

A Probabilistic Learning Approach to Motion Planning

M.H. Overmars and P. Švestka

UU-CS-1994-03

January



Utrecht University

Department of Computer Science

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

A Probabilistic Learning Approach to Motion Planning

M.H. Overmars and P. Švestka

Technical Report UU-CS-1994-03
January

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

ISSN: 0924-3275

A Probabilistic Learning Approach to Motion Planning*

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

Abstract

In this paper a new paradigm for robot motion planning is proposed. We split the motion planning process into two phases: the learning phase and the query phase. In the learning phase we construct a probabilistic roadmap in configuration space. This roadmap is a graph where nodes correspond to randomly chosen configurations in free space and edges correspond to simple collision-free motions between the nodes. These simple motions are computed using a fast local method. The longer we learn, the denser the roadmap becomes and the better it is for motion planning. In the query phase we can use this roadmap to find paths between different pairs of configurations. If a possible path is not found one can always extend the roadmap by learning further. This gives a very flexible scheme in which learning time and success for queries can be balanced.

We will demonstrate the power of the paradigm by applying it to various instances of motion planning : free flying planar robots, planar articulated robots and car-like robots (with non-holonomic constraints). We expect it to be applicable in many other instances as well.

1 Introduction

The *motion planning problem* is well-known in the field of robotics. It asks for computing feasible paths for a given robot \mathcal{A} in a workspace containing some obstacles. Two versions of the problem can be formulated. In one version, a start configuration s and a goal configuration g are given before hand, and the objective is to compute a feasible path for \mathcal{A} from s to g . In the second version, no start and goal configurations are specified, and the objective is to compute a data-structure, which can

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

later be used for queries with arbitrary start and goal configurations. We refer to the former case as a *single shot* problem, and to the latter as a *learning* problem.

The ‘classical’ approaches to motion planning can roughly be divided in the following three classes: *roadmap methods*, *cell decomposition methods*, and *potential field methods*. For a thorough discussion of these approaches see e.g. [Lat91] and [HA92].

Let \mathcal{C} denote the space of all configurations for the robot, and let \mathcal{C}_f be the robots free configuration space, i.e., the subset of \mathcal{C} where the robot does not intersect any obstacles. The roadmap approach (or *skeleton approach*) consists of capturing the connectivity of \mathcal{C}_f in the form of a network of one dimensional curves - the *roadmap* - lying in \mathcal{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 \mathcal{C}_f into a collection of non-overlapping regions (cells), whose union is (exactly or approximately) \mathcal{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).

We see that both the roadmap approach as well as the cell decomposition approach consist of constructing a global data structure that can later be used for solving one or more motion planning problems. This means that both approaches are suitable for learning problems (as well as single shot problems). Another strong point is that cell-decomposition and roadmap algorithms are typically complete, i.e., whenever a path exists a path will be found. Drawbacks though are that (1) the computations of the data structures tend to be very expensive in both time and memory, and (2) they do not seem to be suitable for robots with non-holonomic constraints, like for example car-like robots or multi-body mobile robots.

In the potential field approach no data structure is built. 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 \mathcal{C}_f . The potential field is typically defined by a function $\mathcal{C} \rightarrow \mathbb{R}$ 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.

A typical 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. Recently though much progress has been made in defining good potential functions with few

local minima, and efficient techniques have been developed for escaping from local minima. Currently there exist practical potential field planners for robots with many degrees of freedom, as well as for some types of non-holonomic robots (see for example [BL91]). So it seems that the potential field approach does not have the disadvantages of the two former approaches. A major drawback of the potential field approach though is that the whole concept is unsuitable for learning problems, due to the fact that every goal configuration defines a distinct potential field.

In this paper we describe a new paradigm to the learning motion planning problem, by combining a global roadmap approach with a local planner. In the learning phase, a probabilistic roadmap is built up by repeatedly generating random free configurations and trying to connect these to other (earlier added) configurations by some simple motion planning algorithm. The network thus formed is stored in a graph G . The configurations are stored as nodes in G , and the links, which are paths in free configuration space, are stored as edges in G . After the learning is done, a query consists of trying to connect the given start and goal configurations to some nodes (in the same connected component) of G , with paths which are feasible for the robot. Next, the path between these nodes in G can be transformed into a feasible path in configuration space.

We claim that our paradigm is a very powerful one, and to support this claim, we apply it to a number of different robots : free flying robots, articulated robots, and car-like robots. For each robot type and for a number of different scenes, we test how much learning time is required to be able to solve a certain percentage of 'all' queries in the scene. It turns out that often only very little time (in the order of seconds) is required to solve a percentage of nearly 100%.

The learning phase of this approach is closely related to earlier work that we have done on single shot planners. In [Ove92] a probabilistic single shot planner for free flying planar robots is described in detail, and [Šve93] deals with two types of car-like robots, i.e., normal ones, and cars which can only move forwards. In both papers we gave a lot of experimental results, some of which have guided certain choices made in the learning approach described in this paper. Other related work has independently been done by L.Kavraki and J.-C.Latombe. In [KL93] they describe a probabilistic method for configuration space preprocessing, which also builds up a probabilistic road map that, in a query phase, can be used for motion planning. They deal with articulated robots of high degree of freedom, with links that are line segments. Their method though seems to be more restricted than ours.

This paper is organized in the following way: In section 2 the learning algorithm is described in general terms. In section 3 the query phase is described. In sections 4, 5 and 6 we apply the paradigm to, respectively, free flying planar robots, car-like robots, and (planar) articulated robots. For each of these robot types we fill in the details and give experimental results. Finally, in section 7, we draw some conclusions.

2 The learning paradigm

The learning phase can be described in general terms, without focussing on any specific robot type. The idea is that the data structure built up during the learning phase later (in the query phase) is used to solve individual motion planning problems.

The learning algorithm is a *two-level* approach, consisting of a *global method* and a *local method*. The local method is a (primitive) motion planner, which tries to compute (simple) feasible¹ paths connecting two given free 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 uses the local method to build the mentioned probabilistic roadmap. The basic algorithm is extremely simple. It builds up an undirected graph $G = (V, E)$, by repeatedly generating a random free configuration c , adding c to V , computing a set $N(c) \subset V$ (c 's *neighbors*), and adding an edge (c, n) to E for every $n \in N(c)$ to which the local method can connect from c .

Assume now that we are dealing with a robot \mathcal{A} , and that L is a local method for \mathcal{A} . To describe the global method formally, we need the following :

- A *symmetrical* function $L_d \in \mathcal{C} \times \mathcal{C} \rightarrow \text{boolean}$, that returns whether the local method can compute a feasible path for \mathcal{A} between its two argument-configurations. We refer to this function as L 's decision function.
- A function $D \in \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{R}^+$. It defines the metric² used, and should give a suitable notion of distance for arbitrary pairs of configurations, taking the properties of the robot \mathcal{A} into account. We assume that D is symmetrical.

The algorithm can now be described as follows:

The global method:

```

V = ∅
E = ∅
loop
  c = A randomly chosen free configuration.
  V = V ∪ {c}
  N(c) = A set of neighbors of c, chosen from V.
  E = E ∪ {(c, a) | a ∈ N(c) ∧ Ld(c, a)}
```

When the learning is done, queries can be performed. Given a start configuration s and goal configuration g , we try to connect s and g to nodes \tilde{s} and \tilde{g} in the (same connected component of) the graph. If this succeeds, we compute the shortest path

¹A path P is feasible for a robot \mathcal{A} iff P lies in free configuration space, and the motion described by P is achievable by \mathcal{A} , i.e. it respects \mathcal{A} 's constraints.

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

in the graph from \tilde{s} to \tilde{g} . For each edge on this path we (re)construct a feasible path in configuration space using the local planner. In this way we obtain a feasible path connecting s to g . If \tilde{s} and \tilde{g} cannot be found, the query fails. We will go into some more details about the query phase in section 3.

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

Local method

One of the crucial ingredients in the learning phase is the local method. As mentioned before, the local method must compute paths which are *feasible* for \mathcal{A} . The reason for this is that the global paths computed in the query phase are basically just concatenations of local paths (i.e., paths computed by the local method), so clearly in order for the global paths to be feasible, the local paths must be as well. Furthermore, the local method should be deterministic. Otherwise the existence of a path in G between two nodes a and b does not guarantee that a feasible path in configuration space connecting a and b can be reconstructed in the query phase. A final requirement is that the local method always terminates (some potential field methods do not have this property).

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

Clearly, the choice of the local method should be aimed at maximizing the amount of ‘knowledge’ acquired by the global method per time unit. E.g., if we define the amount of ‘knowledge’ stored in a graph (V, E) as $|E|$, then we should search for a local method with $\frac{\text{chance of success}}{\text{average running time}}$ as high as possible. This is though a rather vague criterion, and we have guided the choice of our local methods by experiments. These clearly indicated that very fast (and, hence, not very powerful) local methods lead to the best performance of the global method.

Neighbors and edge adding methods

Another important choice to be made is that of the neighbors $N(c)$ of a (new) node c . As is the case for the choice of the local method, the choice of $N(c)$ has large impact on the performance of the global method. Reasons for this are that the choice of the node neighbors strongly influences the overall structure of the graph, and that, regardless of how the local method is exactly chosen, the executions of the local method are by far the most time-consuming operations of the global method. This is caused by 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 it is clear that ‘useless’ executions of the local method should be avoided as much as possible. To start with, an execution of the local method which fails is useless, in the sense that it does not extend the ‘knowledge’ stored in the graph. To prevent too many failures of the local method, we only consider nearby nodes (with respect to the metric D), i.e. nodes within distance $maxdist$ of the new node (where $maxdist$ is some well chosen real constant). Thus

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

Now what is the value of successful executions of the local method? This depends on what kind of paths we want to get during the query phase.

One possibility is that we do not care about what these paths look like, as long as many queries succeed. In this case adding any edge (a, b) which creates a cycle in the graph is worthless, because a and b were already connected, and, hence, no query can ever succeed *thanks to* (a, b) . This suggests that we try to connect c at most once to each connected component in the graph, which, in combination with (1), leads to the following definition :

Definition 1

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

So in every connected component of G , the nearest node to c is a neighbor of c , under the condition that $D(c, n) \leq maxdist$. We refer to the edge adding method which results from this definition as the *forest* method, because it leads to a graph that is a forest.

Now suppose that we prefer short paths above arbitrary ones in the query phase. The forest method is not suitable for this purpose. An edge in a cycle is no longer necessarily worthless, because removing it may, in the query phase, result in much longer paths. So we should allow cycles now. One possibility is to just pick the k nearest nodes as the neighbors. We refer to the edge adding method resulting from this choice as the *nearest- k* method. Using this method (with $k \gtrsim 4$) one obtains a very dense graph and near-optimal paths in the query phase, but unfortunately it is quite expensive.

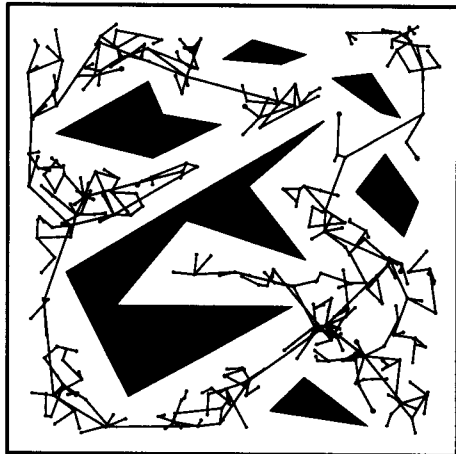


Figure 1: A graph obtained with the forest method.

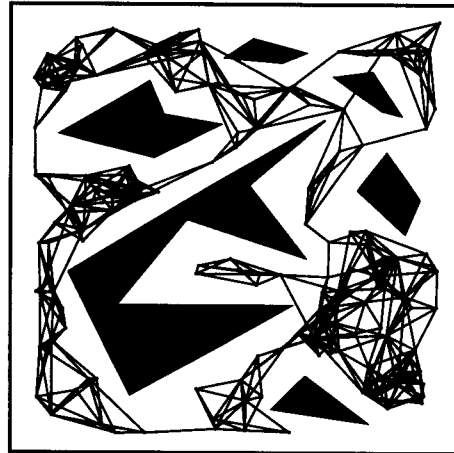


Figure 2: A graph obtained with the nearest-4 method.

A way to obtain reasonable paths at relatively low expense is what we refer to as the *heuristic loops* method. The idea is that, when a new node c has been added, first edges are added using the forest definition. After this is done, one or more nodes in c 's connected component are picked as extra neighbors, in order to obtain some loops containing c . These extra neighbor(s) should be chosen using some heuristics, aimed at reducing (some) path lengths in c 's connected component. We have decided on picking as (only) extra neighbor the node $n \in V$ which minimizes the ratio between $D(c, n)$ and the length of the current shortest path in G from node c to node n .

See figures 1, 2, and 3 for examples of graphs obtained with, respectively, the forest method, the nearest-4 method, and the heuristic loops method, all in the same scene with a rectangular free flying robot. As can be seen, the heuristic loops method captures the topology of the free space in the best way, which results in short paths. (Note that in the figures some edges go through obstacles. This is not an error though. The reason is that we display an edge between two nodes as a straight line segment, while the actual path corresponding to the edge might be different.)

Distance

We have seen that the distance function D is used for choosing the neighbors $N(c)$ of a new node c . For example in the forest method, from every connected component in the neighborhood of the new node c , that node is picked which is nearest to c with respect to the 'metric' D . By picking the nearest node from a component C , we want to maximize the chance of successfully connecting from c to C . But this

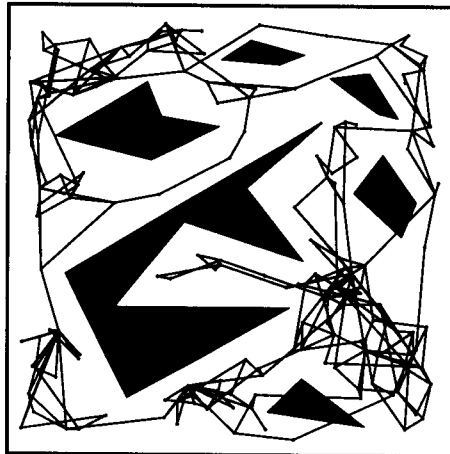


Figure 3: A graph obtained with the heuristic loops method.

means that D should be defined in a way, such that $D(a, b)$ (for arbitrary a and b) somehow reflects the chance that the local method will *fail* to compute a feasible path from a to b . For this we define the distance between two configurations a and b as the *length*³ of the path from a to b computed by the local method in absence of obstacles. In this way any local method induces its own metric.

Adaptive node adding

In the algorithm described above the nodes are added in a fully random way. Experiments indicate that this random node adding strategy performs quite well, e.g. better than adding nodes in some regular pattern. It is though possible to use some heuristics during the node generation, such that more nodes are added in certain ‘interesting’ areas of free configuration space than in others.

For example, areas where two or more connected component of G ‘overlap’ are interesting, because adding nodes in those areas is likely to result in the ‘merging’ of those components. The *adaptive adding heuristic* is aimed at this. When a new configuration c has been (randomly) generated, it is not added to the graph immediately, but instead the *adding chance* of the configuration is computed. If there are no nodes very near to c , or if there are two or more different connected components very near to c , then c ’s adding chance is high (e.g. 1). Otherwise it is low (e.g. 0.05). After this adding chance has been computed, c is added with this probability, and otherwise it is discarded.

³For car-like robots we define the length of a path $P \in [0, 1] \rightarrow \mathcal{C}$ to be the length of P ’s projection in \mathbb{R}^2 . For free flying robots and articulated robots we take the actual length of the path in configuration space, but scaled appropriately along the angular axis, taking into account the robots geometry.

The underlying idea is that if two or more different connected components lie very near to c , then adding c is likely to connect these components. Hence c should be added. If there are no nodes very near to c , then clearly not enough learning has yet been done in c 's near environment, and hence c should be also added. In the case that c has exactly one connected component very near (which typically is the case most often), then c 's near environment is already explored (to some extent), and c is not very likely to connect some different connected components, so probably c is not a very valuable node. Hence its adding chance is low. So adaptive adding adds nodes mainly in 'empty' areas of \mathcal{C}_f , and in 'overlapping' areas of different connected components.

3 Queries

During the query phase, paths are to be found between arbitrary start and goal configurations, using the graph computed in the learning phase. The idea is that, given a start configuration s and goal configuration g , we try to connect s and g to nodes \tilde{s} and \tilde{g} of the same connected component in G , with some feasible paths P_s and P_g . If this succeeds, then we compute the shortest path P_G in G connecting \tilde{s} to \tilde{g} , and a feasible path in configuration space from s to g is constructed by concatenating P_s , the subpaths computed by the local method when applied to pairs of consecutive nodes in P_G , and P_g . Otherwise the query fails.

The main question is how to compute the paths P_s and P_g . The queries should preferably terminate 'instantaneously', so no expensive algorithm is allowed here. One possibility is to use the local method. We have decided on a simple randomized planner, which performs a Brownian-like motion in configuration space for a short period of time (e.g. 0.25 seconds), and during this motion at regular intervals tries to connect to the graph using the local method. For free flying robots and articulated robots, the Brownian-like motion consists of repeatedly picking a random direction in configuration space, and moving in this direction until an obstacle is collided with, or the time runs out. When a collision occurs, a new random direction is chosen. So the paths computed by this planner are sequences of path segments which are straight in configuration space, followed by a path segment computed by the local method. For car-like robots the Brownian-like motion is generated by repeatedly picking random values in control space⁴. (i.e., random values for the steering angle and the velocity of the robot), and performing the motion thus defined until a collision occurs. Experiments show that this Brownian-like method works fine, e.g., better than just using the local method.

⁴Instead of the actual control space $[-\phi_{max}, \phi_{max}] \times [-v_{max}, v_{max}]$ we use the discretisation $\{-\phi_{max}, \phi_{max}\} \times \{-1, 1\}$, where ϕ_{max} is the robots maximal steering angle, and v_{max} its maximal velocity

3.1 Smoothing

Paths computed in the query phase can, especially when the graph has been built using the forest edge adding method, be quite ugly and unnecessarily long.

To improve this, one can apply some path smoothing techniques on these ‘ugly’ paths. The smoothing routine that we implemented is very simple. It just repeatedly picks a pair of random configurations (c_1, c_2) on the ‘to be smoothed’ path P_C , tries to connect these with a feasible path Q_{new} using the local method, and if this is successfully accomplished and Q_{new} is shorter than the path segment Q_{old} in P_C from c_1 to c_2 , then it replaces Q_{old} by Q_{new} (in P_C). So basically it just tries to replace randomly picked segments of the path by shorter ones, using the local method. The longer this is done, the shorter (and nicer) the path gets. Typically, this method smoothes a path very well in just a few seconds.

Still one can argue that a few seconds is too much for a query. In that case one should either accept the ugly paths, or use a more expensive edge adding method, like for example the nearest-k method or the heuristic loops method. The gain obtained by these edge adding methods (in terms of shorter paths in the query phase) is hard to measure accurately, and is very dependent of the scenes that we are dealing with. Experiments though show that for many scenes it is significant. In some cases the path lengths obtained with e.g. the nearest-4 edge adding method are, on the average, only about half of those obtained with the forest method.

4 Free flying planar robots

As a first example of the use of the general paradigm introduced in section 2 we apply it to free flying planar robots, i.e., the robot is a polygon that can rotate and translate freely in the plane among a set of polygonal obstacles. This setting of the motion planning problem is rather simple because the robot has only three degrees of freedom and no non-holonomic constraints. It has been studied in great detail in the past and many methods have been proposed to deal with it. Still the problem is far from trivial. We first fill in a few details of the method, and then give some experimental results.

4.1 Details of the method

To use the global paradigm we need a suitable local method, which we describe below. For the distance between configurations the induced metric will be used. Also we will introduce a non-random node adding strategy, which makes use of the geometry of the workspace, and can be used in combination with the random adding strategy. This strategy is only applicable to planar solid robots, such as free flying and car-like ones.

Local method

For free flying robots we have done a large number of experiments with different types of local methods, and, based on the results, we have decided on the following method that can be regarded as a very simple potential field method: Let ϵ be some small step size. The method will let the robot take steps of this size. To avoid collisions during a step we blow up the robot with a factor ϵ . (In fact we use different step sizes for the translation and rotation of the robot. The rotation step size is chosen automatically such that during a rotation step no part of the robot moves more than a distance of ϵ .) In pseudo C-code the method looks as follows:

ALGORITHM LOCAL METHOD

1. *config* = *source*;
2. **loop**
3. **if** (*config* == *goal*) **return** goal reached;
4. *newconf* = *config* plus step of size ϵ towards *goal*;
5. **if** *newconf* in free space
6. *config* = *newconf*;
7. **else**
8. **for** each of the 26 direct neighbors of *config*
9. determine distance to goal;
10. Sort neighbors by distance in a list *neighbor[]*;
11. **for** (*i*=0; *i*<13; *i*++)
12. **if** *neighbor*[*i*] in free space
13. *config* = *neighbor*[*i*]; **break**;
14. **if** (*i* == 13) **return** goal not reached;

The algorithm simply loops until the goal is reached (line 3) or no progress can be made (line 14). In each loop first we try to make a step in the direction of the goal (line 4-6). If we fail all 26 direct neighbors of the current configuration (by taking an ϵ -step in the x- y- and θ -direction) are considered. We treat them in order of their progress towards the goal (i.e. their distance to the goal) by computing these distances (line 8-9) and sorting them (line 10). Note that only the first 13 of these configurations are better than the current configuration. Hence, we only look at these 13 neighbors and take the first one that is in free space as the new configuration (line 11-13). When none of the neighbors is possible we conclude that no path could be found (line 14). (In the actual implementation the order of the steps is changed to obtain some slight speed-up.)

One can view this method as a potential field method. Here the attracting potential created by the goal is simply inversely proportional to the distance. The repulsive potential is infinite when the robot gets nearer than ϵ to an obstacle and is 0 otherwise. The motion computed this way can turn around corners but often leads to a local minimum where no progress can be made.

The big advantage of the approach is that it is very fast and general. The only basic test we need is whether the (slightly blown-up) robot intersects any of the obstacles. Such a test can be performed very fast after some preprocessing of robot and obstacles; much faster than the distance computations that are normally required for other potential field methods. Note that, when the robot is not near to an obstacle, the step towards the goal will succeed and, hence, only one of the basic tests is performed. This makes things even faster.

Geometric node adding

Many practical scenes have the form of corridors and rooms. In such cases one can boost the planners performance by adding configurations at important positions, in particular along edges of obstacles (i.e., along walls) and next to vertices of obstacles (to facilitate the robot to move around corners). This is implemented in the following way: For the robot we determine the geometric axis, i.e., the axis such that the width of the robot becomes minimal (with respect to the axis). For each edge and vertex of the obstacles (in random order) we determine the outer normal. Next we place the robot with its geometric axis perpendicular to the normal along the edge or convex vertex at such a distance that the robot has some small clearance with respect to the edge or vertex. See figure 4 for an example. If this is a free configuration it is added to the graph, otherwise it is discarded. Often geometric adding captures most of the connectivity of the free configuration space. See figure 5 for an example of a graph obtained after geometric adding (and the nearest-4 edge adding method). Once all edges and convex vertices have been treated, the geometric adding stops. In order for the learning process to continue at this point, the geometric adding should be used in combination with random adding. This means that after all the geometrically obtained nodes have been used, we continue by adding random nodes.

Experimental results

We now present a number of experimental results for free flying planar robots, obtained by applying our learning approach to a number of different scenes. The method is implemented in C++ and the tests were performed on a Silicon Graphics indigo workstation with an R3000 processor running at 33 MHZ. This machine is rated on the SPECMARKS benchmark with 24.2 SPECfp92 and 22.4 SPECint92.

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

We say the programs 'knowledge' of a scene S is $k\%$, if $k\%$ of 'all' solvable queries in scene S are solved successfully (and $(100-k)\%$ are not). Given a particular scene, we want to measure the 'knowledge' that the program acquired about the scene

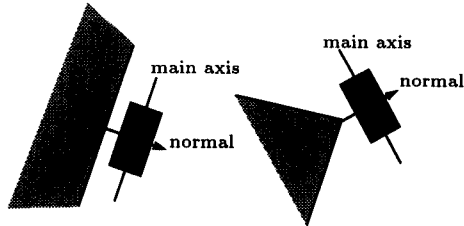


Figure 4: Adding a rectangular robot along an edge or vertex.

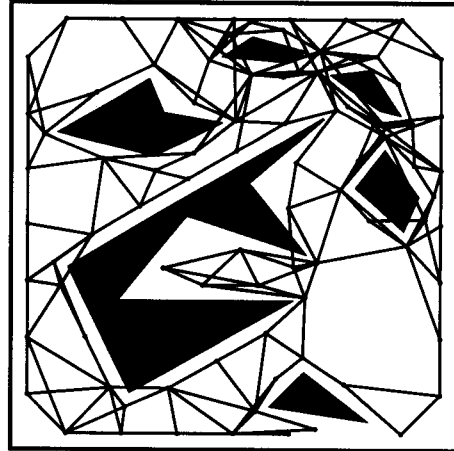


Figure 5: A graph obtained after geometric adding, for a rectangular robot.

after having learned for a certain amount of time. It is not possible to measure this ‘knowledge’ exactly, but we approximate it in the following probabilistic way : First we generate a (large) set $V_c \subset \mathcal{C}_f$ of randomly picked free configurations. Then we generate a (large) set $V_p \subset V_c \times V_c$ of free random configuration pairs, but we discard all pairs (a, b) where a lies in a different connected component of \mathcal{C}_f than b . Finally, we count for how many pairs $(a, b) \in V_p$ the query with arguments a and b succeeds. If s is the number of successful queries, then $\frac{s}{|V_p|} \cdot 100\%$ is our approximation of the programs knowledge. One can expect this approximation to be a good one, if V_p is taken sufficiently large (we use 1000). Instead of giving arrays of numbers, we show some figures where the programs knowledge is set against the learning time. Clearly, the knowledge as defined above not only depends on the graph, but also on the time we allow for answering a query. Remember that during a query we perform a Brownian-like motion for some given amount of time, to connect the start and goal configurations to the graph. The more time we allow for this Brownian-like motion, the higher is the chance that the query will succeed. In our tests we kept the maximal query time very low, i.e., at 0.3 seconds.

In section 2 a number of different edge adding methods have been described, as well as two node adding heuristics. Furthermore, in section 4.1 a geometrically based non-random node adding method has been introduced. So there are many different settings possible. We give results for the tree edge adding method (with $maxdist=0.25$) combined with random node adding, adaptive random node adding, and adaptive geometric node adding. With adaptive random node adding we mean the random node adding method combined with the adaptive adding heuristic, and with geometric node adding we mean the geometric node adding method, followed

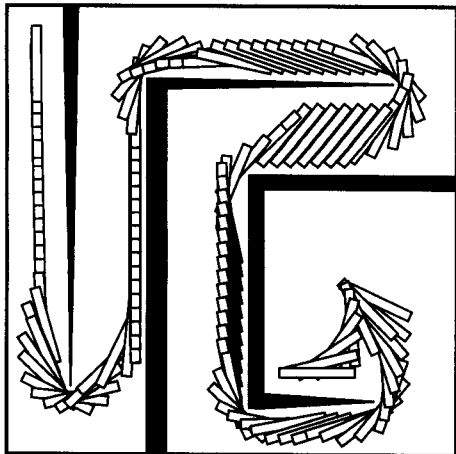


Figure 6: A free flying robot in scene 1.

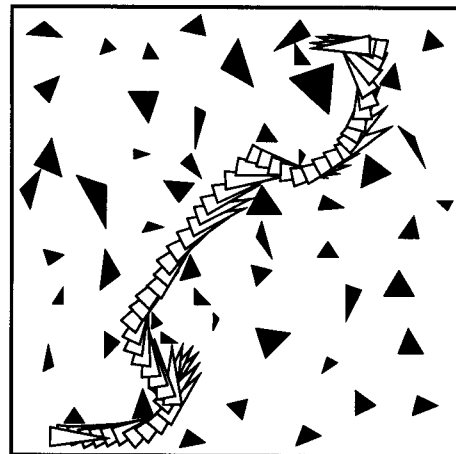


Figure 7: A free flying robot in scene 2.

by adaptive random node adding (when all geometric nodes have been added). So we do not give results for any of the described edge adding methods which allow loops in the graph. These typically slow down the learning process a bit. Experiments that we have performed indicated that the learning process slows down by a factor of approximately 2 when the heuristic loops method is applied, and by a factor of about 3 when the nearest-4 method is used. In return though, as mentioned before, these edge adding methods build graphs which give shorter paths in the query phase than the graphs obtained with the forest method, reducing the average path lengths by up to 50%.

The test scenes

We tested the method on the four scenes shown in figures 6 to 9. In each scene a (smoothed) path computed by our planner is indicated by a number of steps. Scene 1 is a relatively easy scene, consisting of one long corridor which requires the long robot to make some sharp curves. In scene 2 we have a triangular robot amongst a large number of small triangular obstacles. Most motion planning problems in this scene are easy, but there also exist many narrow areas in free configuration space, corresponding to areas in workspace where the robot is tightly surrounded by three or more obstacles. If either the start or the goal configuration of a query is positioned in such an area, then this query is far from trivial. The difficulty in scene 3 is the long and thin non-convex robot. Scene 4 is the most difficult of the four scenes. It contains many narrow passages, it gives the robot little freedom of movement, and most queries can only be solved by relatively long paths.

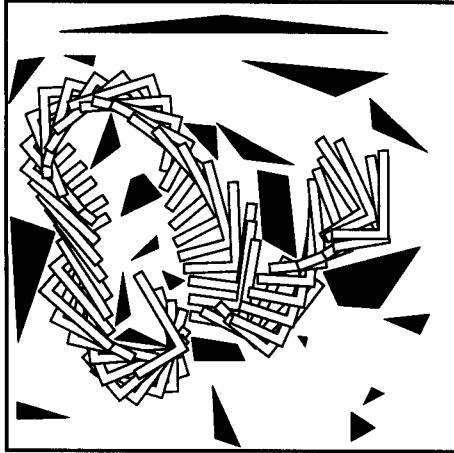


Figure 8: A free flying robot in scene 3.

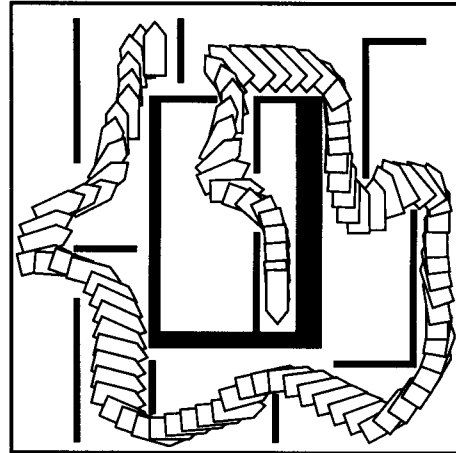


Figure 9: A free flying robot in scene 4.

The test results

Figures 10 to 13 show the amount of knowledge $K(t)$ that the planner has acquired after having learned for a certain time t . The different curves in each figure correspond to different node adding strategies. The dashed ones correspond to random adding, the dotted ones to adaptive random adding, and the solid ones to adaptive geometric adding. The edges are added with the tree method.

Figure 10 shows that the learning problem for scene 1 is solved for 100% within a few seconds. When random adding is used, this takes about 8 seconds. Adaptive random adding gives a significant improvement if the learning time exceeds about 3 seconds, giving 100% knowledge in just over 5 seconds. The relatively bad performance for short learning times is caused by the fact that adaptive adding only begins to pay off when a number of large components are present, which, in their ‘overlapping’ areas, define (not too many) ‘promising’ areas in configuration space. Initially the graph has no structure, and adaptive adding only slows down the node generation. We see that adaptive geometric adding gives by far the best performance for all learning times. In a scene with many long and straight corridors, like scene 1, the graph obtained by adding the geometric nodes typically captures most of \mathcal{C}_f ’s connectivity by itself, not requiring many random nodes to be added. In scene 1 the geometric learning phase (the phase where the geometric nodes are added) takes about $1\frac{1}{2}$ seconds, and the knowledge acquired during this phase is about 75%. Note that the initial knowledge, i.e. the knowledge present after 0 seconds of learning, is already about 22%. This is caused by the fact that our Brownian-like planner, which is used for computing connections between the query configurations and the graph, can solve some easy problems in scene 1 by itself.

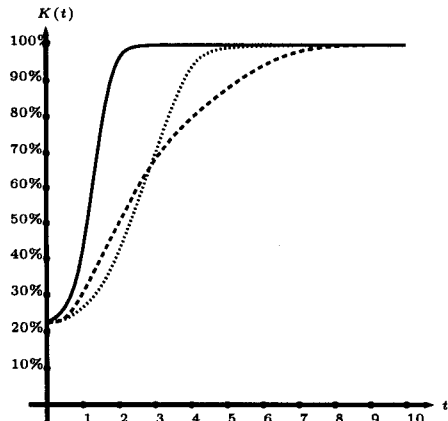


Figure 10: Learning in scene 1.

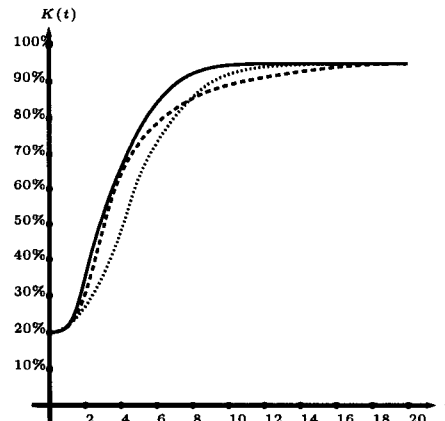


Figure 11: Learning in scene 2.

The plot in figure 11 shows a similar structure as the one in figure 10. Adaptive geometric adding is the best adding method, giving about 95% knowledge in 10 seconds, and the adaptive random method is better than the normal random method if the learning time exceeds some bound, which is about 8 seconds for scene 2. The difference in performance between the geometric adding method and the two random ones is quite small in comparison to scene 1. This is due to the rather chaotic and unstructured character of the scene. Another difference with scene 1 is that the knowledge converges much slower towards 100%. The presence of some very narrow areas in \mathcal{C}_f causes this.

The learning problem in scene 3 is solved for almost 100% in about 12 seconds. We see that geometric adding does not help here. Apparently, the non-convex shape of the robot causes the required motions along obstacle edges to intersect the obstacles too often. Again adaptive adding helps considerably.

The last scene is again one with many straight corridors, and, hence, geometric adding is useful. The adaptive geometric adding strategy solves the learning problem in about 10 seconds. We see that in this relatively difficult scene, which, hence, requires a large graph, adaptive adding really helps a lot. With normal random node adding, it takes more than a minute to solve the problem (for 100%).

The graph in which the computed motions are stored is typically quite small, and, hence, cheap to store. In scene 1 the graph grows with about 80 nodes per second, in scenes 3 and 4 with about 40 nodes per second, and in scene 2 (where the large number of small obstacles causes the intersection tests to be relatively expensive) with about 30 nodes per second. With each of the edge adding methods described in section 2, the number of edges in the graph is linear in the number of nodes. So, if the program has learned for a time t in one of the test scenes, the graph size is something between $30 \cdot t$ and $80 \cdot t$.

As stated before, we kept the query time under 0.3 seconds. If we increase this

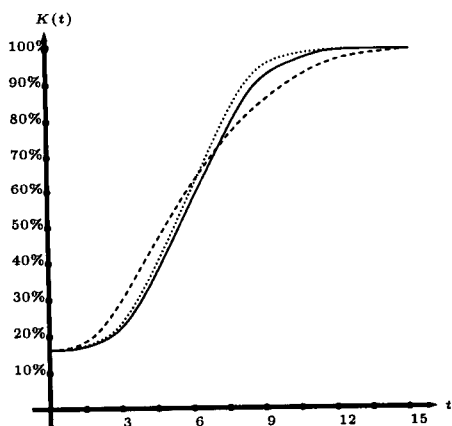


Figure 12: Learning in scene 3.

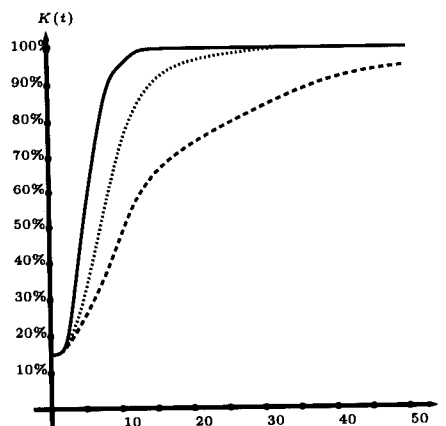


Figure 13: Learning in scene 4.

time, the performance of the method improves. E.g., in scene 2, when we allow 2.0 seconds per query, the knowledge acquired after having learned for time t is about equal to the knowledge acquired after having learned for time $2t$ with 0.3 as maximal query time. So the learning goes twice as fast.

5 Car-like robots

In this section we apply the learning paradigm to car-like robots. As free flying planar robots, car-like robots have three degrees of freedom, but their motions are subject to certain *non-holonomic constraints*, i.e. constraints on their achievable velocities, which makes motion planning considerably more difficult.

Although the motion planning problem for car-like robots has received considerable attention in the past years, there still exist only a few practical (single shot) planners for these robots (see [LTJ90], [LJTMar], [BL93]). It turns out that the learning paradigm can easily deal with nonholonomic constraints. We will briefly indicate the results here. For a more extensive description we refer to [Šve93].

We model a car-like robot \mathcal{A} as a polygon that moves in the plane. It has fixed to itself a *reference point* $R_{\mathcal{A}}$ and line $A_{\mathcal{A}}$, referred to as \mathcal{A} 's *main axis*, and also it has defined a *minimal turning radius* $r_{\mathcal{A}} \in \mathbb{R}^+$. Furthermore, we refer to the line which is perpendicular to $A_{\mathcal{A}}$ and goes through $R_{\mathcal{A}}$, as \mathcal{A} 's *minor axis*. \mathcal{A} can perform certain translations and rotations in the plane, which can be defined in terms of the above. The only translations that \mathcal{A} can perform are translations along $A_{\mathcal{A}}$, and the only rotations that it can perform are such where the center rotation lies on \mathcal{A} 's minor axis, at a distance of at least $r_{\mathcal{A}}$ of $R_{\mathcal{A}}$. See also figure 14.

In real life, car-like robots can perform more complex motions also, but it has been shown that if a feasible motion between two configurations exists for a car-like

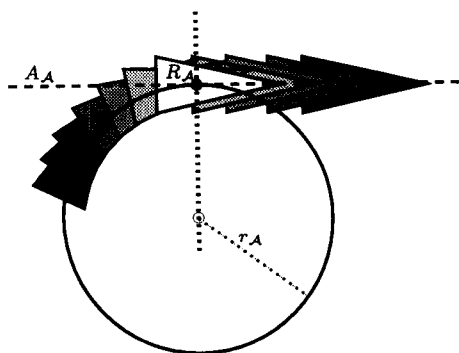


Figure 14: Some simple car-like motions.

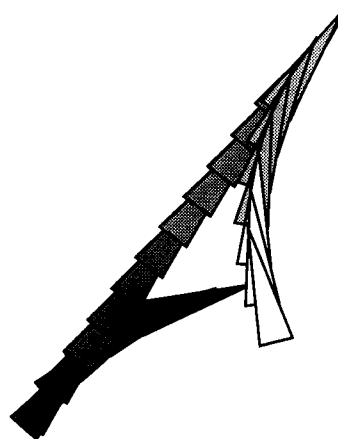


Figure 15: A RTR-path.

robot, then there also exists one which is a sequence of the described rotations and translations.

For applying our paradigm to car-like robots, we need a (good) local method that computes feasible paths for car-like robots.

Local method

As for free flying planar robots, we have done many experiments with different types of car-like local methods. In particular, we have experimented with some simple potential field methods, conceptually similar to the local method for free flying robots. A problem with car-like robots though is, that locally they can only move approximately along their main axis, which makes it hard to 'slide' them along obstacle boundaries. So potential fields should be used which keep them at some distance from the obstacles, but such potential fields turn out to be much too expensive to evaluate for our purposes. As a result, we have decided on a local method which is even simpler than the one for free flying robots.

We refer to a path describing a rotational motion of \mathcal{A} as a *rotational path*, and one describing a translational motion as a *translational path*. If a rotational path describes a rotation of radius $r_{\mathcal{A}}$, then we refer to it as a *maximally curved rotational path*. A *RTR-path* is now defined as the concatenation of a maximally curved rotational path, a translational path, and another maximally curved rotational path (see also figure 15). Analog, a *TRT-path* is defined as being the concatenation of a translational path, a maximally curved rotational path, and another translational path. With these two path constructs we define our local method: Given two argument configurations a and b , if either the shortest RTR-path connecting a to b

or the shortest TRT-path connecting a to b intersects⁵ no obstacles, one of these two paths is returned (depending on which one is free), and otherwise failure is returned. The RTR-path is always tested first (it is most of the time a bit shorter, and, hence, has less chance of intersecting obstacles), which means that the induced metric is the ‘RTR-metric’, i.e., the distance between a and b is defined as the length of the shortest RTR-path connecting a to b . See [Šve93] for more details on the exact shapes and computation of TRT- and RTR-paths.

Experimental results

We give experimental results for scene 1, scene 2, and scene 4 (see section 4). Figure 16 shows a (smoothed) path for a car-like robot, computed by our planner in scene 4. So we skip scene 3. This scene is not very ‘realistic’ if the robot has car-like constraints.

The geometric node adding strategy is applicable to car-like robots, but instead of choosing the geometric axis such that it minimizes the robots width, one should take A_A as geometric axis. In this way the geometric nodes correspond to configurations of the robot where it can move ‘parallel’ along the obstacle boundaries.

See figures 17 to 19 for plots that again show the amount of knowledge $K(t)$ about the corresponding scenes acquired by our planner after having learning for some time t . Again the edges are added with the tree method. The dashed curves correspond to random adding, the dotted ones to adaptive random adding, and the solid ones to adaptive geometric adding. The experiments have been performed with $maxdist=0.5$ and $r_A=0.1$.

The plot in figure 17 shows the same global structure as the plot for free flying robots (see figure 10). So also for car-like robots does adaptive geometric adding give the best performance in scene 1, followed by adaptive random adding. There are two main differences with the results for free flying robots. Firstly, the learning process is about three times slower, and secondly, although the knowledge is almost complete in about 10 seconds (approximately 95% with adaptive geometric adding), it then converges only very slowly towards 100%. E.g., it takes more than a minute to reach 98%. The cause for this slow convergence is that, although all queries performed in the tests are solvable (only configuration pairs in same connected components of C_f are generated), some queries can be very hard to solve for a robot with car-like constraints. This is the typically the case when either the start configuration or the goal configuration of the query is such, that the robot positioned at that configuration faces obstacles very near in front of itself *and* behind itself, thus requiring the robot to perform a large number of reversals to either leave or reach the configuration. Such configurations though can be regarded as unlikely in practice.

⁵We have implemented efficient routines which test whether the sweep volume of a polygon during a rotational or a translational motion intersects any obstacles. These tests do not require the robot to be ‘blown up’.

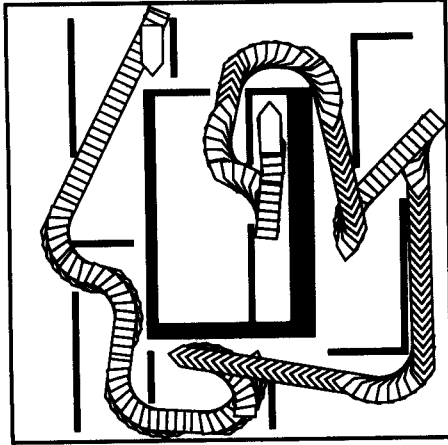


Figure 16: A car-like path in scene 4.

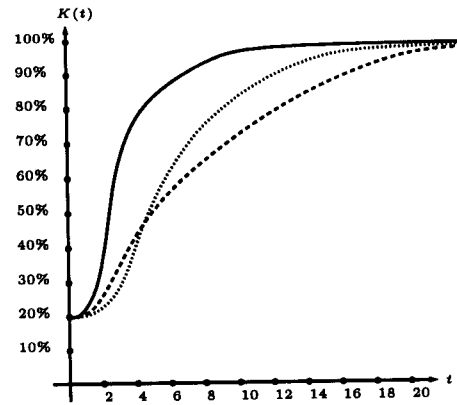


Figure 17: Learning in scene 1.

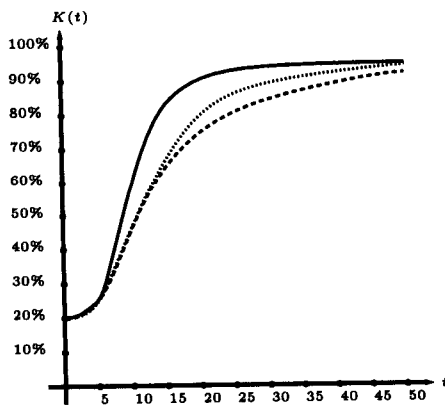


Figure 18: Learning in scene 2.

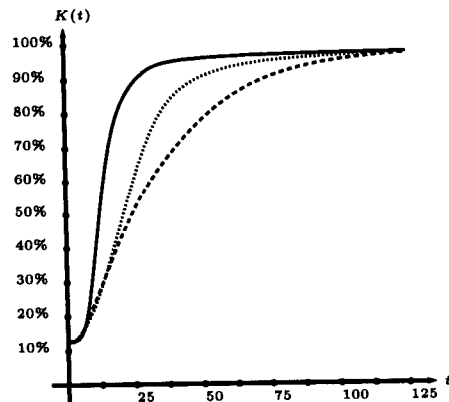


Figure 19: Learning in scene 4.

Hence, basically all ‘reasonable’ paths are found, even though the knowledge is not 100%.

We see that in scene 2 the knowledge reaches 90% in about 20 seconds with the adaptive geometric adding method, which is again about three times slower than for free flying robots. It takes about three minutes to reach 98%.

Finally, the plot for scene 4 shows learning 90% knowledge in scene 4 takes about 25 seconds with adaptive geometric adding.

The graph grows with about 25 nodes per second in scene 1, 15 nodes per second in scene 2, and about 20 nodes per second in scene 4.

6 Articulated robots

As a final example of the use of our learning paradigm, we apply it to planar articulated robots. Many solutions to the motion planning problem for articulated robots have been proposed, but still only a few planners exist that can deal with high degree of freedom articulated robots, and often only in restricted cases (see [FT89], [FT87], [Kon91], [BL90], [KL93]).

A planar articulated robot \mathcal{A} consists of n links L_1, \dots, L_n , which are some solid planar bodies (we use polygons), connected to each other by $n - 1$ joints J_2, \dots, J_n . Furthermore, the first link L_1 is connected to some *base point* in the workspace by a joint J_1 . Each joint can be either a *prismatic joint*, or a *revolute joint*. If J_i is a prismatic joint, then link L_i can translate along some vector, which is fixed to link L_{i-1} (or to the workspace, if $i = 1$), and if J_i is a revolute joint, then link L_i can rotate round some point which is fixed to link L_{i-1} (or to the workspace, if $i = 1$). The range of the possible translations or rotations of each link L_i is constrained by J_i 's *joint bounds*, consisting of a lower bound low_i and an upper bound up_i . The configuration space of a n -linked planar articulated robot can, hence, be represented by $[low_1, up_1] \times [low_2, up_2] \times \dots \times [low_n, up_n]$. In the scenes we show, prismatic joints are denoted by straight arrows, and revolute ones by curved arrows.

Local method

We have implemented and tested different local methods for planar articulated robots. Firstly, we implemented a potential field method, which is analog to the local method for free flying robots. In configuration space, we perform a walk by repeatedly computing all direct neighbors of the ‘current’ configuration c , and going to the best one (with respect to its distance to the goal configuration) which is free, if it is better than c . Of course, we must define what the direct neighbors of a configuration are. For free flying robots we took the set $c + (\{-\epsilon, 0, \epsilon\}^3 - \{(0, 0, 0)\})$. The analog definition for planar articulated robots would be $c + (\{-\epsilon, 0, \epsilon\}^n - \{(0, 0, 0)\})$. For simple robots, with not more than say three degrees of freedom, the definition works fine. The problem though is that, if the number of degrees

of freedom gets high, a configuration will have too many direct neighbors. E.g., if the robot has eight links, a configuration will have 6561 direct neighbors. A possibility is to redefine the direct neighbors. For example, we can take the set $\{(\epsilon, 0, \dots, 0), (-\epsilon, 0, \dots, 0), (0, \epsilon, 0, \dots, 0), (0, -\epsilon, 0, \dots, 0), \dots, (0, \dots, 0, \epsilon), (0, \dots, 0, -\epsilon)\}$. The number of direct neighbors is then only linear in the number of degrees of freedom. The price of such a ‘cheaper’ definition though is, that the local method will more often get stuck in a local minimum. Which definition works best very much depends on the scene, and in particular on the number of degrees of freedom of the robot.

Experiments show that for robots with many degrees of freedom, the best results are obtained by very simple local methods, e.g., the local method which just constructs the straight path (in configuration space) connecting its two argument configurations, and succeeds iff this path intersects no obstacles. Because we are mainly interested in robots with many degrees of freedom, we will give experimental results using this last local method.

Experimental results

We demonstrate the performance of our learning paradigm applied to planar articulated robots with four example scenes, where the robots have, respectively, three, five, four, and nine degrees of freedom. We give results for two different node adding strategies, i.e., the random strategy and the adaptive random strategy, combined with the tree edge adding method. Geometric node adding, as described in section 4.1, cannot be applied to articulated robots. The distance between two configurations c_1 and c_2 is defined as Euclidean distance between c_1 and c_2 in configuration space, but scaled in a way that this distance reflects the sweep volume (in work space) of the straight path (in configuration space) from c_1 to c_2 . In the test scenes used, the coordinates of all workspace obstacles again will lie in the unit square.

The test scenes

See figures 20 to 23 for the test scenes. In each scene a (smoothed) path for the corresponding robot, computed by our planner, is indicated by a few steps. The darkest step corresponds to the start configuration, and the white step to the end configuration of the path. Furthermore, for every joint it is indicated whether it is revolute (a curved arrow) or prismatic (a straight arrow).

Scene 1 is a relatively easy scene. We see a three degrees of freedom robot, with three revolute joints, in a workspace containing two obstacles. Scene 2 is more difficult, due to the robots five degrees of freedom (it has five revolute joints), and the presence of some narrow areas in free configuration space. Furthermore there is an obstacle boundary. In scene 3 we have a four degrees of freedom robot, with three revolute joints, and one prismatic joint. The final scene is the most difficult one, with a nine degrees of freedom robot.

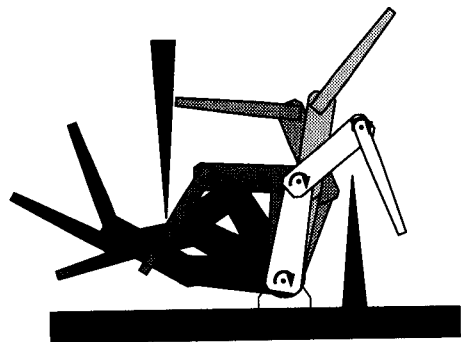


Figure 20: A 3DOF articulated robot in scene 1.

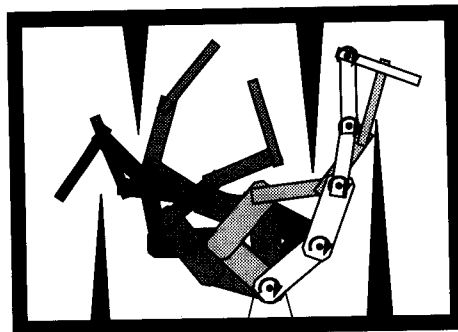


Figure 21: A 5DOF articulated robot in scene 2.

The configuration spaces of the four robots are, respectively, $[-120, 120]^3$, $[-120, 120]^5$, $[-90, 90] \times [0, 0.2] \times [-90, 90] \times [-120, 120]$, and $[-120, 120]^9$, so the links can rotate over at most 240° . The robot is not allowed to intersect itself, restricting the possible motions even further.

The test results

See figures 24 to 27 for plots of $K(t)$, the knowledge acquired by the planner after having learned for time t . The dashed plots correspond to random node adding, and the dotted plots to adaptive random node adding. We see that the learning problem in scene 1 is solved for practically 100% in about four seconds with random node adding, and in about two seconds when the adaptive adding strategy is applied. In scene 2 it takes almost ten seconds to acquire 95% knowledge with adaptive random adding, which is again a clearly better performance than that achieved by normal random adding. In scene 3 the problem is solved for 90% in about fifteen seconds, with both adding strategies. Scene 4 takes much longer, i.e., more than two minutes are required for 90% knowledge.

The growth of the graphs varies from about 8 nodes per second in scene 4 to about 100 nodes per second in scene 1.

7 Conclusions and related work

In this paper we have presented a probabilistic technique for the learning motion planning problem. This technique, which is conceptually simple, proves to be very fast for simple robots (e.g., free flying planar robots, easy articulated robots), and

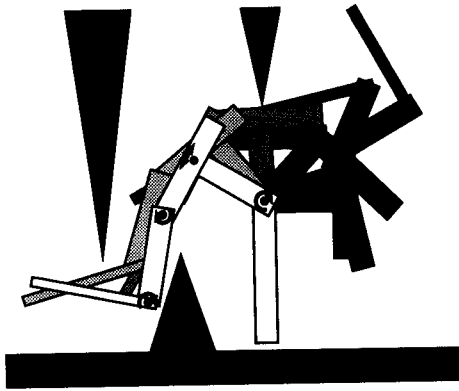


Figure 22: A 4DOF articulated robot in scene 3.

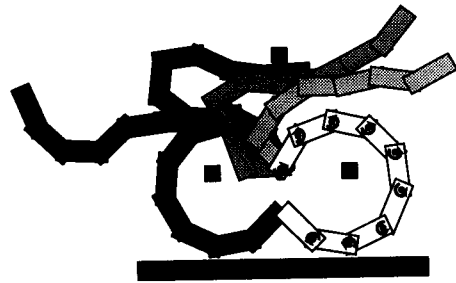


Figure 23: A 9DOF articulated robot in scene 4.

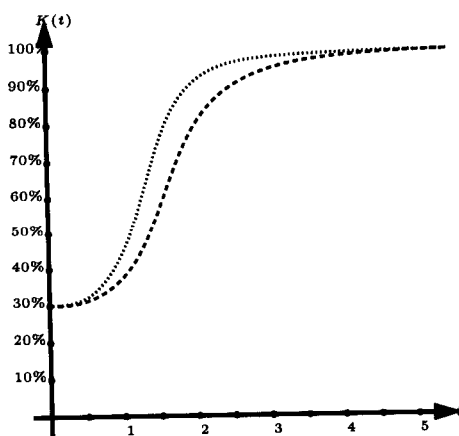


Figure 24: Learning in scene 1.

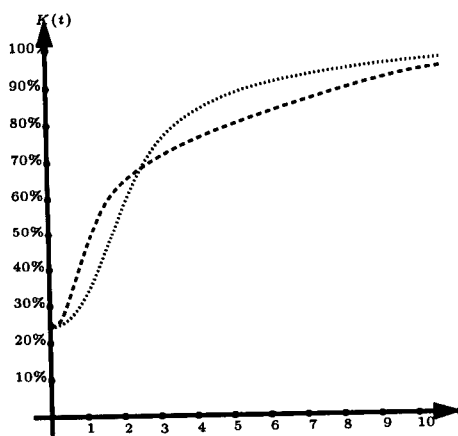


Figure 25: Learning in scene 2.

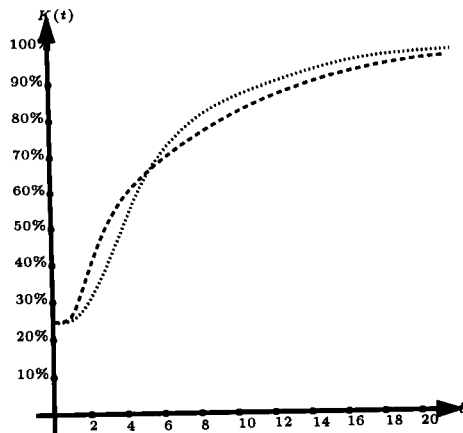


Figure 26: Learning in scene 3.

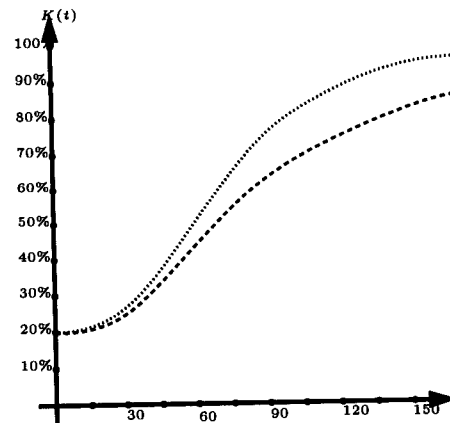


Figure 27: Learning in scene 4.

powerful enough to deal reasonably well with non-holonomically constrained robots (car-like robots) and robots with high degrees of freedom.

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. Experimental results (see [Mas92], [Šve93]) indicate that very primitive local methods achieve the best results. Hence, it should normally be no problem to find a good local method for some given robot type.

Currently we are working on the application of the method to tractor-trailer robots, and furthermore we are planning extensions of the 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 planning to apply our method to 3-dimensional workspaces. Finally, we are also considering some possibilities to do the node adding in a non-random manner, which guarantees completeness in the query phase.

Acknowledgments

We would like to thank Geert-Jan Giezeman for the implementation of many crucial 'geometric' routines (some of which are contained in the *Plageo* library, see [Gie93]), Erik Vermeer, who has written most of the code for articulated robots, Jules Vleugels, Erik van Wessel, and Otfried Schwarzkopf for *Ipe*, with which all figures in this paper were either created or edited.

References

- [BL90] J. Barraquand and J.-C. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. In *Proc. IEEE Intern. Conf. on Robotics and Automation*, pages 1712–1717, 1990.
- [BL91] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Internat. J. Robot. Res.*, 10:628–649, 1991.
- [BL93] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.
- [FT87] B. Faverjon and P. Tournassoud. A local approach for path planning of manipulators with a high number of degrees of freedom. In *Proc. IEEE Intern. Conf. on Robotics and Automation*, pages 1152–1159, 1987.
- [FT89] B. Faverjon and P. Tournassoud. A practical approach to motion planning for manipulators with many degrees of freedom. In *Proc. 5th Intern. Symp. on Robotics Research*, pages 65–73, 1989.
- [Gie93] Geert-Jan Giezeman. *PlaGeo—A Library for Planar Geometry*. Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, August 1993.
- [HA92] Y. K. Hwang and N. Ahuja. Gross motion planning—a survey. *ACM Comput. Surv.*, 24(3):219–291, 1992.
- [KL93] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. Technical Report STAN-CS-93-1490, Dept. Comput. Sci., Stanford Univ., Stanford, CA, September 1993.
- [Kon91] K. Kondo. Motion planning with six degrees of freedom by multistrategic bidirectional heuristic free-space enumeration. *IEEE Transactions on Robotics and Automation*, 7(3):267–277, 1991.
- [Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.
- [LJTMar] Jean-Paul Laumond, Paul E. Jacobs, Michel Taïx, and Richard M. Murray. A motion planner for nonholonomic mobile robots. *IEEE Trans. Robot. Autom.*, ??:??, 1993, to appear.
- [LTJ90] J.-P. Laumond, M. Taïx, and P. Jacobs. A motion planner for car-like robots based on a global/local approach. In *Proc. IEEE Internat. Workshop Intell. Robots Syst.*, pages 765–773, 1990.

-
- [Mas92] J. A. M. Mastwijk. Motion planning using potential field methods. master thesis, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, August 1992. INF/SCR-92-27.
- [Ove92] M. H. Overmars. A random approach to motion planning. Technical Report RUU-CS-92-32, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, October 1992.
- [Šve93] P. Švestka. A probabilistic approach to motion planning for car-like robots. Technical Report RUU-CS-93-18, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, April 1993.