A Probabilistic Approach to Motion Planning for Car-Like Robots

Petr Švestka

RUU-CS-93-18
April 1993

Utrecht University

Department of Computer Science

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

A Probabilistic Approach to Motion Planning for Car-Like Robots

Petr Švestka

Technical Report RUU-CS-93-18
April 1993

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

ISSN: 0924-3275

Contents

Chapter 1

Introduction

The motion planning problem is a well-known problem in the field of robotics. The objective is to find a collision free path in an environment containing some obstacles for a given robot $\mathcal{A}$ from some start configuration to some goal configuration. In this paper we deal with solid robots which move in a planar environment. The techniques described can though be applied to other robot types also. Figure 1.1 shows an example of a situation where a planar robot must move from the bottom right to the top left. The obstacles are shown in black, and a path which solves the problem is indicated by a number of (grey) steps. The robot shown in figure 1.1 is allowed to rotate and translate freely. Such robots are called *free flying robots.* In the past years a number of techniques were developed for the motion planning problem for free-flying solid robots. Globally the current approaches to motion planning can be divided in the following three classes: *roadmap methods*, *cell decomposition methods*, and *potential field methods.*

The motion planning is typically not done in the workspace, i.e., the space in which the robot and the obstacles physically are present, but in configuration space. The configuration space is extracted from the workspace in a way that in configuration space the robot reduces to a point. This is typically achieved by adding some extra dimensions and 'blowing up' the obstacles.

The roadmap approach globally consists of capturing the connectivity of the robot's free configuration space C_f in the form of a network of one dimensional curves - the *roadmap* - lying in C_f. After a roadmap ρ has been constructed, the path planning is reduced to connecting the start and goal configurations to ρ, and searching ρ for a path.

The principle of the cell decomposition approach is to decompose the robots free configuration space C_f into a collection of non-overlapping regions (cells), whose union is exactly C_f. This *cell decomposition* is then used for constructing the *connectivity graph* G which represents the adjacency relation among the constructed cells. Every node in G corresponds to a cell, and two nodes are connected by an edge if and only if their corresponding cells are adjacent. The path planning is then performed by finding a path in G from the node corresponding to the start cell (= the cell containing the start configuration) to the node corresponding to the goal cell (= the cell containing the goal configuration).

So both the roadmap method as well as the cell decomposition method first construct a data structure that is used later for computing motions between different configurations of the robot. Unfortunately, for computing the data structures many expensive geometric operations need to be performed, and the resulting data structures tend to be very large.

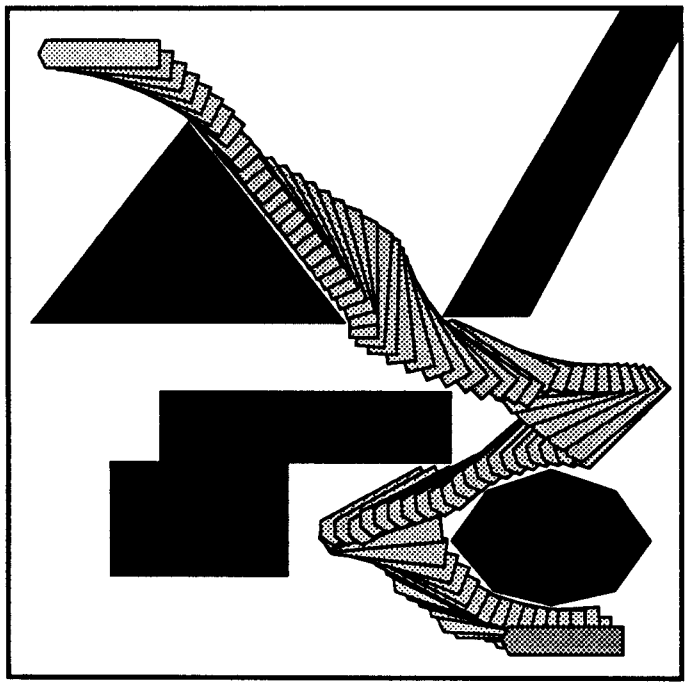

Figure 1.1: A free path for a free flying planar robot.

So the preprocessing is very expensive in both time and memory.

The potential field approach does not have these disadvantages. Globally the idea is that the robot (represented by a configuration in configuration space) is treated as a particle under the influence of an *artificial potential field* whose variations are expected to reflect the 'structure' of the free configuration space C_f. The potential field is typically defined by a function $C \rightarrow \mathbf{R}$ which is a weighed sum of an *attractive* potential pulling the robot towards the goal configuration, and a number of *repulsive* potentials pushing the robot away from the obstacles. The motion planning is performed by repeatedly computing the most promising direction of motion, and moving in this direction by some step size. The major problem with potential field methods is that the robot can get stuck in a local minimum of the potential field. I.e. the robot gets to a configuration m where the (weighed) sum over all the potentials is equal to the null-vector. For a more thorough treatment of the above approaches see [Lat91].

Very recently a new approach for the motion planning problem for free flying solid robots was presented in [Ove92]. We will refer to it as *the random motion planning approach.* The idea is that a network of highways is built up, by repeatedly generating random via-points and trying to connect these to other (earlier added) via-points by some simple motion planning algorithm. The network is stored in a graph G. The via-points, which are configurations, are stored as nodes in G, and the links, which are paths in free configuration space, are stored as edges in G. Initially G has got only two nodes, one node s corresponding to the start configuration, and one node g corresponding to the goal configuration, and G has no edges. Then G is repeatedly extended in the sketched

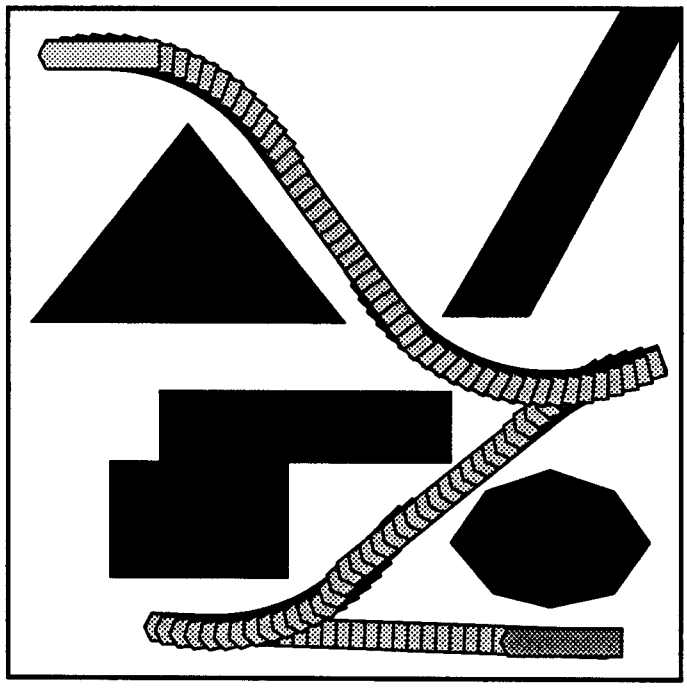

Figure 1.2: A free path for a car-like robot.

way, until there is a path in the graph G. Once such a path is available, a path in free configuration space solving the problem can of course be obtained.

The method turns out to be extremely fast. For example the path shown in figure 1.1 was computed in less than 1 second, on a Silicon Graphics 4D/35 workstation (see also section 8.1). Apart from its speed, the method has another nice property. After a path has been computed, and a network of possible motions has been constructed, this network does not have to be thrown away, but it can be reused and extended for the computation of other paths. So it can be regarded as a learning strategy, which improves its performance in a particular scene with every search that it performs.

As described in [Ove92], the method solves the motion planning problem for *free flying* robots, moving in a planar environment. In this paper we use the concepts of the random motion planning method for solving the motion planning problem for planar robots which have certain *nonholonomic constraints*. First we (re)define the random motion planning method, and we discuss how it can be applied to such robots. Then we apply it to *car-like robots*. A car-like robot is a solid planar robot, which can move forwards and backwards, but has a bounded turning radius. We also deal with car-like robots which cannot make reversals, i.e. can only move forwards.

Experimental results show that the method, for both mentioned car-like robot types, is very fast. For example, the path shown in figure 1.2 was also computed in less than one second. We expect that for robots with other than car-like nonholonomic constraints the method will also achieve very fast running times.

This paper is organized in the following way: In chapter 2 we give some conventions

and basic definitions that will be used throughout this paper. In chapter 3 we give a short and informal description of the random motion planning method. In chapter 4 we define the method more formally. Two versions are described: one based on an undirected graph, and one based on a directed one. We also explain how the constraints of the robot prescribe whether the underlying graph should be be directed or not. Also attention is paid to the way in which edges should be added to the underlying graph, and how to prevent adding useless edges. In chapter 5 we describe a number of (heuristic) techniques for adding via-points in a smarter way then just fully randomly. In chapter 6 two methods for smoothing the (often ugly) paths generated by the basic algorithm are given. Then in chapter 7 we give formal definitions of car-like robots and paths respecting their nonholonomic constraints, and we apply the described techniques to them. In chapter 8 we give experimental results and we draw some conclusions. In chapter 9 some possible improvements of the method are discussed. We conclude this paper with some final conclusions in chapter 10.

Chapter 2

Preliminaries

We consider a solid robot, moving in $\mathbf{R}^2$ among some obstacles. So $\mathbf{R}^2$ is the workspace that we are dealing with. Every robot $\mathcal{A}$ will have a fixed *reference point* and a fixed *main axis*. By fixed we mean fixed to the robot, and not to $\mathbf{R}^2$. A 'positioning' of a robot is uniquely defined by the x and y coordinates of its reference point in $\mathbf{R}^2$ and the angle that $\mathcal{A}$'s main axis makes with the x-axis of $\mathbf{R}^2$. From now on we will simply talk in terms of the *the position* of $\mathcal{A}$ when we (formally) mean the x and y coordinates of $\mathcal{A}$'s reference point in $\mathbf{R}^2$, and we will talk in terms of *the orientation* of $\mathcal{A}$, when we (formally) mean the angle that $\mathcal{A}$'s main axis makes with the x-axis of $\mathbf{R}^2$. A *configuration* c of $\mathcal{A}$ is a specification of a position and orientation of $\mathcal{A}$. We call c a *free configuration* if $\mathcal{A}$ positioned at c does not intersect any obstacles, and otherwise c is a *forbidden configuration*. The *configuration space* of $\mathcal{A}$ is the space C of all possible configurations of $\mathcal{A}$. We represent this space by $\mathbf{R} \times \mathbf{R} \times [0, 2\pi[$.

Furthermore, we use the following convention: If c is a configuration, then we refer to its 'contents' by (x_c, y_c, θ_c). Analog, if p is a point in $\mathbf{R}^2$, then we refer to its 'contents' by (x_p, y_p).

Now we will define *paths* in configuration space C. A *path* for a robot $\mathcal{A}$ from configuration s to configuration g is a continuous map[1] $P \in [0,1] \rightarrow C$ with $P(0) = s$ and $P(1) = g$. We call P a *free path* iff $\forall t \in [0,1] : P(t)$ is a free configuration.

In this paper we will mostly not consider paths as functions of type $[0,1] \rightarrow C$, but simply as some abstract data type *paths*. Also we will use the operator $\oplus$ of type *paths* $\times$ *paths* $\rightarrow$ *paths* which intuitively concatenates its two argument paths. Formally it can be defined by

$$\begin{aligned} \forall t \in [0,1] : P_1(t) \oplus P_2(t) = \ & \textbf{if } P_1(1) = P_2(0) \\ & \quad \textbf{then if } t \leq \tfrac{1}{2} \\ & \qquad \textbf{then } P_1(2t) \\ & \qquad \textbf{else } P_2(2t-1) \\ & \quad \textbf{else } \text{undefined.} \end{aligned}$$

Furthermore, if we refer to the *length* of a path P, then (if not explicitly stated otherwise) we mean the length of P's projection in $\mathbf{R}^2$.

[1] We will not regard configurations $P(m)$ such that $\lim_{t \uparrow m} P(t) = (\tilde{x}, \tilde{y}, 2\pi) \wedge \lim_{t \downarrow m} P(t) = (\tilde{x}, \tilde{y}, 0)$ or $\lim_{t \uparrow m} P(t) = (\tilde{x}, \tilde{y}, 0) \wedge \lim_{t \downarrow m} P(t) = (\tilde{x}, \tilde{y}, 2\pi)$ as discontinuities of P.

For the concatenation of lists, denoted by $[x_1, x_2, \ldots, x_k]$, we use the operator $+\!\!+$.

Chapter 3

The random motion planner

The input of the random motion planner consists of a start-configuration s, a goal-configuration g, and a set of obstacles, which define the forbidden configuration space. The output is a path in free configuration space from s to g, if the random motion planner succeeds in finding one.

The algorithm consists of a *global method* and a *local method.* The local method is a (simple) motion planner, which tries to compute (simple) paths in free configuration space between two given configurations. It is allowed to fail now and then, but it is essential that it always terminates and that it is deterministic. The global method tries to solve the entire problem. The idea is that it subdivides the initial motion planning problem into a number of easier motion planning problems, which are to be solved by the local method.

The way it does this is the following. It gradually builds up a directed graph G, where the nodes correspond to configurations, and every edge (a, b) corresponds to the statement that the local method can compute a free path from a to b (from now on we won't make a distinction between a node and the configuration which corresponds to it). Initially the set of nodes V is $\{s, g\}$, and the set of edges is empty. Then repeatedly a *random free configuration* c is generated, and for each such c the local method tries to compute free paths to and from some nodes ($\in V$) near c. When the local method succeeds in computing a free path from c to say a, then the edge (c, a) is added to E, and when it succeeds in computing a free path from say b to c, then the edge (b, c) is added to E. This continues until s is connected to g by a path in the graph G. Then a free path in configuration space from s to g is reconstructed, by concatenating the subpaths computed by the local method when applied to each pair of consecutive nodes in the path in G.

A nice thing about this approach is that it can be used for all sorts of robots. As mentioned above, we assume that we are dealing with a planar robot $\mathcal{A}$ for which the workspace is $\mathbf{R}^2$, and contains a set of obstacles, which define the forbidden configuration space. But the robot can also have additional constraints (other than those defined by the obstacles) imposed on it, and we want our motion planner to compute paths which respect those constraints, and lie entirely in free space. Such paths will from now on be referred to as *feasible paths.* So if, e.g., we are dealing with a car-like robot, then a straight path in free configuration space which corresponds to a forwards motion is feasible, but one that corresponds to a sidewise motion is not feasible.

The idea now is that to obtain a feasible path for $\mathcal{A}$ from s to g, a local method is used which computes feasible paths for $\mathcal{A}$. A point of concern is that, even if the local

method computes feasible (sub)paths, we are not sure whether the concatenation of those (sub)paths will result in a path that is also feasible. If however $\mathcal{A}$ has the property

$$\forall P \in paths : (\exists P_1, P_2 \in paths : (P = P_1 \oplus P_2 \wedge feasible(P_1) \wedge feasible(P_2)) \Rightarrow feasible(P)$$

then it is obvious that when the local method computes feasible paths, the final path, computed by the global method, will also be feasible. From now on we will assume that the robots we are dealing with have this property.

Chapter 4

The global method

In this chapter the global method will be described in detail. Two versions will be given and motivated. First we will describe a version which uses an undirected underlying graph, and later a version based on a directed underlying graph will be discussed.

4.1 Using an undirected graph

The global method, as briefly sketched in the previous chapter, uses a directed underlying graph. In this section we discuss the possibility of using an undirected graph instead of a directed one. The motivation for this is, that the global method is easier and more efficient to implement when based on an undirected graph, and for many motion planning problems it is sufficient. For example the method in [Ove92], for free flying planar robots, uses an undirected underlying graph.

As indicated in the previous chapter, an (directed) edge (a, b) corresponds to the statement that the local method can compute a feasible path from a to b. Now suppose that the underlying graph is undirected. An edge (a, b) no longer contains any information about the direction in which a feasible path between a and b has been computed, so 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 we could implement the algorithm in such a way that an edge (a, b) is added only if the local method succeeds in both directions. Doing so, a major drawback can be that useful information will 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 it is obvious that this problem will never occur. So if the local method is symmetrical, the underlying graph can surely be undirected, and if it is not symmetrical, then it is safer 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.*

Unconstrained planar robots and car-like robots are two examples of robot types which possess the reversibility property, while for example car-like robots which can only move

forwards (i.e., cannot make reversals) do not possess the reversibility property. An important observation now is that, if the robot $\mathcal{A}$ has the reversibility property, then any local method that computes feasible paths for $\mathcal{A}$ can be made symmetrical in a trivial way. So this implies that if the robot has the reversibility property, then the global method can use an undirected graph. Of course, $\mathcal{A}$ not having the reversibility property does not rule out the possibility that (good) symmetrical local methods for $\mathcal{A}$ exist.

4.2 The global method, based on an undirected graph

In this section a description of the global method, based on an undirected graph, is given. It corresponds to the current implementation of the algorithm. Assume we have the following:

- A robot $\mathcal{A}$ with certain properties, which are defined by some constraints imposed on it.
- A *symmetrical* function *compute_path* $\in C \times C \rightarrow$ *boolean*, that returns whether the local method can compute a feasible path for $\mathcal{A}$, between its two argument-configurations. It has already been said that the local method must be deterministic, so clearly *compute_path* must be deterministic also.
- A function *construct_path* $\in C \times C \rightarrow$ *paths*, that returns the path computed by the local method between its two argument-configurations, if the local method succeeds in computing one. If not, the result is undefined.
- A function $D \in C \times C \rightarrow \mathbf{R}^+$. It defines the metric[1] used, and should give a suitable notion of distance for arbitrary pairs of configurations, taking the properties of the robot into account. We assume that D is symmetrical, so we can talk in terms of distances between configurations.
- A constant *maxdist* $\in \mathbf{R}^+$, which defines the size of the neighborhood of a configuration.

The input consists of a start configuration s, a goal configuration g, and a set of obstacles. V will be the set of nodes, and E the set of edges in the underlying graph G. The algorithm can now be described in the following manner:

The global method:

$V = \{s, g\}$
$E = \{\}$
while there is no path in G from s to g
do c = A randomly chosen free configuration.
  $V = V \cup \{c\}$
  $N(c)$ = A set of neighbors of c, chosen from V.
  $E = E \cup \{(c, a) \mid a \in N(c) \wedge compute\_path(c, a)\}$
P = A path in G from s to g (= a list of nodes in V).

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

Construct a free path from s to g, by concatenating the subpaths computed by *construct_path* when applied to each pair of consecutive nodes in P.

So initially the graph $G = (V, E)$ is defined by $V = \{s, g\}$ and $E = \emptyset$. Then as long as there is no path in the graph G from node s to node g, random configurations are added to V. For each such configuration c its neighbor set $N(c)$ is computed, and for each $n \in N(c)$ *compute_path*(c, n) is tested. If it succeeds, then the edge (c, n) is added to E. Finally, when s and g are connected in G, a free path in configuration space is (re)constructed using the function *construct_path*.

The only thing that remains to be specified, is the set $N(c)$ of neighbors of a (new) configuration c. It is important to note here that the choice of $N(c)$ has large impact on the efficiency of the global method. The reason for this is that executions of the local method are very time-consuming operations, due to the fact that the local method must perform intersection-tests (of the robot with the obstacles) for each path that it, successfully or not, computes. So the choice of $N(c)$ should be such, that only those edges are tried which

1. have a reasonable chance of successfully being added, and
2. are not redundant.

By an edge being redundant, we mean that in the future it can never be a necessary edge in a path (in the graph) from the start node s to the goal node g. Now in undirected graphs, an edge is redundant iff it is part of a cycle.

(1) can be dealt with by simply picking the elements of $N(c)$ from the set $\{n \in V | D(c, n) < maxdist\}$. So we only try to connect to nearby nodes. (2) can be respected by adding the edges in such a way, that the forest-like structure of the graph G (which it initially has) remains intact. This means that G will always be a collection of trees, and no cycles will exist.

It is clear that, if for a new configuration c never more than one connection (edge) to each connected component in G is added, the forest-like structure of the graph will remain intact. Motivated by (1) and (2), the set of neighbors of c is defined as follows: In every connected component of G, the nearest node to c is a neighbor of c, under the condition that $D(c, n) \leq maxdist$. More formally stated:

Definition 2

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

We assume here, for sake of simplicity, that all distances between configurations in V differ from each other. Other definitions of the neighbor set, which also respect (1) and (2), are possible, and some have been implemented and tested. Experimental results though indicate that the above definition works very well.

4.3 The global method, based on a directed graph

There are motion planning problems where no good symmetric local methods exist. E.g., if we want to use a car-like robot that can only move forwards. In this case we need

to use a directed underlying graph. We will now discuss a global method based on a directed graph, which we have implemented. Assume again that we have a robot $\mathcal{A}$, a constant *maxdist*, and functions *compute_path*, *construct_path* and D, as described in the previous section, with the crucial difference that the functions *compute_path* and D are no longer (necessarily) symmetrical. So we should no longer talk in terms of distances between configurations in general. Instead, we will refer to the values of $D(c,*)$ as the distances *from* c, and to the values of $D(*,c)$ as the distances *to* c. The global structure of the algorithm will be the same as in the undirected case, but the key question is how to add (directed) edges in a smart way.

Let us first try doing the edge-adding in a manner, very analog to the way it is done in the undirected case. So for a new via-point c, we will define what its neighbors are, and then try to connect c to those nodes with new edges. Because the edges are now directed, we cannot use just one set $N(c)$ of neighbors in general to which edge-connections will be 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 forward-neighbor set $FN(c)$ and a backward-neighbor set $BN(c)$. $FN(c)$ 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.

The question is: How are $FN(c)$ and $BN(c)$ to be defined ? Let us first try straightforward adaptions of $N(c)$ as defined in section 4.2, and see whether these are suitable.

Definition 3

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

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

where for all $c \in V$,

$forw(c) = \{d \in V | A$ *path (of length* ≥ 0*) from* c *to* d *in* G *exists*$\}$

$backw(c) = \{d \in V | A$ *path (of length* ≥ 0*) from* d *to* c *in* G *exists*$\}$

$FN(c)$ can be computed in the following way:

$FN(c) = \emptyset$
$\tilde{V} = \{n \in V - \{c\} \mid D(c,n) \leq maxdist\}$
Unmark all nodes in $\tilde{V}$
while not all nodes in $\tilde{V}$ are marked
do Let n be the node in $\{\tilde{c} \in \tilde{V} \mid \tilde{c}$ is not marked$\}$ with minimal $D(c,n)$
$FN(c) = FN(c) \cup \{n\}$
Mark all nodes in $forw(n)$

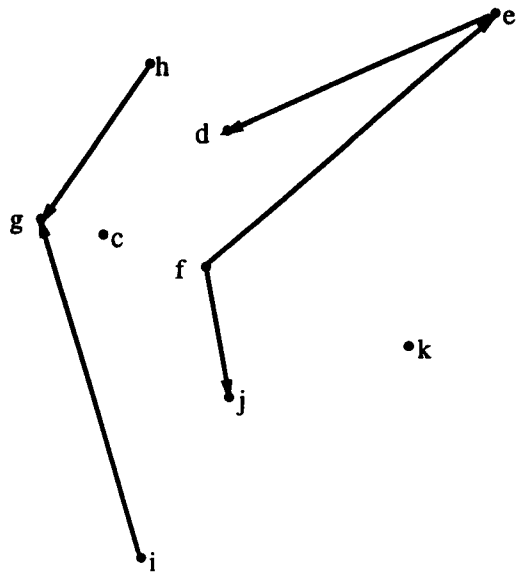

Figure 4.1: Neighbors

And analoguoesly $BN(c)$ can be computed:

$BN(c) = \emptyset$
$\tilde{V} = \{n \in V - \{c\} \mid D(n,c) \leq maxdist\}$
Unmark all nodes in $\tilde{V}$
while not all nodes in $\tilde{V}$ are marked
do Let n be the node in $\{\tilde{c} \in \tilde{V} \mid \tilde{c}$ is not marked$\}$ with minimal $D(n,c)$
$BN(c) = BN(c) \cup \{n\}$
Mark all nodes in $backw(n)$

For example in the graph shown in figure 4.1 the set of forward neighbors of c is $\{g, f, h, i, k\}$, assuming the used metric is the Euclidean. The edge-adding for a new via-point c, will now be as follows:

$$E = E \cup \{(c,a) | a \in FN(c) \wedge compute\_path(c,a)\}$$
$$\cup \{(a,c) | a \in BN(c) \wedge compute\_path(a,c)\}$$

Does this method add good edges? 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. (1) Can again be dealt with by simple only trying to connect c to nearby nodes, so let us now focus on (2).

We have seen that in an undirected graph an edge is redundant iff it is part of a cycle. This is not the case when the graph is directed.

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 the problem (= a path in the graph from the start node to the goal node) will require e. This simply means that no extension $\tilde{G}$ of the current graph G is possible such that in $\tilde{G}$ there is a path from the start node to the goal node and in $\tilde{G}$ without e this is not the case. This is formalized in the following definition:

Definition 4 *Let $G = (V_1, E_1)$ be a directed graph, where V_1 contains a start node s and a goal node g, and assume that $g \notin forw(s)$ in E_1.*
An edge $e \in E_1$ is redundant, **iff**

$$\forall E_2 \supset E_1 : (g \in forw(s) \text{ in } E_2) \Rightarrow (g \in forw(s) \text{ in } E_2 - \{e\}) \tag{4.1}$$

The question is whether redundant edges can be added by the above edge adding method. This appears to be so. Consider again figure 4.1. Assume that the edges (c, g) and (c, h) are successfully added. Then it is clear that (c, g) complies with definition 4, because any (future) path from the start node to the goal node which contains (c, g) can be replaced by a path which does not.

Definition 4 though does not provide suitable criteria which could effectively be used for computing non-redundant edges, which is our aim. We want some simple rules, that exactly state whether an edge is redundant or not, and can easily (efficiently) be verified. Theorem 1 will be of help. It states that an edge (a, b) is redundant iff there is a path not containing (a, b) from the start node to b, from a to b, or from a to the goal node.

Theorem 1 *Let $G = (V_1, E_1)$ be a directed graph, where V_1 contains a start node s and a goal node g, E_1 contains an edge $e = (a, b)$,and assume that $g \notin forw(s)$.*

e is redundant in E_1

iff $b \in forw(a)$ *in* $E_1 - \{e\}$
$\vee$
$b \in forw(s)$ *in* $E_1 - \{e\}$
$\vee$
$a \in backw(g)$ *in* $E_1 - \{e\}$

Proof
Let $G = (V_1, E_1)$ be a directed graph, such that $V_1 \supset \{s, g\}$, $E_1 \ni (a, b)$, and $g \notin forw(s)$ in E_1. Let $e = (a, b)$. We will consider paths in the graph as lists of edges, which can thus be concatenated by the operator $+\!\!+$, as mentioned in chapter 2.

1. Assume that $b \in forw(a)$ in $E_1 - \{e\}$. Let P_{ab} be a path from a to b in $E_1 - \{e\}$, E_2 an arbitrary extension of E_1 such that $g \in forw(s)$ in E_2, and P_{sg} a path from s to g in E_2, with minimal number of edges. If $P_{sg} \not\ni e$, then it is clear that $g \in forw(s)$ in $E_2 - \{e\}$. So now suppose that $P_{sg} \ni e$. Then $P_{sg} = P_{sa} +\!\!+ [e] +\!\!+ P_{bg}$ for certain P_{sa} and P_{bg}, not containing e. But then $P_{sa} +\!\!+ P_{ab} +\!\!+ P_{bg}$ is also a path from s to g in E_2, and this path does not contain e. This means that $g \in forw(s)$ in $E_2 - \{e\}$, which proves that
$$b \in forw(a) \text{ in } E_1 - \{e\} \Rightarrow e \text{ is redundant in } E_1$$

2. Assume that $b \in forw(s)$ in $E_1 - \{e\}$. Let P_{sb} be a path from s to b in $E_1 - \{e\}$, E_2 an arbitrary extension of E_1 such that $g \in forw(s)$ in E_2, and P_{sg} a path from s to g in E_2, with minimal number of edges. If $P_{sg} \not\ni e$, then it is clear that $g \in forw(s)$ in $E_2 - \{e\}$. So now suppose that $P_{sg} \ni e$. Then $P_{sg} = P_{sa} +\!\!+ [e] +\!\!+ P_{bg}$ for certain P_{sg} and P_{bg}, not containing e. But then $P_{sb} +\!\!+ P_{bg}$ is also a path from s to g in E_2, and this path does not contain e. This means that $g \in forw(s)$ in $E_2 - \{e\}$, which proves that
$$b \in forw(s) \text{ in } E_1 - \{e\} \Rightarrow e \text{ is redundant in } E_1$$

3. Assume that $a \in backw(g)$ in $E_1 - \{e\}$. Let P_{ag} be a path from a to g in $E_1 - \{e\}$, E_2 an arbitrary extension of E_1 such that $g \in forw(s)$ in E_2, and P_{sg} a path from s to g in E_2, with minimal number of edges. If $P_{sg} \not\ni e$, then it is clear that $g \in forw(s)$ in $E_2 - \{e\}$. So now suppose that $P_{sg} \ni e$. Then $P_{sg} = P_{sa} + [e] + P_{bg}$ for certain P_{sa} and P_{bg}, not containing e. But then $P_{sa} + P_{ag}$ is also a path from s to g in E_2, and this path cannot contain e. This means that $g \in forw(s)$ in $E_2 - \{e\}$. So this proves that

$$a \in backw(g) \text{ in } E_1 - \{e\} \Rightarrow e \text{ is redundant in } E_1$$

4. Assume that $b \notin forw(a)$ in $E_1 - \{e\}$, $b \notin forw(s)$ in $E_1 - \{e\}$ and $a \notin backw(g)$ in $E_1 - \{e\}$. We want to show that

$$\exists E_2 \supset E_1 : g \in forw(s) \text{ in } E_2 \wedge g \notin forw(s) \text{ in } E_2 - \{e\}$$

Consider $E_2 = E_1 \cup \{(s,a),(b,g)\}$.

- $[(s,a),e,(b,g)]$ is a path from s to g in E_2, so $g \in forw(s)$ in E_2.
- Now assume that $g \in forw(s)$ in $E_2 - \{e\}$. Let P_{sg} be a path from s to g in $E_2 - \{e\}$, with minimal number of edges. Then $P_{sg} \ni (s,a) \vee P_{sg} \ni (b,g)$, because otherwise $g \in forw(s)$ in E_1.

  case $P_{sg} \ni (s,a)$: $P_{sg} = [(s,a)] + P_{ag}$ for a certain path P_{ag} in $E_2 - \{e\}$. P_{ag} must contain either (s,a) or (b,g), because otherwise $a \in backw(g)$ in $E_2 - \{e\}$. Now if $P_{ag} \ni (s,a)$, then P_{sg} contains a loop and is therefore not of minimal length, so $P_{ag} \not\ni (s,a)$. This means that $P_{ag} \ni (b,g)$, so $P_{ag} = P_{ab} + [(b,g)]$ for a path P_{ab} in $E_2 - \{e\}$. Now P_{ab} must contain either (s,a) or (b,g), because otherwise $b \in forw(a)$ in $E_1 - \{e\}$. But this means that P_{ab} contains a loop, and P_{sg} is not of minimal length. So we have a contradiction.

  case $P_{sg} \ni (b,g)$: $P_{sg} = p_{sb} + [(b,g)]$ for a certain path P_{sb} in $E_2 - \{e\}$. P_{sb} must contain either (s,a) or (b,g), because otherwise $b \in forw(s)$ in $E_2 - \{e\}$. But $P_{sb} \ni (s,a)$ has now assumed not to be true. So $P_{sb} \ni (b,g)$, but then P_{sg} contains a loop, which means that it is not of minimal length. Again we have a contradiction.

  Both cases lead to contradictions, so $g \notin forw(s)$ in $E_2 - \{e\}$.

(1),(2),(3) and (4) prove the theorem. □

Theorem 1 thus gives the following rules for adding an edge (a,b):

1. $b \notin forw(a)$ in $E - \{(a,b)\}$
2. $b \notin forw(s)$ in $E - \{(a,b)\}$
3. $a \notin backw(g)$ in $E - \{(a,b)\}$

If all are respected by an edge $(a,b) \in E$, then (a,b) is not redundant, and if any of them is violated by (a,b), then (a,b) is redundant. This suggests the following edge-adding method M_1 for a new node c:

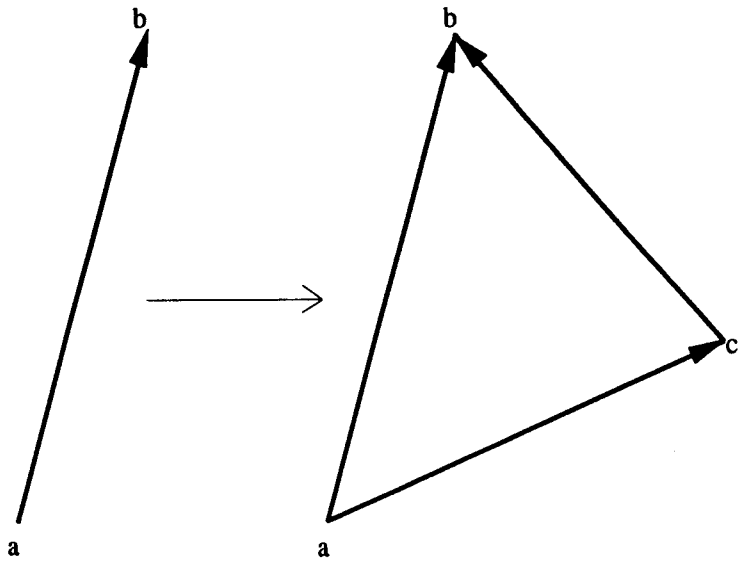

Figure 4.2: redundancies

forall $n \in FN(c)$
  if $n \notin forw(c) \wedge n \notin forw(s) \wedge c \notin backw(g) \wedge compute\_path(c, n)$
  then $E = E \cup \{(c, n)\}$
forall $n \in BN(c)$
  if $c \notin forw(n) \wedge c \notin forw(s) \wedge n \notin backw(g) \wedge compute\_path(n, c)$
  then $E = E \cup \{(n, c)\}$

where $FN(c)$ and $BN(c)$ are defined in some suitable way, for example as in definition 3.

There are though a few problems. From now on we will refer to the set of new edges, added due to a (new) node c, by $E(c)$. One problem is that the set $E(c)$ added by method M_1 depends on the order in which the elements of $FN(c)$ and $BN(c)$ are picked, and also on the ordering of the two loops. This means that the method, in the given form, is not deterministic.

But another, perhaps even more serious problem exists. Although it is guaranteed that an edge e added by method M_1 is not redundant at the moment when it is added, it is very well possible that in the future it will become redundant. An example of this is illustrated in figure 4.2. E contains the (not redundant) edge (a, b), at the moment that c is added. The local method succeeds in computing paths from a to c and from c to b, so the (not redundant) edges (a, c) and (c, b) are added. But now $b \in forw(a)$ in $E - \{(a, b)\}$, which means that (a, b) has become redundant.

The question is whether it is possible, like for undirected graphs, to do the edge-adding in such a manner that the graphs always remains entirely free of redundant edges, without laying restrictions on new edges which have a negative effect on the structure of the graph. Naturally, after a node c and a set of edges $E(c)$ have been added, all edges in E could be tested for redundancy, and redundant ones could simply be removed. This can be done in $\Theta(|E|)$ time, but it is of no use. As mentioned in section 4.2, our aim is to prevent useless executions of the (relatively expensive) local method. So we want to *prevent* computing edges which are redundant, or will be so in the future. Once they are computed the time

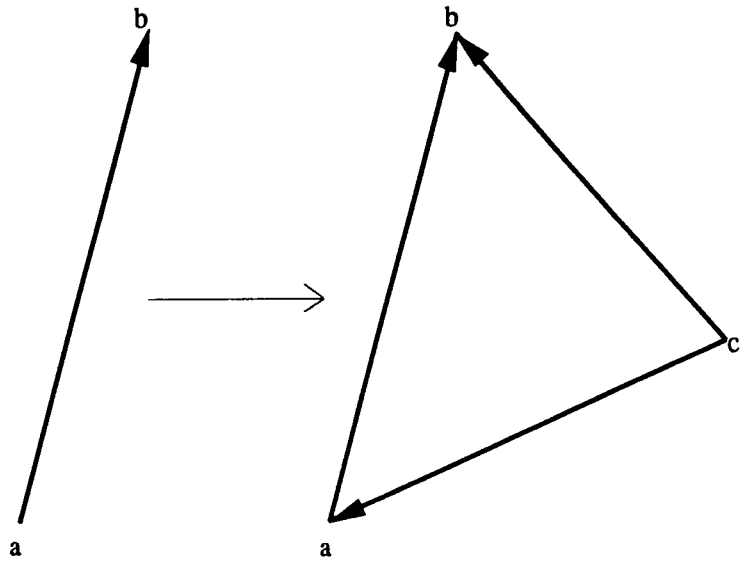

Figure 4.3: Local redundancies

has been spent, so removing them is useless.

In the general case though, it does not seem to be possible doing so. Consider again figure 4.2. At the moment that (a, b) is added, there is no way to predict that it will become redundant, because its future redundancy is caused by a future node c, which is a *randomly* chosen configuration. Of course, we could simply forbid new edges which make some already present edges redundant, but this means that useful (= not redundant) and possible edges would be forbidden.

With method M_1 though an edge in $E(c)$ can not only in the future become redundant (this does not seem preventable), but it can be already redundant in $E \cup E(c)$. We say that $E(c)$ contains *local redundancies*. See figure 4.3. Assume that c is a new node, and that the local method can compute a path from c to a as well as path from c to b. If (c, b) is added before (c, a), then adding (c, a) will cause (c, b) to become redundant. Our goal will now be to formulate good *deterministic* methods, such that for a new node c the set of edges $E(c)$ contains *no local redundancies*.

We will now first focus on the restrictions imposed on the set $E(c)$ by rules (2) and (3), and later we will come to rule (1). The set of outgoing edges $(c, *)$, added due to c, will be denoted by $E_f(c)$, and the set of incoming edges by $E_b(c)$. Rule (2) implies

- $\nexists (c, n) \in E_f(c) : n \in forw(s)$
- $(\exists (n, c) \in E_b(c) : n \in forw(s)) \Rightarrow (E_b(c) = \{(n, c)\})$

and rule (3) implies

- $\nexists (n, c) \in E_b(c) : n \in backw(g)$
- $(\exists (c, n) \in E_f(c) : n \in backw(g)) \Rightarrow (E_f(c) = \{(c, n)\})$

Any edge-adding method should thus be a specialization of the following general method M_2: (assume c is the new node)

Try to add an edge (c, n) with $n \in backw(g)$
if $\neg success$
then
(I) Try adding other outgoing edges (c, n)
with $n \in V - (forw(s) \cup backw(g) \cup \{c\})$

Try to add an edge (n, c) with $n \in forw(s)$
if $\neg success$
then
(II) Try adding other incoming edges (n, c)
with $n \in V - (forw(s) \cup backw(g) \cup \{c\})$

We will now focus on the steps (I) and (II) where 'other' edges are to be added. An important observation is, that edges (n, c) and (c, m) with $n, m \notin forw(s) \cup backw(g) \cup \{c\}$ cannot possibly violate rules (2) or (3), so the only rule to worry about is rule (1).

If an edge (c, a) is added, then any edge (c, b) with $b \in backw(a) \cup forw(a)$ will cause a local redundancy in $E_f(c)$. If $b \in backw(a)$ then (c, a) will violate rule (1), and if $b \in forw(a)$ then (c, b) will violate rule (1). So this demands that whenever an edge (c, a) is added, the forward and backward sets of a are removed from the forward neighbor set of a. An analog argument holds for incoming edges, so whenever an edge (a, c) is added, the forward and backward sets of a must be removed from the backward neighbor set of a. This gives the following specialization M_3 of method M_2:

Try to add an edge (c, n) with $n \in backw(g)$
if $\neg success$
then $V_f = FN(c)$
while $V_f \neq \emptyset$
do $n =$ an element of V_f
$V_f = V_f - \{n\}$
if *compute_path*(c, n)
then $V_f = V_f - forw(n) \cup backw(n)$
$E = E \cup \{(c, n)\}$

Try to add an edge (n, c) with $n \in forw(s)$
if $\neg success$
then $V_b = BN(c)$
while $V_b \neq \emptyset$
do $n =$ an element of V_b
$V_b = V_b - \{n\}$
if *compute_path*(n, c)
then $V_b = V_b - forw(n) \cup backw(n)$
$E = E \cup \{(n, c)\}$

where $FN(c)$ and $BN(c)$ are defined in some suitable manner.

What remains to be specified are the sets $FN(c)$ and $BN(c)$, and some (fixed) order in which the elements of $FN(c)$ and $BN(c)$ are to be picked. This will give a deterministic method that adds sets of edges E(c) which contain no local redundancies.

For the neighbor sets $FN(c)$ and $BN(c)$ we have decided on the following definitions, which are based on definition 3.

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

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

where $\tilde{V} = V - (forw(s) \cup backw(g) \cup \{c\})$

Thus as forward and backward neighbors we take the intersections of the sets of forward and backward neighbors as defined by definition 3, with the set $V - (forw(s) \cup backw(g) \cup \{c\})$. We will show that using such neighbors makes it possible to do the edge adding deterministically, in a way that the added sets of edges $E(c)$ are very 'good'. The key is that the elements of V_f are picked *in order of decreasing distance from* c, and the elements of V_b are picked *in order of decreasing distance to* c. Proceeding in this way, it can easily be shown that (in the first loop of method M_3) $V_f \cap backw(n)$ is always empty, and analog (in the second loop of M_3) $V_b \cap forw(n)$ is always empty. So after each successful execution of *compute_path*(c,n), only $forw(n)$ needs to be removed from V_f, and after each successful execution of *compute_path*(n,c), only $backw(n)$ needs to be removed from V_b (instead of $forw(n) \cup backw(n)$).

So this leads to the following version M_4 of method M_3: (assume c is the new node)

Try to add an edge (c,n) with $n \in backw(g)$
if $\neg success$
then $V_f = FN(c)$
  while $V_f \neq \emptyset$
  do n = the element of V_f with maximal $D(c,n)$
    $V_f = V_f - \{n\}$
    if *compute_path*(c,n)
    then $V_f = V_f - forw(n)$
      $E = E \cup \{(c,n)\}$

Try to add an edge (n,c) with $n \in forw(s)$
if $\neg success$
then $V_b = BN(c)$
  while $V_b \neq \emptyset$
  do n = the element of V_b with maximal $D(n,c)$
    $V_b = V_b - \{n\}$
    if *compute_path*(n,c)
    then $V_b = V_b - backw(n)$
      $E = E \cup \{(n,c)\}$

Let us (again) refer to the set of newly added edges (by method M_4) due to a new node c by $E(c)$. The motivation for picking the neighbors in the described manner is that

1. $forw(c)$ in $E \cup E(c) = forw(c)$ in $E \cup \{(c,n) \mid n \in FN(c) \wedge compute\_path(c,n)\}$
2. $backw(c)$ in $E \cup E(c) = backw(c)$ in $E \cup \{(n,c) \mid n \in BN(c) \wedge compute\_path(n,c)\}$

This means that method M_4 is equally powerful as the method where forward connections to all elements of $FN(c)$ and backward connections to all elements of $BN(c)$ are tried. So $forw(c)$ in $E \cup E(c)$ is surely maximal with respect to the forward sets of c in graphs resulting from other picking orders. And analogously, $backw(c)$ in $E \cup E(c)$ is surely maximal with respect to the backward sets of c in graphs resulting from other picking orders.

Proof of (1) :
Let $E_1 = E \cup \{(c,n) \mid n \in FN(c) \wedge compute\_path(c,n)\}$, $E_2 = E \cup E(c)$, $V_1 = forw(c)$ in E_1, and $V_2 = forw(c)$ in E_2.
The claim is that $V_1 = V_2$.

- $V_1 \supset V_2$ is evident.
- Assume that $V_1 \not\subset V_2$, and let a be a node such that $a \in V_1 \wedge a \notin V_2$. $a \in V_1$ means that $\exists (c,b) \in E_1 : a \in forw(b)$. So $b \in FN(c)$ and $compute\_path(c,b)$. $a \notin V_2$ implies that $(c,b) \notin E_2$. So b is removed from V_f (in method M_4). This means that there exists a $(c,d) \in E(c)$ with $b \in forw(d)$. But $(c,d) \in E(c), b \in forw(d)$ and $a \in forw(b)$ implies that $a \in forw(c)$ in $E \cup E(c)$ $(=E_2)$. Hence we have a contradiction.

□

The second claim can be proven in a very analog manner.

To complete the edge adding algorithm for directed graphs, the steps where edges (c,n) with $n \in backw(g)$ and (n,c) with $n \in forw(s)$ are to be searched for should be specified more precisely. It does not seem possible to do very much smart work here. For the former of the mentioned steps, we simply try to find one (and not more than one) edge (c,n) with $n \in backw(g)$. So the obvious way to do this is to take a number of nearby nodes in $backw(g)$ (if such nodes exist), and try to compute a connection to each of these, stopping when one of these connections is successfully computed (resulting in an outgoing edge, to $backw(g)$). And analog for the latter of the two mentioned steps, where a backward connection to $forw(s)$ is to be searched for.

In our implementation of the method we have proceeded in the sketched way. Assume k is a constant defining the (maximal) number of nodes in both $forw(s)$ and $backw(g)$, that we try to connect to from a new node c. For the step where an edge (c,n) with $n \in backw(g)$ is searched for, we do the following: First we compute the set of c's *goal neighbors* GN(c). $GN(c)$ is defined as the set of the $\tilde{k}$ nearest nodes in $backw(g)$ *from* c, where $\tilde{k}$ is the number of nodes in $backw(g)$ that lie within distance *maxdist from* c, if this number is smaller than k, and k otherwise. Then, in order of increasing distance *from* c, we pick nodes from $GN(c)$, and try to compute forward connections to them. When a connection is successfully computed (resulting in an edge from c to $backw(g)$), or all goal neighbors have been picked, we stop.

Analogously, for the step where an edge (n,c) with $n \in forw(s)$ is searched for, we compute the set of c's *start neighbors* $SN(c)$. $SN(c)$ is defined as the set of the $\tilde{k}$ nearest nodes in $forw(s)$ *to* c, where $\tilde{k}$ is the number of nodes in $forw(s)$ that lie within distance

maxdist to c, if this number is smaller than k, and k otherwise. These start neighbors are then picked in order of increasing distance *to c*, until one of them is successfully connected to c, or all have been picked.

A question of course is which value to choose for the constant k, the maximal size of the start and goal sets. If it is chosen large, then often a lot of effort will be done to connect new nodes to the start and the goal set. This will cause these two sets to grow relatively fast, in comparison to other parts of the graph, but on the other hand the adding of a new node becomes a more time consuming operation, which means that on the whole the graph will grow slower. On the other hand, choosing k small means that the node adding operation will become a bit cheaper, but the start and goal sets will grow slower in relation to other parts of the graph. We have done some tests with different values of k for a number of 'typical' scenes, and as a result we have chosen for $k = 6$ in the implementation of the method.

To complete this chapter, we will give the entire edge adding algorithm for directed graphs. Assume that $G = (V, E)$ is the 'current' (directed) underlying graph. We define the set of forward neighbors $FN(c)$, the set of backward neighbors $BN(c)$, the set of start neighbors $SN(c)$, and the set of goal neighbors $GN(c)$ as follows:

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

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

where $\tilde{V} = V - (forw(s) \cup backw(g) \cup \{c\})$

$GN(c)$ = The set of the $\tilde{k}_g$ nearest elements in $backw(g)$ *from* c,
where $\tilde{k}_g = MIN(k, \#\{n \in backw(g) \mid D(c,n) \leq maxdist\})$

$SN(c)$ = The set of the $\tilde{k}_s$ nearest elements in $forw(s)$ *to* c,
where $\tilde{k}_s = MIN(k, \#\{n \in forw(s) \mid D(n,c) \leq maxdist\})$

The edge adding algorithm can now be stated in the following way: (assume that c is the new node)

$success = false$
$V_g = GN(c)$
while $V_g \neq \emptyset \wedge \neg success$
do n = the element in V_g with minimal $D(c,n)$
$V_g = V_g - \{n\}$
if $compute\_path(c,n)$
then $E = E \cup \{(c,n)\}$
$success = true$

if $\neg success$
then $V_f = FN(c)$
  while $V_f \neq \emptyset$
  do n = the element of V_f with maximal $D(c,n)$
    $V_f = V_f - \{n\}$
    if $compute\_path(c,n)$
    then $V_f = V_f - forw(n)$
      $E = E \cup \{(c,n)\}$

$success = false$
$V_s = SN(c)$
while $V_s \neq \emptyset \wedge \neg success$
do n = the element in V_s with minimal $D(n,c)$
  $V_s = V_s - \{n\}$
  if $compute\_path(n,c)$
  then $E = E \cup \{(n,c)\}$
    $success = true$

if $\neg success$
then $V_b = BN(c)$
  while $V_b \neq \emptyset$
  do n = the element of V_b with maximal $D(n,c)$
    $V_b = V_b - \{n\}$
    if $compute\_path(n,c)$
    then $V_b = V_b - backw(n)$
      $E = E \cup \{(n,c)\}$

Chapter 5

Node adding strategies

5.1 Disadvantages of fully random node adding

In the global methods described in the previous chapter, the nodes of the underlying graph are added in a fully random way. This node adding strategy will be referred to as the *normal node adding strategy*. In this chapter some more sophisticated node adding strategies will be discussed and motivated. Conceptually, the strategies which will be described in this chapter are applicable to both directed and undirected underlying graphs. Where necessary, we will make a distinction between both cases.

Experiments have shown that the normal node adding strategy is not at all a bad method. In fact it performs surprisingly well, better than, e.g., adding configurations in some regular pattern. The method does though have some shortcomings.

1. It typically adds the same amount of nodes at easy places, as at difficult ones, near to obstacles. This disagrees with the intuition that more work should be done at difficult places than at easy ones.

2. No account is taken with the underlying graph computed so far. The graph, even when the entire problem is not yet solved, contains a great deal of information, about motions that are already 'known'. This information about the graph could be used for locating areas in configuration space, where the adding of a new node is relatively likely to bring about valuable extensions of the graph, and also for locating areas where just the contrary is true.

3. 'Unlucky' configurations can be added, which are very hard or impossible to connect to other configurations, even if these are very near. Or stated differently, a configuration c can be added such that the robot, positioned at c, can impossibly or only with great difficulty maneuver its way out. The adding of such 'unlucky' configurations typically results in the presence of unconnected nodes in the graph G. Now why is this a problem? Let us assume that the underlying graph G is undirected, as described in section 4.2, and that c is an unconnected node in G. Then, as long as c remains unconnected, every time a new node $\tilde{c}$ is added within distance *maxdist* of c, c will be chosen as a neighbor of $\tilde{c}$. Consequently, the presence of an unconnected node over a longer period results in many (useless) executions of the local method. An additional drawback is that, when a connection to an unconnected node

is successfully computed, the gain is relatively small. For the case where a directed underlying graph is used, an analog argument holds.

A number of approaches for dealing with the sketched problems will be discussed. (1) and (2) have already been addressed to in [Ove92], and in sections 5.2 and 5.3 we will discuss the solutions proposed in that paper. Then in section 5.4 problem (3) will be addressed.

5.2 The forbidden configurations node adding strategy

Problem (1) can be dealt with by using *forbidden configurations*, i.e., random configurations which are not in free space. The idea is based on the observation that typically near to a difficult passage in free configuration space, there is a large region ζ in forbidden configuration space. Hence, although it is very unlikely that a configuration is added in the difficult passage, the chance that a configuration is added in ζ is quite large. So if the random configurations which 'fall' inside a forbidden region ζ are moved into the difficult allowed regions which are adjacent ζ, then on the whole more nodes will be added at difficult places then at easy ones. Now rather than performing difficult geometric computations for determining which direction is optimal, we (again) use a simple random approach: Whenever a forbidden configuration c is generated, we choose a *random* direction and move the configuration (in small steps) in this direction until it is in free space (or out of boundaries).

It is clear that a trade-off occurs when this strategy is used. On one hand, the average computation time spent on the adding of a node increases, but on the other hand the nodes added are 'better' then when added fully randomly, and, hence, the average size of the graph decreases (on the average).

The approach is very useful for scenes where the robot must pass through some very narrow passage(s), in order to get from the start to the goal configuration. See figure 5.1. Normally a great amount of nodes will be added in the two large free areas, and it will take very long before enough nodes will be added in the narrow passage which connects the two areas. If though the forbidden configurations strategy is applied, then some nodes which are added in the forbidden regions adjacent to the passages, will be moved into the passage, and, hence, relatively many nodes appear in the passage. Figure 5.2 shows the result of applying the forbidden configurations strategy in the discussed scene.

5.3 The adaptive node adding strategy

For problem (2), we can use the *adaptive adding strategy*. The idea here is that, when a node c is to be added, some properties of the underlying graph in the neighborhood of c are computed, and this information is used for computing the *adding chance* for c. So c is not always added, but only with a certain probability. The computation of the adding chances should of course be done in such a way that 'favorable' nodes are more likely to be added than 'unfavorable' ones. For undirected underlying graphs, the following criteria are used for the computation of the adding chance for a new node c:

- The number of nodes (*nodenumb*) within distance *maxdist* of c.

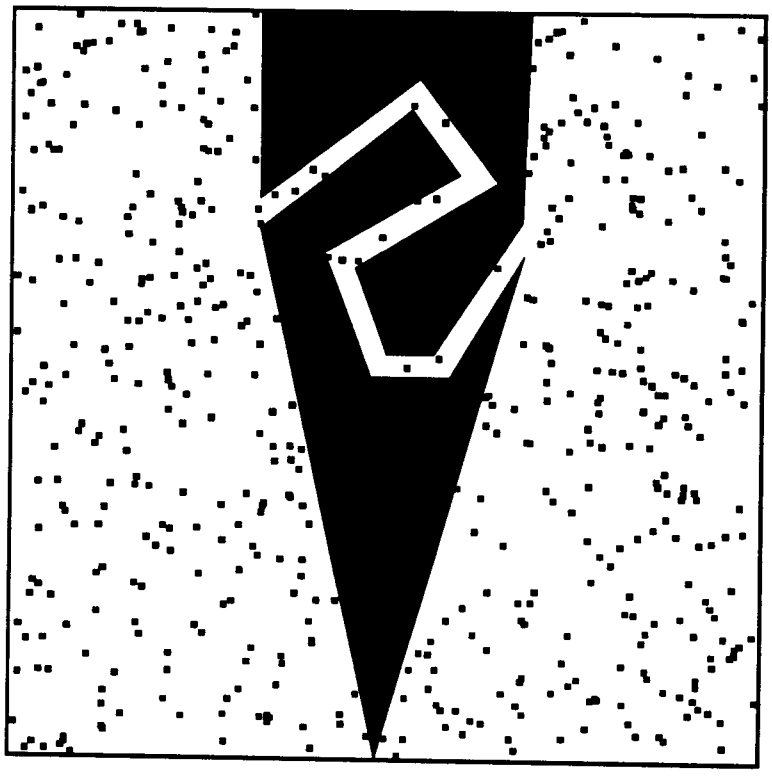

Figure 5.1: Normal node adding.

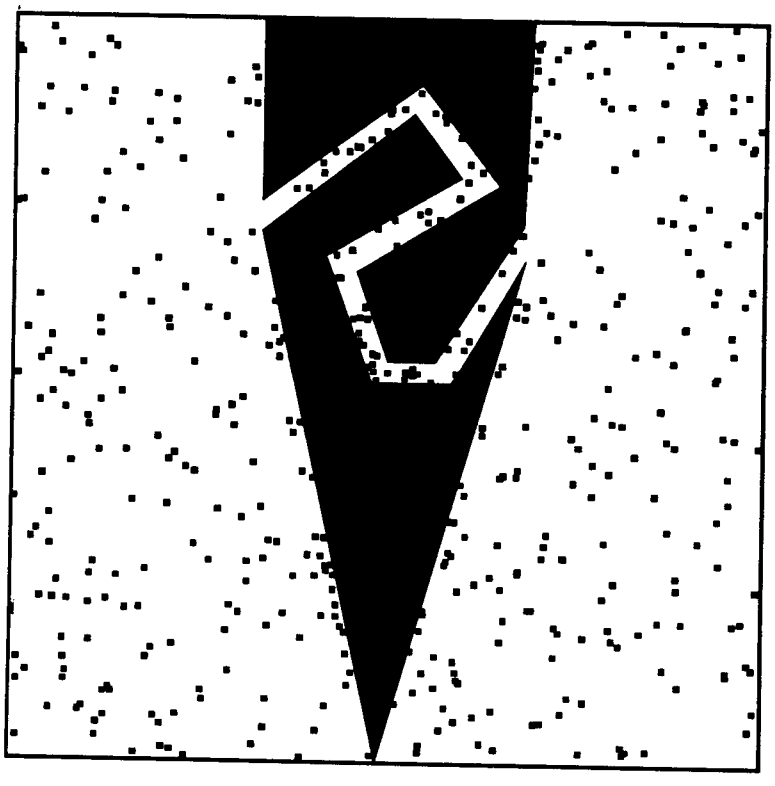

Figure 5.2: Using the forbidden configurations strategy.

- The number of different connected components (*compnumb*) among them.
- Whether the component containing the start configuration (the start component) is within distance *maxdist* of c.
- Whether the component containing the goal configuration (the goal component) is within distance *maxdist* of c.

The computation is as follows:

if $nodenumb \leq 4$
then $chance = 1.0$
else if the start component and the goal component are within distance *maxdist* of c
  then $chance = 1.0$
  else if $compnumb > 1$
    then if the start component or the goal component is within distance *maxdist* of c
      then $chance = 0.75$
      else $chance = 0.5$
    else $chance = 0.25$

So if there are almost no nodes in the neighborhood, then c is always added. The idea here is that too little is known about the neighborhood of c to allow a sensitive decision being made here. Also if both the start and the end component are in the neighborhood, then we always add c. The reason for this of course is that nodes which can be connected to both the start and the goal configuration solve the whole problem. Otherwise we count the number of different connected components which are present within distance *maxdist* of c, and determine whether the start component or the goal component are among them. We then prefer the presence of more components as well as that of either of the two end components.

For directed graphs, the method must be adapted slightly, because of the absence of connected components, and due to the fact that the metric is no longer symmetrical.

So instead of simply counting all nodes n with $D(n,c) \leq maxdist$, we now must count all nodes n with $D(n,c) \leq maxdist \vee D(c,n) \leq maxdist$. The analog for checking the distance from c to the start component in an undirected graph, is checking the distance $D(*,c)$ from the forward set of the start configuration to c. And the analog for checking the distance from c to the goal component in an undirected graph, is checking the distance $D(c,*)$ from c to the backward set of the goal configuration. So the criteria for the computation of the adding chance a new node c now are :

- The number of nodes n (*nodenumb*) with $D(c,n) \leq maxdist \vee D(n,c) \leq maxdist$.
- Whether the forward set ($forw(s)$) of the start configuration is within distance *maxdist* to c.
- Whether the backward set ($backw(s)$) of the start configuration is within distance *maxdist* from c.

The computation chosen is as follows:

if $nodenumb \leq 4$
then $chance = 1.0$
else if $\exists n_s \in forw(s) : D(n_s, c) \leq maxdist \wedge \exists n_g \in backw(g) : D(c, n_g) \leq maxdist$
then $chance = 1.0$
else if $\exists n_s \in forw(s) : D(n_s, c) \leq maxdist \vee \exists n_g \in backw(g) : D(c, n_g) \leq maxdist$
then $chance = 0.67$
else $chance = 0.33$

The motivations for choosing the adding chances in this way are very analog to those given for the undirected case.

When the adaptive adding strategies are applied, then a similar trade-off takes place as is the case for the forbidden configurations strategy. It should be noted that the constants in both the above methods are a bit arbitrary and that they could probably be tuned to more optimal values.

As mentioned before, the adaptive node adding strategy was already proposed in [Ove92], and it was included in the corresponding implementation of the random motion planner for free flying planar robots. For these robots the strategy clearly brings some improvement (for testing results see again [Ove92]). For car-like robots though the performance of the adaptive node adding strategy seems to be somewhat disappointing. The main reason for this probably is, that for car-like robots the used metrics are much more expensive to evaluate than the Euclidean metric which is used for free-flying robots, and application of the adaptive adding strategy brings about that on the average more distance computations are needed for the adding of a node, than when this strategy is not used.

5.4 The edge sensitive/requiring node adding strategy

Now we will address the problem (3) of the 'unlucky' nodes, which stay unconnected in the graph. This problem is very much related to the properties of the robot that we are dealing with. Typically, a robot with kinematic constraints, like for example a car-like robot, brings about more problems than free flying robot. Intuitively, the reason for this is that a constrained robot will have more difficulty in maneuvering itself out of a difficult configuration than an unconstrained one. In fact, for free flying robots the number of unconnected nodes normally remains very low. For this reason problem (3) has not been addressed in [Ove92].

Here we should also mention the notion of *robot controllability* (see chapter 4 in [Lat91]). Assuming that the free configuration space is connected, a robot which is *fully controllable* will in theory always be able to maneuver its way out from any (difficult) configuration, in contrast to a robot which is not fully controllable. So we can expect the problem to be more severe for robots which are not fully controllable, than for such that are. Free flying robots and car-like robots which can move both forwards as well as backwards, are two examples of fully controllable robots, while for example car-like robots which can only move forwards are not fully controllable.

Of course, the robot being fully controllable does by no means guarantee that no unconnected nodes will appear in the graph. Firstly, for a new node c there may simply be no nodes within distance *maxdist* of c, which means that the neighbor set(s) of c is (are) empty, so surely c will not be connected to any nodes. And secondly, even if the

neighbor set(s) of c does (do) contain some nodes, it is very well possible that the local method used will not succeed in computing feasible paths to any of those nodes, even though we know that such paths exist. The reason for this is that the local methods used are not complete (if they where, then we would not need a global method).

We will now describe two approaches to limiting the number of unconnected nodes in the graph. The most straightforward way for preventing unconnected nodes to appear in the graph is the following: For every node that is added, we check whether it is connected to some other node in the graph (during the same iteration of the main loop of the global method, see 4.2). If this is not the case, then the node is removed from the graph. The nodes which are added and not removed, will now be referred to as *successfully* added nodes. This method will be referred to as *the edge requiring node adding strategy.* It is applicable to both directed and undirected underlying graphs. Now how does this adding strategy behave?

Let us first consider the case of an undirected underlying graph G. Initially G consists of exactly two connected components, i.e., the start and the goal component. Then every node which is successfully added must be connected to an existing connected component, because otherwise it is removed. So the number of connected components never increases, and it follows that the only connected components that G ever contains are the start and the goal component. Hence the only extensions of G that take place are extensions of the start and the goal component.

If the underlying graph G is directed, then we have something very similar. Initially there is the start node s, member of $forw(s)$, and the goal node g, member of $backw(g)$. Every node which successfully added to the graph, must either be forwards connected to a node not in $forw(s)$, or backwards connected to a node not in $backw(g)$. But all nodes not in $forw(s)$ are members of $backw(g)$. Hence if a new node is forward connected to a present node n, then n must be a member of $backw(g)$, and consequently the new node will itself become a member of $backw(g)$. Analogously, all nodes not in $backw(g)$, are members of $forw(s)$. Thus a new node which is backwards connected to a present node, is backwards connected to $forw(s)$, and consequently itself becomes a member of $forw(s)$.

This shows that the only possible extensions of G are such which extend $forw(s)$ or $backw(g)$. Note that in the edge adding algorithm for directed graphs (see 4.3), when combined with the edge requiring node adding strategy, the sets of forward and backward neighbors of a new node c ($FN(c)$ and $BN(c)$) will always be empty, which makes some of the given 'code' redundant.

So in both cases the edge requiring adding strategy makes the underlying graph grow from the start and the goal component. Experimental results prove that for many scenes this node adding strategy works very well (see chapter 8).

Sometimes though it seems more favorable to distribute the work more equally over the configuration space, i.e., allow more graph extensions than only such which extend the start or the goal component. This is especially the case for scenes where every feasible path which connects the start configuration to the goal configuration is relatively long. A more equal distribution of the nodes can be achieved by a simple modification of the edge requiring node adding strategy, which will be referred to as the *edge sensitive node adding strategy.* The difference is that apart from new nodes which are immediately connected to others, we now also allow nodes which have no neighbors, i.e., nodes which have no other nodes in their neighborhood. So when a node n is added, it is kept iff it has no neighbors or it is successfully connected to another node.

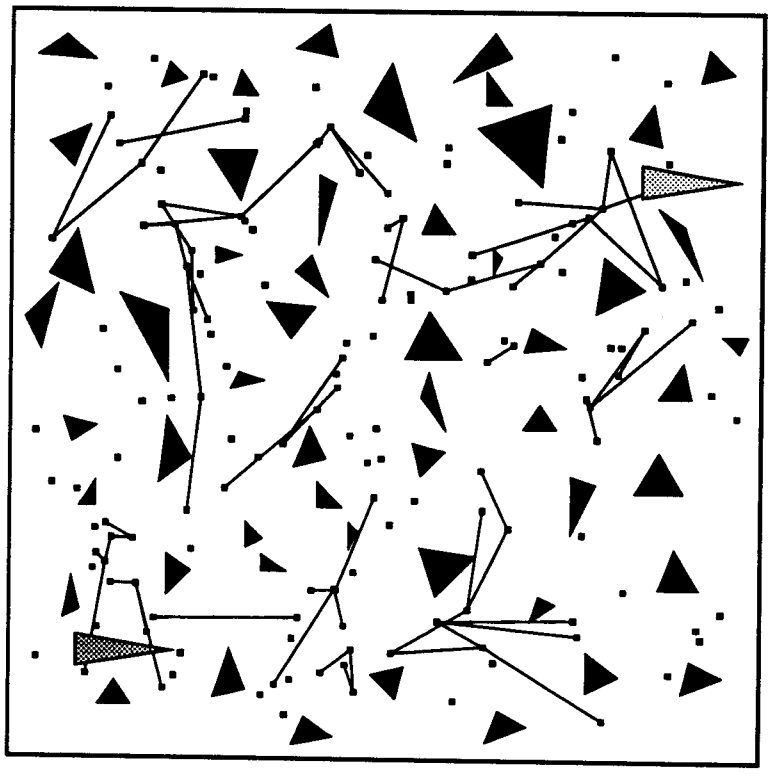

Figure 5.3: Normal node adding.

For both an undirected as well as a directed underlying graph this results in the following behavior: initially a lot of nodes are added which are unconnected. They are allowed because they have no neighbors. But as time passes, more and more new nodes will have neighbors, and, hence, the amount of newly added unconnected nodes decreases. After a while, every new node has some neighbors, which means that the only possible extensions of the graph are extensions of already present components. So initially lots of (small) new components appear, but after a while this stops, and the only thing that happens is that present components grow.

See figures 5.3, 5.4, and 5.5, for visualizations of underlying (undirected) graphs computed with the different node adding strategies, all in the same scene. In none of the shown graphs is the start node connected to the goal node yet. Figure 5.3 shows a graph computed with the normal node adding strategy. We see many unconnected nodes and many small components. In figure 5.4 we see a graph computed by the edge requiring node adding strategy. There are just two components present, one containing the start node, and the other containing the goal node. Figure 5.5 shows a graph computed with the edge sensitive node adding strategy. It shows that this strategy behaves as kind of a compromise of the two former strategies. The number of different connected components typically remains very low.

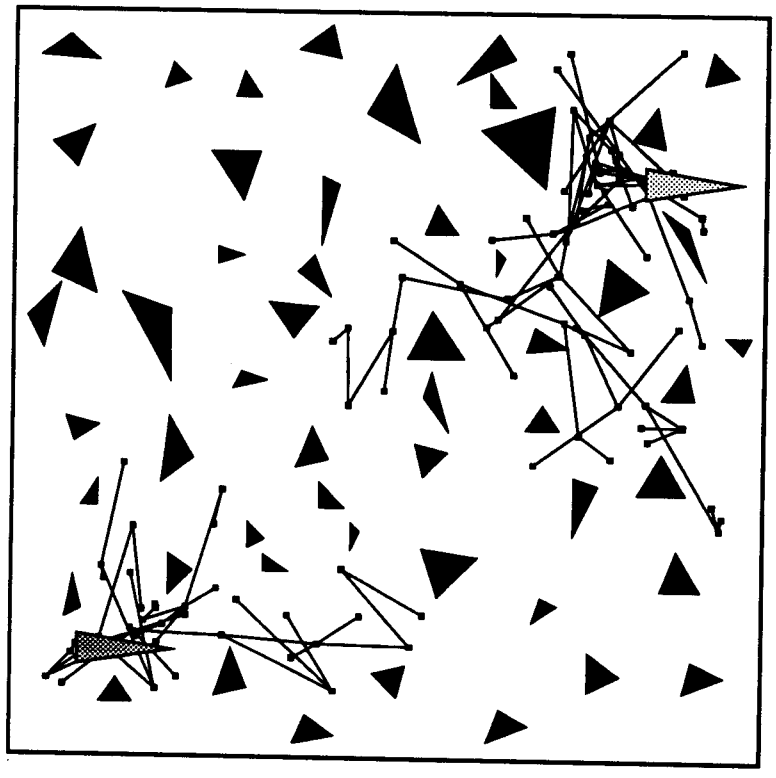

Figure 5.4: Edge requiring node adding.

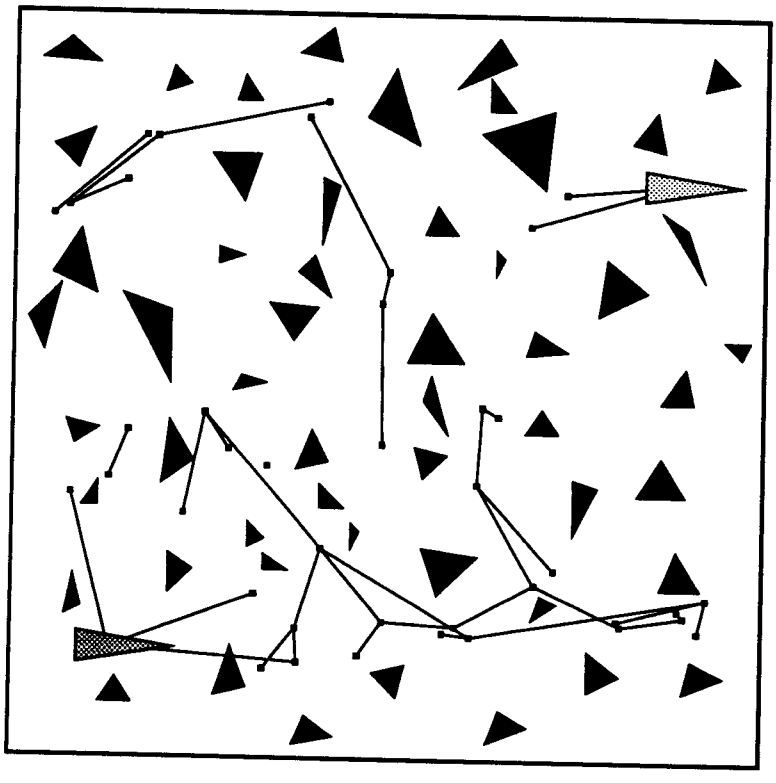

Figure 5.5: Edge sensitive node adding.

5.5 Combining the node adding strategies

Something should be said about how the different discussed node adding strategies can be combined.

When the normal adding strategy is used, no free configuration is more likely to be added then any other, while the forbidden configurations strategy and the adaptive adding strategy bring about that certain (interesting) areas of configuration space are preferred above others, i.e., relatively more nodes are generated in those areas then in others. So we can regard both the forbidden configurations strategy as well as the adaptive adding strategy as possible components of the node generation method.

These two strategies can easily be combined. This means that the node generation can be done with either none, one or both of the two strategies. If both strategies are used, then it is important that the moving of a configuration c according to the forbidden configurations strategy is performed *before* the adding chance for c is computed according to the adaptive adding strategy. The reason for this of course is that the adding chance for c, as computed in section 5.3, is dependent of the (exact) position of c with respect to the current graph.

The generation of a new node can (thus) be described in the following way: First a random node c is generated. If c is a forbidden configuration then if we are using the forbidden configurations strategy we try to move c into a free region, adjacent to the obstacle which causes the intersection, and if we are not using the forbidden configurations strategy then c is thrown away. We repeat the above until a free configuration c is successfully obtained. Then if we are using the adaptive adding strategy, the adding chance for the node c is computed, and with this chance the node c is kept. Now if a node c has been obtained by the above, then we are ready, and otherwise we start all over again with generating another random node.

When a node c has been generated, it is added to the graph, and the edge adding (due to c) is performed. Two edge adding methods have been described (and motivated) in chapter 4 (one for undirected underlying graphs, and one for directed ones). After the edge adding due to c is done, if neither the edge requiring nor the edge depending node adding strategy are used, then we are ready with node c and a new iteration of the main loop of the global method starts (if the problem is not yet solved).

If either of the two strategies is used, then there is a chance that c will be removed. The two strategies are essentially filters which remove some nodes. Whether a node c is removed depends on the outcome of the edge adding due to c.

Furthermore it is clear that the edge requiring and the edge sensitive node adding strategies exclude each other, i.e. cannot be applied both. So the node filtering can be done in 3 different ways. This means that the total number of ways in which it is possible to combine the in this chapter described node adding strategies is equal to 12. Combining the edge requiring node adding strategy with the adaptive adding strategy for undirected graphs though makes no sense. As explained, the edge requiring node adding strategy prevents the appearance of other components than the start and the goal component. Because we already explicitly test for the presence of these two components in the neighborhood of the new node c, it makes no sense to count the number of components near c. Now instead of defining a different version of the adaptive node adding strategy for combining with the edge requiring node adding strategy, we will simple forbid this combination. The performance of the different (allowed) combinations depends on many

factors. See chapter 8 for experimental results.

Chapter 6

Computing and smoothing the final path

In this chapter we describe how the final path in configuration space is computed and how it can be smoothed, after the start node has been connected to the goal node in the underlying graph. It is not relevant here whether the underlying graph is directed or not.

6.1 Computing the final path

As described in chapter 4, the last phase of the global method consists of (re)computing a feasible path from the start configuration s to the goal configuration g. We assume that the global method has successfully computed a path $P_G \in \{0,\ldots,n\} \rightarrow$V in the underlying graph $G = (V, E)$, such that

$$P_G(0) = s \wedge P_G(n) = g \wedge \forall i \in \{0,\ldots,n-1\} : (P_G(i), P_G(i+1)) \in E$$

So we know that for every pair of successive nodes $(P_G(i), P_G(i+1))$ in P_G the local method succeeds in computing a feasible path from $P_G(i)$ to $P_G(i+1)$, i.e., we know that for every i in $\{0,\ldots,n-1\}$ *compute_path*$(P_G(i), P_G(i+1))$ is true.

Now it is easy to compute a path $P_C \in [0,1] \rightarrow C$ in configuration space, which is feasible and connects s to g. We have the function *construct_path* of type $C \times C \rightarrow$ *paths* which can construct feasible paths from $P_G(i)$ to $P_G(i+1)$ ($\forall i \in \{0,\ldots,n-1\}$), and we have the (associative) operator $\oplus$ of type *paths* $\times$ *paths* $\rightarrow$ *paths* which concatenates paths P_1 and P_2 if $P_1(1) = P_2(0)$, preserving feasibility. So P_C can be computed as follows:

$$P_C = \textit{construct\_path}(P_G(0), P_G(1)) \oplus \textit{construct\_path}(P_G(1), P_G(2)) \oplus \ldots\ldots\ldots \\ \oplus \textit{construct\_path}(P_G(n-1), P_G(n))$$

6.2 Smoothing the final path

Unfortunately, the paths computed by the global method can look quite ugly. See for example figure 6.1. It shows a path which is feasible for car-like robots, computed by the random motion planner for car-like robots (see also chapter 7). The shape highly depends on the 'random' configurations $P_G(0)$,$P_G(1)$,...,$P_G(n)$ on the path. To improve this, we

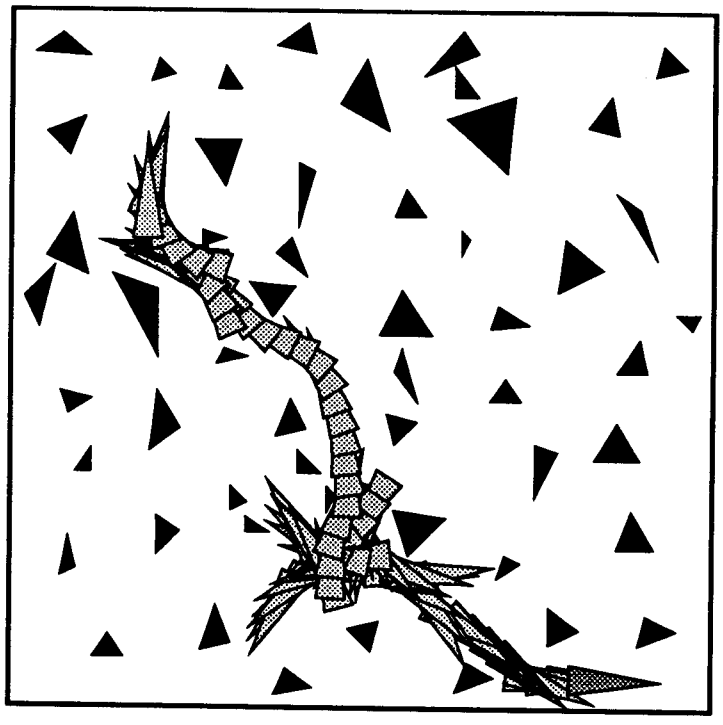

Figure 6.1: A very ugly path, feasible for a car-like robot.

apply some *smoothing methods*, which make the final path 'nicer'. It should be said though that the improvements achieved by the smoothing methods are rather intuitive ones. The paths produced are not 'optimal' in any sense, and, hence, from theoretical point of view the smoothing methods discussed in this chapter are of little value. Intuitively though, they improve the paths a great deal. Lots of useless curves and reversals are removed, and the length of the paths decreases significantly.

We perform the smoothing at two 'levels': First we try to smooth the path P_G, then we compute the path P_C based on the smoothed version of P_G, and finally we smooth P_C. The former smoothing will be referred to as *graph smoothing*, and the latter as *path smoothing*. Conceptually both smoothing methods are very simple.

The idea of graph smoothing is that as long as there exist $P_G(i-1)$ and $P_G(i+1)$ such that the local method can compute a feasible path from $P_G(i-1)$ to $P_G(i+1)$, we discard $P_G(i)$. This is made more precise by the following algorithm:

while $\exists i \in \{2, \ldots, n-1\}$: *compute_path*$(P_G(i-1), P_G(i+1))$
do $n = n - 1$
  for $j = i$ **to** n
  do $P_G(j) = P_G(j+1)$

It is clear that after graph smoothing has been applied, the local method will still be able to compute a feasible path for every pair of successive 'nodes' in P_G. Figure 6.2 shows the result of graph smoothing applied to the 'ugly' path, shown in figure 6.1. This smoothing method is very fast. E.g., the result shown in figure 6.2 was obtained in about 0.2 seconds.

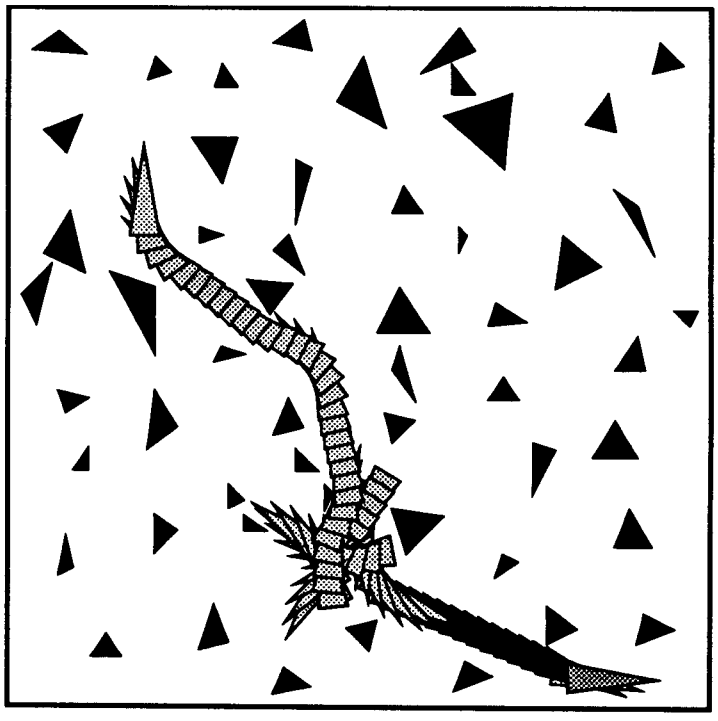

Figure 6.2: The path after applying graph smoothing.

Path smoothing is a bit more time consuming than graph smoothing, but it achieves very good results. The idea of path smoothing is that we repeatedly pick a pair of random configurations (c_1, c_2) on the path P_C (with c_1 positioned 'before' c_2), try 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 we replace Q_{old} by Q_{new} (in P_C). This is formalized by the following algorithm:

repeat a large number of times :
  t_1, t_2 = random elements of $[0,1]$ with $t_1 < t_2$
  $Q_{old} = \lambda t.P_C((t_2 - t_1) \cdot t + t_1)$
  if *compute_path*$(P_C(t_1), P_C(t_2))$
  then Q_{new} = *construct_path*$(P_C(t_1), P_C(t_2))$
    if the length of Q_{new} < the length of Q_{old}
    then $P_C = \lambda t.P_C(t_1 \cdot t) \oplus Q_{new} \oplus \lambda t.P_C(t_2 + (1 - t_2) \cdot t)$

To keep the algorithm compact, we use the functional λ-notation. By $\lambda t.f(t)$ we denote the function which maps t to $f(t)$. A question of course is how many times the loop should be performed. In our current implementation we always perform the loop 500 times, and the results are very nice, while the running times remain reasonable. For example, the path smoothing resulting in the path shown in figure 6.3 took about 1 second. Some heuristic stop criterion could of course be used for the loop. E.g., we could perform the loop until the path P_C has remained unchanged during the past k iterations, where k is some constant.

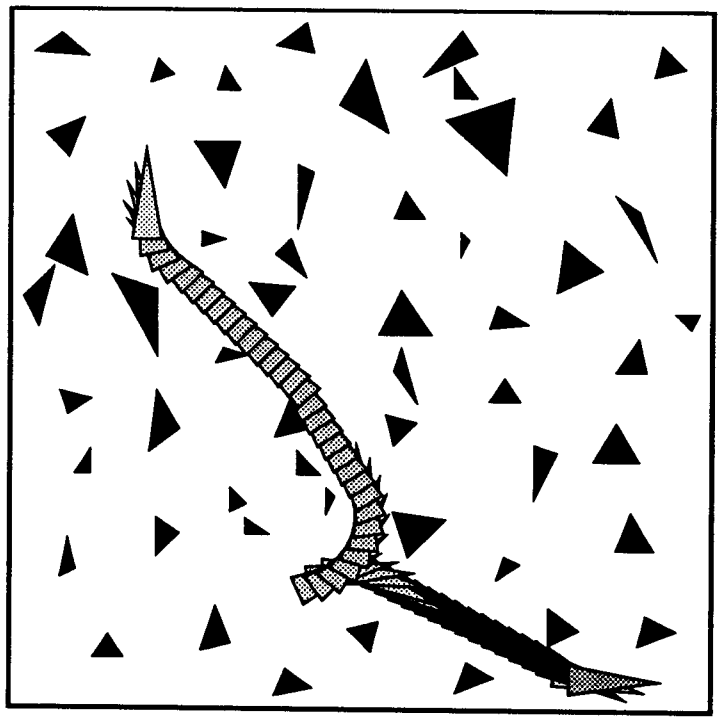

Figure 6.3: The path after applying path smoothing.

Chapter 7

Random motion planning for car-like robots

We will now apply the described random motion planning techniques to the problem of motion planning for robots with nonholonomic car-like constraints. Basically all we need are local methods which compute feasible paths for car-like robots, and (induced) metrics. If these methods are symmetrical, then the global method based on an undirected graph (as described in section 4.2) can be used, and otherwise the global method based on a directed graph (as described in section 4.3) is applicable.

In section 7.1 car-like robots together with paths which respect their nonholonomic constraints will formally be defined. See also chapter 9 in [Lat91] for a thorough treatment of this topic. In section 7.2 some useful paths constructs will be defined and discussed, which will later be used as 'primitive building blocks' by the local methods, for the construction of feasible paths.

Two types of car-like robots will then be dealt with. In section 7.3 a number of local methods will be defined for car-like robots which can make reversals, i.e., can move both forwards as well as backwards. Such robots will from now on be referred to as *general car-like robots*. Because general car-like robots possess the reversibility-property (see section 4.1), the local methods will be symmetrical, and consequently the global method can be based on an undirected graph.

In section 7.4 some local methods for car-like robots which cannot make reversals will be defined. Such robots can only move forwards, and they will therefore be referred to as *forward car-like robots*. They do not possess the reversibility property, so the local methods here will not be symmetrical, and, hence, the global method must be based on a directed graph.

7.1 Definition of car-like robots

As mentioned in the introduction, a car-like robot is a planar robot which can move forwards and backwards with a bounded turning radius. The kinematic constraints of a car-like robot are of non-holonomic nature. This means that, given a configuration c, they restrain the *set of possible velocities* achievable by the robot when positioned at c. In this section we will more formally look at car-like robots. See figure 7.1 for an abstract representation of a car-like robot $\mathcal{A}$, positioned at configuration c. A car-like robot $\mathcal{A}$ is a

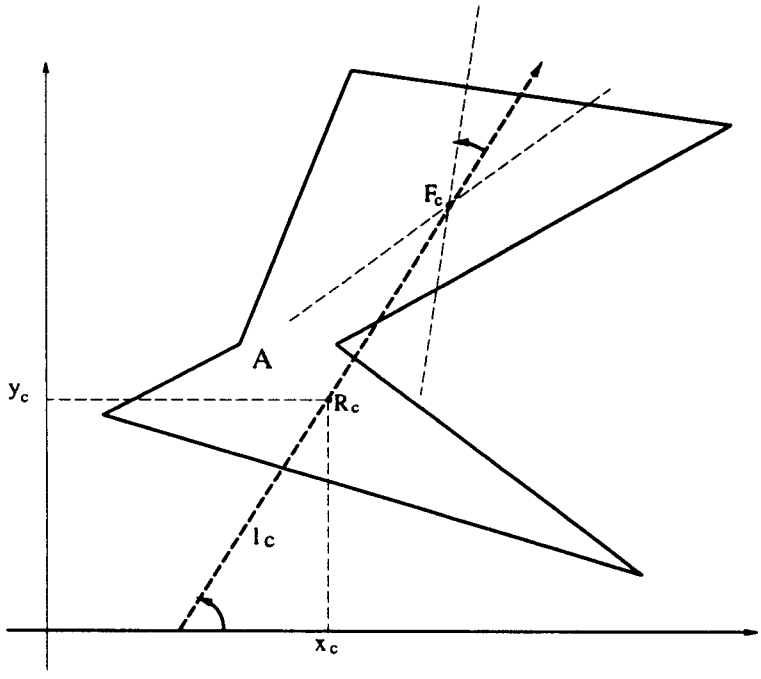

Figure 7.1: An abstract model for the car-like robot.

solid planar object of arbitrary shape, with two points R and F fixed to it. R is referred to as the *rear point*, and F as the *front point*. R and F do not necessarily need to lie inside $\mathcal{A}$, but they must differ from each other. Furthermore $\mathcal{A}$ has a *maximal steering angle* $\phi_{max} \in [0, \frac{1}{2}\pi[$ defined. A car-like robot is uniquely defined by R,F, ϕ_{max}, and its shape. R is used as the reference point of $\mathcal{A}$, and the main axis of $\mathcal{A}$ is defined by the line through R and F. We will use the following convention: If $\mathcal{A}$ is positioned at configuration c, then we will refer by R_c and F_c to the coordinates of R and F in $\mathbf{R}^2$, and analogously we will refer by l_c to the main-axis of $\mathcal{A}$, i.e., to the line in $\mathbf{R}^2$ which contains R_c and F_c.

Now we can define the possible velocities of $\mathcal{A}$ when positioned at configuration c: Exactly those velocities of $\mathcal{A}$ are possible where

1. The direction of R's velocity v points (forwards or backwards) along l_c.
2. The angle ϕ between the direction of F's velocity and l_c, which is referred to as the *steering angle* of $\mathcal{A}$, lies in $[-\phi_{max}, \phi_{max}]$. In other words, the velocity vector of F lies within the double wedge bounded by the two distinct lines containing F and making an angle ϕ_{max} with l_c. See also figure 7.1.

In practice (think of a car), $\mathcal{A}$ is typically approximately rectangular, and moves on four wheels. The two rear wheels have fixed orientation (with respect to $\mathcal{A}$), while the two front wheels can rotate by some limited angle. The rear point R is the midpoint between the two rear wheels, and the front point F is the midpoint between the two front wheels. The orientation of the front wheels determines the steering angle of $\mathcal{A}$. Figure 7.2 shows an example of such a car-like robot, positioned at configuration c.

The restrictions 1 and 2 on the possible velocities of a car-like robot $\mathcal{A}$ translate to restrictions on paths $t \mapsto (x(t), y(t), \theta(t))$, such that these paths respect the nonholonomic

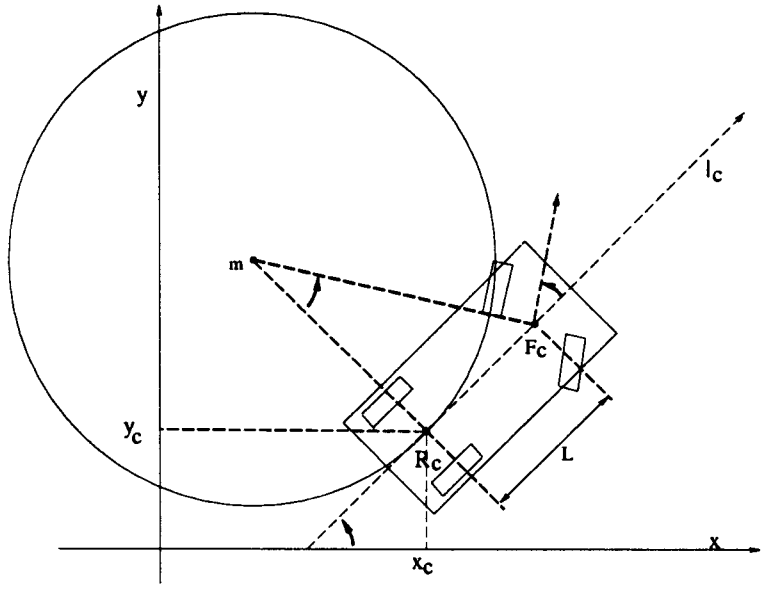

Figure 7.2: A model for the car-like robot.

constraints of car-like robots.

Let us denote the velocity of R (along l_c) at time t by $v(t)$, and the steering angle of $\mathcal{A}$ at time t by $\phi(t)$. The velocity of $\mathcal{A}$ at time t is uniquely defined by $v(t)$ and $\phi(t)$. So a motion of $\mathcal{A}$, say from time 0 to time 1, is uniquely defined by the functions $v \in [0,1] \rightarrow \mathbf{R}$ and $\phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}]$, presuming that its start configuration at time 0 is known.

A choice to be made is whether we require v and ϕ to be continuous functions, or whether we allow (a finite number of) discontinuities, i.e., only require v and ϕ to be piecewise continuous. We choose for the latter, which means that we allow infinite accelerations. This will simplify the construction of feasible paths, and moreover, it can be shown that if a feasible path exists from say a configuration s to a configuration g which is defined by functions v and ϕ that are piecewise continuous, then there also exists a feasible path from s to g which is defined by continuous functions $\tilde{v}$ and $\tilde{\phi}$. We will not prove this claim, but intuitively it is obvious: When driving a car, the 'paths' we can drive are not restricted due to the fact that it is impossible to accelerate or turn the steering wheel with infinite speed.

The motion planning problem in a particular scene asks for a free path from some start configuration s to some goal configuration g. Hence, we want to find piecewise continuous functions v and ϕ such that the motion defined by these functions brings the robot from s to g, without any collisions with the obstacles. We will though not explicitly compute these functions. Instead we will directly compute paths in configuration space, using constructs which guarantee that the car-like constraints of the robot are respected. Or stated otherwise, we compute paths such that there surely exist piecewise continuous functions $v \in [0,1] \rightarrow \mathbf{R}$ and $\phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}]$ which define these paths. We have seen (in chapter 2) that a path P can be described by a continuous function of type $[0,1] \rightarrow \mathbf{R} \times \mathbf{R} \times [0, 2\pi[$ which maps (time) t to $(x(t), y(t), \theta(t))$, where $(x(t), y(t))$ are the coordinates of (reference point) R at time t, and $\theta(t)$ is the orientation of the robot at time t. So $(x(t), y(t), \theta(t))$ is the configuration at which $\mathcal{A}$ is positioned at time t. The

functions x,y and θ must of course also be continuous.

The nonholonomic car-like constraints of $\mathcal{A}$ can be expressed by a set of equalities involving $x'(t), y'(t), \theta'(t), v(t)$ and $\phi(t)$. The fact that at any time t (where v(t) is continuous) the velocity $v(t)$ of R points along l_A can be expressed by

$$x'(t) = v(t) \cdot cos(\theta(t))$$
$$y'(t) = v(t) \cdot sin(\theta(t))$$

For $t \in [0,1]$ where the velocity function v and the steering function ϕ are continuous, $\theta'(t)$ can be expressed in $v(t)$ and $\phi(t)$. I.e., θ' is a function of v and ϕ, defined for all $t \in [0,1]$ where v and ϕ are continuous. Let $m(t)$ be the (instantaneous) center of rotation of $\mathcal{A}$ at time t, and $r(t)$ the distance between R and $m(t)$ (at time t). So $r(t)$ is the radius of the circle which is followed by R as long as the steering angle of $\mathcal{A}$ does not change, and its speed is not 0. It will therefore be referred to as *the turning radius* of the robot at time t. Now $\left|\frac{2\pi \cdot r(t)}{v(t)}\right|$ is the time it would take the robot to perform a full circular motion around $m(t)$, if $v(t)$ and $\phi(t)$ would remain constant. So it is clear that

$$|\theta'(t)| = \frac{2\pi}{\left|\frac{2\pi \cdot r(t)}{v(t)}\right|} = \left|\frac{v(t)}{r(t)}\right|$$

We can now express $\theta'(t)$ in $v(t)$ and $\phi(t)$ using $r(t) = \left|\frac{L}{tan(\phi(t))}\right|$, where L is the distance between the rear point R and the front point F (see also figure 7.2). This leads to

$$|\theta'(t)| = \left|\frac{1}{L} \cdot v(t) \cdot tan(\phi(t))\right|$$

Getting the sign right is easy: (see also figure 7.2) if $\phi(t)$ and $v(t)$ have the same sign, then $\theta'(t) > 0$, and otherwise $\theta'(t) < 0$. So we have

$$\theta'(t) = \frac{1}{L} \cdot v(t) \cdot tan(\phi(t))$$

A path $t \mapsto (x(t), y(t), \theta(t))$ thus respects the nonholonomic constraints of a car-like robot $\mathcal{A}$ if and only if there exist piecewise continuous functions $v \in C \times C \rightarrow \mathbf{R}$ and $\phi \in C \times C \rightarrow [-\phi_{max}, \phi_{max}]$, such that for every $t \in \ ]0,1[$ where v and ϕ are continuous

$$x'(t) = v(t) \cdot cos(\theta(t))$$
$$y'(t) = v(t) \cdot sin(\theta(t))$$
$$\theta'(t) = \tfrac{1}{L} \cdot v(t) \cdot tan(\phi(t))$$

This is formalized by the following definition:

Definition 5 *Let $\mathcal{A}$ be a car-like robot, with maximal steering angle ϕ_{max} and distance L between rear and front point.*

A path $t \mapsto (x(t), y(t), \theta(t))$ ($t \in [0,1]$) respects the nonholonomic constraints of $\mathcal{A}$

iff

$$\begin{array}{l}\exists v \in [0,1] \rightarrow \mathbf{R} : \exists \phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}] : \exists t_1, \ldots, t_n \in [0,1] : \\ \quad t_1 = 0 \ , t_n = 1 \\ \qquad \wedge \\ \quad \forall i \in \{1, \ldots, n-1\} : t_i < t_{i+1} \\ \qquad\qquad \wedge \\ \qquad\qquad v \text{ and } \phi \text{ are continuous in }]t_i, t_{i+1}[\\ \qquad\qquad \wedge \\ \qquad\qquad \forall t \in]t_i, t_{i+1}[\ : \ x'(t) = v(t) \cdot cos(\theta(t)) \\ \qquad\qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad\qquad y'(t) = v(t) \cdot sin(\theta(t)) \\ \qquad\qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad\qquad \theta'(t) = \frac{1}{L} \cdot v(t) \cdot tan(\phi(t))\end{array}$$

Definition 5 is valid for car-like robots which can move both forwards and backwards. As said in the introduction of this chapter, such robots will be referred to as general car-like robots. A analog definition for car-like robots which can only move forwards, i.e., of forward car-like robots, can be obtained by only slightly modifying definition 5: Instead of a function $v \in [0,1] \rightarrow \mathbf{R}$ we require the existence of a function $v \in [0,1] \rightarrow \mathbf{R}^+$. Or to obtain a definition for car-like robots which can move both forwards and backwards, but which cannot steer to the right, we replace $\phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}]$ by $\phi \in [0,1] \rightarrow [0, \phi_{max}]$, etc...

7.2 Basic constructs

In this section some constructs of paths which respect the nonholonomic constraints of car-like robots will be defined. We assume that there are no obstacles, so these paths will 'automatically' be feasible.

Assume that the robot is positioned at a configuration c. We now will describe some possible motions that it can perform.

1. It can perform a *straight motion* over some distance d along $\mathcal{A}$'s main axis l_c. During this motion R moves in a straight line, and the orientation of $\mathcal{A}$ is constant. Such a motion (of constant speed) corresponds to the path

   $$\begin{array}{l}x(t) = x_c + d \cdot cos(\theta_c) \cdot t \\ y(t) = y_c + d \cdot sin(\theta_c) \cdot t \\ \theta(t) = \theta_c\end{array}$$

   for $t \in [0,1]$.

   Now if we define $v \in [0,1] \rightarrow \mathbf{R}$ and $\phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}]$ by $v(t) = d$ and $\phi(t) = 0$ ($\forall t \in [0,1]$), then $x'(t) = d \cdot cos(\theta_c) = v(t) \cdot cos(\theta(t))$, $y'(t) = d \cdot sin(\theta_c) =$

$v(t) \cdot sin(\theta(t))$, and $\theta'(t) = 0 = \frac{1}{L} \cdot v(t) \cdot tan(\phi(t))$. So this shows that the straight path complies to definition 5, which proves that the described motion really is legal, which though is rather obvious.

2. The robot can also perform a *circular motion*, where the center of rotation m lies somewhere on the line through R, perpendicular to $\mathcal{A}$'s main axis l_A, and the distance r between m and R is at least $\left|\frac{L}{tan(\phi_{max})}\right|$, the *minimal turning radius* r_{min} of $\mathcal{A}$. We will refer to r as the *turning radius* of the circular motion. A circular motion (of constant speed) where $\mathcal{A}$ rotates over an angle α around m corresponds to the path

$$\begin{aligned} x(t) &= x_m + r \cdot cos(\alpha_{mR} + t \cdot \alpha) \\ y(t) &= y_m + r \cdot sin(\alpha_{mR} + t \cdot \alpha) \\ \theta(t) &= \theta_c + t \cdot \alpha \end{aligned}$$

for $t \in [0, 1]$, and where α_{mR} is the angle of the vector from m to R.

To prove that this is a legal motion we will again give functions v and ϕ such that the circular path complies to definition 5.

Take $v(t) = \alpha \cdot r$ and $\phi(t) = atan(\frac{L}{r})$. We will show the compliance for the case that m lies to the 'left' of the robot, as in figure 7.2. If m lies to the 'right' of the robot, then it can be done in a very analog way.

$$\begin{aligned} x'(t) &= -r \cdot sin(\alpha_{mR} + t \cdot \alpha) \cdot \alpha = -\alpha \cdot r \cdot sin(\theta_c - \tfrac{1}{2}\pi + t \cdot \alpha) \\ &= \alpha \cdot r \cdot cos(\theta_c + t \cdot \alpha) = v(t) \cdot cos(\theta(t)) \\ y'(t) &= r \cdot cos(\alpha_{mR} + t \cdot \alpha) \cdot \alpha = \alpha \cdot r \cdot cos(\theta_c - \tfrac{1}{2}\pi + t \cdot \alpha) \\ &= \alpha \cdot r \cdot sin(\theta_c + t \cdot \alpha) = v(t) \cdot sin(\theta(t)) \\ \theta'(t) &= \alpha = \tfrac{1}{L} \cdot \alpha \cdot r \cdot \tfrac{L}{r} = \tfrac{1}{L} \cdot \alpha \cdot r \cdot tan(atan(\tfrac{L}{r})) = \tfrac{1}{L} \cdot v(t) \cdot tan(\phi(t)) \end{aligned}$$

The paths for car-like robots that we will compute with the local methods will solely be composed of (sub)paths corresponding to straight motions (as described in 1), or to circular motions (as described in 2) with minimal turning radius r_{min} $(= \frac{L}{tan(\phi_{max})})$. The motivation for disregarding circular paths with larger turning radii than r_{min} is the following: It can be shown that if, in an arbitrary scene containing a set of obstacles, a feasible path for a car-like robot $\mathcal{A}$ from a configuration s to a configuration g exists, then there also exists a feasible path for $\mathcal{A}$ from s to g which is a finite sequence of subpaths corresponding to circular motions with minimal turning radii (see [Lat91]). So we could even do without the straight paths, but we will include them in order to reduce the length of the paths computed by the local methods, and, hence, reduce the total length of the paths computed by our motion planner.

From now on we will refer to paths which correspond to straight motions (as described in 1) as *line paths*, and to paths corresponding to circular motions (as described in 2) with turning radius r_{min} as *arc paths*. The line paths and arc paths will be used as 'primitive building blocks' by the local methods. For this we define the functions L and A, both of type $C \times C \rightarrow paths$, as follows:

$L(a, b) =$ The line path from a to b, if one exists.
Undefined, otherwise.

$A(a,b)$ = The shortest arc path from a to b, if one exists.
Undefined, otherwise.

The motion corresponding to $L(a,b)$ or $A(a,b)$ can be either a forwards motion or a backwards motion [1]. So L and A define constructs of paths which respect the nonholonomic constraints of general car-like robots (this was formally proved earlier in this section), but not those of forward car-like robots. To specify line paths and arc paths which correspond to forward (respectively backward) motions of the robot, we define the functions L_f and A_f (respectively L_b and A_b), both of type $C \times C \rightarrow paths$, as follows:

$L_f(a,b)$ = The line path from a to b corresponding to a forwards motion, if one exists.
Undefined, otherwise.

$L_b(a,b)$ = The line path from a to b corresponding to a backwards motion, if one exists.
Undefined, otherwise.

$A_f(a,b)$ = The shortest arc path from a to b corresponding to a forwards motion, if one exists.
Undefined, otherwise.

$A_b(a,b)$ = The shortest arc path from a to b corresponding to a backwards motion, if one exists.
Undefined, otherwise.

L_f and A_f define constructs of paths which (also) respect the nonholonomic constraints of forward car-like robots. Formally this can be proven analog to how it was proven that $L(*,*)$ and $A(*,*)$ respect the nonholonomic constraints of general car-like robots. There exist functions $v \in [0,1] \rightarrow \mathbf{R}^+$ and $\phi \in [0,1] \rightarrow [-\phi_{max}, \phi_{max}]$ which define these paths.

It is clear that if arc paths are projected on $\mathbf{R}^2$, then they project on circles of radius r_{min}. We define the *left touching circle* $C_L(c)$ and *the right touching circle* $C_R(c)$ of a configuration c as the two circles (in $\mathbf{R}^2$) of radius r_{min} which pass through (x_c, y_c) with angle θ_c. $C_L(c)$ lies to the left of l_c, and $C_R(c)$ to the right. See figure 7.3. Now any arc path P starting at configuration c, i.e. $P(0) = c$, projects to either $C_L(c)$ or to $C_R(c)$.

It is clear that, if two paths P_1 and P_2 both are feasible for our robot $\mathcal{A}$, and $P_1(1) = P_2(0)$, then $P_1 \oplus P_2$ is also a feasible path for $\mathcal{A}$. The functions v and ϕ defining $P_1 \oplus P_2$ may be discontinuous in $t = \frac{1}{2}$, but we have allowed this. We will use this property now for constructing some more complex paths, built up from arc and line paths.

We have implemented efficient routines[2] which test whether a given arc or line path intersects any obstacles. So to test whether a path $P_1 \oplus P_2 \oplus \ldots \oplus P_k$, where the P_i's are arc or line paths, is a free path, we only need to perform k intersection tests.

[1] A line or arc path $t \mapsto (x(t), y(t), \theta(t))$ corresponds to a forward motion if the vector $(x'(t), y'(t))$ has angle $\theta(t)$ for every $t \in [0,1]$, and it corresponds to a backwards motion if the vector $(x'(t), y'(t))$ has angle $(\theta(t) + \pi) \bmod 2\pi$ for every $t \in [0,1]$.

[2] Geert-Jan Giezeman (Department of Computer Science, Utrecht University) has implemented the smear library, which contains two routines that test whether the area covered by a polygon during a rotational or translational motion intersects any (polygonal) obstacles.

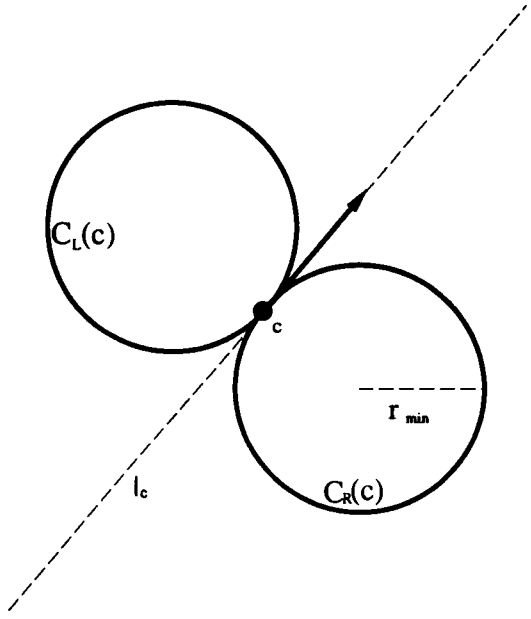

Figure 7.3: The touching circles of configuration c.

7.2.1 ALA paths

Definition 6 *A path P is an* ALA path *from configuration a to configuration b* **iff** $P = A(a, c_1) \oplus L(c_1, c_2) \oplus A(c_2, b)$ *for certain* $c_1, c_2 \in C$.

ALA paths are 'concatenations' of paths which are feasible for general car-like robots, so it is clear that ALA paths themselves are also feasible for these robots. The following properties, valid for an arbitrary pair (a, b) of configurations, are easily verified :

1. Exactly two ALA paths $A(a, *) \oplus L(*, *) \oplus A(*, b)$ exist, such that $A(a, *)$ projects on $C_R(a)$ and $A(*, b)$ on $C_R(b)$. One where the line path $L(*, *)$ corresponds to a forward motion, and one where it corresponds to a backwards motion. See figure 7.4. The two described ALA paths are shown for a triangular car-like robot. The path with $L(*, *)$ corresponding to a forwards motion is indicated by white steps, and the other one by black steps.

2. Exactly two ALA paths $A(a, *) \oplus L(*, *) \oplus A(*, b)$ exist, such that $A(a, *)$ projects on $C_L(a)$ and $A(*, b)$ on $C_L(b)$. One where the line path $L(*, *)$ corresponds to a forward motion, and one where it corresponds to a backwards motion. See figure 7.5 for a visualization of the two described paths.

3. If $C_R(a)$ and $C_L(b)$ do not intersect in more than one point, then exactly two ALA paths $A(a, *) \oplus L(*, *) \oplus A(*, b)$ exist, such that $A(a, *)$ projects on $C_R(a)$ and $A(*, b)$ on $C_L(b)$. One where the line path $L(*, *)$ corresponds to a forward motion, and one where it corresponds to a backwards motion. If the circles do intersect in more than one point, then no such ALA path exist. See figure 7.6 for a visualization of the two described paths.

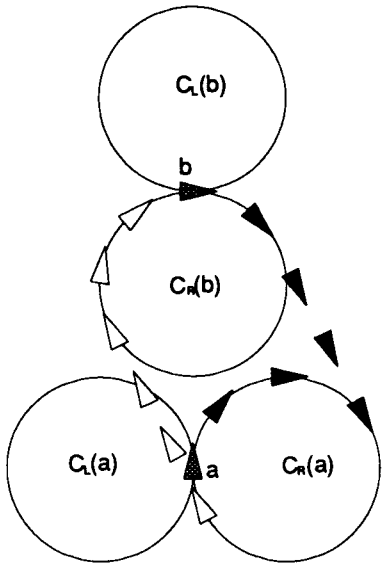

Figure 7.4: Two ALA paths from a to b, which partially project on $C_R(a)$ and $C_R(b)$.

4. If $C_L(a)$ and $C_R(b)$ do not intersect in more than one point, then exactly two ALA paths $A(a,*) \oplus L(*,*) \oplus A(*,b)$ exist, such that $A(a,*)$ projects on $C_L(a)$ and $A(*,b)$ on $C_R(b)$. One where the line path $L(*,*)$ corresponds to a forward motion, and one where it corresponds to a backwards motion. If the circles do intersect in more than one point, then no such ALA path exist. See figure 7.7 for a visualization of the two described paths.

So the total number of ALA paths from one configuration to another varies between 4 and 8.

7.2.2 LAL paths

Definition 7 *A path P is a* LAL path *from configuration a to configuration b* **iff** $P = L(a,c_1) \oplus A(c_1,c_2) \oplus L(c_2,b)$ *for certain* $c_1, c_2 \in C$.

As for ALA paths, it is obvious that LAL paths are also feasible for general car-like robots. The following claims hold for arbitrary $a, b \in C$:

1. $\theta_a \bmod \pi \neq \theta_b \bmod \pi$
   $\Rightarrow$ Exactly two LAL paths from a to b exist. One where the arc path segment corresponds to a forward motion, and one where it corresponds to a backward motion.
   See figure 7.8, for a visualization of these two paths.

2. $\theta_a = \theta_b$
   $\Rightarrow$ If $l_a = l_b$ then exactly one LAL path from a to b exists, and otherwise no LAL path from a to b exists.

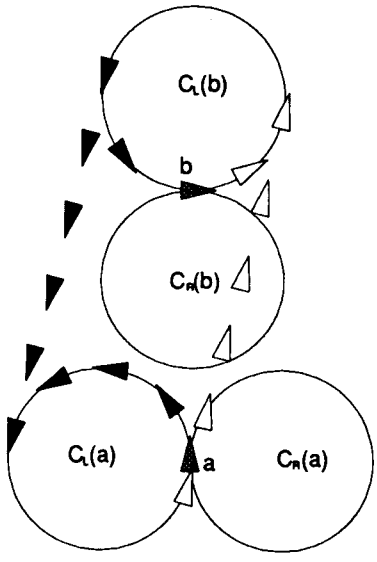

Figure 7.5: Two ALA paths from a to b, which partially project on $C_L(a)$ and $C_L(b)$.

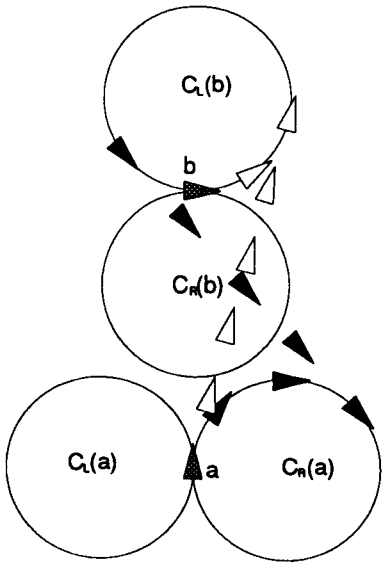

Figure 7.6: Two ALA paths from a to b, which partially project on $C_R(a)$ and $C_L(b)$.

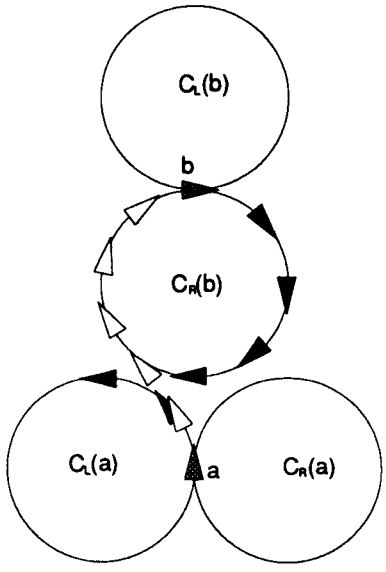

Figure 7.7: Two ALA paths from a to b, which partially project on $C_L(a)$ and $C_R(b)$.

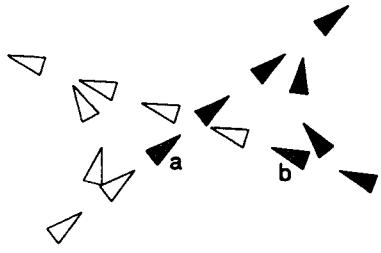

Figure 7.8: Two LAL paths from a to b.

3. $\theta_a = (\theta_b + \pi) \bmod 2\pi$
   $\Rightarrow$ If $distance(l_a, l_b) = 2r$ then infinitely many LAL paths from a to b exists, and otherwise no LAL path from a to b exists.

So the total number of LAL paths from one configuration to another is 0,1,2 or infinite.

7.2.3 ALA forward paths

Definition 8 *A path P is an* ALA forward path *from configuration a to configuration b* **iff** $P = A_f(a, c_1) \oplus L_f(c_1, c_2) \oplus A_f(c_2, b)$ *for certain* $c_1, c_2 \in C$.

So an ALA forward path is the concatenation of three (sub) paths which all are feasible for forward car-like robots, and, hence, ALA forward paths themselves are also feasible for these robots. The following claims hold for arbitrary configurations a and b:

1. Exactly one ALA forward path $A_f(a,*) \oplus L_f(*,*) \oplus A_f(*,b)$ exists, such that $A_f(a,*)$ projects on $C_R(a)$ and $A_f(*,b)$ on $C_R(b)$.

2. Exactly one ALA forward path $A_f(a,*) \oplus L_f(*,*) \oplus A_f(*,b)$ exists, such that $A_f(a,*)$ projects on $C_L(a)$ and $A_f(*,b)$ on $C_L(b)$.

3. If $C_R(a)$ and $C_L(b)$ do not intersect in more than one point, then exactly one ALA forward path $A_f(a,*) \oplus L_f(*,*) \oplus A_f(*,b)$ exists, such that $A_f(a,*)$ projects on $C_R(a)$ and $A_f(*,b)$ on $C_L(b)$. If $C_R(a)$ and $C_L(b)$ do intersect in two or more points, then no such ALA forward path exists.

4. If $C_L(a)$ and $C_R(b)$ do not intersect in more than one point, then exactly one ALA forward path $A_f(a,*) \oplus L_f(*,*) \oplus A_f(*,b)$ exists, such that $A_f(a,*)$ projects on $C_L(a)$ and $A_f(*,b)$ on $C_R(b)$. If $C_L(a)$ and $C_R(b)$ do intersect in two or more points, then no such ALA forward path exists.

So the total number of ALA forward paths from one configuration to another varies from 2 to 4.

7.2.4 LAL forward paths

Definition 9 *A path P is an* LAL forward path *from configuration a to configuration b* **iff** $P = L_f(a,c_1) \oplus A_f(c_1,c_2) \oplus L_f(c_2,b)$ *for certain* $c_1, c_2 \in C$.

Just like ALA forward paths, LAL forward paths are feasible for forward car-like robots. The following claims hold for arbitrary $a, b \in C$:

1. $\theta_a \bmod \pi \neq \theta_b \bmod \pi$
   $\Rightarrow$ The number of LAL forward paths from a to b is 0,1 or 2.

2. $\theta_a = \theta_b$
   $\Rightarrow$ If $l_a = l_b \wedge \theta_a =$ (the angle from (x_a, y_a) to (x_b, y_b) in $\mathbf{R}^2$)
   then 1 LAL forward path from a to b exists,
   and otherwise no LAL forward path from a to b exists.

3. $\theta_a = (\theta_b + \pi) \bmod 2\pi$
   $\Rightarrow$ If $distance(l_a, l_b) = 2r$
   then an infinite number of LAL forward paths from a to b exist,
   and otherwise no LAL forward paths from a to b exist.

The total number LAL forward paths, from one configuration to another, is thus 0,1,2 or ∞.

7.3 Local methods for general car-like robots

Now a number of local methods for general car-like robots will be defined and discussed, making use of the constructs defined in the previous section. As indicated before, local methods should be (very simple) motion planners, which try to compute feasible paths from given start configurations to given goal configurations, amidst some obstacles. So, unlike the constructs given in the previous section, the local methods must try to compute paths which not only respect the constraints of general car-like robots, but also lie entirely in free configuration space. As mentioned in chapter 3, the local methods do not always have to succeed, but it is essential that they always terminate.

As described in section 4.2, a local method is to be defined by functions *compute_path* $\in C \times C \rightarrow$ *boolean*, *construct_path* $\in C \times C \rightarrow$ *paths* and $D \in C \times C \rightarrow \mathbf{R}^+$. Because

the underlying graph is to be undirected, compute_path should be symmetrical. For every defined local method M, we will label the functions *compute_path*, *construct_path* and D with a subscript which refers to M. This will enable us to define local methods in terms of other (previously defined) ones.

In this chapter we will not discuss the performance of the different local methods. For testing results obtained from experiments using different local methods, see chapter 8.

7.3.1 The ALA local method

The ALA method is a very primitive one. It simply tries the shortest ALA path from the start configuration to the goal configuration. If this path lies entirely in free configuration space, then the method succeeds, and otherwise it fails. So this leads to the following definitions :

$compute\_path_{ALA}(a,b)$ = Whether the shortest ALA path from a to b lies entirely in free configuration space.

$construct\_path_{ALA}(a,b)$ = **if** $compute\_path_{ALA}(a,b)$ **then** the shortest ALA path from a to b **else** undefined

$D_{ALA}(a,b)$ = The length of the shortest ALA path from a to b.

We will refer to D_{ALA} as the *ALA metric*. It is easy to see that $compute\_path_{ALA}(a,b)$ is symmetrical: Assume that P_{ab} is the shortest ALA path from a to b, and that P_{ba} is the shortest ALA path from b to a. Then P_{ba} is P_{ab} reversed, so it is clear that P_{ab} is free iff P_{ba} is, which implies the symmetry. In figures 7.9 and 7.10 examples are shown of paths computed by the ALA local method, in a scene without obstacles.

7.3.2 The LAL local method

The LAL method tries the shortest LAL path from the start configuration to the goal configuration. If this path lies entirely in free configuration space, then the method succeeds, and otherwise it fails. So the required functions are defined as follows :

$compute\_path_{LAL}(a,b)$ = Whether the shortest LAL path from a to b lies entirely in free configuration space, if such a path exists, and false otherwise.

$construct\_path_{LAL}(a,b)$ = **if** $compute\_path_{LAL}(a,b)$ **then** the shortest LAL path from a to b **else** undefined

$D_{LAL}(a,b)$ = The length of the shortest LAL path from a to b, if such a path exists, and ∞ otherwise.

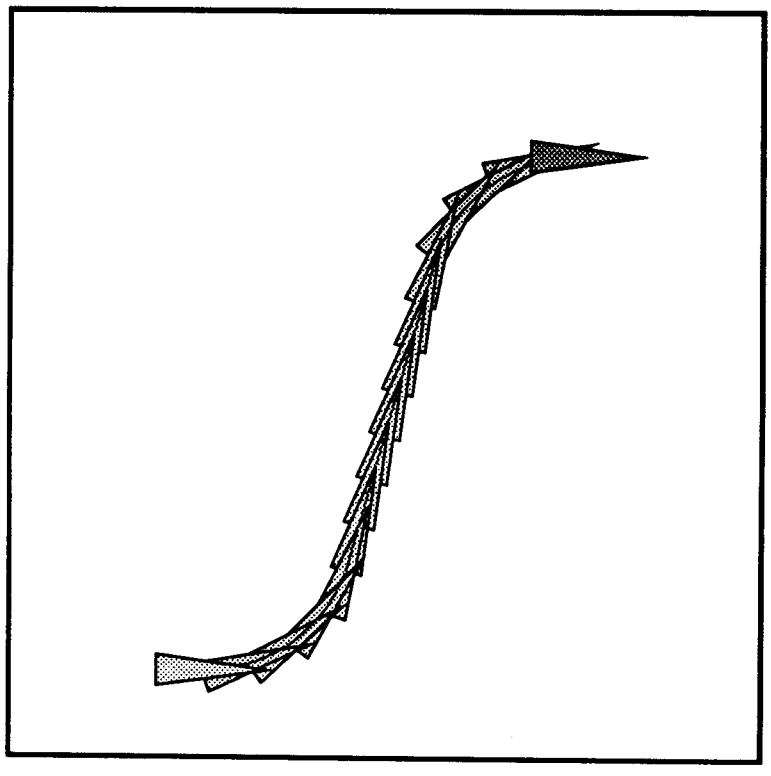

Figure 7.9: A path computed by the ALA local method.

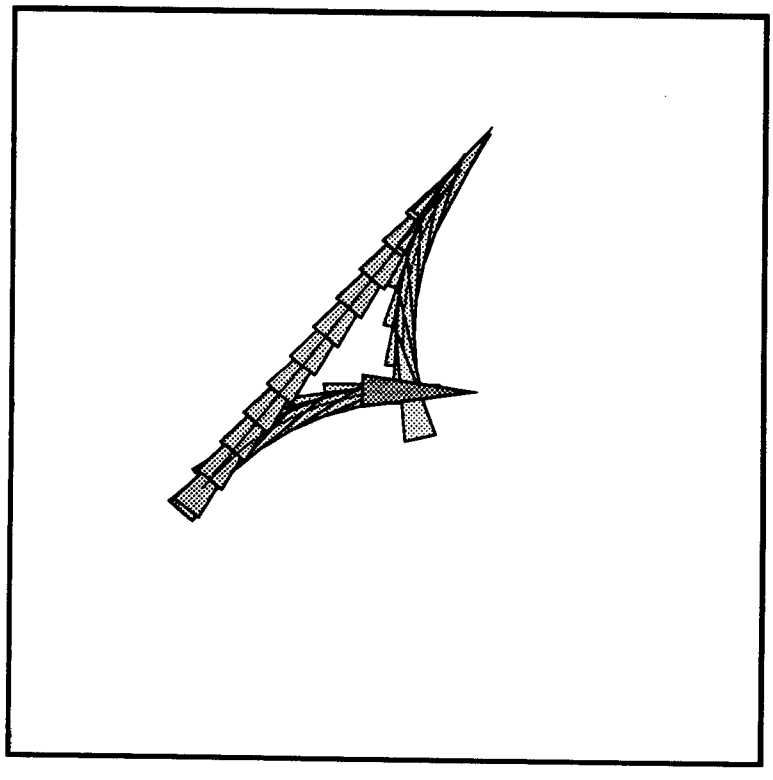

Figure 7.10: Another path computed by the ALA local method.

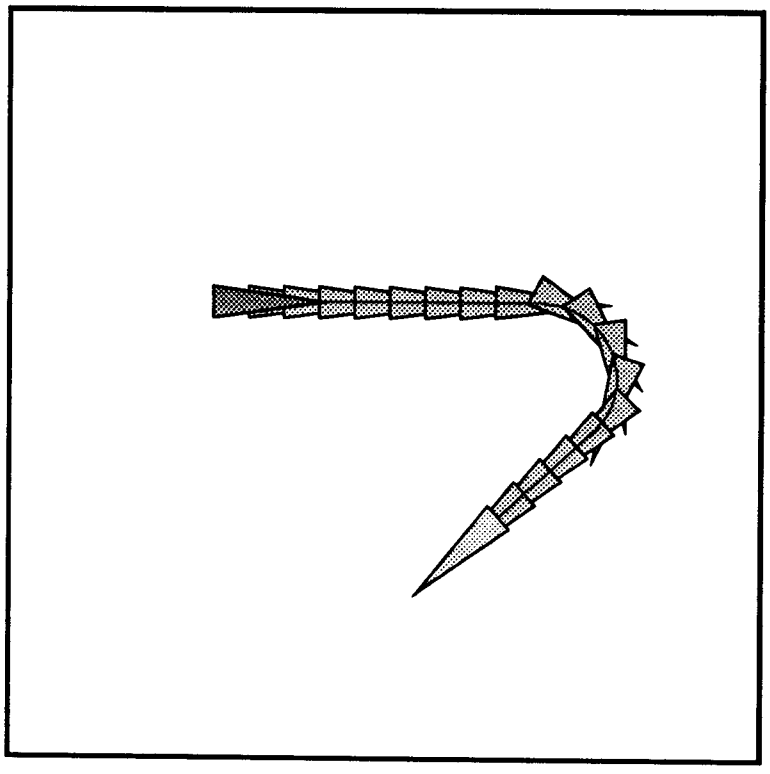

Figure 7.11: A path computed by the LAL local method.

D_{LAL} will be referred to as the *LAL metric*. As for the ALA method, it can easily be verified that $compute\_path_{LAL}(a, b)$ is a symmetric function. In figures 7.11 and 7.12 examples are shown of paths computed by the LAL local method, in a scene without obstacles.

7.3.3 The ALA-LAL local method

The ALA-LAL local method is a little bit more 'complex' than the previous local methods. It first tries the shortest ALA path from the start configuration to the goal configuration. If this is a free path, then the method succeeds. Now if the path is not free, then the shortest LAL path is tried, and the method succeeds if this path is free (and fails otherwise). The metric used is the ALA metric. This leads to the following definitions:

$$compute\_path_{ALA-LAL}(a, b) = compute\_path_{ALA}(a, b) \vee compute\_path_{LAL}(a, b)$$

$$\begin{aligned} construct\_path_{ALA-LAL}(a, b) = \ & \textbf{if } compute\_path_{ALA}(a, b) \\ & \textbf{then} \text{ the shortest ALA forward path from } a \text{ to } b \\ & \textbf{else if } compute\_path_{LAL}(a, b) \\ & \qquad \textbf{then} \text{ The shortest LAL forward path from } a \text{ to } b \\ & \qquad \textbf{else} \text{ undefined} \end{aligned}$$

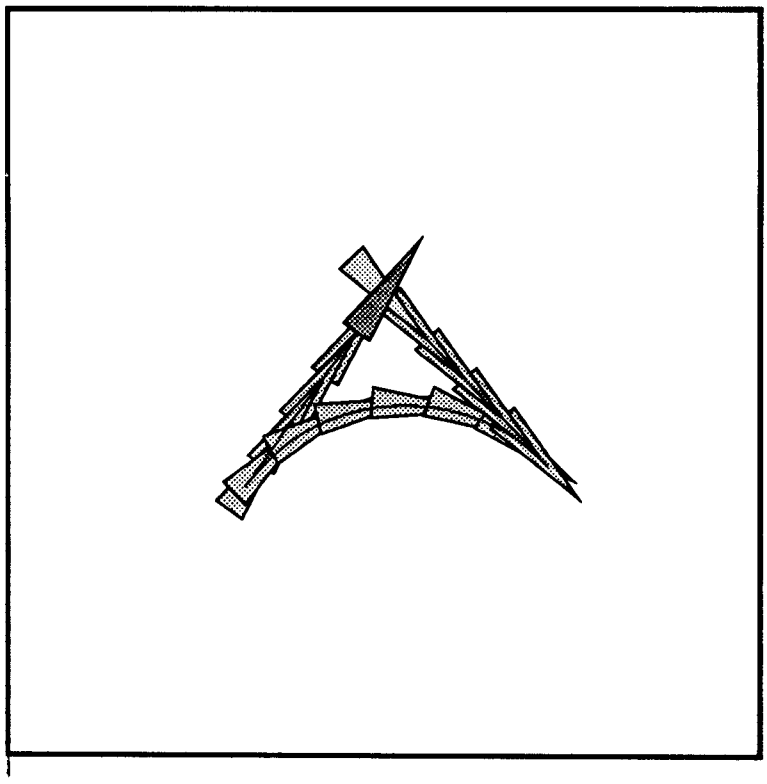

Figure 7.12: Another path computed by the LAL local method.

$D_{ALA-LAL}(a,b) = D_{ALA}(a,b)$

The symmetry of this method is implied by the symmetries of the ALA method and the LAL methods.

7.3.4 Potential field methods

In this section some possibilities will be discussed, concerning the use of potential field techniques for the computation of the local paths. In sections 7.3.1 and 7.3.2 the metrics D_{ALA} and D_{LAL} for (general) car-like robots have been introduced. These metrics can now be used to validate configurations in a way analog to how it is done by 'conventional' potential field methods. The validation typically is such, that configurations near to the goal configuration are preferred above configurations further away, and configurations far away from obstacles are preferred above ones near to obstacles. One can think of a potential field in the configuration space, where the goal configuration g generates a positive potential, which decreases with increasing distance to g, and every obstacle O_i generates a negative potential, which increases with increasing distance to O_i. For the validation of an arbitrary configuration c, the values of all potentials at c are combined in some suitable way. The idea now is that we perform a walk from the start configuration, by repeatedly validating some configurations in the neighborhood and going to the best one, i.e., to that one which validates to the highest value. Our hope is that we will eventually end up in the goal configuration.

So apart from a metric $D_X \in C \times C \rightarrow \mathbf{R}^+$, we also need a definition of *the set of*

neighbors of a configuration c, which define the elementary motions which the robot can perform (not to be confused with the neighbor nodes as defined in chapter 4). These should be defined in such a way that, in absence of obstacles, c can be connected to each of its neighbors by some very short and feasible path. The computation of a feasible path from say c_1 to c_2, will then globally be as follows :

$c = c_1$
$P = [c]$
while $c \not\approx c_2$
    Validate all free neighbors of c.
    c = The best free neighbor n of c, such that a feasible path Q from
        from c to n is known.
    $P = P \oplus Q$

This description still leaves many choices open. First of all a metric should be chosen which takes into account the nonholonomic constraints of a car-like robot. For this cause we can use the ALA metric or the LAL metric, as mentioned above. Then, based on this metric, a potential field must be defined, together with a way in which to validate configurations in the potential field. Globally we have the choice between 'complex' potential fields, whose variations reflect the 'structure' of the potential field well, but are expensive to evaluate, and 'primitive' potential fields, which contain less information about the free configuration space, but are cheap to evaluate.

So it is clear that we are dealing with a trade-off. On one hand we can use a 'smart' potential field method, which will often succeed in finding a feasible path, but will consume a lot of time for every search, or on the other hand we can use a primitive potential field method, which will succeed less often, but is very fast. It seems that the only way to come to a reasonable choice here is by performing experiments with both 'primitive' and 'complex' potential field methods.

Fortunately, for free-flying robots a large number of experiments where performed concerning the use of different kinds of potential field methods as local methods in the random motion planner (See [Mas92]). These experiments clearly indicated that potential field methods using very primitive potential fields lead to a much better performance of the global method than such which use complex potential fields, and, hence, cause the validations of configurations in the potential field to be relatively expensive. By complex potential fields we mean potential fields which, apart from a positive potential which reflects the distance to the goal configuration, also consists of a number of negative potentials which fully reflect the distances to the obstacles. Validations of configurations in such potential fields ask for computations of distances between configurations and obstacles (in configuration space), and for our cause, these operations just seem to be too expensive.

This observation was made for free-flying robots, but we assume that for car-like robots it is valid as well. In fact, computations of distances between configurations and obstacles (in configuration space) are even more expensive for car-like robots than for free-flying ones, because for the latter the Euclidean metric can be used, while for car-like robots some more expensive metric is needed which takes into account the nonholonomic constraints of car-like robots.

For these reasons we have only tried potential field methods which use quite primitive (but cheap to evaluate) potential fields. There is though a price to be paid for doing

so, namely that the robot will often tend to get blocked behind an obstacle (in a local minimum). One should realize that a car-like robot will show such behavior even more often than a free-flying one. The latter can, when it has 'bumped into an obstacle', often 'slide along' the obstacle, without having to move away from the goal configuration, while a car-like robot can locally only move along its main axis.

7.3.4.1 The ALA potential field local method

We will now describe an (implemented) potential field method which uses the ALA metric. The algorithm can easily be generalized, and in fact every distinct metric induces a different version of the method.

We assume that c_1 is the start configuration, and c_2 the goal configuration of the local search. The potential field, used by the ALA-potential field method, is defined in a very primitive way. The positive potential, generated by the goal configuration c_2, is inversely proportional to the ALA distance to c_2, and the negative potentials, generated by the obstacles, are $-\infty$ in configurations where the robot intersects an obstacle, and 0 otherwise. We will refer to such negative potentials as *trivial negative potentials.* The validation of a configuration c is performed by simply adding up the values which the different potentials have in c. The advantage of this simple method is that the validation of a configuration c can be performed in a very efficient manner. The only computations that need to be performed for the validation of a configuration c are the computation of $D_{ALA}(c, c_2)$ (the ALA distance between c and goal configuration c_2) and the intersection test of the robot positioned at c. So the expensive computations of distances between configurations and obstacles are avoided. The drawback of such primitive potential fields is, as mentioned before, that the robot will often get stuck behind an obstacle. The way this problem is tackled is allowing a *limited* amount of backtracking, in order to make escape from local minima possible (in easy cases). We will go into this in more detail later.

Let us now define what the neighbors of a configuration are. A configuration c will have seven neighbors. They not only depend on c, but also on the goal configuration c_2 of the local search, and will be described by the function

$$N_{ALA} \in C \times C \times \{1, ..., 7\} \rightarrow C$$

The first argument corresponds to c, and the second to the goal configuration c_2. We now first define the function

$$P_{ALA} \in C \times C \times \{1, ..., 7\} \rightarrow paths$$

as follows:

Let d in $\mathbf{R}$ be a (small) chosen constant.

$P_{ALA}(c, c_2, 1)$ = The subpath $\tilde{P}$ of the shortest ALA path from c to c_2, such that $\tilde{P}$ starts at c and has length d.
$P_{ALA}(c, c_2, 2)$ = The arc path $A_f(c, *)$ of length d, which projects on $C_L(c)$.
$P_{ALA}(c, c_2, 3)$ = The line path $L_f(c, *)$ of length d, which projects on l_c.
$P_{ALA}(c, c_2, 4)$ = The arc path $A_f(c, *)$ of length d, which projects on $C_R(c)$.

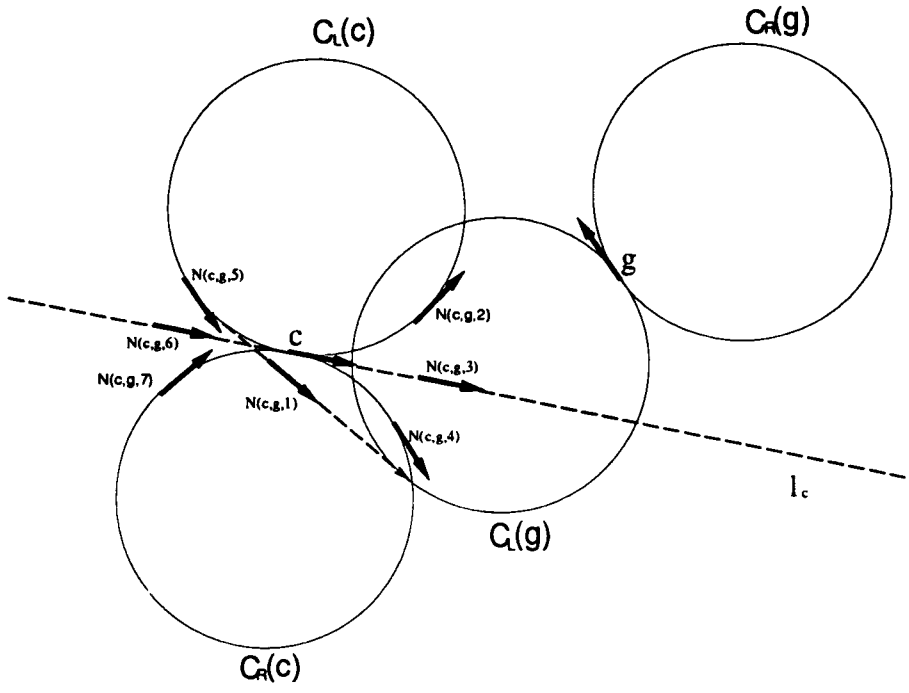

Figure 7.13: The neighbors of configuration c

$P_{ALA}(c, c_2, 5)$ = The arc path $A_b(c, *)$ of length d, which projects on $C_L(c)$.
$P_{ALA}(c, c_2, 6)$ = The line path $L_b(c, *)$ of length d, which projects on l_c.
$P_{ALA}(c, c_2, 7)$ = The arc path $A_b(c, *)$ of length d, which projects on $C_R(c)$.

Intuitively, $P_{ALA}(c, c_2, *)$ defines seven elementary motions from c, over a distance of d. $P_{ALA}(c, c_2, 1)$ defines a motion which is directed 'straight' towards c_2, in configuration space and with respect to the ALA metric. $P_{ALA}(c, c_2, 3)$ and $P_{ALA}(c, c_2, 6)$ define motions which are straight in workspace, while $P_{ALA}(c, c_2, 2)$, $P_{ALA}(c, c_2, 4)$, $P_{ALA}(c, c_2, 5)$, and $P_{ALA}(c, c_2, 7)$ define motions which are (maximally) curved in workspace.

The idea now is, that the neighbors of c are those configurations which are reached by following the paths defined by $P_{ALA}(c, c_2, *)$ from c. So

$$\forall i \in \{1, ..., 7\} : N_{ALA}(c, c_2, i) = (P_{ALA}(c, c_2, i))(1)$$
$$(= \text{The last configuration in } P_{ALA}(c, c_2, i))$$

See also figure 7.13. There c is the 'current' configuration, and g is the goal configuration of the local search. The seven neighbors of c are shown. We assume that the $A_b(c, *) \oplus L_f(*, *) \oplus A_f(*, g)$ path, where $A_b(c, *)$ projects on $C_L(c)$ and $A_f(*, g)$ projects on $C_L(g)$, is the shortest ALA path from c to g.

The given definition of neighbors is surely not the only possibility. We have though tried some alternative definitions, and experimental results indicated that the above definition works relatively well. It appeared for example that the first neighbor $N_{ALA}(c, c_2, 1)$ is quite essential.

Now a functional description of the decision function $compute\_path_{ALAPF}$ can be given. In this version the backtracking is not limited, and in this it differs from the final (and implemented) version which will be given later.

$$
\begin{aligned}
&compute\_path_{ALAPF}(a,b) = \\
&\quad (D_{ALA}\ (a,b) < d \wedge compute\_path_{ALA}(a,b)) \\
&\qquad \vee \\
&\quad \exists i \in \{1,...,7\} : D_{ALA}(N_{ALA}(a,b,i),b) < D_{ALA}(a,b) \\
&\qquad\qquad\qquad \wedge \\
&\qquad\qquad\qquad P_{ALA}(a,b,i) \text{ is a free path} \\
&\qquad\qquad\qquad \wedge \\
&\qquad\qquad\qquad compute\_path_{ALAPF}(N_{ALA}(a,b,i),b)
\end{aligned}
$$

The symmetry of the ALA potential field method can easily be obtained by performing the top level call of the above function in both directions (if the first fails).

Now conceptually an execution of $compute\_path_{ALAPF}(c_1,c_2)$ corresponds to a depth first traversal of a computation tree T, where the nodes are configurations and the edges correspond to elementary motions. The root of T is the start configuration c_1, and for every node n in T its children are exactly those neighbors $N_{ALA}(n,c_2,i)$ such that $N_{ALA}(n,c_2,i)$ validates to a higher value then n and $P_{ALA}(n,c_2,i)$ is a free path.

As mentioned before, the backtracking is limited in order to keep the complexity of the algorithm low. This is done by using a constant max_fail_depth, which defines the maximal allowed depth of subtrees in T which do not lead to a solution of the local motion planning problem, i.e., which do not contain any node c such that $D_{ALA}(c,c_2) < d$ and $compute\_path_{ALA}(c,c_2)$.

Doing so, large parts of the computation tree will be 'cut off'. It is important to realize that now the order in which the neighbors of a node are picked, is of considerable influence on the eventual shape of the tree traversed, and thus on the outcome of the search. In order to maximize the chances of success, a best-first instead of a normal depth-first search should be performed. This leads to the following (final) version:

$$
\begin{aligned}
&compute\_path_{ALAPF}(a,b) = \\
&\quad max\_depth = 0 \\
&\quad \textbf{return } (compute\_sub\_path_{ALAPF}(a,b,0))
\end{aligned}
$$

where the function $compute\_sub\_path_{ALAPF}$ of type $C \times C \times \mathbb{N}^+ \rightarrow boolean$ is defined as follows:

$compute\_sub\_path_{ALAPF}(a, b, depth) =$
$max\_depth = MAX(max\_depth, depth)$
if $D_{ALA}(a, b) < d$
then return$(compute\_path_{ALA}(a, b))$
else forall $j \in \{1, ..., 7\}$ in order of increasing $D_{ALA}(N_{ALA}(a, b, j), b)$
do if $D_{ALA}(N_{ALA}(a, b, j), b) < D_{ALA}(a, b)$
$\wedge$
$P_{ALA}(a, b, j)$ is a free path
then if $compute\_sub\_path_{ALAPF}(N_{ALA}(a, b, j), b, depth + 1)$
then return$(true)$
else if $max\_depth - depth > max\_fail\_depth$
then return$(false)$
return$(false)$

The (global) variable max_depth is used to store the maximal depth reached in the computation tree. After each failed recursion the value of max_depth is compared with the 'current' depth, and when the difference between these two values exceeds max_fail_depth, then the local search is given up (and fails).

Some good values must be chosen for the constants d and max_fail_depth. We have done quite a lot of experiments; for different values of d and max_fail_depth 30 runs of the global method (using the ALA potential field method) where performed, for a number of different scenes. These experiments indicated that the optimal setting of the two parameters lies somewhere near $d = \frac{3}{4}r_{min}$ and $max\_fail\_depth = 1$. Hence in the implementation of the algorithm we use these values. Actually the implementation still differs in one minor point from the above: to guarantee (fast) termination, we do not merely test whether the distance to the goal node decreases when we go to a neighbor, but we test whether it decreases by at least some (small) fixed constant. In figure 7.14 an example is given of a path that is computed by the ALA potential field method. So this motion planning problem is solved entirely be the local method. The computation time of the example is about 0.04 seconds.

7.3.4.2 The LAL potential field local method

The LAL potential field method is the LAL analog of the ALA potential field method. The basic difference is that it uses the LAL metric as underlying metric, instead of the ALA metric. Assume again that c_1 is the start configuration and c_2 the goal configuration of the local search. $P_{LAL}(c, c_2, 2), \ldots, P_{LAL}(c, c_2, 7)$ are defined in exactly the same manner as $P_{ALA}(c, c_2, 2), \ldots, P_{ALA}(c, c_2, 7)$, and $P_{LAL}(c, c_2, 1)$ is the subpath of the shortest LAL path from c to c_2, starting at c with length d. The neighbors of c $N_{LAL}(c, c_2, 1), \ldots, N_{LAL}(c, c_2, 7)$ are again defined in terms of P_{LAL}, as in the previous section. We will here give a functional description of the LAL potential field method, where the backtracking is not limited.

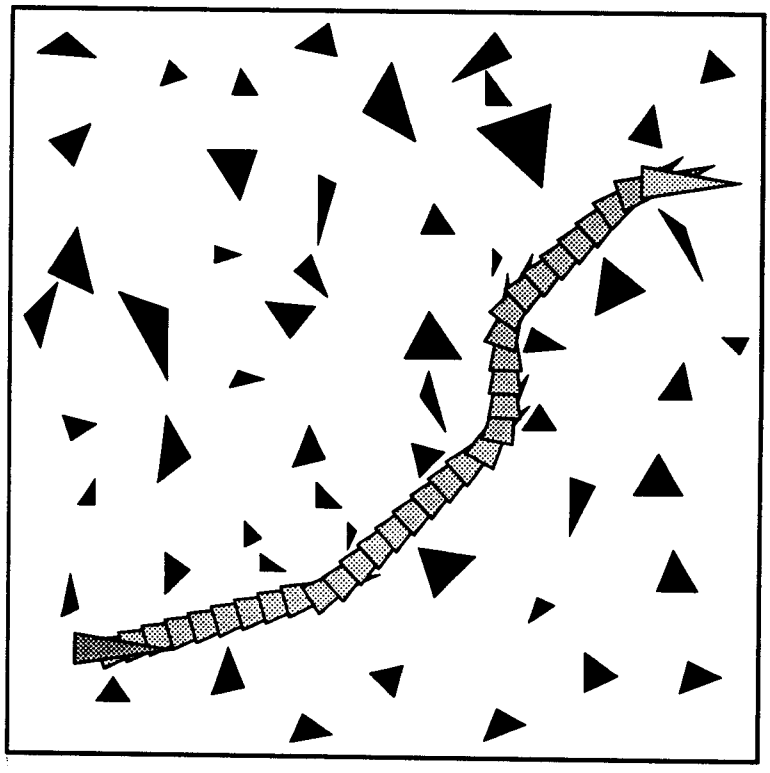

Figure 7.14: A path computed by the ALA potential field local method.

$$
\begin{aligned}
&\mathit{compute\_path}_{LALPF}(a,b) = \\
&\quad (D_{LAL}\ (a,b) < d \wedge \mathit{compute\_path}_{LAL}(a,b)) \\
&\qquad \vee \\
&\quad \exists i \in \{1,...,7\} : D_{LAL}(N_{LAL}(a,b,i),b) < D_{LAL}(a,b) \\
&\qquad\qquad\qquad \wedge \\
&\qquad\qquad\qquad P_{LAL}(a,b,i) \text{ is a free path} \\
&\qquad\qquad\qquad \wedge \\
&\qquad\qquad\qquad \mathit{compute\_path}_{LALPF}(N_{LAL}(a,b,i),b)
\end{aligned}
$$

The symmetry of the LAL potential field method can again be obtained by performing the top level call of $\mathit{compute\_path}_{LALPF}$ in both directions (if the first one fails). In order to keep the algorithm 'cheap', the backtracking is limited as in the ALA potential field method, using a constant max_fail_depth which defines the maximal allowed depth of subtrees in the computation tree which do not lead to a solution. Again d and max_fail_depth should be experimentally 'tuned'.

Unfortunately some experiments we did concerning the performance of the global method using the LAL potential field method as local method indicated that this performance is relatively bad, in comparison to other local methods, e.g. the ALA potential field method. Two important causes of this are the following:

- LAL paths tend to be longer than ALA paths, and, hence, the LAL potential field method tries to 'approximate' paths which are longer than those 'approximated' by the ALA potential field method. It is clear that this negatively effects the chances for success.

- For configurations of similar or approximately opposite orientation, the LAL metric typically defines a very large distance. The two straight path segments then tend to be quite long. So often a neighbor of the 'current' configuration c will validate to a lower value than c, solely because its orientation approximates the orientation of the goal node c_2, or c_2's opposite orientation, more than c's orientation does. This means that, in absence of obstacles, the number of 'better' neighbors typically is low, and hence that, in presence of obstacles, the chance of getting into a local minimum is large.

Due to the relatively bad performance of the LAL potential field method, it is of little practical value. It is included here for completeness.

7.3.4.3 Other potential field approaches

Some other potential field approaches that we tried will be briefly sketched here. We have tried an approach, based on the ALA metric, where no backtracking is performed to escape from local minima. Instead, when the robot gets stuck in a local minimum c_m, then the potential field is extended with a new (non-trivial) negative potential which retracts the robot from configuration c_m. So the idea is that during the search for a feasible path from the start to the goal configuration, the collisions of the robot with obstacles lead to adaptions of the potential field, such that the robot is pushed away from the places where it got stuck.

One advantage of such negative potentials with respect to negative potentials which retract the robot from the obstacles in general, is that they are quite cheap to evaluate. Assume that c_m is a configuration where the robot got stuck, and consequently a negative potential P_{c_m} was added, which retracts the robot from c_m. Then for the evaluation of P_{c_m} in some configuration c, the computation of the distance $D_{ALA}(c, c_m)$ is the only expensive operation that needs to be performed. The actual value returned by the evaluation of P_{c_m} in c should of course be somehow inversely proportional to $D_{ALA}(c, c_m)$, to prevent the robot getting near to c_m again.

Another advantage is that the number of (non-trivial) negative potentials can be kept low. Their number is equal to the number of times that the robot got stuck in a local minimum during the performed search, and we can simply give up the search when this number exceeds some given value. This provides a reasonable stop criterion for the method.

We have implemented and tested such a potential field method, but the performance of the global method when using this potential field method was not very encouraging. It appeared that in most cases where the robot must defect more than marginally from the 'ideal' path, the required number of added negative potentials tends to be quite high, which means that for every evaluation of the potential field a large number of distance computations need to be performed. This makes the method rather expensive. For this reason the idea of 'dynamic' potential fields, as sketched here, was dropped, and we will not go into further details.

Another idea that we tried, was to use a method like the ALA potential field method, but to deal with local minima by performing some *random motion* when the robot gets stuck in a local minimum, instead of doing the (more expensive) backtracking. This idea also was dropped, due to relatively poor performance.

7.4 Local methods for forward car-like robots

Here we will give some local methods which compute paths feasible for forward car-like robots, again making use of the constructs defined in 7.2. As mentioned in the introduction of this chapter, these local methods will not be symmetric. We will (again) define each local method by the functions *compute_path* $\in C \times C \rightarrow$ *boolean*, *construct_path* $\in C \times C \rightarrow$ *paths* and $D \in C \times C \rightarrow \mathbf{R}$, where *compute_path* now does not need to be a symmetrical function. For experimental results we again refer to chapter 8.

7.4.1 The ALA forward local method

The ALA forward motion method is analog to the ALA motion method. It tries the shortest ALA forward path from the start configuration to the goal configuration, and succeeds if this path is free, and fails otherwise. So the following definitions define the method :

$compute\_path_{ALAFW}(a,b)$ = Whether the shortest ALA forward path from a to b lies entirely in free configuration space.

$construct\_path_{ALAFW}(a,b)$ = **if** $compute\_path_{ALAFW}(a,b)$
then the shortest ALA forward path from a to b
else undefined

$D_{ALAFW}(a,b)$ = The length of the shortest ALA forward path from a to b.

We will refer to D_{ALAFW} as the *ALA forward metric.*

7.4.2 The LAL forward local method

The LAL forward method is the analog of the LAL local method. It tries the shortest LAL forward path from the start configuration to the goal configuration. If such a path exists and is free, then the method succeeds, and otherwise it fails. So the required functions are defined as follows :

$compute\_path_{LALFW}(a,b)$ = Whether the shortest LAL forward path from a to b lies entirely in free configuration space, if such a path exists, and false otherwise.

$construct\_path_{LALFW}(a,b)$ = **if** $compute\_path_{LALFW}(a,b)$
then the shortest LAL forward path from a to b
else undefined

$D_{LALFW}(a,b)$ = The length of the shortest LAL forward path from a to b, if such a path exists, and ∞ otherwise.

D_{LALFW} will be referred to as the *LAL forward metric.*

7.4.3 The ALA-LAL forward motion method

This method first tries the shortest ALA forward path from the start configuration to the goal configuration. If this is a free path, then the method succeeds. If the path is not free, then the shortest LAL forward path is tried, and the method succeeds if this path is free (and fails otherwise). The metric used is the ALA forward metric. This leads to the following definitions:

$$compute\_path_{ALAFW-LALFW}(a,b) = compute\_path_{ALAFW}(a,b) \vee compute\_path_{LALFW}(a,b)$$

$$\begin{aligned} construct\_path_{ALAFW-LALFW}(a,b) = \ & \textbf{if } compute\_path_{ALAFW}(a,b) \\ & \textbf{then } \text{the shortest ALA forward path from } a \text{ to } b \\ & \textbf{else if } compute\_path_{LALFW}(a,b) \\ & \quad \textbf{then } \text{The shortest LAL forward path from } a \text{ to } b \\ & \quad \textbf{else } \text{undefined} \end{aligned}$$

$$D_{ALAFW-LALFW}(a,b) = D_{ALA}(a,b)$$

7.4.4 The ALA forward potential field method

As for general car-like robots, we can define potential field methods for forward car-like robots, using metrics which take into account the nonholonomic constraints of such robots (e.g., the ALA forward metric and the LAL forward metric). In section 7.3.4 we have seen the ALA potential field method and the LAL potential field method, which both are specializations of the same abstract potential field method, conceptually only differing in the metric that is used.

Here we will give a another specialization of this abstract potential field method, which uses the ALA forward metric as underlying metric, to compute paths which respect the nonholonomic constraints of forward car-like robots. It will be referred to as the ALA forward potential field method.

Assume again that c_1 is the start configuration and c_2 the goal configuration of the local search. The potential field (again) consists of a positive potential, which is inversely proportional to the (ALA forward) distance to the goal configuration c_2, and for each obstacle a (trivial) negative potential which evaluates to $-\infty$ in configurations where the robot intersects the obstacle, and 0 elsewhere. The neighbor definition, as provided in 7.3.4.1, is not suitable for forward car-like robots. The reason for this is that, for a forward car-like robot, the neighbors $N(c,c_2,5)$,$N(c,c_2,6)$, and $N(c,c_2,7)$ cannot be reached from c by a path which is short and respects the constraints of the robot. These neighbors namely correspond to configurations which can be reached from c (by a general car-like robot) by performing a short *backwards* motion, and forward car-like robots cannot move backwards. So we will drop these three neighbors, and only use the neighbors $N(c,c_2,1)$,$N(c,c_2,2)$,$N(c,c_2,3)$, and $N(c,c_2,4)$. Of course, $N(c,c_2,1)$ must be redefined as the configuration which is reached from c by following the shortest ALA *forward* path from c to c_2 over a distance of d. Formally the four neighbors of a configuration c are thus defined as follows:

Let d in $\mathbf{R}$ be a (small) chosen constant,
and let the function P_{ALAFW} of type $C \times C \times \{1,\ldots,4\} \rightarrow paths$ by defined by

$P_{ALAFW}(c, c_2, 1)$ = The subpath $\tilde{P}$ of the shortest ALA forward path from c to c_2, such that $\tilde{P}$ starts at c and has length d.
$P_{ALAFW}(c, c_2, 2)$ = The arc path $A_f(c, *)$ of length d, which projects on $C_L(c)$.
$P_{ALAFW}(c, c_2, 3)$ = The line path $L_f(c, *)$ of length d, which projects on l_c.
$P_{ALAFW}(c, c_2, 4)$ = The arc path $A_f(c, *)$ of length d, which projects on $C_R(c)$.

Then

$$\forall i \in \{1, ..., 4\} : N_{ALAFW}(c, c_2, i) = (P_{ALAFW}(c, c_2, i))(1)$$
$$(= \text{The last configuration in } P_{ALAFW}(c, c_2, i))$$

$P_{ALAFW}(c, c_2, i)$ defines a (short) path which connects c to its neighbor $N_{ALAFW}(c, c_2, i)$, and is feasible (for a forward car-like robot) in absence of obstacles. The decision function of the ALA forward potential field method can now be defined as follows:

$compute\_path_{ALAFWPF}(a, b) =$
    $max\_depth = 0$
    return $(compute\_sub\_path_{ALAFWPF}(a, b, 0))$

where the function $compute\_sub\_path_{ALAFWPF}$ of type $C \times C \times \mathbb{N}^+ \rightarrow boolean$ is defined in the following way:

$compute\_sub\_path_{ALAFWPF}(a, b, depth) =$
    $max\_depth = MAX(max\_depth, depth)$
    if $D_{ALAFW}(a, b) < d$
    then return$(compute\_path_{ALAFW}(a, b))$
    else forall $j \in \{1, ..., 4\}$ in order of increasing $D_{ALAFW}(N_{ALAFW}(a, b, j), b)$
        do if $D_{ALAFW}(N_{ALAFW}(a, b, j), b) < D_{ALAFW}(a, b)$
                $\wedge$
            $P_{ALAFW}(a, b, j)$ is a free path
            then if $compute\_sub\_path_{ALAFWPF}(N_{ALAFW}(a, b, j), b, depth + 1)$
                then return$(true)$
                else if $max\_depth - depth > max\_fail\_depth$
                    then return$(false)$
        return$(false)$

The constants d and max_fail_depth should again experimentally be 'tuned'. In the current implementation of the algorithm, we take the same values as for the ALA potential field method, namely $d = \frac{3}{4} r_{min}$, and $max\_fail\_depth = 1$.

Chapter 8

Experimental results

Because the method presented in this paper is a heuristic one, it seems impossible to provide some good theoretical bounds on its efficiency. We do though claim that the method is very fast, and to support this claim we present in this chapter a fairly large number of experimental results, obtained through the performance of many test runs of the motion planner on a number of 'typical' scenes. We have implemented two versions of the method. One solves the motion planning problem for general car-like robots and uses an undirected underlying graph, as described in section 4.2, and the other deals with forward car-like robots and uses a directed underlying graph, as described in section 4.3. We will refer to the former version as the general version, and to the latter as the forward version of the random motion planning method for car-like robots. In section 8.3 we give experimental results for the general version, and in section 8.4 we give experimental results for the directed version. We will first though say something about the implementation (in section 8.1) and the parameters of the method (in section 8.2).

8.1 Implementation aspects

Both versions were implemented in C (partially C++) on a Silicon Graphics 4D/35 work station, which is based on a R3000 processor and has a performance of about 33 MIPS/4.5 MFLOPS/25.4 SPECMARKS. Apart from the 'computational parts', as described in this paper, they also contain graphical user interfaces which make interactive use possible. These interfaces enable the user to play with the different parameters, and they display the computed graphs and motions. The implementations also contain testing routines, which can be used for conveniently testing different aspects of the method. The coordinates of all obstacles should lie in the unit square, and all random 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 the unit square.

8.2 Parameters of the method

As mentioned before, there are a number of parameters to set when running the program for a particular scene. These parameters are used by both the general version, as well as by the forward version.

minimal turning radius Every car-like robot has some fixed minimal turning radius, which is defined by its maximal steering angle. See section 7.1 for more details. The user can choose an arbitrary minimal turning radius for the robot in the scene, and the program will compute paths which are feasible with respect to this value.

maximal distance As described in chapter 4, when connecting nodes, we only consider nodes that lie near enough to each other. Recall that the set of neighbors of a configuration c is a subset of the set of nodes which lie within distance *maxdist* of c. The value of this parameter has large impact on the efficiency of the global methods. Choosing it small means that the adding of a node results in only a few executions of the local method, making the node adding 'operation' relatively cheap. The paths computed by the local method will though all be quite short, and, hence, we typically need a large graph for solving the entire problem, which means that a large number of nodes need to be added. We can on the other hand chose a large value for the maximal distance, which means that the local method will also (try to) compute longer paths, hence, reducing the required size of the graph which solves the problem. The consequences though are that every newly added node leads to many executions of the local method (it has many neighbors), and that the local method will fail more often (on the average it tries to compute longer paths). So there is a clear trade-off. The optimal value very much depends on the particular scene that we are dealing with. It must be such that there is a reasonable chance that two configurations which lie within maximal distance of each other can successfully be connected by the local method. This implies that for difficult scenes the optimal value for the maximal distance will lower be than for an easy ones.

local method We have implemented most of the local methods described in chapter 7. For the general version (for general car-like robots) we have the ALA method, the LAL method, the ALA-LAL method, and the ALA potential field method. The three former methods are very primitive methods which often fail, but are very fast. The ALA potential field method is a 'smarter' method, which succeeds more often in computing feasible paths, but is (consequently) also more time-consuming. In fact there is a similar trade-off here like is the case for the maximal distance parameter. Cheap local methods bring about that extensions of the graph are relatively cheap, but not always of great value, which typically results in a large graph. Expensive local methods on the other hand result in extensions of the graph which are in a way 'better', but also more time consuming. For forward car-like robots we have four local methods, which are analog to the local methods implemented for general car-like robots. These are the ALA forward method, the LAL forward method, the ALA-LAL forward method, and the ALA forward potential field method.

node adding strategy We have implemented all of the node adding strategies described in chapter 5. The goal of these strategies is to add nodes in a somewhat smarter way than fully random, and through this to improve the running times of the global methods. We have the choice between the normal, the edge sensitive, and the edge requiring node adding strategies, and further more each of these can be combined with the forbidden configurations and the adaptive node adding strategies.

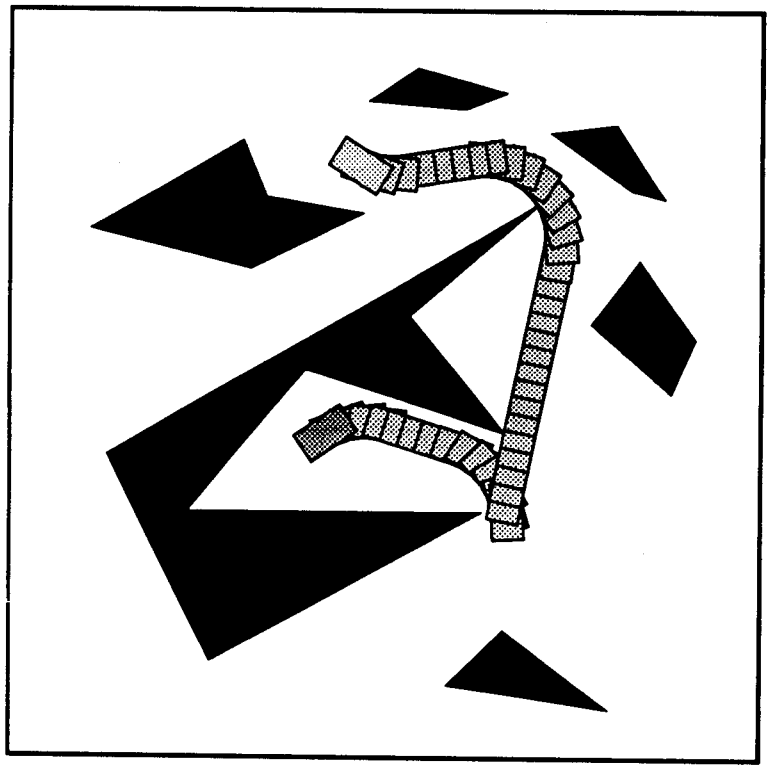

Figure 8.1: Scene 1.

8.3 Experimental results for general car-like robots

In this section we give a some experimental results for the general version of the random motion planner, obtained by performing a large number of test runs on 8 'typical' scenes. Based on these results, we will also draw some conclusions.

8.3.1 The test scenes

We will briefly discuss each of the test scenes used. See figures 8.1 to 8.8 for the scenes together with paths solving the corresponding motion planning problems, computed by (the general version of) the random motion planner. In each scene the start configuration s is indicated by a darker color of the robot (positioned at s).

1. The first test scene is a very simple one. There are only a few obstacles, there are no narrow passages through which the robot must go, and there are a number of

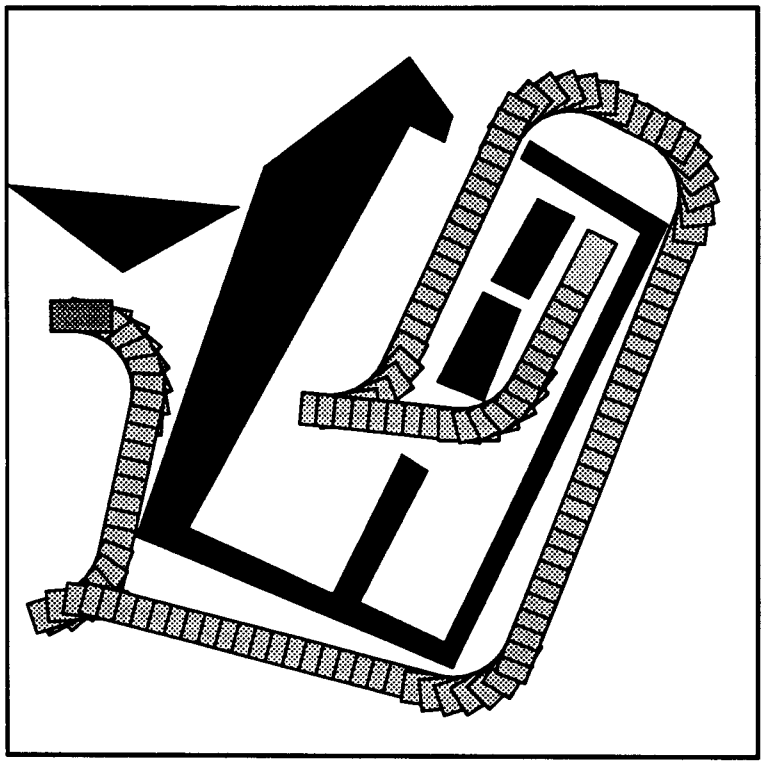

Figure 8.2: Scene 2.

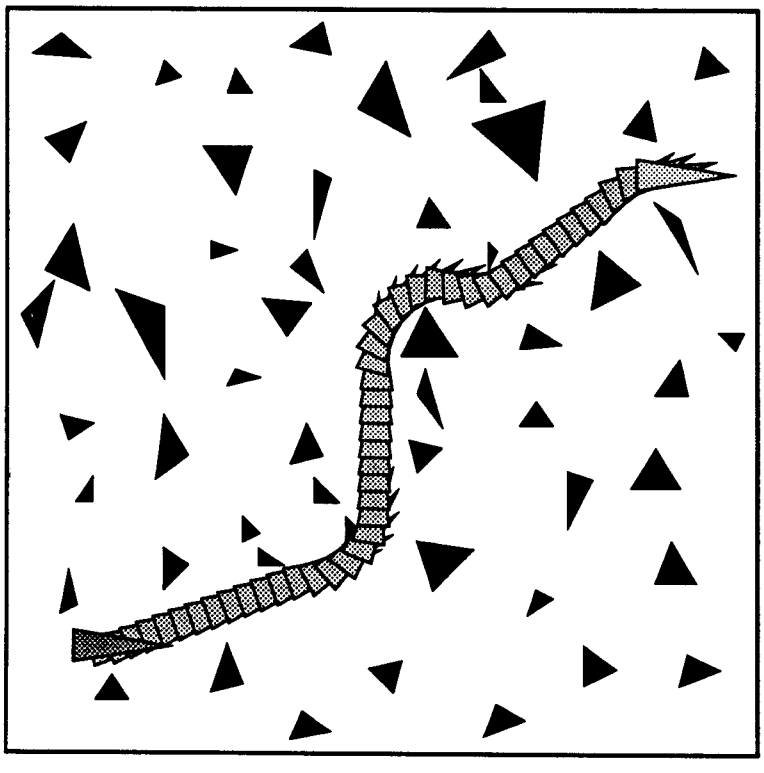

Figure 8.3: Scene 3.

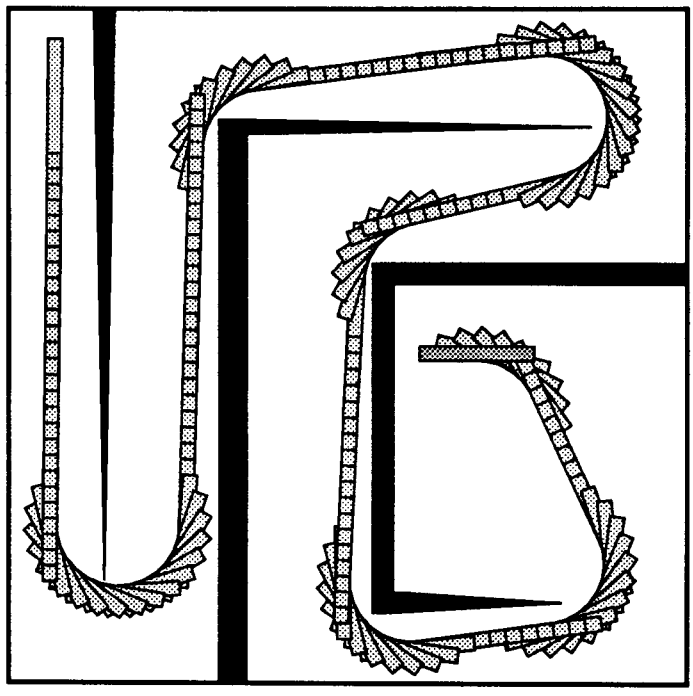

Figure 8.4: Scene 4.

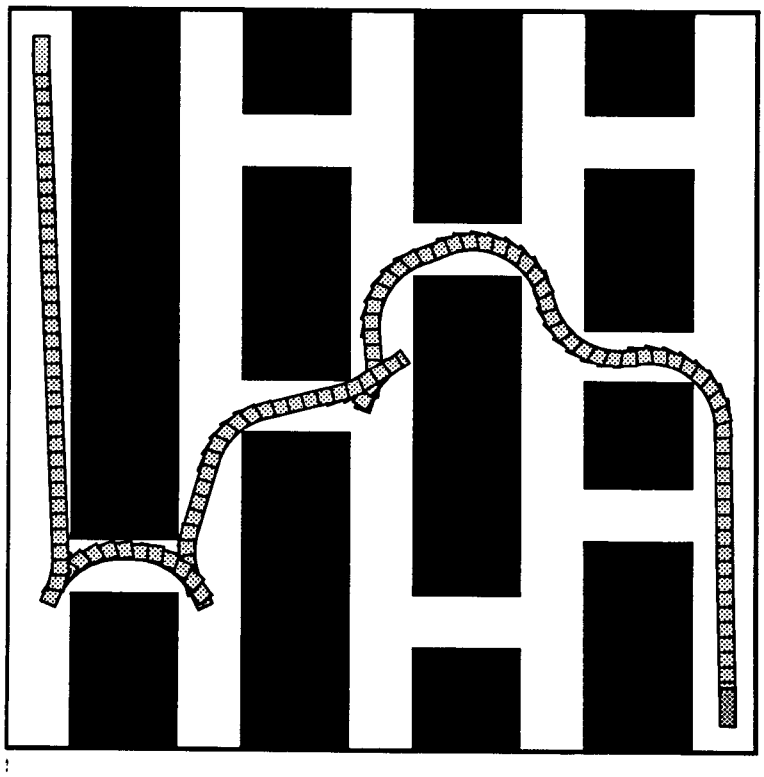

Figure 8.5: Scene 5.

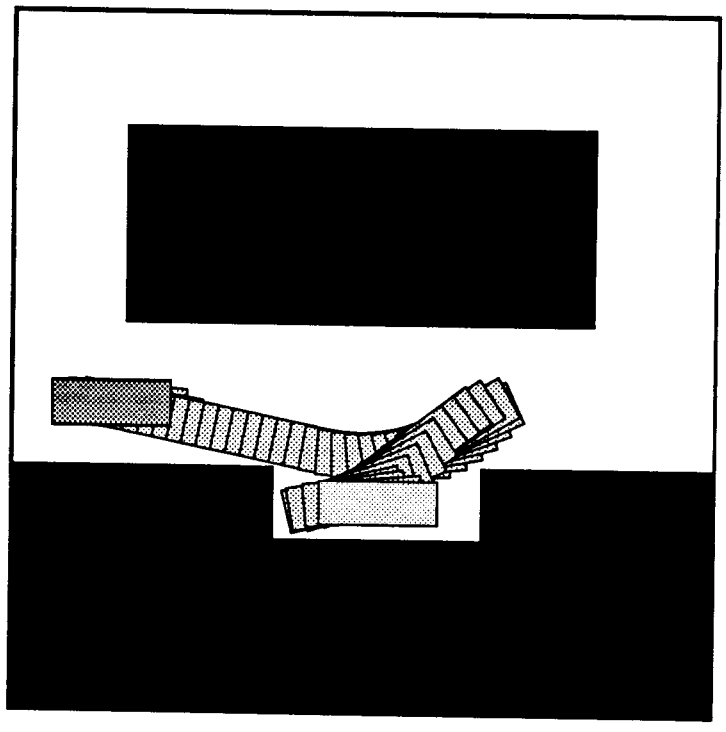

Figure 8.6: Scene 6.

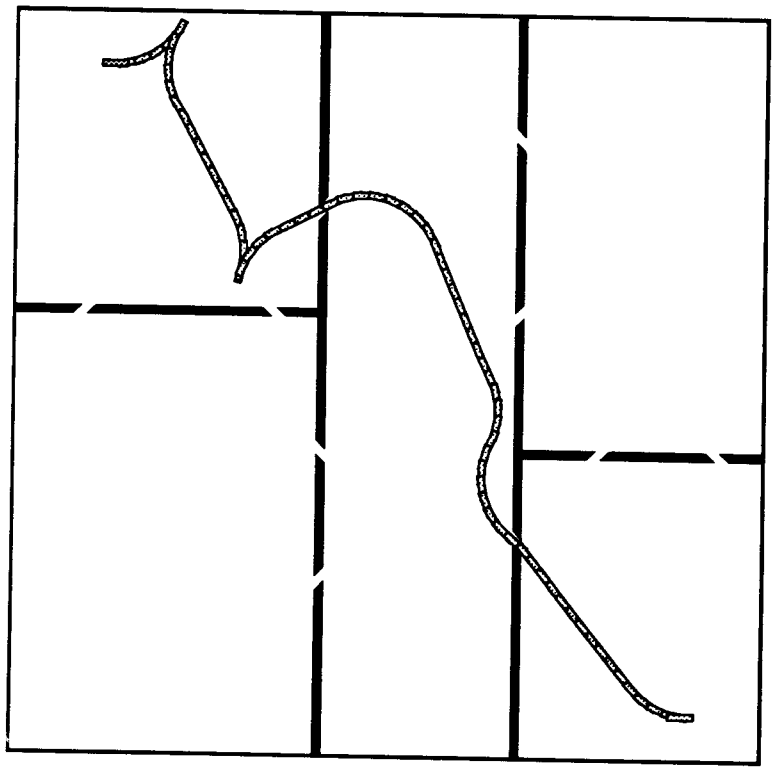

Figure 8.7: Scene 7.

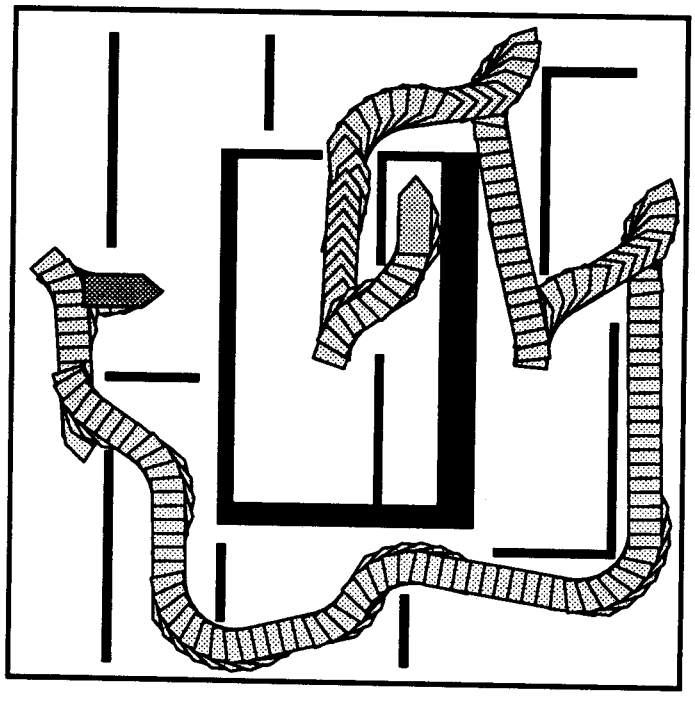

Figure 8.8: Scene 8.

different ways to solve the problem.

2. The second scene is a bit more difficult. Any path solving the problem is quite long, and near to the goal configuration the robot does not have much freedom of movement. This is typically a scene where any potential-field-like approach (for the whole problem) would have great difficulty, because the robot has to move away from the goal configuration a number of times.

3. The third scene looks quite difficult, but is in fact relatively easy. There are many possible paths which solve the problem, and many of these correspond to motions which are approximately directed towards the goal configuration. Hence, we can expect potential field approaches to also do a good job here. In fact the ALA potential field method, as described in section 7.3.4 can solve the whole problem (see also figure 7.14). Still, in the test runs that we performed with the global method using the ALA potential field method, the problem was never solved in one step. The reason for this is that we use lower values for *maxdist* than is the distance between the start and the goal configuration in scene 3. Recall that *maxdist* defines the size of the neighborhood of a new node c, from which the other nodes are picked to which connections are tried.

   A concern could be the large number of obstacles, which may cause high computation times for intersection tests (of the robot with the obstacles). The intersection routines used in our implementation though are very efficient in discarding obstacles which are far away from the robot (see [Gie93]), and (consequently) we never had

problems related to the complexity of the scene.

4. In contrast to the previous scene, scene 4 looks quite easy (and for humans also is easy), but is relatively hard to solve for our motion planner. One difficulty is that in areas other the rectangular space around the start configuration, it is very hard for the robot to turn around. To do so, a number of reversals are needed. So about half of all nodes which are added have a 'wrong' orientation, and are therefore practically useless. Another difficulty is that the robot, due to its long and thin shape, has little space to perform the necessary curves.

5. Scene 5 is a maze-like scene, which again does not give the robot much freedom of movement.

6. Scene 6 is a very well-know scene, which is mostly referred to as *the parallel parking problem*. It again is not a very easy scene for our motion planner, because it requires a few nodes to be added near the goal configuration, with a similar orientation as the goal configuration. So one could say that it requires a certain amount of 'luck'.

7. Scene 7 is a difficult scene for our method. The only way to get from the start configuration to the goal configuration, is by passing through some of the narrow passages. Now our (dumb) local methods will not very often succeed in computing paths which go through one of these passages, and, hence, it is necessary that some nodes are added inside or nearby the narrow passages. The problem though is that in the scene there are large free areas, and the chance that a random node is added in these areas is much higher than that it is added in (or near) a narrow passage. Therefore it can take quite a while before enough (good) nodes have been added in (or near) the passages. Fortunately we have the forbidden configurations node adding strategy (see section 5.2), which brings about that more nodes are added in 'difficult' areas. Testing results show that this strategy really helps here.

8. The final scene is a very difficult one. It contains many narrow passages, it provides little freedom of movement for the robot, and any path solving the problem is very long.

8.3.2 The tests, their results, and some conclusions

We will now give some experimental results, obtained by the performance of many test runs of the random motion planner for general car-like robots on the test scenes shown. We do not use one default parameter setting here, but instead give results for a number of different settings. For every test scene we performed 100 test runs for 12 different settings of the parameters, and we give the average running time of the method for each of these settings. We tested each combination of the four implemented local methods with the normal, the edge sensitive, and the edge requiring node adding strategies. We will refer to these tests as the *main tests*.

For these tests we do not use the forbidden configurations strategy, nor the adaptive adding strategy. For scenes where the forbidden configurations strategy is of help, we will though give some additional test results obtained by using it. It was already mentioned in chapter 5 that for car-like robots the adaptive adding strategy does not bring much good, and that this is caused by the relatively high costs of the distance computations for

car-like robots. In fact, we did some tests with the adaptive adding strategy, but did not find one scene where the strategy improved the running times. For this reason we will not use it here.

We use 0.50 as the maximal distance in the main tests of the scenes 1 to 7. We have done some preliminary tests (in the test scenes) concerning the use of different values for the maximal distance, and it turned out that for these scenes the optimal value always was something between 0.40 and 0.70. For scene 8 though the optimal maximal distance appeared to be a considerably lower value. The reason for this clearly is that scene 8 is a more difficult scene than the others, and consequently the local methods fail too often for configurations which lie relatively far away from each other. We decided on 0.25 as default maximal distance for scene 8. Some tests indicated that the running times obtained with this value are about twice as good as those obtained when using 0.50 as maximal distance.

Furthermore, all main tests except the one corresponding to scene 6 (the parking scene) were performed with 0.10 as value for the minimal turning radius. We will give a few additional results for other (larger) turning radii. For the parking scene the motion planning problem is quite trivial (and not 'realistic') if the minimal turning radius of the robot is 0.10, and therefore we use a higher value for this scene, i.e., 0.25.

We give the results of the main tests in tables where every row corresponds to one of the four local methods, and every column to one of the three node adding strategies. The running times are in seconds, and in every table the best value(s) is (are) emphasized.

The computation time consumed by the smoothing of the final paths is not included here. The time consumed by the graph smoothing varies from about 0.2 seconds for the easy scenes, up to about 2 seconds for scene 8, and for path smoothing it varies from approximately 1 second for the easy scenes, up to about 8 seconds for scene 8. All the paths shown in figures 1 to 8 where obtained by performing both types of smoothing on the 'ugly path', but often graph smoothing alone seems sufficient.

scene 1	normal	edge sensitive	edge requiring
ALA	0.6	**0.4**	**0.4**
LAL	2.2	1.6	1.1
ALA-LAL	0.6	**0.4**	**0.4**
ALA pot.field	1.1	0.7	0.8

scene 2	normal	edge sensitive	edge requiring
ALA	7.0	4.8	4.3
LAL	4.1	3.4	3.1
ALA-LAL	2.6	2.4	**1.9**
ALA pot.field	4.8	3.8	3.6

scene 3	normal	edge sensitive	edge requiring
ALA	1.7	0.8	0.7
LAL	2.1	1.2	0.9
ALA-LAL	1.4	0.6	**0.5**
ALA pot.field	2.0	0.8	0.7

scene 4	normal	edge sensitive	edge requiring
ALA	6.2	10.9	17.4
LAL	8.4	10.7	29.8
ALA-LAL	**4.7**	7.2	11.8
ALA pot.field	8.5	8.9	12.8

scene 5	normal	edge sensitive	edge requiring
ALA	10.5	5.9	7.0
LAL	8.8	4.9	7.2
ALA-LAL	6.4	**3.2**	4.1
ALA pot.field	14.6	3.8	5.1

scene 6	normal	edge sensitive	edge requiring
ALA	49.6	17.0	8.5
LAL	22.9	11.7	7.1
ALA-LAL	7.5	5.7	**1.9**
ALA pot.field	11.7	6.1	4.9

scene 7	normal	edge sensitive	edge requiring
ALA	33.2	37.2	33.8
LAL	68.2	37.2	52.8
ALA-LAL	10.7	14.4	**10.6**
ALA pot.field	14.3	12.1	11.4

scene 8	normal	edge sensitive	edge requiring
ALA	12.7	11.1	57.7
LAL	32.6	22.4	87.2
ALA-LAL	9.3	**7.2**	36.9
ALA pot.field	13.7	12.5	40.9

First we will draw some conclusions about the performance of the four tested local methods. For very easy scenes, like for example scene 1 and scene 3, the ALA method and the ALA-LAL method seem to give the best running times of the global method. As the scenes get more complex, the performance of the ALA method gets relatively worse (with respect to the performance of the other local methods), while that of the ALA potential field method improves. In none of the scenes does the ALA potential field method perform as well though as the ALA-LAL method, and over the whole the ALA-LAL method clearly is the best one. The LAL method does not perform very well in any scene, except perhaps in scene 2, where though the goal configuration is such that it is much easier to reach with a LAL path than an ALA path. This is of course a very scene dependent reason, and we will therefore not draw any conclusions from it. In fact it is logical that the LAL method performs worse than the ALA method, because a LAL path P_1 connecting two configurations is typically longer than an ALA path P_2 connecting the same two configurations, and, hence, the chance that P_1 intersects an obstacle is larger than the chance that P_2 does.

So the results are quite clear. The ALA-LAL method is the best one. Only for very easy scenes does the ALA method give comparable results. For more complex scenes the ALA potential field method performs quite well, but still does not achieve the performance of the ALA-LAL method. So if we want a default local method, the obvious choice is the ALA-LAL method.

For the node adding strategies the results are less clear. Each of the three strategies achieves the best running time for at least one of the test scenes. For most cases, the edge requiring node adding strategy seems to give the best results. If though we have a scene where any path which solves the problem is relatively long and contains many curves and/or cusps (like for example scenes 4,5 and 8), then the other strategies are better. In such scenes it takes quite long for the start and the end component to grow all the way to each other if they are only extended with 'loose' nodes (like is the case when the edge requiring node adding strategy is applied), and it appears to be better to do some intermediate work between the two components. Scene 4 is the only test scene where the normal node adding strategy is the best one. The problem here with the edge sensitive node adding strategy probably is that early added nodes with a 'wrong' orientation (see also (4) in section 8.3.1), prevent (necessary) nodes with a 'right' orientation to be added.

It is hard to decide which strategy should be chosen as default strategy. On one hand, for most scenes the edge requiring strategy seems to give more spectacular results than the edge sensitive one, but on the other hand, the latter strategy seems to be 'safer'. For every scene it gives reasonable running times, while the edge requiring strategy performs bad in difficult scenes which ask for long paths (like scene 8).

We have done some additional tests concerning the use of the forbidden configurations node adding strategy, and other values for the minimal turning radius of the robot.

It was already mentioned (and explained) in 8.3.1 that scene 7 typically is a scene where the forbidden configurations strategy should help. We have therefore repeated the main test for scene 7, but with application of the forbidden configurations node adding strategy. The results where as follows:

	normal	edge sensitive	edge requiring
ALA	9.1	8.2	8.1
LAL	10.4	9.5	8.3
ALA-LAL	4.7	**4.6**	4.7
ALA pot.field	7.6	7.6	6.4

So we see a great improvement. For all other test scenes though the forbidden configurations strategy eats up some extra time.

The exact influence of a larger minimal turning radius is very dependent of the particular scene we are dealing with. We will just give some results, to give an indication of the influence of using a larger turning radius. We have not done enough testing with different turning radii to draw any concrete conclusions here, except that the running times get worse for larger values of the minimal turning radius, which though is rather obvious.

- The average running time for scene 1 with the ALA-LAL local method, the edge sensitive node adding strategy (= an optimal setting in the main test for scene 1), and 0.25 as value of the minimal turning radius, over 100 runs of the motion planner is about 1.4 seconds. So this is $3\frac{1}{2}$ times the running time of the method when using 0.10 as minimal turning radius.

- For scene 2 the average running time for the optimal setting in the main test with 0.25 as minimal turning radius, is about 15 seconds, which is more than 7 times the average running time when using 0.10.

- For scene 3 the average running time for the optimal setting in the main test with 0.25 as minimal turning radius, is about 1.4, which 3 times the running time when using 0.10.

The sizes of the underlying graphs which solve the problems are quite small. For the easy test scenes, like for example scene 1 and scene 3, on the average only about 50 nodes are required. Of course this number increases for more difficult scenes. The average number of required nodes for scene 4 is about 300, and for scene 8 it is something like 400.

8.4 Experimental results for forward car-like robots

Now we will give some experimental results for the forward version of the random motion planner, obtained by the performance of a large number of test runs on 6 'typical' test scenes, and we will draw some conclusions.

8.4.1 The test scenes

As test scenes, we will use the scenes 1, 2, 3, 4, 5, and 7 as described in section 8.3. So we do not use the parking scene. This scene is not solvable for forward car-like robots, unless a very small turning radius is used, which of course makes the scene trivial. Scene 8 is also not solvable for car-like robots with turning radius 0.10.

See figures 8.9 to 8.12 for a few paths, feasible for forward car-like robots (with turning radius 0.10), computed by the forward version of the random motion planner.

8.4.2 The tests, their results, and some conclusions

For each test scene, we again perform the main tests. These consist of 100 test runs for each test scene, and each combination of the ALA forward method, the ALA-LAL forward method, and the ALA forward potential field method, with the normal, the edge sensitive, and the edge requiring node adding strategies. So we do not use the LAL forward method. The reason for this is that some preliminary tests we have done showed that the LAL forward method gives really bad running times. A cause for this surely is, that LAL forward paths do not always exist, in contrast to for example ALA forward paths, and that if they exist, they often are relatively long.

Again, the forbidden configurations strategy and the adaptive adding strategy are not used for these tests, but where the former is of help, we will give some additional results. The used maximal distance is 0.50, and the used turning radius is 0.10. As for the general car-like robots, the smoothing costs (which are comparable to those for general car-like robots) are not included.

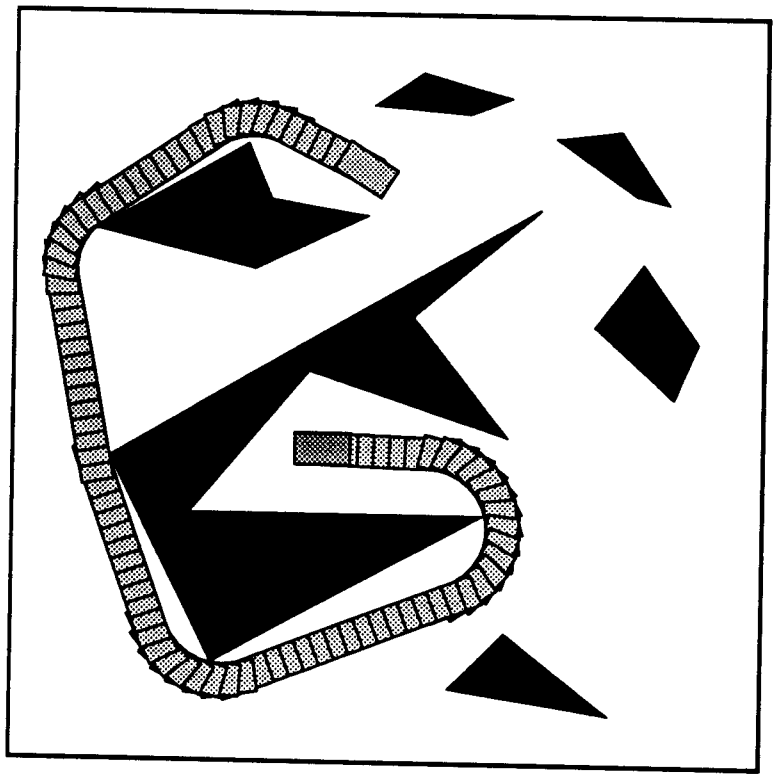

Figure 8.9: A reversal free path in scene 1.

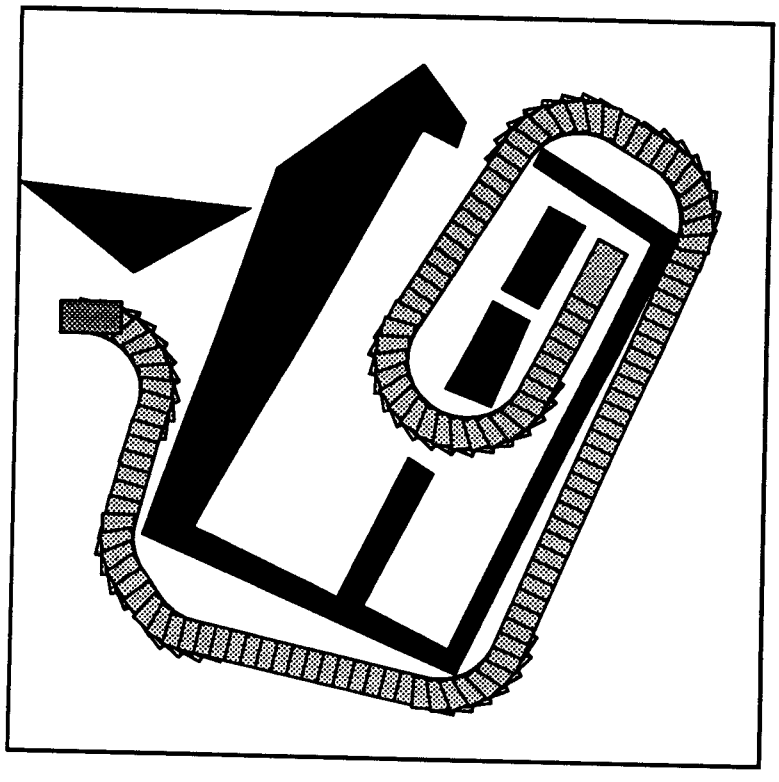

Figure 8.10: A reversal free path in scene 2.

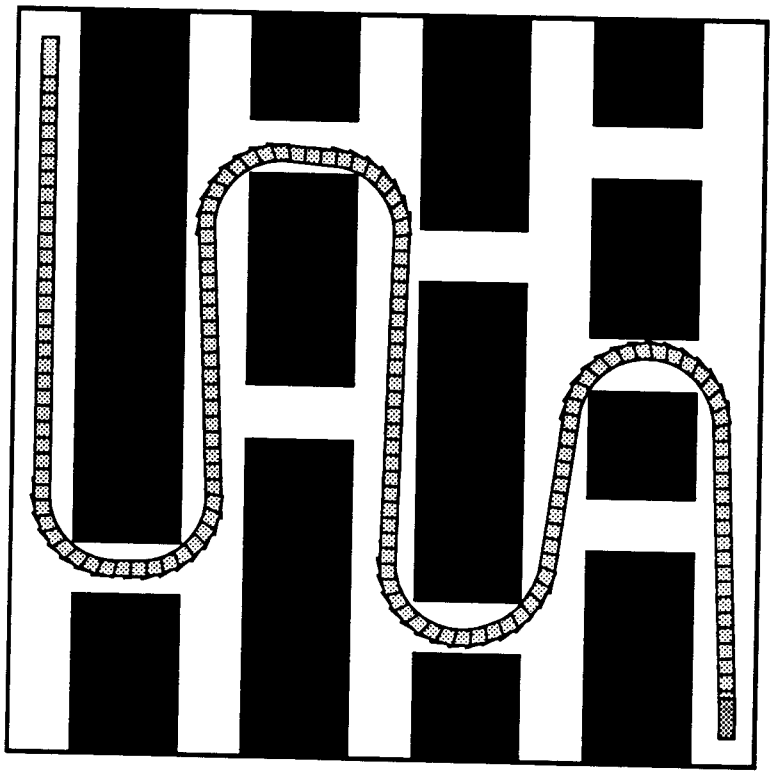

Figure 8.11: A reversal free path in scene 5.

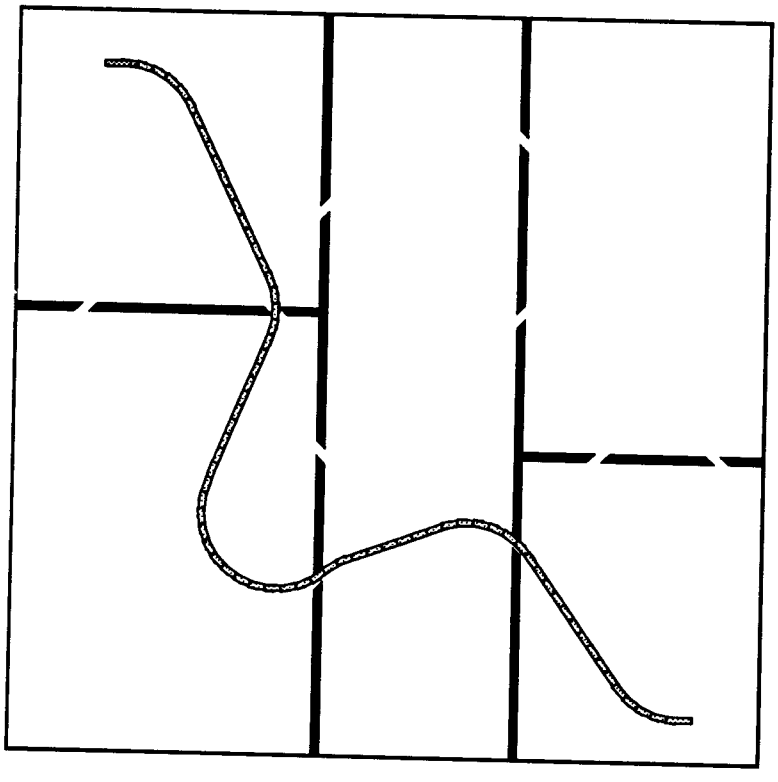

Figure 8.12: A reversal free path in scene 7.

We again give the results of the main tests in tables where the (average) running times are in seconds, and the best value for each test scene is emphasized.

scene 1	normal	edge sensitive	edge requiring
ALA	2.7	1.6	1.0
ALA-LAL	1.9	1.3	**0.8**
ALA pot.field	7.2	6.1	2.5

scene 2	normal	edge sensitive	edge requiring
ALA	35.1	22.8	12.2
ALA-LAL	11.6	9.1	**3.6**
ALA pot.field	90.1	73.6	29.8

scene 3	normal	edge sensitive	edge requiring
ALA	5.2	3.1	1.3
ALA-LAL	4.1	2.7	**0.9**
ALA pot.field	10.3	6.7	3.1

scene 4	normal	edge sensitive	edge requiring
ALA	7.8	7.1	**4.4**
ALA-LAL	8.0	6.3	4.8
ALA pot.field	27.0	18.7	12.0

scene 5	normal	edge sensitive	edge requiring
ALA	80.6	41.4	25.6
ALA-LAL	40.9	24.3	**15.4**
ALA pot.field	243.4	98.6	71.8

scene 7	normal	edge sensitive	edge requiring
ALA	80.1	52.6	27.9
ALA-LAL	39.0	28.2	**11.5**
ALA pot.field	211.3	96.6	48.7

The ALA-LAL forward method is clearly the best local method, followed by the ALA forward method. We see that the performance of the ALA forward potential field method is very bad. While for undirected underlying graphs the ALA potential field method performs better than the ALA method, for directed underlying graphs the ALA forward potential field method produces running times which are about 3 times as bad as those produced by the ALA forward method. Apparently, the 4 neighbors defined for forward car-like robots (see section 7.4.4) do not provide enough 'maneuvering freedom', in contrast to the 7 neighbors defined for general car-like robots (see section 7.3.4.1).

Furthermore it is clear that the edge requiring node adding strategy is the best one. For undirected graphs, the edge requiring node adding strategy gave relatively bad results for the 'difficult' test scenes, which ask for long paths. This appears no longer to be true for directed graphs, where for all test scenes the edge requiring node adding strategy gives

(clearly) the best running times. For scenes 4 and 5 though, we see that the relative difference in performance between the edge requiring node adding strategy and the edge sensitive node adding strategy is much smaller than for the easier test scenes. This suggests that perhaps there exist difficult scenes where the edge sensitive (or normal) node adding strategy performs better than the edge requiring one.

Now why does the edge requiring node adding strategy perform better (with respect to the other node adding strategies) for directed underlying graphs than for undirected ones? Intuitively there is an easy explanation. The start and goal components in undirected underlying graphs are trees. It is therefore clear that the edges in those components can be used in only one direction. In the start component an edge can only be used (i.e. correspond to a motion) in the direction 'away' from the start node, and in the goal component an edge can only be used (i.e. correspond to a motion) in the direction 'towards' the goal node. An edge (a, b) though which is not in the start or goal component, can be used in both directions. I.e., in the final path it can both correspond to a motion from a to b, as well as to a motion from b to a. An edge (a, b) in a directed underlying graph can though always only be used in one direction. It can only correspond to a motion from a to b. So for both directed and undirected underlying graphs, when extending either of the two end components [1] with a set of k new edges, then k new possible motions between nodes in the graph are stored. When though extending some intermediate component with a set of k new edges, in a directed underlying graph this results in the storage of k new possible motions, while for undirected underlying graphs $2k$ new possible motions are stored. So it follows that relatively less gain is achieved by extending intermediate components (which is done by the normal and the edge sensitive node adding strategies) in directed underlying graphs than in undirected ones.

The test results clearly indicate that the ALA forward local method in combination with the edge requiring node adding strategy is the best combination, and, hence, this should be the default setting.

Again the forbidden configurations strategy appears to help for scene 7. The results of the main test for scene 7, performed with use of the forbidden configurations strategy, are as follows:

	normal	edge sensitive	edge requiring
ALA	24.3	17.8	8.3
ALA-LAL	19.6	12.5	**6.8**
ALA pot.field	57.4	49.5	25.0

The required number of nodes in the underlying graph is again typically quite small, not exceeding a few hundred.

[1] By end components in directed graphs we here refer to the forward set of the start configuration $forw(s)$, and the backward set of the goal configuration $backw(g)$.

Chapter 9

Some possible improvements

The two most (time) expensive 'operations' that are performed by (both versions of) our motion planner, are the distance computations, i.e., executions of the function D, and the path computations, i.e., executions of the function *compute_path*. Tests have indicated that practically all ($> 95\%$) computation time is consumed by these two operations. So the costs of for example all graph operations are neglectible.

We have therefore focussed on implementing these operations in an efficient way. E.g., a simple optimization with regard to the distance computations that we have performed, which is based on the observation that the ALA (forward) distance between two configurations c_1 and c_2 is always lower bounded by the Euclidean distance between c_1 and c_2, is the following: To reduce the number of (expensive) computations of ALA (forward) distances, we first compute the Euclidean distances from the new node c to all present nodes in the graph, and we discard those which lie further away than *maximaldist* from c (with respect to the Euclidean metric). Only for the remaining nodes do we compute the ALA (forward) distances.

We have looked at the ratio between the costs of the distance computations and those of the path computations, and the results where similar for both directed and undirected underlying graphs. We will therefore not discuss these results separately. By a ratio of $a : b$ we mean that $(\frac{a}{a+b} \cdot 100)\%$ of the total computation time is spent on distance computations, and the rest on path computations.

For easy scenes the ratio is about 1:1 when using the ALA-LAL (forward) local method, and about 1:3 when using the ALA (forward) potential field method. For more complex scenes, think of for example scene 8, the ratio appears to be quite different. When using the ALA-LAL (forward) local method, it is something like 4:1, and when using the ALA (forward) potential field method it is about 1:1.

Assume now that we are dealing with an undirected underlying graph, and (hence) that the used metric is the ALA one. The fact that the relative costs of the path computations are higher for a complex method like the ALA potential field method than for a simple one like the ALA-LAL method is to be expected. More interesting is the great increase of the 'distance computation costs' when the scenes get more complex.

The reason for this increase is the following: For every new node c that is added to the graph, the distances to all other nodes in the graph are computed in order to find those nodes which lie within distance *maximaldist* of c. So if the 'current' graph is (V, E), then the adding of a new node asks for $|V|$ distance computations, which means that

the total number of these computations is quadratic in the size of the final graph. Now because the individual distance computations are very cheap operations in comparison to the path computations, the total costs of the distance computations remain 'acceptable' for problems where the graph remains relatively small, say where the number of nodes does not exceed something like 100. If though the graph gets larger, the costs of distance computations 'explode'.

We will now discuss a number of possibilities for attacking this problem. We could try making the metric cheaper. Instead of computing the exact ALA distance between two configurations, we could compute some (cheaper) approximation of it. The drawback of course is that doing this results in 'worse' neighbor sets of new nodes. When using the exact ALA metric, we are sure that every node n that is picked as a neighbor of c, is the nearest node in n's connected component to c. Hence, the chance that a feasible path will successfully be computed from c to n is greater for n than for any other node in its connected component. When using some inexact metric, it is clear that often we will no longer choose the nearest nodes in each connected component. The result is that relatively less path computations (executions of *compute_path*) will succeed, with the consequence that typically more nodes in the graph and more path computations will be required for solving the motion planning problems. So we have a trade off.

We have done some experiments, where instead of the ALA metric the Euclidean metric was used. One can see the Euclidean metric as a very cheap but bad approximation of the ALA metric. For each of the eight test scenes, we have performed 100 test runs with the optimal parameter setting for the particular scene, i.e., with the setting for which the average running time is emphasized in the tables shown in section 8.3.2. For scene 1, where there are 4 different optimal settings, we use the ALA-LAL local method in combination with the edge sensitive node adding strategy, and for scene 7 the forbidden configurations strategy is applied. The results of the described tests are shown in the following table :

	ALA metric	Euclidean metric
scene 1	**0.4**	**0.4**
scene 2	**1.9**	2.2
scene 3	**0.5**	0.6
scene 4	**4.7**	6.0
scene 5	**3.2**	3.6
scene 6	**1.9**	2.4
scene 7	4.6	**3.2**
scene 8	**7.2**	9.6

So we see that for most scenes, using the Euclidean metric results in (slightly) higher running times. The only real exception is scene 7. In that scene, the running time very much depends on whether some 'lucky' nodes are added in the narrow passages. As long as this does not happen, the global method in general does not succeed in connecting the isolated free areas to each other, no matter what metric is used. At the moment that the necessary nodes have been added, the problem is very easy to solve, and it typically is solved 'instantaneously'.

Another approach (than using cheaper metrics) to decreasing the total costs of the distance computations is to decrease the number of distance computations performed for the computation of the neighbor set of a new node. In the current algorithm, we use the

set V of all nodes in the graph as the set of *candidate neighbors* (the set of nodes from which the real neighbors are picked) of a new node c. To decrease the total number of distance computations performed for a new node c, it is necessary to use some smaller set of candidate neighbors than V.

Now if the Euclidean metric is used, then we can use some 'smart' data structures for storing the nodes, like e.g. K-d trees. With such data structures it is possible to efficiently discard nodes which do not lie in the neighborhood of the new node c, without explicitly having to compute the distances between c and those distant nodes. The ALA metric though is a very 'strange' metric, and it seems very difficult to find such data structures for it.

An easier way to decrease the size of the candidate neighbor sets of new nodes, is to simply pick random subsets of V, of which the size is bounded by some constant k. In other words, instead of computing the distances to all present nodes, we compute distances to k randomly picked nodes from V, and for the nodes which have not been picked, we say the distance is ∞. Again, the quality of the neighbor sets will be a bit worse, resulting in more nodes and more path computations, but the total number of distance computations performed for the solving of a motion planning problem will now be linear (instead of squared) in the number of nodes in the final graph. So no matter how large the graph becomes, the costs of the distance computations will never 'explode'. A question of course is how to choose the constant k, and (again) it seems that the best approach here is to do experiments with different values for k. This though is a topic of further research.

For directed underlying graphs, the above ideas are applicable in a straight forward way. When adding a new node c to the directed graph (V, E), $2 \cdot |V|$ distance computations are performed. For every node in V its distance *to* c, and its distance *from* c are computed. So again the total number of distance computations required for solving a motion planning problem is squared in the number of nodes in the final graph, which leads to 'explosive' costs of the distance computations when the underlying graph gets 'too' big.

To attack this problem, one can again try to replace the ALA forward metric by some cheaper approximation of it. Like for undirected graphs, we have done some tests with using the Euclidean metric instead of the ALA forward metric. For all test scenes (with optimal parameter settings) the Euclidean metric gave worse results, even for scene 7, where it gave an improvement for undirected graphs. This did not really come as a surprise though, because the Euclidean metric is a very bad approximation for the ALA forward metric, considerably worse than for e.g. the ALA metric.

Another approach again is to decrease the number of distance computations performed for the adding of a new node to the graph, by decreasing the sizes of the candidate neighbor sets. Using some 'smart' data structures for storing the nodes would be nice, but just like the ALA metric, the ALA forward metric is a 'strange' metric, and perhaps a better approach is to pick random subsets of V of bounded size as candidate neighbor sets for new nodes.

Chapter 10

Conclusions and future work

In this paper we have described the random motion planning technique, and we have applied it to two types of car-like robots. For car-like robots which can move both forwards and backwards, the method uses an undirected underlying graph, and for such which can only move forwards, a directed graph is used.

The method, which has also proven to be extremely fast for free flying planar robots, achieves very good results for both mentioned car-like robot types, in many different types of scenes. Conceptually it is a simple (and therefore easy to implement) method, and the graphs that it builds are very small representations of the free configuration space.

A nice property furthermore is the great flexibility of the method. In order to apply the method to some particular robot type, all that is needed is a local method which computes feasible paths for this robot type, and some (induced) metric. When plugged into the global method, either the directed or the undirected version, depending on the robot type, this results in a (fast) motion planner for the given robot type. Experimental results in this paper, and in [Mas92] (concerning free flying planar robots), indicate that very primitive local methods achieve very good results. Hence, it should normally be no problem to find a good local method for some given robot type.

Another useful property of the random motion planner is that it uses a learning approach. Once a search in a particular scene has been done, the constructed 'random' network can be reused and extended for other searches in the same scene. So the running times of the method improve with every search that it performs in the same scene with the same robot. After a few searches the graph typically has almost complete knowledge about the structure of the free configuration space, and, hence, further motions are then computed instantaneously.

Currently we are working on the application of the method to a number of different robot types, such as articulated robots, solid robots in 3D, and robots with other than car-like nonholonomic constraints, like e.g. tractor-trailer robots. This can all be done in a rather straightforward manner, as sketched above.

Furthermore, we are planning extensions of the random motion planning method in various directions, aimed at solving more difficult motion planning problems, such as motion planning in scenes with multiple robots, moving obstacles, or perhaps some amount of uncertainty. Also we are considering some possibilities to use more information about the geometry of the workspace in order to achieve better graph extensions than the basically random ones achieved by the current method. Another interesting question, and topic of

research, is whether the method can be made deterministic.

Bibliography

[Gie93] G.-J. Giezeman. PlaGeo — a library for planar geometry. Technical report, Department of Computer Science, Utrecht University, April 1993.

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

[Mas92] P. Mastwijk. Motion planning using potential fields. Technical report, Department of Computer Science, Utrecht University, October 1992.

[Ove92] M.H. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Department of Computer Science, Utrecht University, October 1992.