Motion Planning for Car-like Robots using a Probabilistic Learning Approach

P. Svestka and M. Overmars

UU-CS-1994-33
August 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Motion Planning for Car-like Robots using a Probabilistic Learning Approach

P. Svestka and M. Overmars

Technical Report UU-CS-1994-33
August 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Motion Planning for Car-like Robots using a Probabilistic Learning Approach*

Petr Švestka, Mark H. Overmars
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: petr@cs.ruu.nl, markov@cs.ruu.nl

Abstract

In this paper a recently developed learning approach for robot motion planning is extended and applied to two types of car-like robots: normal ones and robots which can only move forwards. In this learning approach the motion planning process is split into two phases: the learning phase and the query phase. In the learning phase a probabilistic roadmap is incrementally constructed in configuration space. This roadmap is an undirected graph where nodes correspond to randomly chosen configurations in free space and edges correspond to simple collision-free paths between the nodes. These simple motions are computed using a fast local method. In the query phase this roadmap can be used to find paths between different pairs of configurations.

The approach can be applied to normal car-like robots (with non-holonomic constraints) by using suitable local methods, which compute paths feasible for the robots. Application to car-like robots which can move only forwards demands a more fundamental adaptation of the learning method. That is, the roadmaps must be stored in directed graphs instead of undirected ones.

We have implemented the planners and we present experimental results which demonstrate their efficiency for both robot types, even in cluttered workspaces.

1 Introduction

The motion planning problem is a well-known problem in the field of robotics ([Lat91], [HA92]). The objective is to find collision-free paths in an environment containing some obstacles for a robot $\mathcal{A}$. Throughout this paper we assume that the obstacles are fixed and known in advance. Motion planning is typically not performed in the workspace, that is, the space in which the robot and the obstacles physically are present, but rather in configuration space, or *C-space*. In C-space the robot reduces to a point. This is generally achieved by adding some extra dimensions and 'growing' the obstacles.

Many different instances of the motion planning problem can be identified. In simple cases, the robot is a solid object, and the constraints on its motions are of a holonomic nature. That is, given a free configuration c, the directions of the velocities (in C-space) achievable by the robot when positioned in c are not constrained. Such robots are called *free-flying robots* (see Figure 1). Things get more complicated if the robot is no longer solid

*This research was partially supported by the ESPRIT III BRA Project 6546 (PROMotion) and by the Dutch Organization for Scientific Research (N.W.O.)

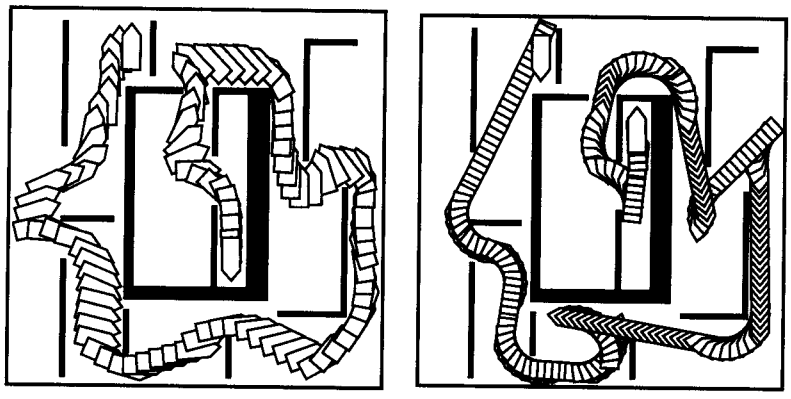


Figure 1: Feasible paths for solid planar robots. At the left, the robot is free-flying, while at the right it has non-holonomic car-like constraints.

or of it is subject to certain non-holonomic constraints. Articulated robots (robot arms) are examples of non-solid (but holonomic) robots. Car-like robots on the other hand are solid, but they are subject to non-holonomic constraints. The directions of their achievable velocities are constrained in a way that they can move forwards and backwards, and can make (bounded) turns, but cannot move sidewise (Figure 1). Intuitively, the motions that car-like robots can perform correspond to those performable by ordinary cars. Multi-body mobile robots (car-like robots with one or more trailers) are examples of non-solid robots with non-holonomic constraints.

Regardless of the robot type, the motion planning problem can be formulated in two, fundamentally different, versions. In one version, a start configuration s and a goal configuration g are given beforehand, and the objective is to compute a feasible path for the robot $\mathcal{A}$ from s to g. In the second version, no start and goal configurations are specified, and the objective is to compute a data structure, which can later be used for queries with arbitrary start and goal configurations. We refer to the former case as a *single shot* problem, and to the latter as a *learning* problem. Given a robot $\mathcal{A}$ in some environment S, solving a single shot problem for $\mathcal{A}$ in S is typically cheaper (that is, less time-consuming) than solving the corresponding learning problem. However, if many motions of the robot are to be computed in S, then it pays off to solve the learning problem. Of the "classical" approaches to motion planning (see [Lat91],[HA92]), that is, *roadmap methods*, *cell decomposition methods*, and *potential field methods*, the former two can be regarded as learning methods, while potential field methods are inherently single shot methods. The single shot nature of potential field methods is caused by the dependence of the involved potential fields on the given goal configurations.

In this paper we apply a recently developed probabilistic learning approach ([OŠ94]), which we refer to as the *random learning paradigm*, to motion planning problems involving two types of car-like robots: robots which can move both forwards and backwards, and robots which can only move forwards. We refer to the former ones as *general car-like robots*, and to the latter ones as *forward car-like robots*. Application to the latter requires, as we will show, a fundamental adaption of the paradigm.

The random learning paradigm consists of a *learning phase* and a *query phase*. In the

learning phase a probabilistic roadmap is constructed by repeatedly generating random free configurations and trying to connect these to other (earlier added) configurations by a simple but fast motion planner, called the *local method.* The network thus formed is stored in an undirected graph G. The configurations are stored as nodes in G, and the paths computed by the local method as edges in G. After the learning is done, a *query* consists of trying to connect some given start configuration and goal configuration to two (suitable) nodes of G, with paths that are feasible for the robot. Next, a graph search is performed to find a sequence of edges connecting these nodes, and, using the local method, this sequence is transformed into a feasible path in configuration space.

The flexibility of the described approach lies in the fact that the properties of the paths computed in the query phase are induced by the paths computed by the local method. Hence, application of the approach to a particular robot type asks for a suitable local method. In many cases, "plugging" a suitable local method into the random learning paradigm yields a fast and practical planner. This is demonstrated in this paper for general car-like robots. There are however certain robots, non-holonomic robots with non-symmetric control systems, which require the probabilistic road-maps to be stored in directed graphs instead of undirected ones. As mentioned above, the random learning paradigm, in its original form, uses undirected graphs for the storage of the roadmaps. Therefore it is not directly applicable to robots with non-symmetric control systems. In this paper we describe an adaptation of the random learning paradigm, in which directed underlying graphs are used. Subsequently, we apply it to forward car-like robots. These fall into the class of robots for which undirected graphs do not suffice.

We present experimental results which show that, for both types of car-like robots, the obtained planners are very fast, even in cluttered workspaces. Typically, the learning times required for the construction of adequate roadmaps, that is, roadmaps that capture the connectivity of the free C-space well, are in the order of a few seconds.

Apart from the low running times, our approach has another major advantage. Although there currently exist a few practical planners for car-like robots (see Section 2), these are all single shot methods. To our knowledge, our method is the first true learning approach for car-like robots, which makes it particularly suitable for situations in which some car-like robot is to perform several collision-free motions in a static environment.

This paper is organized as follows: In Section 2 we review previous work on motion planning for car-like robots and probabilistic/heuristic motion planning techniques. In Section 3, we describe the random learning paradigm in its original form. We do this in general terms, without focussing on any specific robot type. Then, in Section 4, we show that for some robot types directed (instead of undirected) graphs are required for storing the roadmaps, and we subsequently discuss how the approach can be adapted in this direction. In Section 5 car-like robots are formally defined, along with some simple (car-like) path constructs. In the rest of the paper, we describe how the general techniques described in Sections 3 and 4 can be applied to car-like robots. Section 6 deals with general car-like robots, and Section 7 with forward car-like robots. In both sections we give experimental results, and also we consider possibilities for guiding the node adding by the geometry of the workspace, improving the running times considerably. In Section 8 we give a probabilistic completeness proof for the method applied to general car-like robots. That is, we prove that any query that is solvable in open free C-space will be solved in the query phase, provided that a sufficient amount of time has been spent on learning. Some conclusions are drawn in Section 9.

2 Previous work

The motion planning problem for car-like robots has received considerable attention in the past years. Yet, there still exist only few planners for car-like robots which are practical in the presence of obstacles.

In [LTJ90] and [LS89] efficient planners for general car-like robots are described. The approach consists of three steps. In Step one, given a start configuration s and a goal configuration g, a collision-free path P_1 connecting s and g is computed, without taking into account the non-holonomic car-like constraints. Then, in Step two, a set of configurations $c_1, \ldots, c_n \in P_1$ is picked, such that each c_i can be connected to c_{i+1} by some feasible (simple) path, computed by a local planner. In this way P_1 is transformed into a feasible path P_2, that takes into account the non-holonomic constraints. Finally, in Step three, the resulting path P_2 is smoothed. Step two of this method bares some resemblance with the learning phase of our approach: A global method generates a set of free configurations, and tries to connect certain pairs of these configurations by some local planner. The method exploits the fact that a general car-like robot is fully controllable[1]. For non-holonomic robots which are not fully controllable, like for example forward car-like robots, the method cannot be used. Furthermore, the method is not a learning approach. Although some learning method can be used for solving the holonomic problem (step 1), the most time-consuming part of the method is step 2, where holonomic paths are transformed into feasible paths, and such a transformation must be carried out for each query.

Another approach is proposed in [BL93]. It consists of decomposing the configuration space into an array of small rectangloids and heuristically searching a graph whose nodes are these rectangloids. Two rectangloids are adjacent in this graph if there is a feasible path between a configuration lying in the first rectangloid and a configuration lying in the second rectangloid. The method is suitable for both general car-like robots, as well as forward car-like robots. Furthermore, the method is also applicable to multi-body mobile robots. In practice, however, the complexity of the method becomes overwhelming if the number of trailers exceeds one. Another drawback of the method again is the fact that it is a single shot method, unsuitable for solving learning problems.

The random learning paradigm, which forms the basis for the work described in this paper, has recently been applied to a number of robot types, such as free flying robots and articulated robots with many degrees of freedom ([OŠ94],[KŠLO94]). It is related to earlier work that we have done on probabilistic single shot planners for free-flying robots and car-like robots ([Ove92], [Šve93]), as well as a randomized C-space preprocessing method developed by Kavraki and Latombe for high dof articulated robots ([KL94]).

Other, more loosely related, heuristic approaches are described in [MATB92], [ATBM92], and [HST94]. The planners in [MATB92] and [ATBM92] embed genetic algorithms to help the search for paths in high dimensional configuration spaces. In [HST94] a probabilistic planner is proposed that uses random reflections at C-space obstacles. This idea bares some similarity with techniques that we use for connecting configurations to our roadmaps, in the query phase.

[1] A robot is fully controllable iff the existence of a path in the open free C-space implies the existence of a feasible path.

3 The random learning paradigm

The random learning paradigm can be described in general terms, without focussing on any specific robot type. The idea is that during the learning phase a data structure is incrementally constructed in a probabilistic way, and that this data structure is later, in the query phase, used for solving individual motion planning problems.

The data-structure constructed during the learning phase is an undirected graph $G = (V, E)$, where the nodes V are randomly generated free configurations and the edges E correspond to (simple) paths, such that an edge (a, b) corresponds to a feasible path connecting the configurations a and b. These simple paths, which we refer to as *local paths*, are computed by a *local method*, which should be a very simple but fast and deterministic motion planner.

In the query phase, given a start configuration s and a goal configuration g, we try to connect s and g to suitable nodes $\tilde{s}$ and $\tilde{g}$ in V. Then we perform a graph search to find a sequence of edges in E connecting $\tilde{s}$ to $\tilde{g}$, and we transform this sequence into a feasible path in configuration space. So the paths generated in the query phase (which is described in detail later) are basically just concatenations of local paths, and therefore the properties of these "global paths" are induced by the local method. This makes our approach a flexible one.

3.1 The learning phase

We assume that we are dealing with a robot $\mathcal{A}$, and that L is a local method that computes paths feasible for $\mathcal{A}$. L is allowed to fail rather often, but should be able to solve the motion planning problem in easy cases, e.g., in the absence of obstacles (in Section 8 we formulate a general property for local methods which guarantees probabilistic completeness of global planner).

As mentioned above, in the learning phase a probabilistic roadmap is constructed, and stored in an undirected graph $G = (V, E)$. The construction of the probabilistic roadmap is performed incrementally in a probabilistic way. Repeatedly a random free configuration c is generated and added to V. We try to connect each such node c to the graph by adding a number of edges (c, n) to E, such that the local method can connect from c to n.

This edge adding is done as follows: First, a set $N(c)$ of neighbors is chosen from N. This set consists of nodes lying within a certain distance from c, with respect to some metric D. Then, in order of increasing distance from c, we pick nodes from $N(c)$. We try to connect c to each of the selected nodes if it is not already graph-connected to c. Hence, no cycles can be created and the resulting graph is a forest, i.e., a collection of trees. The motivation for preventing cycles is that no query would ever succeed *thanks to* an edge that is part of a cycle. Hence, adding an edge that creates a cycle can impossibly improve the planners performance in the query phase.

A price to be paid for disallowing cycles in the graph is that in the query phase often unnecessarily long paths will be constructed. Suppose that a and b are two configurations that can easily be connected by some short feasible path. Due to the random nature of the learning algorithm, it is very well possible that, at some point, a and b get connected by some very long path. Obtaining a shorter connection between a and b would require the introduction of a cycle in the graph, which we prevent. So, for any pair of nodes, the first graph path connecting them blocks other possibilities.

There are a number of ways for dealing with this problem. One possibility is to apply an edge adding method which does allow cycles in the graph. See [OŠ94], where we discuss a number of such "cycle-allowing" methods. These methods, however, have the disadvantage that they slow down the learning algorithm, due to the fact that the adding of a node requires more executions of the local method to be performed. Another possibility is to build a forest as described above, but, before using the graph for queries, "smoothing" the graph by adding some edges which create cycles. Some experiments that we have done indicated that smoothing the graph for just a few seconds significantly reduces the path lengths in the query phase. Finally, it is possible to apply some smoothing techniques on the paths constructed in the query phase. We briefly describe a simple but efficient probabilistic path smoothing technique in Section 3.2.

Let $\mathcal{C}$ be the configuration space of the robot. To describe the learning algorithm formally, we need the following :

- A symmetrical function $L_d \in \mathcal{C} \times \mathcal{C} \rightarrow \mathit{boolean}$, that returns whether the local method can compute a feasible path for $\mathcal{A}$ between its two argument-configurations. We refer to this function as L's decision function.
- A function $D \in \mathcal{C} \times \mathcal{C} \rightarrow \mathbf{R}^+$. It defines the metric[2] used, and should give a suitable notion of distance for arbitrary pairs of configurations, taking the properties of the robot $\mathcal{A}$ into account. We assume that D is symmetrical.

The algorithm can now be described as follows:

The learning algorithm:

(1) $V = \emptyset$
(2) $E = \emptyset$
(3) **loop**
(4) $\quad c =$ a randomly chosen free configuration
(5) $\quad V = V \cup \{c\}$
(6) $\quad N(c) =$ a set of neighbors of c chosen from V
(7) $\quad$ **forall** $n \in N_c$, in order of increasing $D(c,n)$ **do**
(8) $\quad\quad$ **if** $\neg \mathit{connected}(c,n) \wedge L_d(c,n)$ **then**
(9) $\quad\quad\quad E = E \cup \{(c,n)\}$

The learning method, a described above, leaves a number of choices to be made: A local method must be chosen, a metric must be defined, and it must be defined what the neighbors of a node are. Some choices must be left open as long as we do not focus on a particular robot type, but certain global remarks can be made here.

Local method One of the crucial ingredients in the learning phase is the local method. As mentioned before, the local method must compute paths which are *feasible* for $\mathcal{A}$. Furthermore, the local method should be deterministic. Otherwise the existence of a path in G between two nodes a and b does not guarantee that a feasible path in configuration space connecting a and b can be reconstructed in the query phase.

[2]By metric we simply mean a function of type $\mathcal{C} \times \mathcal{C} \rightarrow \mathbf{R}^+$, without any restrictions.

Another requirement is that the local method always terminates (some potential field methods do not have this property). Finally, the local method should guarantee probabilistic completeness of the learning algorithm. In section 8 we give a sufficient property.

There are still many possible choices for such an algorithm. On one hand one could take a very powerful method. Such a method would very often succeed in finding a feasible path when one exists, and, hence, relatively few nodes would be required in order to obtain a graph which captures the connectivity of the free configuration space well. Such a local method would (probably) be slow, but one could hope that this is compensated by the fact that only a few executions of the method need to be performed. On the other hand, one could choose a very simple and fast algorithm that is much less successful. In this case many more nodes will have to be added in order to obtain a reasonable graph, which means that many more executions of the local method will be required. But this might be compensated by the fact that each execution is very cheap. So it is clear that there is a trade-off, and it is not trivial to make a smart choice here.

We have guided the choice of our local methods by experiments ([Šve93],[Ove92], [Mas92]). These clearly indicated that very fast (and, hence, not very powerful) local methods lead to the best performance of the learning algorithm.

Neighbors and edge adding methods Another important choice to be made is that of the neighbors $N(c)$ of a (new) node c. As is the case for the choice of the local method, the definition of $N(c)$ has large impact on the performance of the learning algorithm. Reasons for this are that the choice of the neighbors strongly influences the overall structure of the graph, and that, regardless of how the local method is exactly defined, the executions of the local method are by far the most time-consuming operations of the learning algorithm (due to the intersection tests that must be performed).

So it is clear that executions of the local method that do not effectively extend the knowledge stored in the roadmap should be avoided as much as possible. Firstly, as mentioned before, attempts to connect to nodes which are already in c's connected component are useless. For this reason the learning algorithm builds a forest. Secondly, local method executions which fail add no knowledge to the roadmap. To avoid too many local method failures we only submit pairs of configurations whose relative distance (with respect to D) is relatively small, that is, less than some constant threshold *maxdist*. Thus:

$$N(c) \subset \{\tilde{c} \in V | D(c, \tilde{c}) \leq maxdist\} \tag{1}$$

This criterion still leaves many possibilities open, regarding the actual choice for $N(c)$. We have decided on taking all nodes within distance *maxdist* as neighbors. Experiments with various definitions for $N(c)$ on a wide range of problems lead to this choice.

Hence, according to the algorithm outline given above, we try to connect to all "nearby" nodes of c, in order of increasing distance D, but we skip those nodes which are already in c's connected component at the time that the connection is to

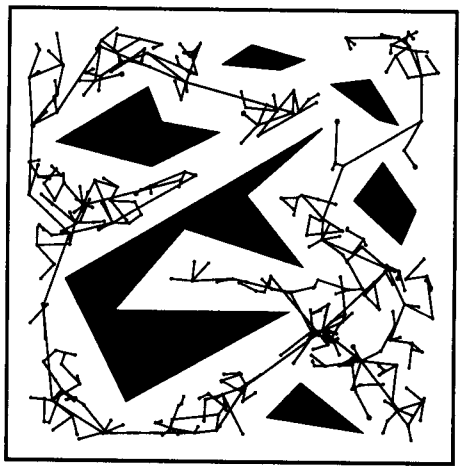


Figure 2: A graph constructed by the forest method (for a general car-like robot).

be attempted. By considering elements of $N(c)$ in this order we expect to maximize the chances of quickly connecting c to other configurations and, consequently, reduce the number of calls to the local method (since every successful connection results in merging two connected components into one). We refer to the described edge adding method as the *forest method.* See Figure 2 for an example of a graph constructed for a general car-like robot by the forest method (the node configurations are projected on $\mathbf{R}^2$).

Distance We have seen that the distance function D is used for choosing and sorting the neighbors $N(c)$ of a new node c. It should be defined in such a way, that $D(a, b)$ (for arbitrary a and b) somehow reflects the chance that the local method will *fail* to compute a feasible path from a to b. For example, given two configurations a and b, a possibility is to define $D(a, b)$ as the size of the sweep volume (in the workspace) obtained when the local method computes a path connecting a to b, in the absence of obstacles. In this way each local method L induces its own metric, which reflects the described "failure-chance" very well. In fact, if the obstacles were randomly distributed points, then this definition would reflect the local method's failure chance exactly. In the general case however, exact computations of the described sweep-volumes tend to be rather expensive, and in practice it turns out that certain rough but cheap to evaluate approximations of the sweep volumes are to be preferred. See Sections 6 and 7 for details on the metrics chosen for car-like robots.

3.2 The query phase

During the query phase, paths are to be found between arbitrary start and goal configurations, using the graph G computed in the learning phase. The idea is that, given a start configuration s and a goal configuration g, we try to find feasible paths P_s and P_g, such that P_s connects s to a graph node $\tilde{s}$, and P_g connects g to a graph node $\tilde{g}$, with $\tilde{s}$ graph-connected to $\tilde{g}$ (that is, they lie in the same connected component of G). If this succeeds, then we compute the shortest path P_G in G connecting $\tilde{s}$ to $\tilde{g}$, and a feasible path in con-

figuration space from s to g is constructed by concatenating P_s, the subpaths computed by the local method when applied to pairs of consecutive nodes in P_G, and P_g reversed. Otherwise, the query fails. The queries should preferably terminate 'instantaneously', so no expensive algorithm is allowed for computing P_s and P_g.

For finding the nodes $\tilde{s}$ and $\tilde{g}$ we use the function *query_mapping* $\in \mathcal{C} \times \mathcal{C} \rightarrow V \times V$, defined as follows :

query_mapping(a, b) $=(\tilde{a}, \tilde{b})$, such that $\tilde{a}$ and $\tilde{b}$ are connected, and
$D(a, \tilde{a}) + D(b, \tilde{b}) = \text{MIN}_{(x,y)\in W} : D(a, x) + D(y, b)$

where $W = \{(x, y) \in V \times V | \textit{connected}(x, y)\}$

So *query_mapping*(a, b) returns the pair of connected graph nodes $(\tilde{a}, \tilde{b})$ which minimize the total distance from a to $\tilde{a}$ and from b to $\tilde{b}$. We will refer to $\tilde{a}$ as a's graph retraction, and to $\tilde{b}$ as b's graph retraction.

The most straightforward way for performing a query with start configuration s and goal configuration g is to compute $(\tilde{s}, \tilde{g}) =$ *query_mapping*(s, g), and to try to connect with the local method from s to $\tilde{s}$ and from $\tilde{g}$ to g. The local method though typically is a rather weak method, and, in unlucky cases, it may fail to find the connections even if the graph captures the connectivity of free C-space well.

Experiments with different robot types indicated that simple probabilistic methods that (repeatedly) perform short random walks from s and g, and try to connect to the graph retractions of the end-points of those walks with the local method, achieve significantly better results than the straightforward method by itself. These random walks should be aimed at maneuvering the robot out of narrow C-space areas (that is, areas where the robot is tightly surrounded by obstacles), and hereby improving the chances for the local method to succeed. For free flying robots and articulated robots, good performance is obtained by what we refer to as the *random bounce walk* (see also [KŠLO94]). The idea is that repeatedly a random direction (in C-space) is chosen, and the robot is moved in this direction until a collision occurs (or time runs out). When a collision occurs, a new random direction is chosen. This method performs much better than for example pure Brownian motion in C-space. In Sections 6 and 7 we describe the methods used for car-like robots, which is of a similar nature.

Paths computed in the query phase can be quite ugly and unnecessarily long. The main reason for this is, as stated before, the fact that the graph is forced to be a forest. Furthermore, the path segments connecting the start and goal configurations of a query to the graph are often, when computed with random-walk techniques as sketched above, ugly as well.

To improve the quality of the paths computed in the query phase, one can apply some path smoothing techniques on these 'ugly' paths. The smoothing routine that we decided on is very simple. It just repeatedly picks a pair of random configurations (c_1, c_2) on the 'to be smoothed' path P_C, tries to connect these with a feasible path Q_{new} using the local method, and if this is successfully accomplished and Q_{new} is shorter than the path segment Q_{old} in P_C from c_1 to c_2, then it replaces Q_{old} by Q_{new} (in P_C). So basically it just tries to replace randomly picked segments of the path by shorter ones, using the local method. The more time is spent on this, the shorter (and nicer) the path gets. Typically, this method produces a nice path within a very short period of time. See Sections 6 and 7 for some car-like paths, smoothed in one second.

4 The random learning paradigm using a directed graph

In the random learning paradigm, as described in the previous section (and in [OŠ94]), the computed roadmaps are stored in undirected graphs. For many motion planning problems this is sufficient, and it appears that the method is easier and more efficient to implement when based on undirected graphs. For example, motion planning problems involving free flying robots, articulated robots, and general car-like robots can all be dealt with using undirected underlying graphs. There are however motion planning problems for which undirected underlying graphs not sufficient, and directed ones are required instead. For example, problems involving forward car-like robots require directed underlying graphs.

We first, in Section 4.1, give a simple criterion for deciding which type of graph is to be used for a particular robot type. Then, in Sections 4.2 and 4.3, we describe the modifications/extensions of the random learning paradigm that are required in the case of directed underlying graphs.

4.1 Symmetrical local methods

The existence of an edge (a, b) in the underlying graph G corresponds to the statement that the local method can compute a feasible path from a to b. If though G is undirected, then the edge contains no information about the direction in which the local method can compute the path, and, hence, it must correspond to the statement that the local method can compute both a feasible path from a to b, as well as one from b to a. So an edge (a, b) can be added only if the local method succeeds in both directions. Doing so, useful information might be thrown away. This will happen in those cases where the local method is successful in exactly one direction, and the fact that it has successfully computed a feasible path will not be stored. If however the local method is *symmetrical*, which means that it succeeds for say (a, b) whenever it succeeds for (b, a), then obviously this problem will never occur. So if the local method is symmetrical, the underlying graph can be undirected, and if it is not symmetrical, then it is better to use a directed graph.

Whether it is possible to implement (good) local methods which are symmetrical, depends on the properties of the robot $\mathcal{A}$, defined by the constraints imposed on it.

Definition 1 *A robot $\mathcal{A}$ has the* reversibility property **iff** *any feasible path for $\mathcal{A}$ remains feasible when reversed.*

Free flying robots and general car-like robots are two examples of robot types which possess the reversibility property, while for example forward car-like robots do not possess the reversibility property. In terms of control theory, a robot has the reversibility property iff its control system is symmetric. That is, it can attain a velocity v (in configuration space) if and only if it can also attain velocity $-v$.

An important observation now is that, if the robot $\mathcal{A}$ has the reversibility property, then any local method L that computes feasible paths for $\mathcal{A}$ can be made symmetrical in a trivial way, by reversing computed paths when necessary. So this implies that if the robot has the reversibility property, then an undirected graph can be used for storing the local paths, and otherwise a directed graph is required.

4.2 The learning phase using a directed graph

Assume now that we are dealing with a robot $\mathcal{A}$ which does not possess the reversibility property. Let again L be a local method for $\mathcal{A}$, and D an associated metric, with the (crucial) difference that both are not symmetrical. We assume that D is defined such that it reflects the failure chance of the local method in a reasonable way.

As pointed out above, the asymmetry of L requires directed graphs for the storage of the the roadmaps. The global structure of the learning algorithm will be the same as in the undirected case, but the key question is how to add (directed) edges in a smart way.

In the previous section, the following criteria for candidate edges were established: (1) they should have a reasonable chance of successfully being added, and (2) they should not be redundant. We dealt with (1) by defining only nearby nodes as neighbors, and attempting connections to them in a sensible order. Criterion (2) was met by preventing cycles in the graph.

For directed graphs we proceed in an analog manner. Regarding the node neighbors, we now cannot do with just one set $N(c)$ of neighbor nodes to which connections are tried. A node can now be connected to other nodes in two different ways. There can be *forward* connections, corresponding to outgoing edges, and there can be *backward* connections, corresponding to incoming edges. So it is clear that, instead of one (general) neighbor set $N(c)$, we now need two neighbor sets: a set of *forward neighbors* $FN(c)$, and a set of *backward neighbors* $BN(c)$. $FN(c)$ will contain the nodes $\in V$ to which forwards connections $(c, *)$ will be tried, and BN(c) will contain the nodes $\in V$ to which backwards connections $(*, c)$ will be tried. Preventing too many failures of the local method can again be achieved by only trying to connect nodes a and b which lie relatively close to each other, that is, within some distance *maxdist* of each other. Thus:

$$FN(c) \subset \{\tilde{c} \in V | D(c, \tilde{c}) \leq maxdist\} \tag{2}$$

$$BN(c) \subset \{\tilde{c} \in V | D(\tilde{c}, c) \leq maxdist\} \tag{3}$$

In an undirected graph an edge is redundant iff it is part of a cycle. When the graph is directed, this is no longer the case. An edge e being redundant intuitively means that it surely will be of no use in the future, in the sense that no conceivable solution of a problem (= a path in the graph between two nodes) will require e. We define the *forward set* of a node c as the set of all nodes which are reachable from c (including c), and we denote it by *forw*(c). Analog, we define the *backward set* of a node c as the set of all nodes from which c is reachable (including c), and we denote it by *backw*(c). We say that the nodes in *forw*(c) are *forwards reachable* from c, and the nodes in *backw*(c) are *backwards reachable* from c. Lemma 1 (which, in an extended version, is proven in [Šve93]) now states that an edge $e = (a, b)$ is redundant iff b is forwards reachable from a even if edge e is removed from the graph:

Lemma 1 *An edge $e = (a, b)$ in a graph $G = (V, E)$ is redundant if and only if $b \in$ forw(a) in the graph $(V, E - \{e\})$.*

Lemma 1, together with (2) and (3), suggests the following algorithm outline:

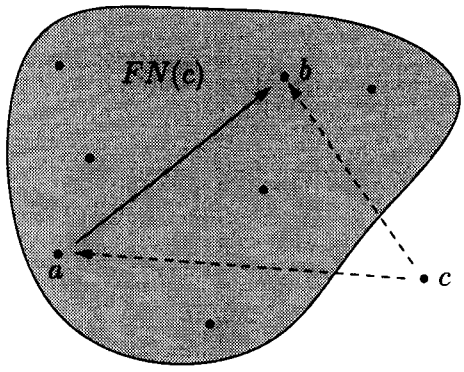


Figure 3: A redundant edge (c, b) has been added.

Directed edge adding outline (c is the new node):

$FN(c)$ = a set of forward neighbors of c chosen from V
$BN(c)$ = a set of backward neighbors of c chosen from V
forall $n \in FN(c)$, in some order **do**
  if $\neg n \in forw(c) \wedge L_d(c, n)$ **then**
    $E = E \cup \{(c, n)\}$
$BN(c)$ = a set of backward neighbors of c chosen from V
forall $n \in BN(c)$, in some order **do**
  if $\neg n \in backw(c) \wedge L_d(n, c)$ **then**
    $E = E \cup \{(n, c)\}$

Recall that in the undirected case the neighbors of a node c consist of all nodes in c's neighborhood, and they are picked in order of increasing distance to c. Unfortunately, this scheme is not feasible for directed graphs, in the sense that it leads to an edge adding algorithm that adds redundant edges. This is illustrated in Figure 3. The nodes a and b are forward neighbors of c, and they are connected by an edge (a, b). Assume that $D(c, b) < D(c, a)$ and that L succeeds to connect to both a and b from c. In that case, both edges will be added, which means that the edge adding algorithm has added the redundant edge (c, b).

To prevent such behavior, we use more constrained neighbor definitions. We define the set of c's forward neighbors $FN(c)$ and the set c's backward neighbors $BN(c)$ as follows:

Definition 2

$$FN(c) = \{n \in V - \{c\} |\ D(c,n) \leq \text{maxdist} \wedge \forall m \in V - \{c\} : n \in \text{forw}(m) \Rightarrow D(c,n) < D(c,m)\}$$

$$BN(c) = \{n \in V - \{c\} |\ D(n,c) \leq \text{maxdist} \wedge \forall m \in V - \{c\} : n \in \text{backw}(m) \Rightarrow D(n,c) < D(m,c)\}$$

According to these definitions, a "nearby" node n is a forward neighbor of c if and only if there exists no node $\tilde{n}$ with $D(c,\tilde{n}) < D(c,n)$ from which n is forwards reachable. Analog, a "nearby" node n is a backward neighbor of c if and only if there exists no node $\tilde{n}$ with $D(\tilde{n},c) < D(n,c)$ from which n is backwards reachable. Picking the nodes from $FN(c)$ in order of decreasing distance from c, and the nodes from $BN(c)$ in order of decreasing distance to c, guarantees that the edge adding algorithm adds no redundant edges, as we will show below. The edge adding algorithm can hence be described as follows:

Directed edge adding algorithm (c is the new node):

compute FN(c) and BN(c)
forall $n \in FN(c)$, in order of decreasing $D(c,n)$ **do**
  if $\neg n \in forw(c) \wedge L_d(c,n)$ **then**
    $E = E \cup \{(c,n)\}$
$BN(c)$ = a set of backward neighbors of c chosen from V
forall $n \in BN(c)$, in order of decreasing $D(n,c)$ **do**
  if $\neg \in backw(c) \wedge L_d(n,c)$ **then**
    $E = E \cup \{(n,c)\}$

This algorithm adds edges with the nice property that none of the edges added will be redundant when it finishes.

Lemma 2 *Let $E(c)$ be a set of edges added by the directed edge adding algorithm (for some new node c) to the graph $G = (V,E)$. Then*

$$\neg\exists e \in E(c) : e \text{ is redundant in } E \cup E(c).$$

Proof
We denote c's outgoing edges by $E_f(c)$, and c's incoming edges by $E_b(c)$ (note that $E(c) = E_f(c) \cup E_b(c)$). Assume that $(c,a) \in E_f(c)$ is redundant in $E \cup E(c)$. So there exists a path in $E \cup E(c)$ connecting c to a, and not containing the edge (c,a). If we consider the shortest path P with these properties, then surely P contains no incoming edges of c. This means that (c,a) is redundant in $E \cup E_f(c)$ also. At the point that (c,a) was added, $a \notin forw(c)$, and, hence, it was not redundant. So an edge (c,b) with $a \in$ forw(b) must have been added later, causing (c,a)'s redundancy. But, due to the fact that the elements of $FN(c)$ are picked in order of decreasing distance from c, this implies that $D(c,a) > D(c,b)$. By definition 2 though, a is not a forward neighbor of c if $a \in forw(b) \wedge D(c,a) > D(c,b)$. So edge (c,a) was never added, and we have a contradiction.

For the case where $e \in E_b(c)$, a contradiction can be shown by an analog argument. □

Consider a graph $G = (V,E)$, with $V = \{c_1, c_2, \ldots, c_n\}$ where c_i is the i^{th} node added, and $E(c_i)$ is the set of edges added to E when c_i was added to V. In the undirected case, E had the property :

$$\forall (c_j,c_k) \in E : (c_j,c_k) \text{ not redundant in } E.$$

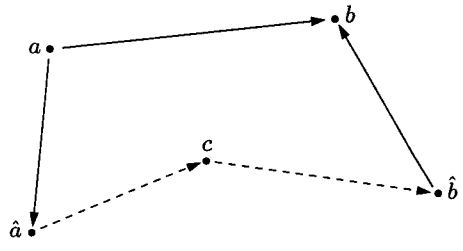


Figure 4: The edge (a, b) becomes redundant due to the adding of the two new (non-redundant) edges $(\hat{a}, c)$ and $(c, \hat{b})$.

We say that E is *globally free of redundancies.* When though we are dealing with a directed graph, and the edges are added by the algorithm described above, only a weaker property holds for E, namely:

$$\forall (c_j, c_k) \in E : (c_j, c_k) \text{ not redundant in } \bigcup_{i=1}^{\max(j,k)} E(c_i).$$

We say that E is *locally free of redundancies.* It is impossible to obtain directed graphs which are globally free of redundancies. This is illustrated in Figure 4. E contains the (non-redundant) edges (a, b),$(a, \hat{a})$, and $(\hat{b}, b)$ at the point where c is added. Now, because $\hat{b}$ is not forward reachable from any other node than itself, surely $\hat{b} \in FN(c)$ and the algorithm will try to connect from c to $\hat{b}$ (assuming that *maxdist* is large enough). Analog, $\hat{a} \in BN(c)$ and the algorithm will try to connect from $\hat{a}$ to c. Now, if both connections succeed, then clearly b will reachable from a by a path not containing (a, b), and, hence, (a, b) will be redundant. Could this have been prevented? At the point where (a, b) was added, there was no way to predict its future redundancy, because it is caused by a *randomly* chosen future node. This shows that obtaining an edge set that is locally free of redundancies is the best we can achieve. (Of course, it is useless to remove redundant edges later, because the time on computing them has already been spent.)

4.3 The query phase using a directed graph

As in the undirected case, during the query phase paths connecting arbitrary start and goal configurations are to be found, using the graph constructed in the learning phase. This can be done using a scheme very similar to that proposed in Section 3.2 for the undirected case.

The only difference is that the random walks performed from the goal node g must be feasible *when reversed* (recall that the robot does not possess the reversibility property). Furthermore, due to the fact that, for directed graphs, node-connectivity is not symmetric, the computation of the "query mappings" is more complicated than for undirected graphs. In an undirected graph, if $(\tilde{s}, \tilde{g})$ is the query mapping of (s, g), then $\tilde{s}$ and $\tilde{g}$ lie in the same connected component $(\tilde{V}, \tilde{E})$ of G. Furthermore, $\tilde{s}$ is the nearest node to s in $\tilde{V}$, and $\tilde{g}$ is the nearest node to g in $\tilde{V}$. In a directed graph though the nodes can (in general) no longer be partitioned into disjunct sets within which all nodes are (symmetrically)

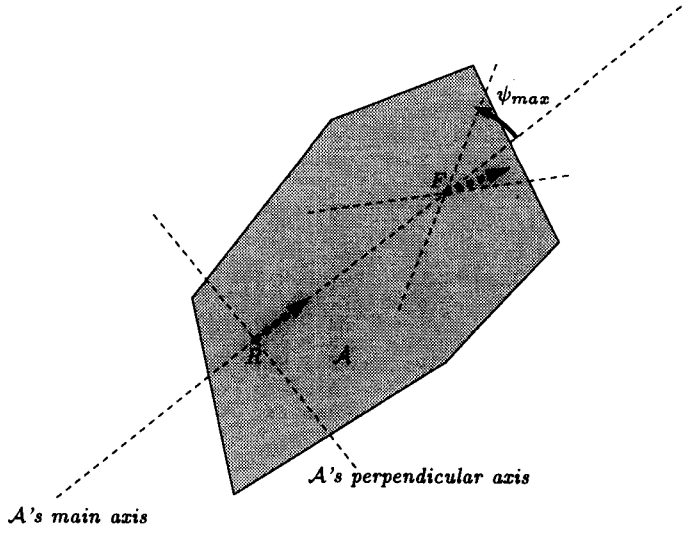


Figure 5: An abstract car-like robot $\mathcal{A}$. The thick dashed arrows show feasible velocity directions for the rear point R and the front point F.

connected. This makes the computation of the "query mappings" somewhat more time consuming.

The paths constructed in the query phase can again be smoothed by the same probabilistic algorithm as used in the undirected case (see Section 3.2), which repeatedly tries to replace randomly picked segments of the path by shorter ones, computed by the local method.

5 Car-like robots

In the rest of this paper, we apply the described motion planning techniques to car-like robots. For this, we now formally define car-like robots, and we introduce a few useful notations (This definition is partially taken from [Lat91]).

A car-like robot is a solid planar object (we use a polygon) that moves in $\mathbf{R}^2$. It has 3 degrees of freedom, and its configuration space can be represented by $\mathbf{R}^2 \times [0, 2\pi[$. The robots possible motions are restricted by certain non-holonomic constraints, that is, constraints on its achievable velocities.

A car-like robot $\mathcal{A}$ has defined a *rear point* R, a *front point* F, and a *maximal steering angle* ψ_{max}. R and F are points fixed to $\mathcal{A}$, and ψ_{max} is a positive angle less than $\frac{1}{2}\pi$. Furthermore we refer to the line through R and F as $\mathcal{A}$'s *main axis*, and to the line through R and perpendicular to $\mathcal{A}$'s main axis as $\mathcal{A}$'s *perpendicular axis*. A configuration (x, y, θ) corresponds to a positioning of $\mathcal{A}$ such that R coincides with (x, y) and the angle that the vector $F - R$ makes with the positive x-axis is θ.

Suppose that $\mathcal{A}$ is a general car-like robot (it can move both forwards and backwards). Then exactly those velocities of $\mathcal{A}$ are possible, where the direction of R's velocity v points along $\mathcal{A}$'s main axis, and the angle ψ that F's velocity makes with $\mathcal{A}$'s main axis lies in $[-\psi_{max}, \ldots, \psi_{max}]$. See also Figure 5. We refer to v as the *drive velocity* of $\mathcal{A}$, to ψ as $\mathcal{A}$'s

steering angle, and to a pair $(\psi, v) \in [-\psi_{max}, \psi_{max}] \times \mathbf{R}$ as a *control* of the robot. Positive values of v describe robot velocities where R's velocity points towards F, and we refer to the corresponding motions as *forward motions* of $\mathcal{A}$. Analog, negative values of v describe robot velocities where R's velocity points away from F, and we refer to the corresponding motions as *backward motions* of $\mathcal{A}$.

A feasible motion of a car-like robot with constant steering angle ψ is either a *translational* motion along its main axis ($\psi = 0$), or a *rotational* one around a point on $\mathcal{A}$'s perpendicular axis ($\psi \neq 0$). The radius of such a rotational motion, defined as the distance between R and the center of rotation, is induced by the steering angle ψ. $\mathcal{A}$'s *minimal turning radius* r_{min} is the rotation radius induced by $\psi = \psi_{max}$. We refer to paths describing translational motions (with $\psi = 0$) as *translational paths*, and to paths describing rotational motions (with ψ constant and $\neq 0$) as *rotational paths*. If a path describes a rotational motion with $|\psi| = \psi_{max}$, then we refer to it as an *extreme rotational path*.

The paths constructed by our planner will be sequences of translational paths and extreme rotational paths only. It is a well-known fact ([Lat91]) that if, for a general car-like robot, a feasible path in the open free C-space exists between two configurations, then there also exists one which is a (finite) sequence of extreme rotational paths. We include translational paths to enable straight motions of the robot, hence reducing the path lengths.

Now suppose that $\mathcal{A}$ is a forward car-like robot, that is, one that can only move forwards. In this case its possible velocities are further constrained: $\mathcal{A}$'s drive velocity must be positive. So only such translational and rotational paths are feasible which describe forward motions of the robot. We refer to such paths as *forward translational/rotational paths*. For forward car-like robots, our planner will construct paths that are concatenations of forward translational paths and extreme forward rotational paths only. Again, this does not restrict the set of solvable queries.

6 Application to general car-like robots

We will now apply the random learning paradigm, using an undirected graph, to general car-like robots, as defined in Section 5. This asks for filling in some of the (robot specific) details which we have left open in the discussion of the general method. This is done in Section 6.1. In Section 6.2 we introduce a non-random node adding strategy, which uses the geometry of both the robot and the workspace, and in many cases improves the performance of the global method, as we shall see later. Then, in Section 6.3, we describe some experiments that we have done with an implementation of the method, and we give experimental results.

6.1 Filling in the details

The local method: We have done a large number of experiments with different types of local methods (see [Šve93]). For example, we have experimented with some simple potential field methods. This was motivated by the fact that for free flying planar robots simple potential field methods seemed to give the best results (see [Ove92],[Mas92]). A problem with car-like robots however is, that locally they can only move approximately along their main axis, which makes it hard to 'slide' them along obstacle boundaries. So potential fields should be used which keep them at

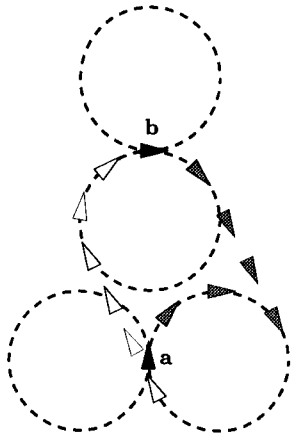


Figure 6: Two RTR paths for a triangular car-like robot, connecting configurations a and b.

some distance from the obstacles, but such potential fields turn out to be much too expensive to evaluate for our purposes. As a result, we decided on a very simple local method, which constructs a path connecting its argument configurations in the absence of obstacles, and then tests whether this path intersects any obstacles.

This local method makes use of the primitives introduced in Section 5. A *RTR path* is defined as the concatenation of an extreme rotational path, a translational path, and another extreme rotational path. With this path construct we define the *RTR local method*: Given two argument configurations a and b, if the shortest RTR path connecting a to b intersects no obstacles, then the method succeeds and returns this path, and otherwise failure is returned. Figure 6 shows two RTR paths. It can easily be proven that any pair of configurations is connected by a number of RTR paths (See [Šve93] for more details).

An alternative to the RTR local method is to use a local method which constructs and tests the shortest (car-like) path connecting its argument configurations ([RS90], [SL92]). Constructing shortest car-like paths is however a rather expensive operation, and the construct requires more expensive intersection checking routines than does the RTR construct. On the other hand, RTR paths will, in general, be somewhat longer than the shortest paths, and, hence, they have a higher chance of intersection with the obstacles. However, the (slightly) higher failure chance of the RTR local method does not outweigh the outlined advantages.

Collision checking for a RTR path can be done very efficiently, performing three intersection tests for translational and rotational sweep volumes. These sweep volumes are bounded by linear and circular segments (such objects are called *generalized polygons* in [Lat91]) and hence the intersection tests can be done exactly and efficiently. Moreover, the intersection tests for the rotational path segments can be eliminated by storing some extra information in the graph nodes, hence reducing the collision check of a RTR path to a simple intersection test for a polygon.

The metric: We use a metric, which is induced by the RTR local method, and which can be regarded as an approximation of the (too expensive) induced "sweep volume metric", as described in Section 3.1. The distance between two configurations is defined as the length (in workspace) of the shortest RTR path connecting them. We refer to this metric as the *RTR metric*, and we denote it by D_{RTR}.

Although the evaluation of the D_{RTR} metric is quite cheap, the total number of distance computations performed for the construction of a graph (V, E) is quadratic in $|V|$. Hence, for large graphs, the time consumed by distance computations can be significant. To avoid this, we have implemented a simple optimization, which we refer to as the *metric grid optimization*. The idea is that we define a grid over $[-1,1] \times [-1,1] \times [0,2\pi[$, and that with each grid-point c (which is a configuration) we store the RTR distance from $(0,0,0)$ to c, that is, $D_{RTR}((0,0,0),c)$. Then, during the learning phase, we index this grid to get (an approximation of) the RTR distance between arbitrary configurations in $[0,1] \times [0,1] \times [0,2\pi[$ (which will be the actual configuration spaces of our scenes). The underlying trick is that the RTR distance between configurations $a = (x_a, y_a, \theta_a)$ and $b = (x_b, y_b, \theta_b)$ is equal to the RTR distance between $(0,0,0$ and $(x_p, y_p, \theta_b - \theta_a)$, where (x_p, y_p) is $(b_x - a_x, b_y - a_y)$ rotated around $(0,0)$ over an angle of $-\theta_a$.

The resolution does not have to be very high in order for the approximations to be good. For example, we use a grid of resolution $20 \times 20 \times 20$, and the error is typically less than 0.04 (for configurations in $[0,1] \times [0,1] \times [0,2\pi[$ and a minimal turning radius of 0.1). The computation of a $20 \times 20 \times 20$ metric grid is quite cheap (it takes us about one second). Moreover, this grid depends only on the robots minimal turning radius, and, hence, metric grids can be precomputed and stored for the minimal turning radii one is interested in.

The random walks in the query phase: In Section 3.2 we have described a general scheme for solving a query using a graph constructed in the learning phase. Multiple random walks were performed from the query configurations s and g, aimed at connecting the end-points of these walks to their graph retractions with the local method. Remains to define the random walks that we use for (general) car-like robots. The (maximal) number of these walks (per query) and their (maximal) lengths are parameters of the method, which we denote by, respectively, N_W and L_W.

Let c_s be the start configuration of a random walk. As mentioned above, the parameter L_W defines the maximal length of the walk. As actual length l_W of the walk we take a random value from $[0, L_W]$. The random walk is now performed in the following way: First, the robot is placed at configuration c_s, and a random control (ψ, v) is picked from the discrete set $\{-\psi_{max}, \psi_{max}\} \times \{-1, 1\}$. Then, the motion defined by (ψ, v) is performed until either a collision of the robot with an obstacle occurs, or the total length of the random walk has reached l_W. In the former case, a new random control is picked, and the process is repeated. In the latter case, the random walk ends.

Good values for N_W and L_W must be experimentally derived (the values we use are given in Section 6.3).

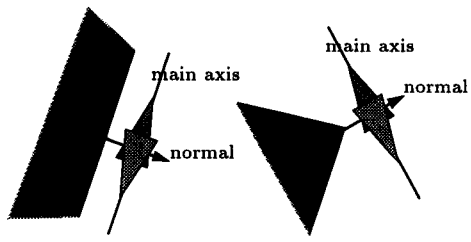


Figure 7: Geometrically defined configurations, for a triangular robot.

6.2 Guiding the node adding by the geometry of the workspace

Many practical scenes (partially) consist of corridors and rooms. In such cases one can boost the planners performance by adding configurations at important positions, in particular along edges of obstacles (along walls) and next to vertices of obstacles (to facilitate the robot to move around corners). The *geometric node adding strategy* implements this idea.

First, a set of geometric configurations V_g, defined by the workspace obstacles, is computed. Each edge and convex vertex of a workspace-obstacle defines two such geometric configurations. The two configurations defined by a convex obstacle-vertex v correspond to placements of the robot $\mathcal{A}$ on the outer side of v, such that $\mathcal{A}$'s main axis is perpendicular to v's outer normal, and the (workspace) clearance between $\mathcal{A}$ and v is ϵ (some small real constant). Analog, the two configurations defined by an obstacle-edge e correspond to placements of $\mathcal{A}$ on the outer side of e, with $\mathcal{A}$'s main axis perpendicular to e's outer normal, and a clearance of ϵ between $\mathcal{A}$ and e. See Figure 7 for an example.

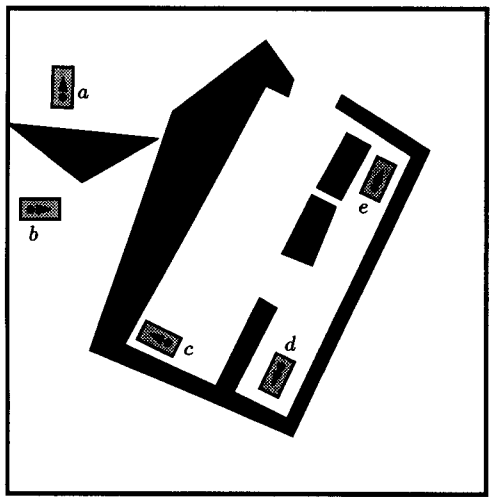
After V_G has been computed, we add the configurations from V_G in a random order to the graph, but we discard those which are not free. Once all nodes from V_G have been picked, the learning process can be continued by adding random nodes.

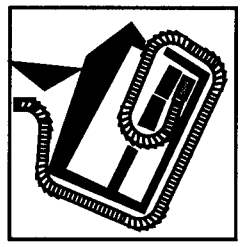
6.3 Experimental results

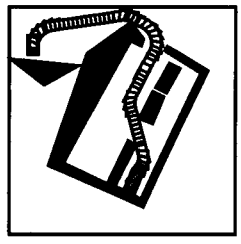
We now present a number of experimental results for general car-like robots, obtained by applying our learning approach to a number of different scenes. The method is implemented in C++ and the tests were performed on a Silicon Graphics Indigo^2 workstation with an R4400 processor running at 150 MHZ. This machine is rated with 96.5 SPECfp92 and 90.4 SPECint92.

In the test scenes that are used, the coordinates of all workspace obstacles lie in the unit square, and all node configurations are chosen such that they project on this unit square. Paths between configurations are allowed to run outside this square, but we prevented this in the test scenes by adding a small obstacle boundary around it.

The experiments are aimed at measuring the "knowledge" acquired by the method after having learned for certain periods of time. This is done by testing how well the method solves certain (interesting) queries. For each scene S we define a *query test set* $T_Q = \{(s_1, s_1), (s_2, g_2), \ldots, (s_m, g_m)\}$, consisting of a number of configuration pairs (that is, queries). Then, we repeatedly build a graph by learning for some specified time t, and we count how many of these graphs solve the different queries in T_Q. This experiment is







	0.1 sec.	0.2 sec.	0.3 sec.	0.4 sec.	0.5 sec.
(a,b)	20%				
(a,d)	35%				
(b,e)	15%				
(c,e)	80%				
(d,e)	90%				

	1 sec.	2 sec.	3 sec.	4 sec.	5 sec.
(a,b)					
(a,d)	40%				
(b,e)	45%				
(c,e)	30%				
(d,e)					

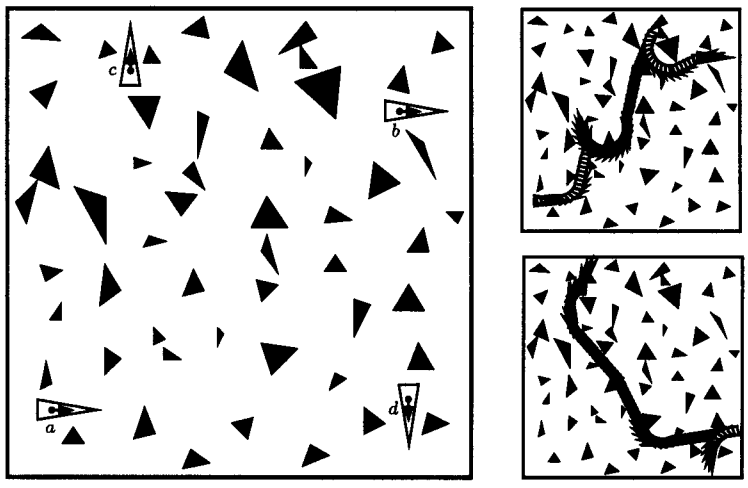
Figure 8: Scene 1, and its configuration test set. At the top right, two paths computed by the planner and smoothed in 1 second are shown. The lower left table gives results for geometric node adding, and the table at the lower right for random node adding.

repeated for a number of different learning times t.

The values for the random walk parameters N_W and L_W are, respectively, 10 and 0.05. This guarantees that the time spent per query is bounded by approximately 0.3 seconds (on our machine). Clearly, if we allow more time per query, the method will be more successful in the query phase, and vice versa. So there is a trade-off between the learning time and the time allowed for a query. We give results for both random node adding, as well as for geometric node adding. The geometric adding is implemented such that at the point where all geometric nodes have been added, the algorithm switches to random adding (unless the learning time has run out).

Scene 1 is a relatively easy scene. It is shown, together with the robot $\mathcal{A}$ positioned at a set of configurations $\{a, b, c, d, e\}$, in Figure 8. The topology is simple and there are only a few narrow passages. Furthermore, the scene is "corridor-like", that is, most useful paths consist of motions parallel to obstacle edges and curves around (convex) obstacle vertices. Hence, it typically is a type of scene for which geometric node adding was envisaged. As query test set T_Q we use $\{(a,b),(a,d),(b,e),(c,e),(d,e)\}$ (At the top-right of Figure 8 paths solving the queries (a,d) and (b,e), smoothed in 1 second, are shown). The minimal turning radius r_{min} used in the experiments is 0.1, and the neighborhood size *maxdist* is 0.5.

The bottom-left table in Figure 8 gives results for geometric node adding. For each of



	0.25 sec.	0.5 sec.	0.75 sec.	1.0 sec.	1.5 sec.	2.0 sec.
(a,b)	30%					
(a,c)	20%					
(a,d)						
(c,d)	35%					

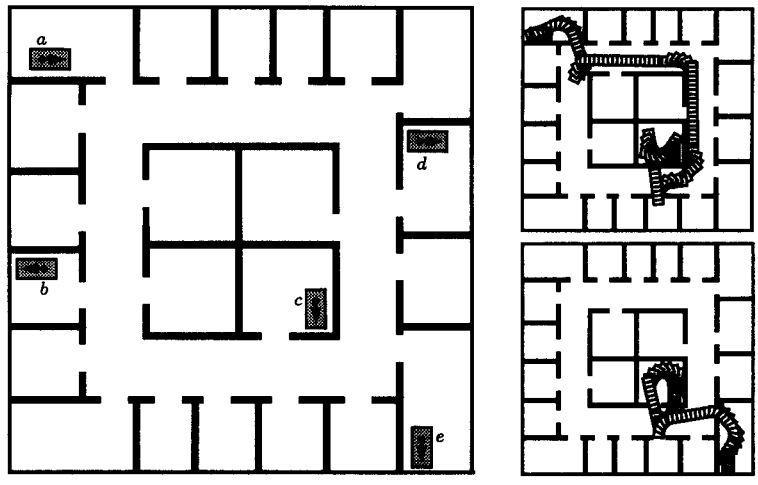
	0.25 sec.	0.5 sec.	0.75 sec.	1.0 sec.	1.5 sec.	2.0 sec.
(a,b)	30%					
(a,c)	15%					
(a,d)	25%					
(c,d)	15%	35%				

Figure 9: Scene 2, and its configuration test set. At the top right, two paths computed by the planner and smoothed in 1 second are shown. The lower left table gives results for geometric node adding, and the table at the lower right for random node adding.

the queries q and for a number of different learning times t it is shown how many percent of 20 (independently) in time t generated networks solve query q. We see that after only 0.3 seconds of learning, the constructed networks solve each of the queries in most cases (but not all). Half a second of learning is sufficient for solving each of the queries, in all 20 trials. The bottom-right table gives the results obtained with random node adding. We see that in this case the learning process is much slower than in the case of geometric adding. 4 seconds are required to obtain networks that solve the queries in all cases.

So there is a great difference in performance between the two node adding strategies. The spectacular performance of the geometric adding (with respect to random node adding) in Scene 1 is caused by the fact that the graph constructed from solely geometric nodes appears to capture most of the connectivity of the free C-space by itself. It takes about 0.2 seconds to add all the geometric nodes, and we see (in Figure 8) that the queries are mostly solved (in about 80% of the cases) by networks constructed in just this time.

Scene 2, which is shown in Figure 9 (again together with a robot $\mathcal{A}$ placed at different configurations $\{a, b, c, d\}$) is a completely different type of scene. It contains many (small) obstacles and is not at all "corridor-like". Although many individual motion planning problems in this scene are quite simple, the topology of the free C-space is quite complicated, and can only be captured well in relatively complicated graphs. As query test set T_Q we use $\{(a, b), (a, c), (a, d), (c, d)\}$. Furthermore, as in the previous scene, $r_{min} = 0.1$ and $maxdist = 0.5$. Again, we show two (smoothed) paths computed by our planner (solving the queries (a, b) and (c, d)).



	1.0 sec.	2.0 sec.	3.0 sec.	4.0 sec.	5.0 sec.	7.5 sec.
(a, c)	15%	35%				
(a, d)	40%					
(b, c)	30%					
(b, d)	25%					
(c, e)	30%					

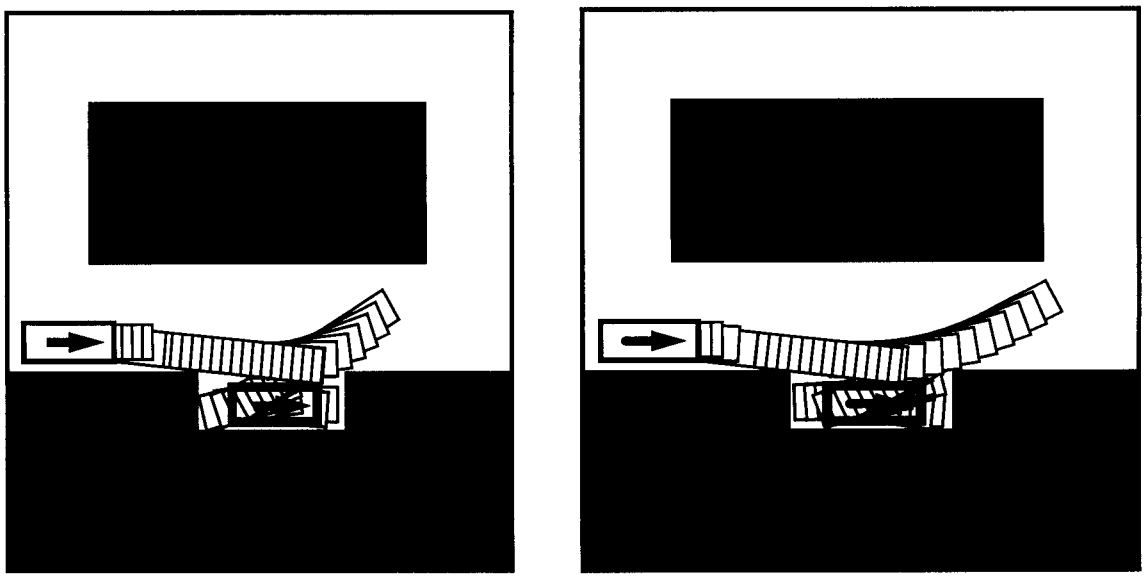
	1.0 sec.	2.0 sec.	3.0 sec.	4.0 sec.	5.0 sec.	7.5 sec.
(a, c)	15	20%	40%			
(a, d)	25%	40%	45%			
(b, c)	25%	25%				
(b, d)	20%	40%				
(c, e)	10%	45%				

Figure 10: Scene 3, and its configuration test set. At the top right, two paths computed by the planner and smoothed in 1 second are shown. The lower left table gives results for geometric node adding, and the table at the lower right for random node adding.

The experimental results obtained with geometric node adding are given in the bottom-left table of Figure 9, and those obtained with random node adding in the bottom-right table. We see that in both cases about 2 seconds of learning are required to obtain networks which are (almost) guaranteed to solve each of the queries. Geometric adding does not help much here, which is not a great surprise, due to the "chaotic" structure of the scene. Only for relatively small learning times does geometric adding give a significant improvement. This is caused by the fact that the many obstacle vertices and edges are distributed rather uniformly over the workspace, and, hence, geometric adding (quickly) generates a set of nodes that covers the free C-space in a fairly uniform way. This makes it possible for the method to (with some luck) capture much of free C-space connectivity very early.

Scene 3 is another "corridor-like" scene, consisting of 22 "rooms" and a "hallway" (see Figure 10). The main difficulty is the large number of narrow passages, connecting the rooms to the hallway. The rectangular robot can only just pass through. Also it is difficult for the robot to reverse its orientation when in the hallway. As query test set we take $\{(a, c), (a, d), (b, c), (b, d), (c, e)\}$. We use a smaller minimal turning radius than in the two previous scenes, namely $r_{min} = 0.05$. The size of a node's neighborhood (defined by *maxdist*) is again 0.5.

We again give experimental results for both geometric node adding and random node adding. We see that geometric adding beats random adding in performance. The difference between the two strategies though is not as spectacular as it was the case for Scene 1. The



	0.25 sec.	0.5 sec.	0.75 sec.	1.0 sec.	1.5 sec.	2.0 sec.
$r_{min} = 0.25$	60%	65%	90%	90%		
$r_{min} = 0.5$	30%	60%	70%	75%	90%	

Figure 11: Scene 4. Parking with large minimal turning radii. In the left case r_{min} is 0.25, and in the right case 0.5. Results are shown in the table.

reason for this is that the graphs constructed from solely geometric nodes typically do not capture the connectivity of the free C-space well. The geometric phase during geometric adding takes about 1.5 seconds, and we see (in the bottom-left table of Figure 10) that at this point the queries are often not yet solved. Hence, subsequent to the geometric nodes still a large number of random nodes need to be added to obtain the required connectivity.

The final scene corresponds to the problem of parking a car (Figure 11). We test how often the problem is solved over 20 independently constructed graphs, for a number of different learning times and two different turning radii. The left picture in Figure 11 shows a path solving the problem for $r_{min} = 0.25$, and the right one shows a path solving the problem for $r_{min} = 0.5$. As can be seen, in the latter case an extra reversal is required. We only use random node adding (geometric node adding does not help here) and again $maxdist = 0.5$.

See the table in Figure 11 for the results. The first row corresponds to the case where $r_{min} = 0.25$ and the second row to the case where $r_{min} = 0.5$. So after 1.5 seconds of learning the problem is always solved if $r_{min} = 0.25$, and if $r_{min} = 0.5$ it takes 2 seconds.

7 Application to forward car-like robots

Forward car-like robots are those that can only move forwards (as described in Section 5), and ask for roadmaps in the form of directed underlying graphs. So we must use the

method described in Section 4. In Section 7.1 we fill in some (robot specific) details, such as the local method, the metric, and the random walks in the query phase. In Section 7.2 we consider the possibility of guiding the node adding by the geometry of the workspace. Then, in Section 7.3, we give some experimental results, in the same form as those given for general car-like robots.

7.1 Filling in the details

The local method: We use a local method that is very similar to the RTR local method used for general car-like robots. An *RTR forward path* is defined as the concatenation of extreme forward rotational path, a forward translational path, and another extreme forward rotational path (see Section 5). This construct defines the *RTR forward local method* as follows: Given two argument configurations a and b, if the shortest RTR forward path connecting a to b intersects no obstacles, then the method succeeds and returns this path, and otherwise failure is returned. As is the case for regular RTR paths, any pair of configurations is connected by a number of RTR forward paths. Furthermore, the RTR forward construct allows for highly optimized collision checking routines (See [Šve93] for more details on RTR forward paths).

The metric: We use the *RTR forward metric.* This metric is induced by the RTR forward local method in the same way as the *RTR metric* is by the RTR local method. The distance between two configurations is defined as the length (in workspace) of the shortest forward RTR path connecting them. Furthermore, we apply the metric grid optimization, described in Section 6.1.

The random walks in the query phase: As mentioned in Section 4.3, we need to define two types of random walks: random walks which are feasible themselves (for the random walks from the start configuration), and random walks which are feasible when reversed (for the random walks from the goal configuration). We refer to the former ones as *forward random walks*, and to the latter as *backward random walks.*

The forward random walks are defined similarly to the random walks described in Section 6.1. The main difference is that the random controls (ψ, v) are picked from $\{-\psi_{max}, \psi_{max}\} \times \{1\}$ instead of $\{-\psi_{max}, \psi_{max}\} \times \{-1, 1\}$. Analog, the random controls for the backward random walks are picked from $\{-\psi_{max}, \psi_{max}\} \times \{-1\}$.

7.2 Guiding the node adding by the geometry of the workspace

The geometric node adding strategy is directly applicable to forward car-like robots, and, in Section 7.3, we give some experimental results obtained with this strategy.

We now describe a method which, using the geometry of the workspace, prevents the adding of certain nodes which are unlikely to be of use in the query phase. We use this method in the experiments described later.

Given an obstacle B, we define a free configuration c to be *forwards blocked by* B iff any forward motion starting at c will eventually cause a collision with B, and we define it to be *backwards blocked by* B iff any backwards motion starting at c will eventually cause a collision with B. We say c is *blocked by* B, or *B-blocked*, iff it is forwards blocked by B or backwards blocked by B. Furthermore, we refer to the set of all configurations that are blocked by an obstacle B as the *area blocked by* B, and we denote it by $\Upsilon(B)$. The key

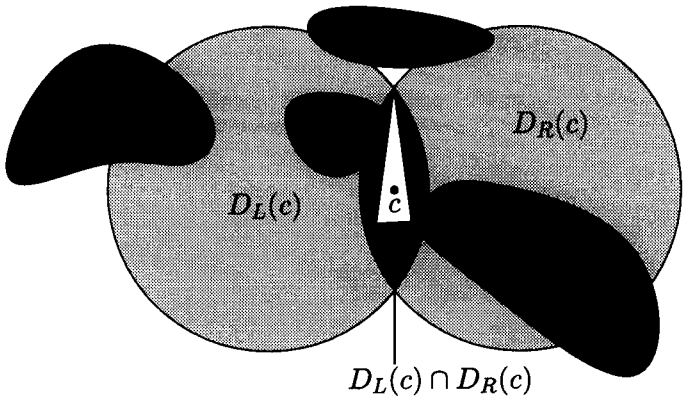


Figure 12: Blocked configurations. Obstacle B_1 intersects both $D_L(c) - D_R(c)$ and $D_R(c) - D_L(c)$, so c is B_1-blocked. Obstacle B_2 does not intersect both $D_L(c) - D_R(c)$ and $D_R(c) - D_L(c)$, but it does intersect $D_L(c) \cap D_R(c)$. The same is the case for obstacle B_3. Finding out whether c is blocked by B_2 (respectively B_3) requires further analysis of $B_2 \cap D_L(c) \cap D_R(c)$ (respectively $B_3 \cap D_L(c) \cap D_R(c)$). It turns out that c is B_2-blocked, but not B_3-blocked. Obstacle B_4 does not intersect $D_R(c)$, so surely c is not B_4-blocked.

observation now is, that if $x \in \Upsilon(B)$ for some $B \in \beta$, then x cannot be part of a feasible path between two configurations which are both not blocked by B. In other words, x cannot contribute to capturing the connectivity of $\mathcal{C}_f - \Upsilon(B)$ in a roadmap for forward car-like robots. So adding x to the graph can only be of use for capturing the connectivity of $\Upsilon(B)$. Now if we assume that the random walk methods performed in the query phase are capable solving all (sub)problems in the $\Upsilon(B)$ areas (for each $B \in \beta$), then clearly we are allowed to discard all nodes which are blocked by an obstacle B.

Whether this assumption holds depends on the complexity of the workspace obstacles and the robot. Experiments have indicated that if all workspace obstacles as well as the robot are convex, then the assumption typically holds. This suggests the following optimalisation: First, partition the workspace obstacles into convex pieces. Then, during the learning phase, add only those configurations to the graph which are not blocked by any obstacle B (assume here that the robot is convex).

There remains the question how to (efficiently) test whether a configuration is blocked a convex obstacle B. Lemma 3 gives an exact criterion (see also Figure 12):

Lemma 3 *Let c be an arbitrary free configuration and B a convex obstacle. Let $D_R(c)$ be the (smallest) disc containing the robots sweep volume when it performs an extreme right rotational motion, over an angle of 2π, and let, analog, $D_L(c)$ be the (smallest) disc containing the robots sweep volume when it performs an extreme left rotational motion, over an angle of 2π. Then $c \in \Upsilon(B)$ if and only if*

$$B \cap (D_L(c) - D_R(c)) \neq \emptyset \wedge B \cap (D_R(c) - D_L(c)) \neq \emptyset$$

$$\vee$$

$$c \in \Upsilon(B \cap D_L(c) \cap D_R(c))$$

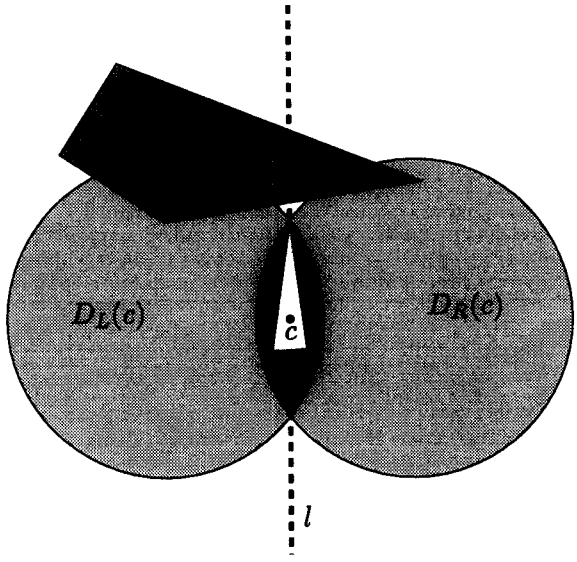


Figure 13: Edge e forward blocks the robot.

Proof
Let B be a convex obstacle and c a free configuration. We denote the portion of B which lies in $D_L(c) \cap D_R(c)$ by $\check{B}$, and the portion lying outside $D_L(c) \cap D_R(c)$ by $\hat{B}$.

1. Assume that the disjunct is not true. So assume, without loss of generality, that (1) $B \cap (D_L(c) - D_R(c)) = \emptyset$ and (2) $c \notin \Upsilon(\check{B})$. The robot can perform an infinite (extreme) left rotational motion, without colliding with $\hat{B}$, due to (1). Hence, $c \notin \Upsilon(\hat{B})$. Together with (2), this proves that $c \notin \Upsilon(\hat{B} \cup \check{B}) = \Upsilon(B)$.

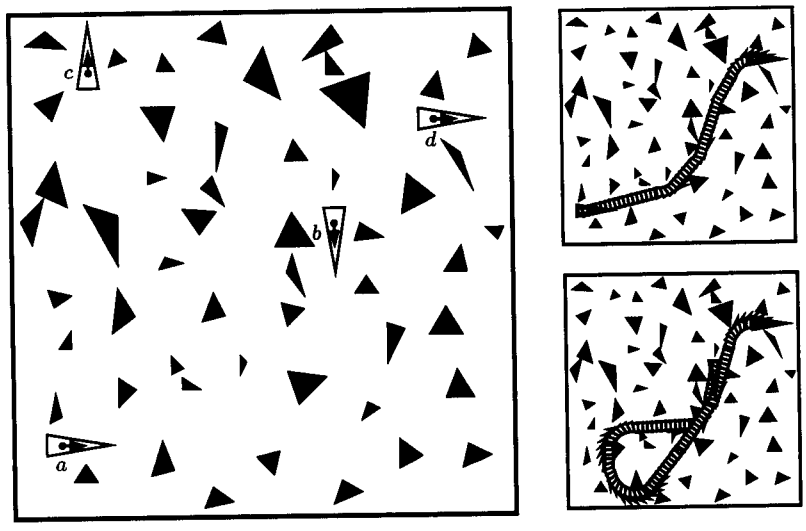
2. Now assume that the disjunct is true.

   - First let us assume the case $B \cap (D_L(c) - D_R(c)) \neq \emptyset \wedge B \cap (D_R(c) - D_L(c)) \neq \emptyset$. Let p_l be a point lying in the intersection of B and $(D_L(c) - D_R(c))$, and let p_r be a point lying in the intersection of B and $(D_R(c) - D_L(c))$. Let e be the line segment connecting p_l and p_r, and let l be the main axis of the robot (see also section 5), when positioned at configuration c. Clearly, l and e intersect. Due to the convexity of B, $e \subset B$. If e intersects l "in front of" c, than any forward motion will (eventually) cause the robot to intersect e, and, hence, also B. Analog, if e intersects l "behind" c, than any backward motion will (eventually) cause collision with B. Hence, $c \in \Upsilon(B)$. See also figure 13.
   - $c \in \Upsilon(\check{B})$ directly implies $c \in \Upsilon(B)$.

   It follows that $c \in \Upsilon(B)$.

The above proves the lemma. □

We use the criterion of Lemma 3 in an inexact way (for efficiency reasons). When a new random configuration has been generated, we only test for every obstacle B whether



	1.0 sec.	2.0 sec.	3.0 sec.	4.0 sec.	5.0 sec.	7.5 sec.
(a, c)	35%	40%				
(a, d)	30%					
(b, c)	15%	35%				
(b, d)	20%	45%	45%			

Figure 14: Scene 2, and its configuration test set. At the top right, two paths computed by the planner and smoothed in 1 second are shown. The table gives results for random node adding.

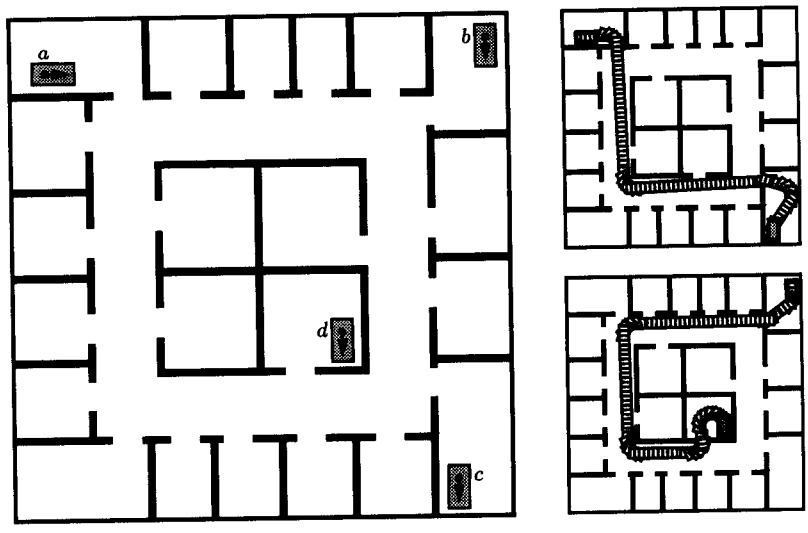
$B \cap (D_L(c) - D_R(c)) \neq \emptyset \wedge B \cap (D_L(c) - D_R(c)) \neq \emptyset$. If so, we know that c is blocked by B, and it is discarded. If not, we conclude that it is not blocked by B, and it is kept. The latter conclusion may be wrong, due to the fact that we do not test whether c is blocked by $B \cap D_L(c) \cap D_R(c)$. Hence, some blocked configurations will pass the test and be added to the graph (see also Figure 12). However, there number typically is low, and it does not pay-off to use a more expensive test in order to filter out all blocked nodes.

Experiments indicate that filtering out blocked nodes, as described above, speeds up the learning process by up to 40%.

7.3 Experimental results

We have performed the same type of experiments for forward car-like robots, as described in Section 6.3. We give results for Scene 2 and Scene 3. See Figures 14 and 15. In both scenes we use four different queries (a, c),(a, d),(b, c), and (b, d) to test the performance of the method after having learned for certain periods of time. In Scene 2 we use random node adding (geometric adding does not help), while in Scene 3 we use geometric adding. Furthermore, we do not add nodes which are blocked by an (convex) obstacle. The used values for *maxdist* and r_{min} in Scene 2 are, respectively, 0.5 and 0.1, and in Scene 3, respectively, 0.5 and 0.05.

We see that in Scene 2 the queries are most likely to be solved after 5 seconds of learning, and almost surely after 7.5 seconds. In Scene 3 the learning process appears to



	2 sec.	4 sec.	6 sec.	8 sec.	10 sec.	15 sec.
(a,c)	40%					
(a,d)	20%					
(b,c)	20%	30%				
(b,d)	40%					

Figure 15: Scene 3, and its configuration test set. At the top right, two paths computed by the planner and smoothed in 1 second are shown. The table gives results for geometric node adding.

be slower by roughly a factor 2.

8 Probabilistic completeness

The experimental results, presented in the previous sections, clearly confirm that the more time is spent on learning, the more queries are solved in the query phase. In this section we will formulate a general property for symmetrical local methods (for robots with the reversibility property), which guarantees that *any* query which is solvable in the open free C-space (that is, no contacts with the obstacles are allowed), will be solved by our planner, provided that a sufficient amount of time has been spent on learning. We then say that the learning method is *probabilistically complete*. We will show that the property holds for the RTR local method. Hence, we prove that the planner for general car-like robots, as described in the Sections 3 and 6, is probabilistically complete. The theory in this section only applies to the planner using undirected underlying graphs. Throughout this section we assume that the configuration space of the robot is a Euclidean space, and we denote the Euclidean distance (in C-space) between two configurations a and b by $D_E(a, b)$.

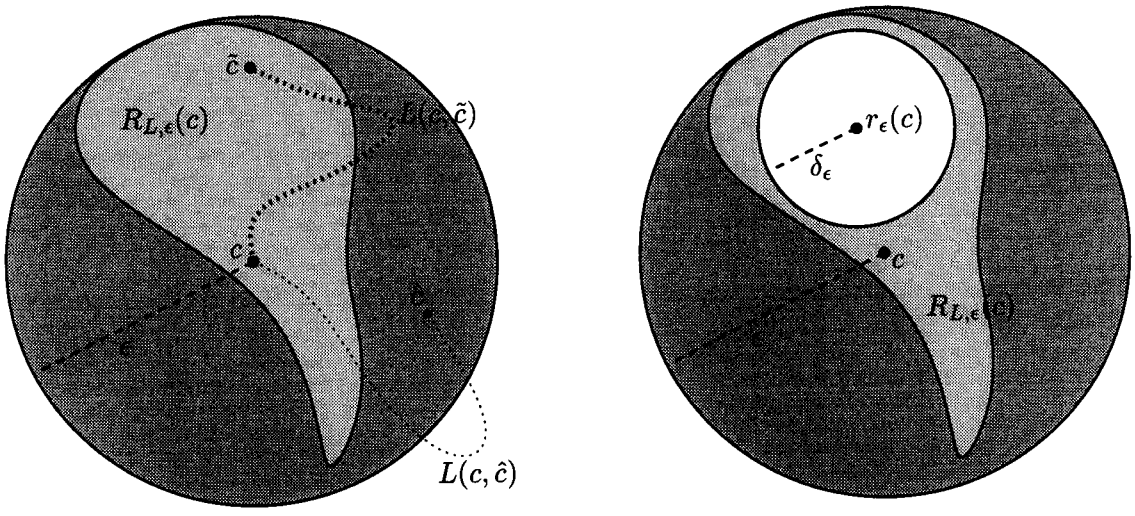


Figure 16: At the left, the ϵ-reachable area of a (fictive) local method L is indicated in light grey, in a two dimensional configuration space. Local paths connecting c to configurations lying in this area stay within $B_\epsilon(c)$, while those going to configurations outside the area leave $B_\epsilon(c)$. At the right, a (white) δ_ϵ-ball fits into c's ϵ-reachably area. This is required in order for L to have the ϵ-RB property.

8.1 ϵ-Reachability of local methods

We first formally define the notion of symmetrical local methods. Given a configuration space $\mathcal{C}$, we regard a *path* in $\mathcal{C}$ as a continuous function of type $[0,1] \to \mathcal{C}$. Furthermore, we denote the set of all paths (in $\mathcal{C}$) by $\mathcal{P}_\mathcal{C}$, and the set of all paths (in $\mathcal{C}$) connecting two configurations a and b, by $\mathcal{P}_\mathcal{C}(a,b)$.

Definition 3 *A* symmetrical local method *in $\mathcal{C}$ is defined as a function L of type $\mathcal{C}\times\mathcal{C} \to (\mathcal{P}_\mathcal{C} \cup \{\bot\})$ with $L(c_1,c_2) \in \mathcal{P}_\mathcal{C}(c_1,c_2) \cup \{\bot\}$, and*

$$\forall (a,b) \in \mathcal{C}\times\mathcal{C} : L(a,b) = L(b,a) = \bot \vee \forall t \in [0,1] : L(a,b)(t) = L(b,a)(1-t)$$

So we define a local method to be a function that takes two argument configurations, and returns either a path connecting them, or returns that it cannot connect them. Note that it does not take into account any constraints induced by possible obstacles. The local method is symmetrical if $L(a,b)$ is the reverse of $L(b,a)$.

The *ϵ-ball* of a configuration c, denoted by $B_\epsilon(c)$, is defined as the ball in $\mathcal{C}$ that is centered at c and has radius ϵ. We denote the set of all ϵ-balls in $\mathcal{C}$ by $\mathcal{B}_\epsilon$. Given an ϵ-ball of a configuration c and a local method L, we are interested in the area within this ball that is reachable from c by a local path (that is, a path computed by the local method L) without leaving the ϵ-ball. We refer to this area as the *ϵ-reachable area* of c (See also figure 16). This notion is formalized in the following definition ($L(a,b) \subset A$ denotes $\forall t \in [0,1] : L(a,b)(t) \in A$) :

Definition 4 *Given a local method L, the* ϵ-reachable area *of a configuration c is defined as the set of all configurations $\tilde{c}$ in $B_\epsilon(c)$ such that $L(c,\tilde{c}) \neq \perp$ and $L(c,\tilde{c})$ lies within $B_\epsilon(c)$:*

$$R_{L,\epsilon}(c) = \{\tilde{c} \in B_\epsilon(c) \mid L(c,\tilde{c}) \neq \perp \wedge L(c,\tilde{c}) \subset B_\epsilon(c)\}$$

The first requirement we make is that the ϵ-reachable area's are of a certain "fatness". More specifically, we require that, given an arbitrary $\epsilon > 0$, there exists a $\delta > 0$ such that balls of radius δ can be fit into all ϵ-reachable areas of configurations in $\mathcal{C}$. We refer to this local method property, which is made precise in Definition 5, as the *ϵ-reachable ball property*, or *ϵ-RB property* (See also figure 16).

Definition 5 *A local method L has the* ϵ-RB property, *iff for every $\epsilon > 0$ there exists a $\delta_\epsilon > 0$ and a function r_ϵ mapping configurations to δ_ϵ-balls, such that for each configuration $c \in \mathcal{C}$ the δ_ϵ-ball $r_\epsilon(c)$ is contained in c's epsilon-reachable area. Formally:*

$$\forall \epsilon > 0 : \exists \delta_\epsilon > 0, r_\epsilon \in \mathcal{C} \rightarrow \mathcal{B}_{\delta_\epsilon} : \forall c \in \mathcal{C} : r_\epsilon(c) \subset R_{L,\epsilon}(c)$$

We refer to r_ϵ as an ϵ-reachability function *of L, and to δ_ϵ as r_ϵ*'s radius.

The second requirement uses the notion of *Lipschitz continuity*. A function f is called Lipschitz continuous if and only if f is continuous and its (first) derivative is bounded, in absolute value, by some constant. Such a constant is called a *Lipschitz constant* of the function f.

We say that a local method has the *Lipschitz ϵ-RB property*, iff it has the ϵ-RB property and, for each $\epsilon > 0$, it has a Lipschitz continuous ϵ-reachability function. Formally:

Definition 6 *A local method L has the* Lipschitz ϵ-RB property, *iff for every $\epsilon > 0$ there exists a $\delta_\epsilon > 0$ and a* Lipschitz continuous function *r_ϵ mapping configurations to δ_ϵ-balls*[3], *such that for each configuration $c \in \mathcal{C}$ the δ_ϵ-ball $r_\epsilon(c)$ is contained in c's epsilon-reachable area. Formally:*

$$\forall \epsilon > 0 : \exists \delta_\epsilon > 0, r_\epsilon \in \mathcal{C} \rightarrow \mathcal{B}_{\delta_\epsilon} : r_\epsilon \textit{ is Lipschitz continuous } \wedge \forall c \in \mathcal{C} : r_\epsilon(c) \subset R_{L,\epsilon}(c)$$

8.2 Probabilistic completeness proof

We will now prove that a symmetrical local method having the Lipschitz ϵ-RB property guarantees the earlier mentioned probabilistic completeness of the learning algorithm, as described in Section 3. We will here refer to this algorithm as *the undirected learning algorithm.*

For ease of presentation, we assume here that $N(c)$ (the neighbors of c) consists of *all* nodes in V. If however the used metric D has the property that, for any configuration pair (a,b) with $b \in R_{L,\epsilon}(a)$, $D(a,b) \leq d \cdot \epsilon$ for some constant d, than the above assumption can be dropped. That is, $N(c)$ can be defined as the set of all nodes lying within a distance *maxdist* of c (with respect to D). In practice, any well defined metric (that is, one reflecting the failure chances of the local method well) will have the above property. This is for example also the case for the induced RTR-metric.

[3]We define the Euclidean distance $D_E(B_1, B_2)$ between two δ-balls B_1 and B_2 as the distance between their centers.

We denote the free C-space by $\mathcal{C}_f$ and we make the (realistic) assumption that it is bounded. The portion of $\mathcal{C}_f$ lying further than ϵ away from any C-space obstacle (in C-space, with respect to the Euclidean metric), is denoted by $\mathcal{C}_{f,\epsilon}$. A path lying in $\mathcal{C}_{f,\epsilon}$ is referred to as an *ϵ-free path.* Furthermore, given a graph $G = (V, E)$ computed by the undirected learning algorithm, we refer to the nodes lying in $\mathcal{C}_{f,\epsilon}$ as *ϵ-free nodes*, and we denote this set by V_ϵ.

Lemma 4 states that an ϵ-free node a which contains another node b in its ϵ-reachable area, must be graph-connected to this node.

Lemma 4 *Let $G = (V, E)$ be a graph computed (at some point) by the undirected learning algorithm, and let $\epsilon > 0$. If $a \in V_\epsilon$ and $b \in V \cap R_{L,\epsilon}(a)$, then a and b are graph-connected in G.*

Proof
Assume a and b are not graph-connected. Either a is added to V after b, or b after a.

Case a is added after b : When a is added, $N(a) \ni b$. Because $a \in \mathcal{C}_{f,\epsilon}$ and $b \in R_{L,\epsilon}(a)$, surely $L(a, b) \subset \mathcal{C}_f$, and hence the edge (a, b) is added to E_G.

Case b is added after a : Analog, when b is added, $N(b) \ni a$. We know that $L(a, b) \subset \mathcal{C}_f$, but because L is symmetrical, this means that $L(b, a) \subset \mathcal{C}_f$ as well. It follows that the edge (b, a) is added to E_G.

So it follows that a and b are connected. □

The second lemma that we need shows that, provided the used local method has the Lipschitz ϵ-RB property, a certain node density guarantees nodes lying sufficiently near to each other to be graph-connected.

Lemma 5 *Let $G = (V, E)$ be a graph computed (at some point) by the undirected learning algorithm, and let $\epsilon > 0$. Furthermore, let L be a local method with the Lipschitz ϵ-RB property. So L has an ϵ-reachability function r_ϵ with some radius δ_ϵ, and Lipschitz constant d_ϵ. Now if each free ball of radius $\frac{\delta_\epsilon}{2}$ (in $\mathcal{C}_f$) contains a node of G, then each pair of ϵ-free nodes which lie at most $\frac{\delta_\epsilon}{d_\epsilon}$ apart will be graph-connected in G. Formally :*

$$\forall B \in \mathcal{B}_{\frac{\delta_\epsilon}{2}} : B \subset \mathcal{C}_f \Rightarrow B \cap V \neq \emptyset$$
$$\Longrightarrow$$
$$\forall a, b \in V_\epsilon : D_E(a, b) \leq \tfrac{\delta_\epsilon}{d_\epsilon} \Rightarrow \textit{graph-connected}(a, b)$$

Proof
Assume that $G = (V, E)$ is a graph, computed by the undirected learning algorithm, such that

$$\forall B \in \mathcal{B}_{\frac{\delta_\epsilon}{2}} : B \subset \mathcal{C}_f \Rightarrow B \cap V \neq \emptyset$$

Furthermore, assume that $\{a, b\} \subset V_\epsilon$ with $D_E(a, b) \leq \frac{\delta_\epsilon}{d_\epsilon}$. The fact that d_ϵ is a Lipschitz constant of the Lipschitz continuous function r_ϵ, induces $D_E(r_\epsilon(a), r_\epsilon(b)) \leq \delta_\epsilon$. This means that the intersecting area of $r_\epsilon(a)$ and $r_\epsilon(b)$ contains a ball B of radius $\frac{\delta_\epsilon}{2}$, and this ball contains a node c. So a is an ϵ-free node which contains node c in its ϵ-reachable area.

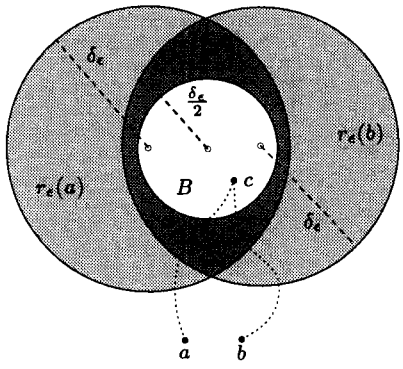


Figure 17: Intersecting δ-balls $r_\epsilon(a)$ and $r_\epsilon(b)$. Their intersection contains a $\frac{\delta_\epsilon}{2}$-ball B. The presence of a node c in B guarantees a and b to be graph-connected.

Hence, by Lemma 4, a and c are graph-connected in G. Analog, b is an ϵ-free node with c in its ϵ-reachable area, and therefore b and c are graph-connected in G as well. By the fact that G is undirected, it follows that a is graph-connected to b. See also Figure 17. □

With the above properties, we now prove the claim of probabilistic completeness.

Theorem 1 *If the undirected learning algorithm uses a local method with the Lipschitz ϵ-RB property, then it is probabilistically complete.*

Proof
Let L be the local method used by the undirected learning algorithm, and assume it has the Lipschitz ϵ-RB property. Furthermore, let s and g be two configurations lying in the same connected component of the open free C-space. We must prove that the undirected learning algorithm solves the query (s, g) if a sufficient amount of time is spent on learning. One can view s and g as nodes of G (the last two added). Hence it suffices to prove that any pair of graph nodes lying in the same component of the open free C-space gets graph-connected if the algorithm runs for a long enough period of time.

Let a and b be such a pair of nodes. Take $\epsilon > 0$ such that a and b lie in the same connected component of $\mathcal{C}_{f,\epsilon+\frac{\delta_\epsilon}{4d_\epsilon}}$ (where δ_ϵ is the radius of an ϵ-reachability of L, and d_ϵ is a finite Lipschitz constant of this function). In other words, there exists a $(\epsilon + \frac{\delta_\epsilon}{4d_\epsilon})$-free path $\mathcal{P}$ connecting connecting configuration a to configuration b.

Let now $G = (V, E)$ be a graph, computed by the undirected learning algorithm, such that

$$\forall B \in \mathcal{B}_{\frac{\delta_\epsilon}{4d_\epsilon}} : B \subset \mathcal{C}_f \Rightarrow B \cap V \neq \emptyset$$

Due to the boundedness of $\mathcal{C}_f$, the probability of obtaining such a graph grows toward 1 when the running time grows toward infinity.

We now show that there exists a path P_G in G which connects the nodes a and b. Let $\{B_1, B_2, \ldots, B_n\}$ be a set of $\frac{\delta_\epsilon}{4d_\epsilon}$-balls (i.e., balls of radius $\frac{\delta_\epsilon}{4d_\epsilon}$) centered at configurations of $\mathcal{P}$, such that $B_1 \ni a$, $B_n \ni b$, and $\forall i \in \{1, \ldots, n-1\} : B_i \cap B_{i+1} \neq \emptyset$. See Figure 18.

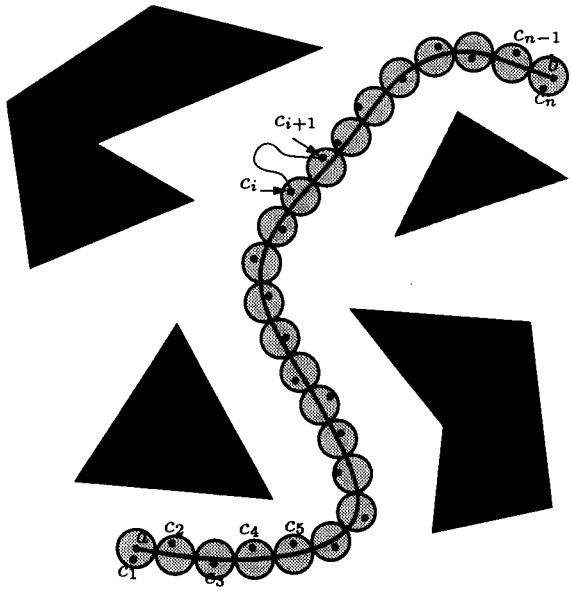


Figure 18: A "covering" of an $(\epsilon + \frac{\delta_\epsilon}{4d_\epsilon})$-free path by $\frac{\delta_\epsilon}{4d_\epsilon}$-balls, which contain the vertices of a graph path connecting a to b. Nodes c_i and c_{i+1} lying in "neighboring" balls must be graph-connected.

Each ball B_i is a $\frac{\delta_\epsilon}{4d_\epsilon}$-ball lying in free configuration space, so each B_i contains at least one node $c_i \in V$. Clearly

$$\forall i \in \{1, \ldots, n-1\} : D_E(c_i, c_{i+1}) \leq \frac{\delta_\epsilon}{d_\epsilon}$$

Each c_i is an ϵ-free node, so, by Lemma 4, it follows that

$$\forall i \in \{1, \ldots, n-1\} : \textit{graph-connected}(c_i, c_{i+1})$$

Furthermore, $D_E(a, c_1) \leq \frac{\delta_\epsilon}{d_\epsilon}$ and $D_E(c_n, b) \leq \frac{\delta_\epsilon}{d_\epsilon}$ (and a and b are ϵ-free nodes), so, again by Lemma 4, it follows that

$$\textit{graph-connected}(a, c_1) \wedge \textit{graph-connected}(c_n, b)$$

Hence, node a is graph-connected to node b, which concludes the proof. □

8.3 Probabilistic completeness with the RTR local method

It remains to show that the RTR local method, as defined in Section 6, has the Lipschitz ϵ-RB property. So let L be the RTR local method. That is :

$$L \in \mathcal{C} \times \mathcal{C} \to \mathcal{P}_\mathcal{C} = (a, b) \mapsto \text{The shortest RTR path connecting } a \text{ to } b.$$

Lemma 6 *L has the Lipschitz ϵ-RB property, with ϵ-reachability function*

$$r_\epsilon = (x, y, \Theta) \mapsto B_{\delta_\epsilon}((x, y, \Theta) + \frac{1}{2}\epsilon \cdot (\cos\Theta, \sin\Theta, \Theta))$$

where (for all $\epsilon > 0$) $\delta_\epsilon > 0$.

Proof
Given a configuration c, let us denote the configuration $(x, y, \Theta) + \frac{1}{2}\epsilon \cdot (\cos\Theta, \sin\Theta, \Theta)$ by $\tilde{r}_\epsilon(c)$. So $\tilde{r}_\epsilon(c)$ is the center of the δ_ϵ-ball to which c is mapped by r_ϵ. Positioned at configuration c, $\tilde{r}_\epsilon(c)$ is the configuration that is reached when the car-like robot drives straight forward over a distance of $\frac{1}{2}\epsilon$.

We denote the metric induced by the RTR local method (as also described in Section 6) by D_{RTR}. D_{RTR} is a continuous function on $\mathcal{C} \times \mathcal{C} - \{(a, a) \in \mathcal{C} \times \mathcal{C}\}$. This is due to the fact that, given two circles C_1 and C_2 in $\mathbf{R}^2$ and a line l which is (and stays) tangent to both circles, continuous motions of the circles bring about continuous motions of the tangent line l, as long as they do not coincide.

Now let $\epsilon > 0$ and $c \in \mathcal{C}$. $L(c, \tilde{r}_\epsilon(c))$ is a straight path (in C-space) connecting c to $\tilde{r}_\epsilon(c))$, and, hence, $D_{RTR}(c, \tilde{r}_\epsilon(c)) = \frac{1}{2}\epsilon$. Since D_{RTR} is continuous, there must exist a $\delta_\epsilon > 0$ such that $\forall a \in B_{\delta_\epsilon}(\tilde{r}_\epsilon(c)) : D_{RTR}(c, a) < \epsilon$. However, if $D_{RTR}(c, a) < \epsilon$ then $L(c, a)$ must be contained in $B_\epsilon(c)$. Hence, $B_{\delta_\epsilon}(\tilde{r}_\epsilon(c)) \subset R_{L,\epsilon}(c)$, that is, $r_\epsilon(c) \subset R_{L,\epsilon}(c)$. The observations that δ_ϵ is independent of c and that r_ϵ is Lipschitz continuous now conclude the proof. □

Lemma 6 and Theorem 1, together, prove the probabilistic completeness claim for general car-like robots.

Theorem 2 *The undirected learning algorithm using the RTR local method is a probabilistically complete method for general car-like robots.*

Unfortunately, although the RTR forward local method has the Lipschitz ϵ-RB property as well, the probabilistic completeness proof as given for general car-like robots does not hold for forward car-like robots. The reason is that if directed underlying graphs are used, then graph-connectivity is no longer symmetrical. This symmetry is however required for the validity of the Lemmas 4 and 5. Proving (or disproving) probabilistic completeness for the directed learning method remains a topic of further research.

9 Discussion and conclusions

We have described and extended a probabilistic technique for solving the learning motion planning problem in static environments, and we have applied it to two types of car-like robots. In the learning phase a probabilistic roadmap is incrementally constructed, which can subsequently be used for solving individual motion planning problems in the given scene. The method, which has previously proven to be very fast for free-flying robots and (high dof) articulated robots, achieves very good results for both general car-like robots as well as for forward car-like robots, in different types of scenes. Conceptually it is a very simple method, and the graphs that it builds are small representations of the free configuration space.

The method is highly flexible. In order to apply it to some particular robot type, all that is needed is a local method which computes paths feasible for this robot type, and some (induced) metric. Experimental results ([Mas92], [Šve93]) indicate that very simple local methods achieve the best results. Hence, it is usually easy to find a suitable local method for some given robot type. For robots with the reversibility property, we have also shown that a proper choice of the local method guarantees probabilistic completeness of the learning method.

Another property which one may wish is that (concatenations of) local paths describe "smooth" motions of the robot. The paths generated by the planners described in this paper are solely composed of straight line- and circle segments. In practice, due to various mechanical constraints, such paths will cause the robot to jerk while traveling along its path (in points where the straight line- and circle segments meet). There exist however local methods which take into account the smoothness constraints imposed by the control engineering level ([Nau94]). We believe we can incorporate such local methods into our global approach. This will be investigated in the future.

Currently we are working on the application of the method to a number of different robot types, such as solid robots in 3 dimensional workspaces, and robots with other than car-like non-holonomic constraints, for example tractor-trailer robots. Furthermore, we are working on extensions of the method in various directions, aimed at solving more difficult motion planning problems, such as motion planning in scenes with multiple robots, or in scenes which are subject to dynamic changes (for example, moving obstacles).

Finally we note that it should be easy to parallelize the method. Most work done in the learning phase is highly independent.

Acknowledgments

We thank Geert-Jan Giezeman for the implementation of many crucial 'geometric' routines (some of which are contained in the *Plageo* library, see [Gie93]) used both implementations. Also, we thank Anil Rao, Otfried Schwarzkopf and Mark de Berg for useful comments.

References

[ATBM92] J. M. Ahuactzin, E.-G. Talbi, P. Bessiere, and E. Mazer. Using genetic algorithms for robot motion planning. pages 671–675. John WIley and Sons, Ltd., 1992.

[BL93] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.

[Gie93] G.-J. Giezeman. *PlaGeo—A Library for Planar Geometry.* Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, August 1993.

[HA92] Y. Hwang and N. Ahuja. Gross motion planning—a survey. *ACM Comput. Surv.*, 24(3):219–291, 1992.

[HST94] Th. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom - random reflections at c-space obstacles. In *Proc. IEEE Internat. Conf. on Robotics and Automation*, pages 2138–2145, San Diego, CA, 1994.

[KL94] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. IEEE Internat. Conf. on Robotics and Automation*, San Diego, CA, 1994.

[KŠLO94] L. Kavraki, P. Švestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high dimensional configuration spaces. Technical Report UU-CS-94-32, Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, August 1994.

[Lat91] J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, USA, 1991.

[LS89] J.-P. Laumond and T. Siméon. Motion planning for a two degrees of freedom mobile robot with towing. Technical Report 89148, LAAS/CNRS, Toulouse, France, April 1989.

[LTJ90] J.-P. Laumond, M. Taïx, and P. Jacobs. A motion planner for car-like robots based on a global/local approach. In *Proc. IEEE Internat. Workshop Intell. Robots Syst.*, pages 765–773, 1990.

[Mas92] J. Mastwijk. Motion planning using potential field methods. master thesis, Utrecht, the Netherlands, August 1992.

[MATB92] E. Mazer, J. M. Ahuactzin, E.-G. Talbi, and P. Bessiere. The ariadne's clew algorithm. Technical report, Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle, Grenoble, France, July 1992.

[Nau94] J.M. Nauta. A tom thumb algorithm for planning simple vehicle motions. Technical report, University of Twente, Enschede, the Netherlands, May 1994.

[OŠ94] M. Overmars and P. Švestka. A probabilistic learning approach to motion planning. In *Proc. The First Workshop on the Algorithmic Foundations of Robotics.* A. K. Peters, Boston, MA, 1994.

[Ove92] M. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, October 1992.

[RS90] J.A. Reeds and R.A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific Journal of Mathematics*, 145, 1990.

[SL92] P. Souères and J.P. Laumond. Shortest paths synthesis for a car-like robot. Technical report, LAAS/CNRS, Toulouse, France, September 1992.

[Šve93] P. Švestka. A probabilistic approach to motion planning for car-like robots. Technical Report RUU-CS-93-18, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, April 1993.