, 7th edition, 2001.

[34] Z. Butler. Corridor planning for natural agents. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 499–504, 2006.

[35] J. Canny. *The complexity of robot motion planning*. MIT Press, Cambridge, MA, 1988.

[36] J. Canny. Computing roadmaps of general semi-algebraic sets. *The Computer Journal*, 36(5):504–514, 1993.

[37] E. Chang, S. Choi, D. Kwon, H. Park, and C. Yap. Shortest path amidst disc obstacles is computable. In *Proc. ACM Symp. on Computational Geometry*, pages 116–125, 2005.

[38] S. Chaudhuri and V. Koltun. Smoothed analysis of probabilistic roadmaps. Unpublished manuscript.

[39] B. Chazelle. Approximation and decomposition of shapes. In J. Schwartz and C. Yap, editors, *Algorithmic and Geometric Aspects of Robotics*, pages 145–185. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[40] H. Chitsaz, J. O'Kane, and S. LaValle. Exact pareto-optimal coordination of two translating polygonal robots on an acyclic roadmap. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3981–3986, 2004.

[41] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, 2005.

[42] C. Clark, T. Bretl, and S. Rock. Applying kinodynamic randomized motion planning with a dynamic priority system to multi-robot space systems. In *Proc. IEEE Aerospace Conference*, pages 3621–3631, 2002.

[43] G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Lecture Notes in Computer Science*, 33:135–183, 1975.

[44] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[45] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, 1993.

[46] C. Duncan, A. Efrat, S. Kobourov, and C. Wenk. Drawing with fat edges. In *Proc. Int. Symp. on Graph Drawing*, pages 162–177, 2001.

[47] M. Erdmann and T. Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

[48] P. Fiorini and Z. Shiller. Time optimal trajectory planning in dynamic environments. *Journal of Applied Mathematics and Computer Science*, 7(2):101–126, 1997.

[49] P. Fiorini and Z. Shiller. Motion planning in dynamic environments using velocity obstacles. *International Journal of Robotics Research*, 17(7):760–772, 1998.

[50] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. European Symp. on Algorithms*, pages 664–676, 2004.

[51] S. Fortune. Voronoi diagrams and Delaunay triangulations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 23, pages 513–528. Chapman & Hall/CRC, 2nd edition, 2004.

[52] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation*, 4(1):23–33, 1997.

[53] T. Fraichard. Trajectory planning in a dynamic workspace: a 'state-time' approach. *Advanced Robotics*, 13(1):75–94, 1999.

[54] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[55] K. Fujimura. *Motion planning in dynamic environments*. Springer-Verlag, Tokyo, Japan, 1991.

[56] K. Fujimura. Time-minimum routes in time-dependent networks. *IEEE Transactions on Robotics and Automation*, 11(3):343–351, 1995.

[57] K. Fujimura and H. Samet. Planning a time-minimal motion among moving obstacles. *Algorithmica*, 10:41–63, 1993.

[58] R. Geraerts and M. Overmars. A comparative study of probabilistic roadmap planners. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 40–54, 2002.

[59] R. Geraerts and M. Overmars. Clearance based path optimization for motion planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2386–2392, 2004.

[60] R. Geraerts and M. Overmars. Reachability analysis of sampling based planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 406–412, 2005.

[61] R. Geraerts and M. Overmars. Sampling and node adding in probabilistic roadmap planners. *Journal of Robotics and Autonomous Systems*, 54:406–412, 2005.

[62] R. Geraerts and M. Overmars. Creating high-quality roadmaps for motion planning in virtual environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 4355–4361, 2006.

[63] R. Geraerts and M. Overmars. The corridor map method: A general framework for real-time high-quality path planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2007.

[64] S. Ghosh and D. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.

[65] R. Ghrist, J. O'Kane, and S. LaValle. Computing pareto optimal coordinations on roadmaps. *International Journal of Robotics Research*, 24(11):997–1010, 2005.

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

[67] K. Goldberg. Completeness in robot motion planning. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 419–429, 1994.

[68] A. Gray. *Modern Differential Geometry of Curves and Surfaces with Mathematica*, chapter Logarithmic Spirals, pages 40–42. CRC Press, Boca Raton, FL, 2nd edition, 1997.

[69] D. Halperin and M. Sharir. A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment. *Discrete Computational Geometry*, 16(2):121–134, 1996.

[70] P. Hart, N. Nilsson, and B. Rafael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[71] S. Hirsch and D. Halperin. Hybrid motion planning: Coordinating two discs moving among polygonal obstacles in the plane. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 225–241, 2002.

[72] S. Hirsch and E. Leiserowitz. Exact construction of Minkowski sums of polygons and a disc with application to motion planning. Technical Report ECG-TR-181205-01, Tel-Aviv University, 2002.

[73] C. Holleman and L. Kavraki. A framework for using the workspace medial axis in PRM planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1408–1413, 2000.

[74] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 4420–4426, 2003.

[75] D. Hsu, R. Kindel, J. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255, 2002.

[76] D. Hsu, J. Latombe, and H. Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. *International Journal of Robotics Research*, 25(7):627–643, 2006.

[77] D. Hsu, J. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2719–2726, 1997.

[78] P. Isto. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2323–2328, 2002.

[79] L. Jaillet and T. Simeon. A PRM-based motion planner for dynamically changing environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1606–1611, 2004.

[80] L. Jaillet, A. Yershova, S. LaValle, and T. Simeon. Adaptive tuning of the sampling domain for Dynamic-Domain RRTs. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2005.

[81] M. Kallmann and M. Matarić. Motion planning using dynamic roadmaps. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 4399–4404, 2004.

[82] A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. In *Proc. Eurographics/ACM SIGGRAPH Symp. on Computer Animation*, pages 1–10, 2004.

[83] A. Kamphuis and M. Overmars. Motion planning for coherent groups of entities. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3815–3822, 2004.

[84] A. Kamphuis, J. Pettre, M. Overmars, and J. Laumond. Path finding for the animation of walking characters. In *Proc. Eurographics/ACM SIGGRAPH Sympos. Computer Animation*, pages 8–9, 2005.

[85] K. Kant and S. Zucker. Toward efficient planning: the path-velocity decomposition. *International Journal of Robotics Research*, 5(3):72–89, 1986.

[86] M. Karavelas. Segment Voronoi diagrams in CGAL, 2004. http://www.cgal.org/UserWorkshop/2004/svd.pdf.

[87] L. Kavraki and J. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3020–3026, 1994.

[88] L. Kavraki, J. Latombe, R. Motwani, and P. Raghavan. Randomized preprocessing of configuration space for fast path planning. In *Proc. ACM Symp. on Theory of Computing*, pages 353–362, 1995.

[89] L. Kavraki, P. Švestka, J. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Trans. Robotics and Automation*, 12(4):566–580, 1996.

[90] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, 1:59–70, 1986.

[91] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[92] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 968–975, 2002.

[93] J. Kuffner and S. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 995–1001, 2000.

[94] J. Kuffner and S. LaValle. An efficient approach to path planning using balanced bidirectional RRT search. Technical Report CMU-RI-TR-05-34, Carnegie Mellon University, 2005.

[95] H. Kurniawati and D. Hsu. Workspace importance sampling for probabilistic roadmap planning. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1618–1623, 2004.

[96] J. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

[97] S. LaValle. *Planning Algorithms*. Cambridge University Press, New York, 2006.

[98] S. LaValle, M. Branicky, and S. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research*, 23(7/8):673–692, 2004.

[99] S. LaValle and S. Hutchinson. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*, 14(6):912–925, 1998.

[100] S. LaValle and J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, 2001.

[101] S. LaValle and J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001.

[102] D. Lee and R. Drysdale. Generalization of Voronoi diagrams in the plane. *SIAM Journal on Computing*, 10(1):73–87, 1981.

[103] 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.

[104] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An anytime, replanning algorithm. In *Proc. Int. Conf. on Automated Planning and Scheduling*, 2005.

[105] M. Lin and D. Manocha. Collision and proximity queries. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, FL, 2nd edition, 2004.

[106] F. Lingelbach. Path planning using probabilistic cell decomposition. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 467–472, 2004.

[107] Y. Liu and S. Arimoto. Finding the shortest path of a disc among polygonal obstacles using a radius-independent graph. *IEEE Transactions on Robotics and Automation*, 11:682–691, 1995.

[108] T. Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983.

[109] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[110] J. Mitchell. Shortest paths and networks. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 27, pages 607–642. Chapman & Hall/CRC, 2nd edition, 2004.

[111] J. Mitchell and C. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.

[112] C. Nielsen and L. Kavraki. A two level fuzzy PRM for manipulation planning. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1716–1721, 2000.

[113] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, and M. Overmars. Automatic construction of roadmaps for path planning in games. In *Proc. Int. Conf. Computer Games: Artificial Intelligence, Design and Education*, pages 285–292, 2004.

[114] D. Nieuwenhuisen and M. Overmars. Motion planning for camera movements. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3870–3876, 2004.

[115] D. Nieuwenhuisen and M. Overmars. Useful cycles in probabilistic roadmap graphs. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 446–452, 2004.

[116] N. Nilsson. A mobile automaton: an application of artificial intelligence techniques. In *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 509–520, 1969.

[117] N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[118] C. Ó'Dúnlaing, M. Sharir, and C. Yap. Retraction: A new approach to motion-planning. In *Proc. ACM Symp. on Theory Computing*, pages 207–220, 1983.

[119] C. Ó'Dúnlaing and C. Yap. A "retraction" method for planning the motion of a disc. *Journal on Algorithms*, 6:104–111, 1985.

[120] M. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Universiteit Utrecht, 1992.

[121] M. Overmars. Path planning for games. In *Proc. Int. Game Design and Technology Workshop*, pages 29–33, 2005.

[122] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.

[123] J. Peng and S. Akella. Coordinating the motions of multiple robots with kinodynamic constraints. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 4066–4073, 2003.

[124] S. Petty and T. Fraichard. Safe motion planning in dynamic environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 3726–3731, 2005.

[125] E. Plaku, K. Bekris, B. Chen, A. Ladd, and L. Kavraki. Sampling-based roadmap of trees for parallel motion planning. *IEEE Transactions on Robotics*, 21(4):597–608, 2005.

[126] M. Pocchiola and G. Vegter. The visibility complex. *International Journal of Computational Geometry and Applications*, 6(3):279–308, 1996.

[127] J. Reif. Complexity of the mover's problem and generalizations. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 421–427, 1979.

[128] J. Reif and M. Sharir. Motion planning in the presence of moving obstacles. *Journal of the ACM*, 41:764–790, 1994.

[129] J. Reif and J. Storer. Shortest paths in Euclidean space with polyhedral obstacles. Technical Report CS-85-121, Brandeis University, 1985.

[130] J. Reif and H. Wang. Social potential fields: a distributed behavioral control for autonomous robots. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 431–459, 1995.

[131] E. Rimon and D. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Transactions on Robotics and Automation*, 8(5):501–518, 1992.

[132] H. Rohnert. Moving a disc between polygons. *Algorithmica*, 6:182–191, 1991.

[133] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[134] M. Saha, J. Latombe, Y. Chang, and F. Prinz. Finding narrow passages with probabilistic roadmaps: The small-step retraction method. *Autonomous Robots*, 19(3):301–319, 2005.

[135] G. Sánchez and J. Latombe. A single query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proc. Int. Symp. on Robotics Research*, 2001.

[136] G. Sánchez and J. Latombe. Using a PRM planner to compare centralized and decoupled planning for multi-robot systems. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2112–2119, 2002.

[137] J. 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, 1983.

[138] J. 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, 1983.

[139] J. 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 obstacles. *Int. J. of Robotics Research*, 2(3):46–75, 1983.

[140] F. Schwarzer, M. Saha, and J. Latombe. Exact collision checking of robot paths. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 25–41, 2002.

[141] H. Shaul and D. Halperin. Improved construction of vertical decompositions of 3D arrangements. In *Proc. ACM Symp. on Computational Geometry*, pages 283–292, 2002.

[142] T. Simeon, J. Laumond, and C. Nissoux. Visibility based probabilistic roadmaps for motion planning. *Advanced Robotics Journal*, 14(6), 2000.

[143] T. Siméon, S. Leroy, and J. Laumond. Path coordination for multiple mobile robots: a resolution complete algorithm. *IEEE Transactions on Robotics and Automation*, 18(1):42–49, 2002.

[144] D. Spielman and S. Teng. Smoothed analysis: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

[145] C. Stachniss and W. Burgard. An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 508–513, 2002.

[146] A. Stentz. The Focussed D* algorithm for real-time replanning. In *Proc. Int. Joint Conf. on Artificial Intelligence*, 1995.

[147] K. Sutner and W. Maass. Motion planning among time dependent obstacles. *Acta Informatica*, 26:93–122, 1988.

[148] P. Švestka. On probabilistic completeness and expected complexity of probabilistic path planning. Technical Report UU-CS-96-20, Utrecht University, 1996.

[149] P. Švestka. *Robot motion planning using probabilistic road maps*. PhD thesis, Universiteit Utrecht, 1997.

[150] P. Švestka and M. Overmars. Coordinated path planning for multiple robots. *Robotics and Autonomous Systems*, 23(3):125–152, 1998.

[151] S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, California Institute of Technology, CA, 1977.

[152] J. Vannoy and J. Xiao. Real-time adaptive and trajectory-optimized manipulator motion planning. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 497–502, 2004.

[153] D. Vasquez and T. Fraichard. Motion prediction for moving objects: a statistical approach. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3931–3936, 2004.

[154] D. Vasquez, F. Large, T. Fraichard, and C. Laugier. High-speed autonomous navigation with motion prediction for unknown moving obstacles. In *Proc. IEEE Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 82–87, 2004.

[155] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. European Symp. on Algorithms*, pages 884–895, 2002.

[156] R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. In *Proc. European Workshop on Computational Geometry*, pages 151–154, 2005.

[157] R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. In *Proc. ACM Symp. on Computational Geometry*, pages 63–72, 2005.

[158] R. Wein, J. van den Berg, and D. Halperin. Planning near-optimal corridors amidst obstacles. In *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006.

[159] R. Wein, J. van den Berg, and D. Halperin. Planning near-optimal corridors amidst obstacles. Technical report, Tel-Aviv University, 2006.
http://www.cs.tau.ac.il/~wein/publications/pdfs/corridors.pdf.

[160] R. Wein, J. van den Berg, and D. Halperin. The visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications*, 36:66–87, 2007.

[161] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Proc. Library-Centric Software Design Workshop*, 2005.

[162] E. Weisstein. Logarithmic spiral. In *MathWorld – a Wolfram web resource*. http://mathworld.wolfram.com/LogarithmicSpiral.html.

[163] S. Wilmarth, N. Amato, and P. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1024–1031, 1999.

[164] Y. Yang and O. Brock. Adapting the sampling distribution in PRM planners based on an approximated medial axis. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 4405–4410, 2004.

Samenvatting in het Nederlands

Padplanning speelt een belangrijke rol in verscheidene toepassingsgebieden, zoals CAD-ontwerp, computerspellen en virtuele omgevingen, moleculaire biologie, en robotica. In de meest algemene vorm is het padplanningprobleem te formuleren als het vinden van een botsingsvrij pad voor een bewegende entiteit tussen een start- en een doelpositie in een omgeving met obstakels. Dit probleem is uitgebreid bestudeerd in de afegelopen decennia. De meeste aandacht is besteed aan padplanning in *statische* omgevingen. Dat zijn omgevingen waarin alle obstakels stationair zijn en niet bewegen, en de posities en de vorm van de obstakels bekend worden verondersteld. Minder aandacht is uitgegaan naar padplanning in *dynamische* omgevingen, waarin naast stationaire obstakels, ook bewegende obstakels voorkomen. Padplanning in dynamische omgevingen is het voornaamste onderwerp van dit proefschrift. We behandelen zowel het geval waarin de trajecten van de bewegende obstakels vooraf bekend zijn, als het geval waarin deze informatie online vergaard moet worden, bijvoorbeeld door middel van sensoren. Hieronder geven we een kort overzicht van de inhoud van dit proefschrift.

In Hoofdstuk 3 presenteren we een offline planner voor bekende dynamische omgevingen. De methode bestaat uit een voorbewerkingsfase en een queryfase. In de voorbewerking wordt alleen rekening gehouden met de stationaire obstakels; er wordt een dichte roadmap gecreëerd die het vrije deel van de configuratieruimte bedekt. In de queryfase kan er dan door middel van een efficiënte zoekstrategie een pad op deze roadmap worden berekend tussen een start- en een doelpostitie die botsingen met bewegende obstakels vermijdt. Eerdere werken gebruiken ofwel direct een single-shotbenadering [75], of een ontkoppelde benadering waarin de beweging van de robot wordt gecoördineerd op een voorgepland pad [53]. Onze benadering heeft voordelen ten opzicht van beide. Ten eerste, als de stationaire obstakels nauwe doorgangen veroorzaken in de configuratieruimte, heeft een RRT-achtige benadering die direct wordt toegepast in de configuratie-tijdruimte mogelijk problemen om daar doorheen een pad te vinden binnen een redelijke hoeveelheid tijd. In onze benadering wordt dit probleem aangepakt in de voorbewerkingsfase. Ten tweede kan het beperken van de bewegingen van de robot tot een enkel pad zo rigoureus zijn, dat er geen geldige coördinatie bestaat op dit pad met betrekking tot de bewegende obstakels, terwijl er wel degelijk geldige bewegingen zijn die van dit pad afwijken. Als de roadmap die wij gebruiken dicht genoeg is, en hij cykels bevat die alternatieve routes verschaffen, verdwijnt dit probleem grotendeels. Het gebruik van een roadmap heeft meer voordelen, waarvan een is dat de methode zeer algemeen toepasbaar is en werkt voor bijna alle robottypes. Een nadeel is dat

onze aanpak geen beperkingen op de versnelling toelaat, in tegenstelling tot de twee eerder genoemde methoden. Experimentele resultaten tonen aan dat onze methode zeer snelle looptijden behaalt in ingewikkelde dynamische omgevingen.

Een van de meest prominente toepassingen van offline planners voor bekende dynamische omgevingen is padplanning voor meerdere robots. In dit gerelateerde probleem is het doel om paden te plannen voor een verzameling robots die elke robot van zijn start- naar zijn doelconfiguratie brengt, zonder onderlinge botsingen en botsingen met stationaire obstakels. In principe kan de verzameling robots worden gezien als één robot die uit meerdere onderdelen bestaat met één samengestelde configuratieruimte. Alhoewel direct plannen in deze ruimte in het algemeen te langzaam is om practisch te kunnen worden toegepast (vanwege de hoge dimensionaliteit), toont deze formulering van het probleem aan dat we te maken hebben met een statisch padplanningprobleem waarin tijd geen intrinsieke rol speelt. Het is daarom dat we een offline planner kunnen gebruiken. Deze wordt als volgt toegepast: Eerst krijgt elke robot een prioriteit toegewezen. Vervolgens wordt in volgorde van prioriteit een pad gepland voor elke robot tussen zijn start- en doelconfiguratie, waarbij botsingen met eerder geplande robots van hogere prioriteit worden vermeden. Dus, voor elke robot moet een pad worden gepland in een bekende dynamische omgeving, waarin de eerder geplande robots worden gezien als bewegende obstakels. Het plannen kan offline worden gedaan, omdat er geen real-timevereisten zijn. Als voor alle robots een geldig pad is gevonden, is de combinatie van deze paden een oplossing voor het multi-robotprobleem. Deze aanpak volgt een zogenoemde *geprioriteerde* benadering. Die is zeer snel, maar niet compleet, omdat de robots niet in staat zijn hun bewegingen te coördineren, hetgeen noodzakelijk is in sommige situaties. Echter, in de meeste practische omstandigheden zal de geprioriteerde methode perfect werken. De offline planner die wordt gepresenteerd in Hoofdstuk 3 is bij uitstek geschikt voor geprioriteerde multi-robotplanning. Deze toepassing wordt behandeld in Hoofdstuk 4.

In Hoofdstuk 5 presenteren we een online planner voor gedeeltelijk bekende omgevingen. Om met de real-timevereisten om te kunnen gaan, heeft de planner de anytime-karakteristiek, wat inhoudt dat initieel zeer snel een compleet pad wordt gepland, die echter van matige kwaliteit is. De kwaliteit van het pad wordt constant verbeterd, zolang de reactietijd het toelaat. Wanneer tijdens de executie van het pad veranderingen worden waargenomen in de trajecten van obstakels moet een nieuw pad worden gepland. Een voornaam doel van dit werk is om informatie van eerdere plans opnieuw te gebruiken om het oude pad snel bij te kunnen werken, en daarmee een hogere uitvoerkwaliteit te kunnen garanderen. Eerdere werken gebruiken de informatie van eerdere plans niet opnieuw, maar plannen een nieuw pad van begin af aan [124, 154]. De gekozen strategie in ons werk combineert een PRM-representatie van de statische configuratieruimte met een variant van het *D**-algoritme [104] om te plannen en herplannen in de roadmap-tijdruimte. Er is veel onderzoek verricht naar D*-achtige planners [146, 92], die kunnen worden gebruikt om te herplannen in statische omgevingen waarin veranderingen kunnen worden waargenomen aan de stationaire obstakels (bijvoorbeeld vanwege onprecieze kaarten). In ons werk is het doel om deze toepassing uit te breiden naar dynamische omgevingen waarbij veranderingen kunnen worden waargenomen aan de bewegende obstakels (bijvoorbeeld een koerswijziging).

In Hoofdstuk 6 presenteren we een planner voor volledig onvoorspelbaar bewegende ob-

stakels. De planner is in principe offline, maar hij kan ook online worden gebruikt, daar hij specifiek ontworpen is om de fundamentele moeilijkheden (die worden genoemd in Sectie 1.3.4) die inherent zijn aan plannen in gedeeltelijk bekende omgevingen te vermijden. Om dit te bereiken, worden de toekomstige trajecten van de bewegende obstakels zeer conservatief voorspeld. We nemen aan dat de obstakels ronde schijven zijn en een bekende maximale snelheid hebben. Dus, de gebieden waarin de obstakels kunnen zijn, zijn schijven die groeien met de tijd. Het doel in dit werk is om het korste pad te vinden die deze groeiende schijven vermijdt. Zulke paden zijn inherent veilig, ongeacht wat de bewegende obstakels doen. Dit is een geometrisch probleem, waarvoor een efficiënt algoritme en een snelle implementatie worden gepresenteerd. Om de groeiende schijven klein te houden, moeten paden continu opnieuw worden gepland met nieuwe sensorgegevens. Tevens moet de robot sneller bewegen dan alle bewegende obstakels om de groeiende schijven te kunnen omzeilen.

In de inleiding wordt verklaard dat een PRM-roadmap niet direct gebruikt kan worden in een configuratie-tijdruimte van een (bekende) dynamische omgeving. Dit is omdat de vooronderstelling dat de roadmap meerdere keren gebruikt kan worden niet opgaat in vergankelijke configuratie-tijdruimten. Dit is jammer, omdat PRM's de aantrekkelijke eigenschap hebben dat queries zeer snel kunnen worden beantwoord, én binnen een precies in te schatten hoeveelheid tijd. Het kiezen van een waarde voor τ in een real-timeomstandigheid (τ is de hoeveelheid tijd waarbinnen een pad moet worden gepland) zou dan zeer eenvoudig zijn. Er is echter een klasse van dynamische omgevingen waarvoor de vooronderstelling wél geldt, en waarin we wél een voorbewerkte roadmap, met al zijn aantrekkelijke eigenschappen, kunnen gebruiken. Deze klasse van omgevingen bevat de zogenaamde *repetitieve* omgevingen, waarin de bewegingen van de obstakels cyclisch zijn. Dat betekent dat na een bepaalde hoeveelheid tijd de obstakels terugkeren naar hun initiële configuratie en ad infinitum dezelfde beweging opnieuw uitvoeren. Een eenvoudig voorbeeld is een automatische draaideur. In Hoofdstuk 7 presenteren we een methode voor padplanning in zulke omgevingen.

In de omstandigheden die we tot dusverre hebben genoemd nemen we een zekere vorm van kennis over de bewegende obstakels aan. Echter, in veel complexe omgevingen zijn de posities en afmetingen van slechts de stationaire obstakels precies bekend ten tijde van het plannen, maar niets is bekend over de hoeveelheid en aard van de bewegende obstakels in de omgeving. In zulke dynamische omgevingen is het soms beter om de taak van het plannen van een pad dat globaal naar het doel leidt en de taak om bewegende obstakels te vermijden te scheiden. Dit vereist dat het globale pad niet vastligt, maar de flexibiliteit bevat om lokaal af te wijken van het globale pad om bewegende obstakels te vermijden. Aan de andere kant moet deze flexibiliteit niet ongelimiteerd zijn, omdat het globale pad een richtlijn moet zijn voor de globale bewegingsrichting. Deze overwegingen hebben geleid tot de ontwikkeling van *corridors* [34, 121, 63]. Corridors bestaan uit een *backbonepad* met aan beide zijden een vrij gebied (de corridor). Het backbonepad geeft een globale bewegingsrichting, en de corridor geeft de ruimte voor lokale afwijkingen van het globale pad.

Een goed backbonepad voor een corridor is kort, maar houdt een bepaalde afstand tot de obstakels. Als deze afstand overal aangehouden kan worden, zou het kortste pad uit de visibilitygraaf van de obstakels, vergroot met een laag met een breedte gelijk aan de gewenste afstand, een zinvol backbonepad van een corridor zijn. Als deze afstand niet over-

al aangehouden kan worden, wordt de maximale afstand lokaal verkregen op het voronoidiagram van de obstakels. Deze delen van het voronoidiagram kunnen worden verbonden met de visibilitygraaf van de vergrote obstakels om een graaf te genereren die corridors bevat voor alle homotopische klassen van de omgeving. We noemen deze graaf het visibility-voronoidiagram, en wordt geïntroduceerd in Hoofdstuk 8. Verder presenteren we een constructie die we het visibility-voronoicomplex noemen. Dat is een datastructuur die alle visibility-voronoidiagrammen bevat voor alle gewenste afstanden tot de obstakels, en die direct kan worden gequeried voor een geschikte corridor, zonder expliciet het visibility-voronoidiagram te construeren voor de betreffende gewenste afstand tot de obstakels.

Het visibility-voronoicomplex is een efficiënte datastructuur voor corridors. Deze corridors optimaliseren echter niet een zinvol wiskundig kwaliteitscriterium. In Hoofdstuk 9 introduceren we een natuurlijke maat voor de kwaliteit van corridors, waarin de lengte van een corridor wordt afgewogen tegen de breedte van een corridor (binnen de gewenste afstand tot de obstakels), en we verkennen de eigenschappen van corridors die optimaal zijn onder deze maat. Ook presenteren we een benaderingsalgoritme voor het plannen van bijna-optimale corridors.

Acknowledgments

I like to express my sincerest gratitude to all people that have contributed to the realization of this thesis. In particular, I would like to thank the following people.

First of all, I would like to thank Mark Overmars, who has been my supervisor during my PhD. I really appreciated our regular discussions, and Mark's sharp vision helped a lot in shaping solutions to many problems discussed in this thesis. Apart from scientific matters, Mark has been a very good mentor in the process of being a PhD student, which can be difficult sometimes.

Next I would like to thank Dan Halperin, Jean-Paul Laumond, Steve LaValle, Jan van Leeuwen and Gert Vegter for being on the reading committee of this thesis and for their helpful comments.

Fruitful collaborations with Tel-Aviv University and Carnegie Mellon University have been the basis for three chapters of this thesis. I like to thank Dan Halperin for giving me the opportunity to visit Israel and to host me at his institute. I like to thank Ron Wein, for the interesting discussions, the pleasant cooperation and for coauthoring the papers that have been the foundation of chapters 8 and 9. I like to thank James Kuffner for hosting me at his institute *and* his house, and I like to thank Dave Ferguson for the pleasant cooperation and for coauthoring the paper founding chapter 5 of this thesis.

Special thanks go to Dennis Nieuwenhuisen, who has been my officemate throughout my PhD. I really appreciated the fun we had and the support we gave each other. I thank him for making the workplace a very pleasant place to be.

Further, I would like to thank all of my coworkers, and Arno Kamphuis, Roland Geraerts and Frank van der Stappen in particular, for fruitful discussions, for the support I got and for the fun we had.

Lastly, I thank all of my friends and family for being my friends and my family.

List of Publications

- Ron Wein, Jur van den Berg, Dan Halperin; The visibility-Voronoi complex and its applications. *Computational Geometry: Theory and Applications* 36(1):66-87, 2007.
- Jur van den Berg, Mark Overmars; Using Workspace Information as a Guide to Non-uniform Sampling in Probabilistic Roadmap Planners. *International Journal on Robotics Research* 24(12):1055-1072, 2005.
- Jur van den Berg, Mark Overmars; Roadmap-based Motion Planning in Dynamic Environments. *IEEE Transactions on Robotics* 21(5):885-897, 2005.
- Henk Bekker, Jur van den Berg, Tsjerk Wassenaar; A Method to Obtain a Near-Minimal-Volume Molecular Simulation of a Macromolecule, Using Periodic Boundary Conditions and Rotational Constraints. *Journal of Computational Chemistry* 25(8):1037-1046, 2004.
- Ron Wein, Jur van den Berg, Dan Halperin; Planning Near-Optimal Corridors amidst Obstacles. *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006.
- Jur van den Berg, Mark Overmars; Planning the Shortest Safe Path amidst Unpredictably Moving Obstacles. *Proc. Workshop on Algorithmic Foundations of Robotics*, 2006.
- Jur van den Berg, Mark Overmars; Path Planning in Repetitive Environments. *Proc. IEEE International Conference on Methods and Models in Automation and Robotics*, pp. 657-662, 2006.
- Jur van den Berg, Dave Ferguson, James Kuffner; Anytime Path Planning and Replanning in Dynamic Environments. *Proc. IEEE International Conference on Robotics and Automation*, pp. 2366-2371, 2006.
- Jur van den Berg, Mark Overmars; Prioritized Motion Planning for Multiple Robots. *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2217-2222, 2005.
- Jur van den Berg, Dennis Nieuwenhuisen, Léonard Jaillet, Mark Overmars; Creating Robust Roadmaps for Motion Planning in Changing Environments. *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2415-2421, 2005.

- Ron Wein, Jur van den Berg, Dan Halperin; The Visibility-Voronoi Complex and its Applications. *Proc. ACM Annual Symposium on Computational Geometry*, pp. 63-72, 2005.
- Jur van den Berg, Mark Overmars; Roadmap-based Motion Planning in Dynamic Environments. *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1598-1605, 2004.
- Jur van den Berg, Mark Overmars; Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. *Proc. IEEE International Conference on Robotics and Automation*, pp. 453-460, 2004.
- Henk Bekker, Jur van den Berg, Tsjerk Wassenaar; Constructing a Near-Minimal-Volume Computational Box for Molecular Dynamics Simulations with Periodic Boundary Conditions. *Proc. International Conference on Computational Science, Lecture Notes in Computer Science* 2659(70-79), 2003.
- Jur van den Berg, Mark Overmars; Computing Shortest Paths amidst Growing Discs in the Plane. *Proc. European Workshop on Computational Geometry*, pp. 59-62, 2006.
- Ron Wein, Jur van den Berg, Dan Halperin; The Visibility-Voronoi Complex and its Applications. *Proc. European Workshop on Computational Geometry*, pp. 151-154, 2005.
- Jur van den Berg, Mark Overmars; Using watershed segmentation in robot motion planning. *Proc. Annual Conference of the Advanced School for Computing and Imaging*, pp. 151-158. 2004.

Curriculum Vitae

Jur Pieter van den Berg was born on 27th May 1981 in Groningen, The Netherlands. In 2003, he received his Master's degree in computer science from the University of Groningen, Groningen, The Netherlands. In the same year, he started as PhD-student under the supervision of prof. Mark Overmars at the computer science department of Utrecht University, Utrecht, The Netherlands. In 2007, he completed his thesis there.