# A Random Approach to Motion Planning

Mark H. Overmars

Technical Report RUU-CS-92-32
October 1992

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

ISSN: 0024-3275

# A random approach to motion planning*

Mark H. Overmars†

**Abstract**

The motion planning problem asks for determining a collision-free path for a robot amidst a set of obstacles. In this paper we present a new approach for solving this problem, based on the construction of a random network of possible motions, connecting the source and goal configuration of the robot. Experiments show that the method is very fast compared to other techniques, also when paths are complicated. The method can be seen as a learning strategy in the sense that the network can be reused for further motions (extending it where required) as long as the obstacle environment does not change. The method has only been tested for moving solid robots (allowing translation and rotation) in a planar environment but it can easily be extended to three-dimensional workspaces and articulated robots.

## 1 Introduction

A basic problem in the design of autonomous robots, that must perform different tasks without human intervention, is the motion planning problem:

> Given a robot $R$ and an environment containing a number of obstacles, compute a motion to move $R$ from some source configuration *source* to some goal configuration *goal*, without colliding with any of the obstacles.

See figure 1 for a typical planar situation in which an L-shaped robot has to move from the bottom left to the top right, allowing for both translation and rotation. The obstacles are shown in black, the path is indicated by a number of steps.

The motion planning problem has received considerable attention over the past years. See the book of Latombe [8] for an account of the different techniques that

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

†Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands, Email: markov@cs.ruu.nl.

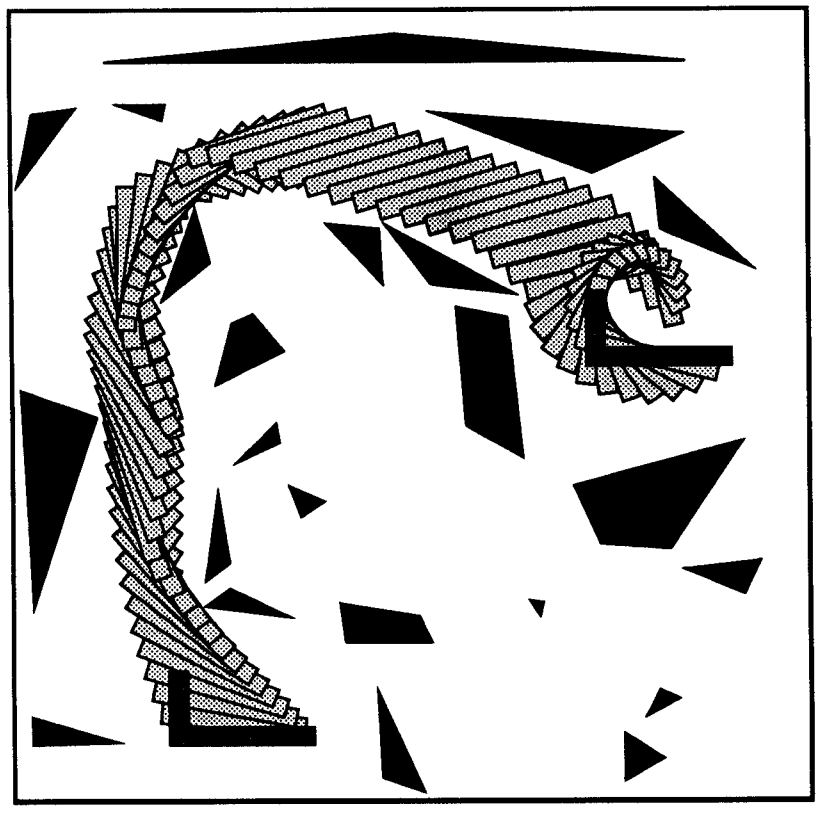

Figure 1: A typical planar motion.

exist. Despite the effort efficient, general algorithms for solving motion planning problems are still lacking. Only special cases have been dealt with in a satisfactory way.

Current approaches to motion planning can roughly be divided in three classes: roadmap methods, cell decomposition methods, and potential field methods. We will briefly review them. Here, and in the rest of this paper, we assume that the robot is a solid body, moving freely among the obstacles. So we do not consider articulated robots or robots with non-holonomic constraints. (See the concluding section 5 for some remarks on generalizations though.)

Roadmap methods try to construct a network of highways amidst the obstacles. Once this network is available, a motion for the robot can be planned as follows: first we move the robot from the source configurations to a nearby position on the network. Next we move the robot along the network until it is near to the goal configuration and finally we leave the network and move the robot directly to its goal. The network can either be constructed in the 2- or 3-dimensional workspace of the robot or in the corresponding 3- or 6-dimensional configuration space (i.e., the space of all possible configurations, consisting of a position and an orientation,

of the robot). A typical network used is the Voronoi diagram of the obstacles that tries to maximize the clearance for the robot (see e.g. [4, 13]).

Cell decomposition methods try to divide the free space, i.e., the part of the configuration space where the robot does not collide with any obstacle, into simple cells. Such a decomposition can either be exact or can approximate free space. Next, neighboring cells are connected in a graph. Between such cells a motion can easily be constructed. To find a motion between a source and goal configuration the algorithm determines the cells containing the source and goal, computes a path in the graph between the cells and computed a motion for each of the edges in the path. Typical examples of this approach can be found in [11] (see also [12]) for exact decompositions, and [2, 5, 9] for approximate decompositions. (See also [8].)

Both roadmap methods and cell decomposition methods construct a data structure that can later on be use for computing motions between different configurations of the robot. Unfortunately, they often require difficult (and expensive) geometric computations and the data structures produced tend to be very large (especially for cell decomposition). The potential field method works in a completely different way. In its most basic form (see e.g. [7]) it starts at the source configuration and works its way towards the goal by taking small steps. The direction of these steps is determined by some force. The goal is assumed to produce an attractive force and the obstacles produce a repulsive force. So the goal pulls the robot towards itself while the obstacles push it away. The big problem with this approach is that the robot might end up in a local minimum where the different forces sum up to 0, resulting in no direction to move. To remedy this problem different techniques have been tried like using backtracking, adding random Brownian motion ([1]), or modifying the potentials created by the obstacles ([14]). Such techniques can be either very slow (especially backtracking) or are only applicable in limited situations. The problem is illustrated in figure 2. Even though source and goal are very near to each other the robot has to make a big detour to reach the goal. Hence, it has to go completely against the attractive force of the goal.

Looking again at figure 2 it is clear that a solution would be easy to find if we told the robot to move first to the top left, next to the top right and finally to the goal. A solution that uses this idea by global learning was proposed in [6]. The idea is to put a grid of (large) cells over the configuration space and keep track of the probability that the potential field method finds a path from a particular cell to a particular neighboring cell. After some learning phase the system would know that there is no chance of going directly from the source to the goal configuration but would choose the detour instead.

We will exploit a different strategy to solve this type of problems. Again referring to figure 2, a motion would be easy if we gave the planner two via-points on the path, one at the top left and one at the top right. The planner would now have no problem to compute the motion. It is though difficult to compute such via-points. We could though try to create random via-points and see whether they are of any use. This was one of the ideas that led to the new motion planning algorithm described in this

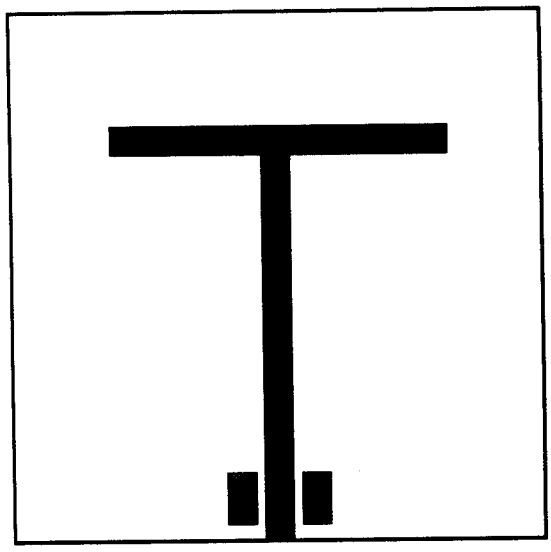

Figure 2: A problematic situation.

paper. The method uses a very simple potential field technique to build up a random network of highways between random via-points, trying to connect the source and goal configuration. The method turns out to be extremely fast. For example, the motion in figure 1 took only about 2 seconds to compute on a Silicon Graphics Indigo (30 MIPS, 4.2 MFLOP) workstation. Also the "problematic" situation in figure 2 took only 0.2 seconds to solve. The method is easy to implement and requires only simple geometric calculations. Although at the moment only implemented for moving a solid robot (with translation and rotation) in a planar environment, the method can easily be extended to 3-dimensional workspaces and to more complicated (articulated) robots.

The method can be seen as a learning strategy. It builds up a network as long as it does not know a path from source to goal. This network can be reused and extended when new motions are asked for. The network provides a very small data structure representation of possible motions. It simply consists of a graph with typically somewhere between 100 and 1000 nodes and a similar number of edges, which can easily be stored for later use.

This paper is organized as follows. In section 2 we describe the basic approach taken. This approach leaves many details to be filled in. In section 3 we discuss the different possibilities for this and motivate the choices made. In section 4 we give a number of experimental results and indicate the influence of particular parameters on the speed of the method. Finally, in section 5 we give some conclusions and describe possible extensions and improvements of the method.

## 2 Global approach

As indicated in the introduction, the underlying idea of our approach is to construct a random network of possible motions between different configurations of the robot. We continue doing so until the start and goal configuration have been connected in the network in which case a path has been found. We will represent the network as a graph. Each node of the graph corresponds to a tested configuration in free space and each edge between two nodes corresponds to a motion that was computed between the two configurations. We do not explicitly store the motion. This motion can be recomputed later once we connected the source and goal node in the graph.

The method now works as follows: We first construct a graph consisting of two nodes: the source node *source* and the goal node *goal*. (We do not make a distinction between a node and the configuration it represents. So we will simply use the notation *source* to indicate both the source node and the source configuration.) Next we add nodes (configurations) to the graph that lie in free space and try to connect them with other nodes in the graph, using some simple motion finding algorithm. This simple algorithm need not be successful in all cases. It is only required to be able to compute simple motions when they exist (see the next section for more details on this algorithm). Once *source* and *goal* are in the same connected component of the graph we compute a path between them, recompute the motions for the different edges of this path, and combine them into a motion from *source* to *goal*.

Below we describe this procedure in more detail. Note that a number of aspects still have to be filled in. In the next session we discuss the possible choices.

ALGORITHM PLAN-MOTION(*source*,*goal*)
1. Let $G$ be an empty graph;
2. Add nodes *source* and *goal* to $G$;
3. Try to connect them with a simple motion finding algorithm;
4. **while** *source* and *goal* not connected in $G$
5. Take some (random) configuration *config* in free space;
6. Add *config* to $G$;
7. Select some nodes in $G$;
8. Try to connect *config* to these nodes, using the motion finding algorithm and, if successful, add an edge in the graph;
9. Find a path between *source* and *goal* in $G$;
10. For each edge on this path recompute the corresponding motion;
11. Combine these motions into the final motion;

In figures 3 to 5 the shape of the graph constructed is visualized at different stages in the process. The small squares are the nodes of the graph, the edges correspond to possible motions. Note that the motion does not follow the edge directly. Hence,

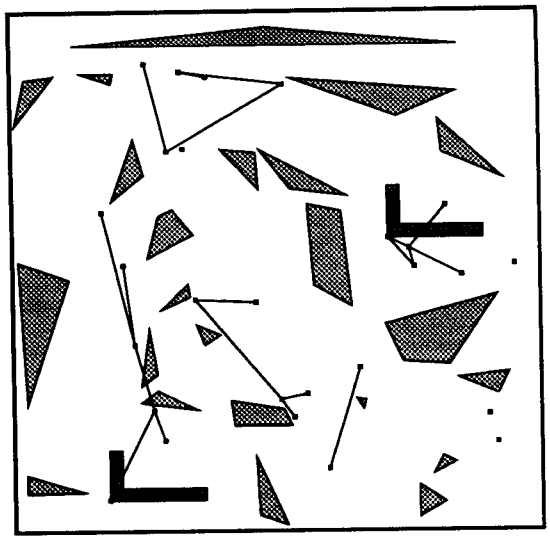

Figure 3: A number of components.

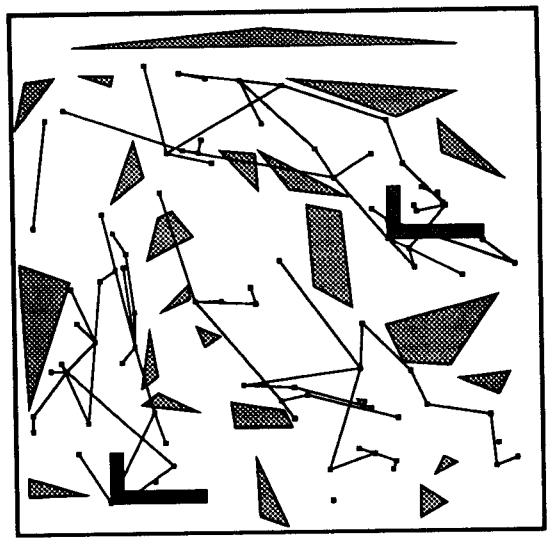

Figure 4: Some components left.

even though edges run through obstacles the corresponding motion does not. Also note that the graph is actually constructed in the 3-dimensional configuration space; each node corresponds to a position and orientation. We simply projected the nodes back to the workspace, discarding the rotation angle component. In the first picture there are still a large number of small components. One is connected to the source node at the bottom left and another to the goal node at the top right. Some other components lie in between. In the second picture some more nodes have been added, connecting different components. Finally, in the third picture the source and goal node have been connected and, hence, the algorithm has found a motion.

# 3 Specific choices

The global approach described in the previous section leaves many decisions to be made. In particular one has to decide where to add "random" configurations, with which nodes in the graph we try to connect the new configuration, and the way of connecting, i.e., the algorithm used for constructing simple motions. In the following subsections we motivate our choices and indicate some alternative possibilities.

## 3.1 Finding simple motions

One of the crucial ingredients in the global approach is the algorithm for finding simple motions between the added configuration and a number of "nearby" configurations in the graph. This algorithm does not always need to find a motion but it is important that it halts when it cannot succeed. (Some potential field approaches do

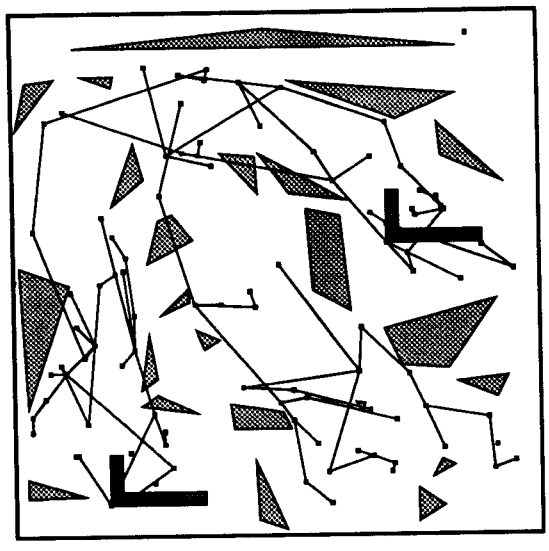

Figure 5: Source and goal connected.

not have this property. They will simple search forever if no motion can be found. Such methods are not suited for our purposes.) There are many possible choice for such an algorithm. On one hand we could take a very powerful method. As a result we would almost always find a motion when it exists and, hence, only very few nodes will be required and the resulting graph will be small. The algorithm would (probably) be slow but, one can hope that this is compensated by the fact that only a few motions need to be computed. On the other hand, one could choose a very simple and fast algorithm that is much less successful. In this case the graph would be much larger and we need to compute many more motions but the time per motion would be a lot smaller.

Some experiments we did with different types of potential field methods (e.g. with and without backtracking) indicate that a very simple underlying algorithm is to be preferred (see Mastwijk [10]). The method though should not be too simple. For example, the approach in which one only tries to connect the configurations with a straight motion (interpolating between the two configurations) was only successful for very simple situations. As soon as the scene of obstacles became more complicated (e.g. with small passages) the method needed a large graph to solve the problem and, hence, was much slower.

In the end we decided on the following method that can be regarded as a very simple potential field method: Let $\epsilon$ be some small stepsize. The method will take steps of this size. To avoid collisions during a step we blow up the robot with a factor $\epsilon$. (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 $\epsilon$.) In pseudo C-code the

method looks as follows:

ALGORITHM SIMPLE-MOTION(*source,goal*)

```
1.  config = source;
2.  while (not ready)
3.      if (config == goal) return goal reached;
4.      for each of the 26 direct neighbors of config
5.          determine distance to goal;
6.      Sort neighbors by distance in a list neighbor[];
7.      for (i=0; i<13; i++)
8.          if neighbor[i] in free space
9.              config = neighbor[i]; break;
10.     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 10). In each loop all 26 direct neighbors of the current configuration (by taking an $\epsilon$-step in the x- y- and $\theta$-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 4-5) and sorting them (line 6). 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 7-9). When none of the neighbors are possible we conclude that no path could be found (line 10). (In the actual implementation the order of the steps is slightly changed to obtain some slight speed-up.)

One can view this method as a potential field method. Here the attracting potential created by the source is simply inversely proportional to the distance. The repulsive potential is infinite when the robot gets nearer than $\epsilon$ 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 loop in lines 7-9 is executed only once, i.e., only one of the basic tests is performed. This makes things even faster.

Of course, there are many other possibilities for finding motions. As indicated above we did try to use some more sophisticated potential field techniques which turned out to be much slower. We also looked at straight line motions which were slightly faster for very simple scenes of obstacles but performed quite bad for complicated scenes (see also section 4). One other possibility we considered was to look at only 6 direct neighbors rather than 26 (by taking an $\epsilon$ step in only one of the x- y- or $\theta$-direction). This also turned out to be slower (see section 4).

One final option we tried was to compute motions both from source to goal and from goal to source. When our algorithm cannot find a motion from source to goal it is still possible that it finds a motion from goal to source which, reversed, is of course a legal motion. In this way we find more motions and, hence, obtain a smaller graph. The idea though did not have much success. In a typical scene it increased the chances for finding a motion with 10% or 20%, i.e., from 10% success to 12% or from 25% success to 30%. But the computation time for motions almost doubled. The size of the resulting graph became only slightly smaller and the time for finding the solution became much larger. Hence, we soon dropped this idea.

## 3.2 Neighbors

When adding a "random" configuration to the graph we have to try and connect it to a number of nodes in the graph. Again one has a trade-off here. On one hand one can try to connect the new configuration to all existing nodes in the graph. In this way normally only a small graph suffices but we have to compute many motions (of which a large number will fail) which is expensive. It turns out that a much better strategy is to try and connect the new node only to nodes that lies relatively near to the new node. This reduces the number of nodes and, hence, the number of paths, and also makes the paths computed short (and, hence, faster). Of course, occasionally we will miss a long possible connection but this connection will probably be found later when a few nodes have been added "in between".

There is though another optimization we can use here. The goal of the method is to connect up different component such that the source and goal component will get connected. Hence, there is no reason to connect the new node to multiple nodes in the same component. This leads to the following approach: For each connected component in the graph we select the node nearest to the new node and, if near enough (i.e., nearer than some predefined distance *dist*; see section 4 for best values) try to connect it to the new node.

ALGORITHM SELECT-NEIGHBORS(*config*)
1. **for** each connected component in the graph
2. find the node *node* nearest to *config*;
3. **if** distance between *node* and *config* < *dist*
4. **if** SIMPLE-MOTION(*node*,*config*) successful
5. add an edge in the graph between *node* and *config*;

Note that in this way the graph will always look like a forest; no cycles will appear. We did try connecting multiple nodes per component but this did not improve the running time of the method. We also tried to connect the new node with even fewer nodes in the graph but this also did not seem more efficient.

We still have to define what "nearest" means, i.e., what is the distance between two configurations. One would expect that the distance should be measured in

configuration space. The role of the rotation angle in the distance though is more subtle. If the robot is roundish (e.g., a square) rotation without translation is normally possible. Hence, we should not take the rotational component of the configuration into account when calculating distances; a configuration at the same position but rotated 180 degrees should be much "nearer" that a configuration with the same orientation but far away. On the other hand, when the robot is long and thin the rotation angle is important and should be taken into account; a small rotation changes the place where the robot is much more than a small translation. To solve this problem we take as $\theta$-distance between two configurations the distance traveled by the point on the robot furthest away from the origin of the robot. The total distance then is

$$\sqrt{x\text{-}dist^2 + y\text{-}dist^2 + \theta\text{-}dist^2}.$$

This is easy to compute (of course we do not need to actually compute the square root as we only compare distances) and resembles quite well the difference between the two configurations. (Experiments with different methods of defining distances showed that the method chosen performs very good.)

## 3.3 Adding strategy

One decision to take is where to add new configurations. The simplest solution is to add them in a fully random way. So we take a random configuration, check whether it lies in free space (using a simple intersection test with the robot in the configuration and the set of obstacles) and, if so, add it to the graph and compute motions to nearby nodes. Such an approach is surprisingly successful. More successful than, e.g., adding configurations in a regular pattern. The method though has a few disadvantages. Firstly, it adds the same number of nodes at "easy" places in free space, i.e., far away from obstacles, as at difficult places, near to obstacles. This results in a waste of nodes and, hence, a waste of time. See for example figure 6. Here we have give a typical distribution of the nodes of the graph (no edges shown) computed for moving a small rectangle from left to right. Of course, one would prefer that nodes are added in the passage but, because the passage is narrow (especially in configuration space) the chance for a node to appear in the passage is small. Secondly, adding configurations randomly does not take into account the graph computed so far. The reason for adding configurations is to connect the goal with the source. The graph will consist of a number of connected components and we would prefer to add configurations that can be connected to at least two components, in this way reducing the number of components. In particular one would like configurations that (have a large chance to) connect the source component with the goal component.

A solution, in particular to the first problem, is to use forbidden configurations, i.e., random configurations that are not in free space. This is based on the following observation. Near to difficult passage in configuration space there is normally a large

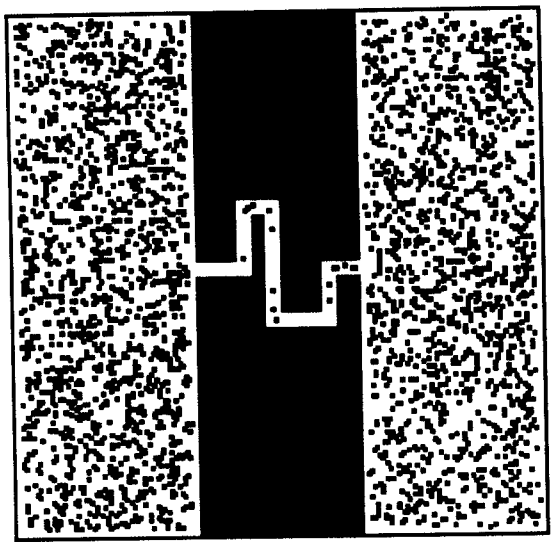

Figure 6: Adding random.

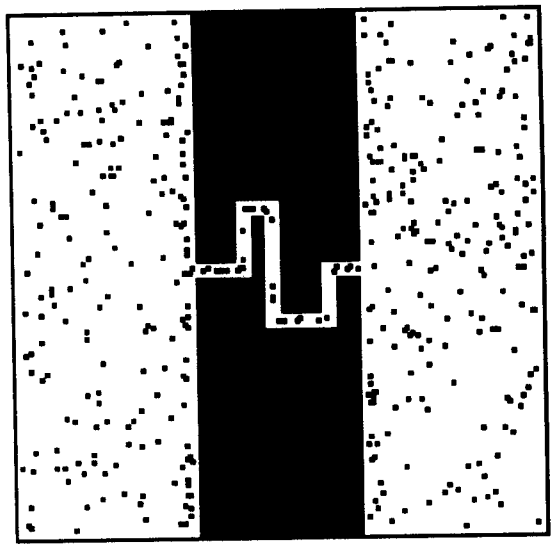

Figure 7: Using forbidden configurations.

forbidden region (otherwise the passage would probably not be important). Hence, though the chance is small that a random configuration falls inside the passage, the chance that it falls inside this nearby forbidden region is large. So we would like to move this forbidden configuration into the passage. Rather than performing difficult geometric computations for this we (again) use a simple random approach: Whenever we find a forbidden configuration we choose a random direction (in configuration space) and move the configuration in small steps in this direction until it is in free space (or out of the boundaries). In this way many configurations will be added near to obstacles. See figure 7 for the result of applying this technique. As you can see the number of nodes required to solve the problem is a lot less. A disadvantage of the approach is that the walking with the configuration towards free space takes time. Hence, the idea is only useful for difficult scenes (see section 4 for the effect on the running time).

A second approach implemented is to use a more adaptive adding strategy. Whenever the algorithm wants to add a configuration it checks all nodes that lie nearby, i.e., within distance *dist* of the new configuration (see subsection 3.2) and counts their number and, in particular, the number of different components that appear among them. Based on this information it determines a chance with which the configuration is added or discarded. The method is made more precise in the following algorithm:

ALGORITHM ADAPTIVE-ADDING()
1. *config* = random configuration in free space;
2. *numb* = number of nodes within distance *dist* from *config*;
3. *comp* = number of different graph components among them;

4. **if** ($numb \leq 4$) $chance = 1.0$;
5. **else if** (both source and goal component nearby) $chance = 1.0$;
6. **else if** ($comp > 1$)
7. **if** (source or goal component nearby) $chance = 0.75$;
8. **else** $chance = 0.50$;
9. **else** $chance = 0.25$;
10. with chance *chance* call SELECT-NEIGHBORS(*config*);

The different constants are a bit arbitrary but turn out to work nicely. In line 4 we check whether the number of nodes in the region is large enough to make any sensible decision here. If not we always add the configuration. Also, when both the source and goal component are nearby, we always add the configuration (line 5). Otherwise, when there are at least two different components nearby we increase the chance. Here we make a distinction between the cases when either the goal or source component is nearby and the other cases. The idea is that we prefer to extend the source and goal components. Note that the adaptive strategy does give us some penalty: some configurations will be discarded even though some time has been spent on them. This is though normally compensated by the increase in success in connecting components, resulting in a smaller graph. See section 4 for some example of the effect of using this adaptive adding strategy.

Other, more complicated, adding strategies exist that take more of the geometry into concern. For example, in a large workspace one might start adding nodes in the vicinity of the source and goal node and slowly extend the range. Also one might try to locate "difficult" parts of the workspace in advance. Such approaches are a topic of further study.

## 3.4 The final motion

Once the source and goal node are connected in the graph we are able to compute a motion for the robot. First we compute a path between the two nodes in the graph. (Because the graph is a forest only one path exists. If one would allow multiple connections between a new configuration and existing connected components multiple paths could exist. In this case one can store the length of each computed motion with the corresponding edge and compute a shortest path in the graph.) Secondly, for each edge on this path we have to recompute the corresponding motion using the simple motion finding algorithm SIMPLE-MOTION. Note though that the original motion might have been computed in the opposite direction and, hence, the algorithm might fail when computing in the wrong direction. In this case we have to compute the motion in the other direction (which must be successful, assuming SIMPLE-MOTION is deterministic) and reverse it. (Alternatively one could store the "direction" with each edge.) So the algorithm looks as follows:

ALGORITHM FINAL-MOTION(*source,goal*)

1. Find a path *node[0]* = *source*, ... , *node[z]* = *goal* in the graph;
2. **for** $(i = 0; i < z; i{+}{+})$
3. SIMPLE-MOTION(*node[i],node[i+1]*);
4. **if** (not successful) reverse SIMPLE-MOTION(*node[i+1],node[i]*);
5. Combine the pieces into one motion;

Unfortunately, the motion found can look quite ugly. See for example figure 8. The shape highly depends on the "random" configurations of the nodes on the path. Because the graph is a forest only one path exists and there is no way of selecting a nice path. To improve this we smoothen the motion in two ways. First of all we try to eliminate nodes on the path in the graph in the following way. Whenever SIMPLE-MOTION can find a motion between nodes *node[i-1]* and *node[i+1]* we discard *node[i]*. This will remove many unnecessary diversions. The result is shown in figure 9.

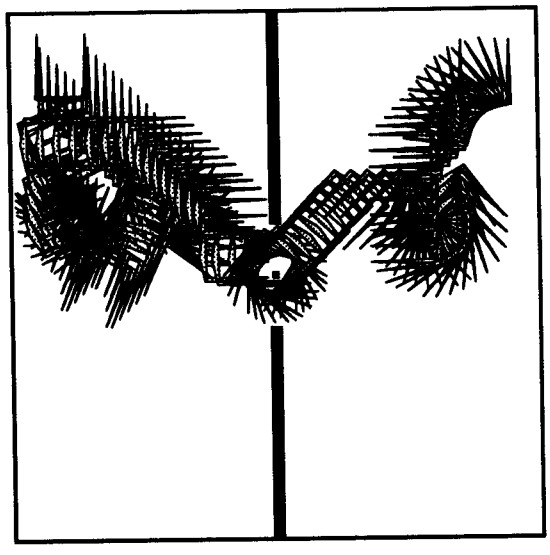

Figure 8: A motion without smoothing.

The new motion is still not very smooth. The reason for this non-smoothness though lies more in the way simple paths are computed than in the random approach. When computing simple paths only horizontal, vertical and diagonal steps are allowed. This means that paths will always be jaggy. To improve this we slightly change the simple motion planning algorithm as follows: We first try to perform an $\epsilon$-step straight towards the goal. Only if this does not succeed we try the other 26 directions. As a result, whenever possible, we get a nice interpolated movement between two configurations. A typical motion obtained is depicted in figure 10. Using this different simple motion planning algorithm has a slight effect on the efficiency of the approach, but this effect can be both positive and negative. (See the next section.)

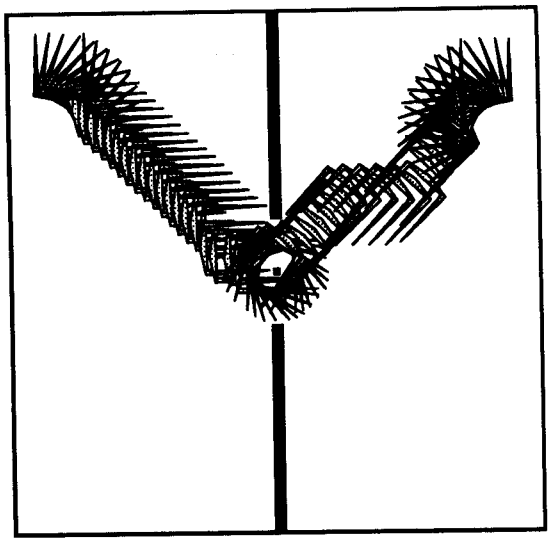

Figure 9: Simple smoothing.

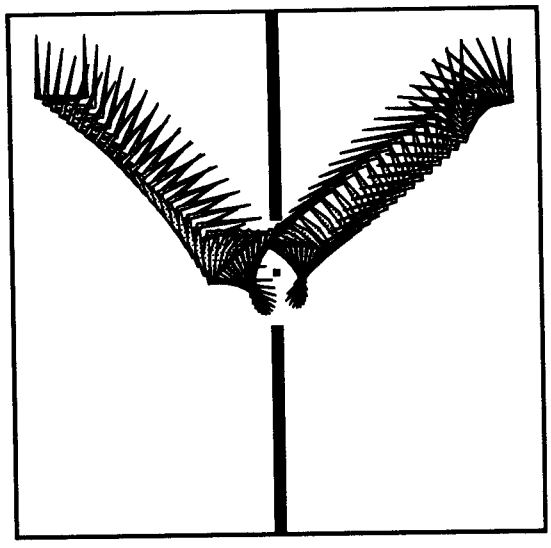

Figure 10: Expensive smoothing.

Still the path may not be the one one prefers. It could easily take long detours. Improving this is a topic of further research.

# 4 Experimental results

Because the method presented in this paper is a heuristic method, of which it seems impossible to give any theoretical bounds on its efficiency, we need to base our claim of efficiency on a number of experiments we did on "typical" scenes. In this section we describe the results obtained.

## 4.1 The implementation

The algorithm was implemented in C (partially C++) on a Silicon Graphics Indigo computer (based on a R3000 processor, about 30 MIPS and 4.2 MFLOPs, 25.4 SPECMARKS). It contained a graphical user interface, display of the graph constructed and the motions computed, routines for testing different aspects, e.g. the influence of a particular parameter, etc. Because of the simple nature of the algorithm it took us less than a month to program. The code is about 3500 lines of which about half consists of efficient routines to compute intersections between the robot and obstacles. The implementation allowed us to play with the different parameters and test different approaches. It finally led us to the approach described above.

The coordinates of all the obstacles are supposed to lie in the unit square. All random configurations are also chosen within this unit square, adding a random

orientation. Paths between configurations though can run outside this square. To avoid this in all scenes we have added a small obstacle boundary around the unit square.

There are a number of parameters to set when running the program for a particular scene:

**rotation** It is possible to switch off the use of rotation, allowing only translational motions. As expected, not allowing for rotations (when translations suffice) is faster (although not very much). All scenes tested except for the last one use rotations.

**gridsize** The gridsize indicates the number of steps in which to divide the $x$- and $y$-coordinates, resulting in the size of the $\epsilon$-steps taken in the simple motion planning algorithm. The gridsize should be large enough to allow for a motion (because the robot is blown up by the stepsize to avoid collisions during steps). On the other hand, the time required for computing paths is linear in the number of steps. Hence, a small gridsize decreases the running time. Typical values chosen were a $100 \times 100$ or a $200 \times 200$ grid (see below). As indicated in subsection 3.1 the stepsize in the $\theta$-direction is computed from this gridsize.

**maximal distance** As indicated in subsection 3.2, when connecting nodes, we only consider nodes that lie near enough to each other. The maximal distance allowed is an important parameter in the method. When choosing this distance small, paths are short and only a few nodes are tested which makes adding a configuration very fast. We need to add though many configurations to connect two far away configuration. Choosing the maximal distance too large though means that we spend a lot of time on computing paths that won't be successful. We should choose the maximal distance such that there is still a reasonable chance that two configurations with this distance can be connected. The default value chosen was 0.25. See below for the dependence of the time bound on this parameter.

**number of directions** The simple motion planning algorithm looks at the neighbors of a configuration in 26 directions to find the best new configuration on the path. We also tried looking in 6 directions (a small step in only one of the $x$-, $y$- or $\theta$-direction), and 1 direction (a small step straight towards the goal). See below for the effect. The default value was 26 directions.

**forbidden configuration** One can indicated whether to use forbidden configurations by shifting them until in free space. The default was not to use them.

**adaptive adding** Finally it is possible to switch adaptive adding on and off. Default it was on.

## 4.2 The test cases

We will now describe the eight scenes on which we tested our algorithm. Each of the scenes has its own peculiarities. The scenes, together with a typical motion computed, can be found in figures 11 to 18.

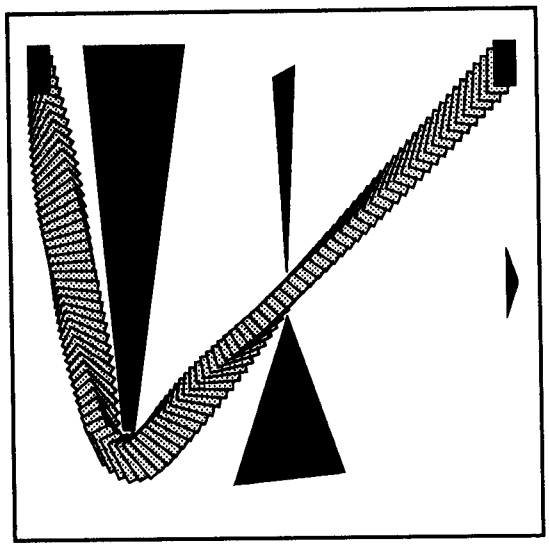

Figure 11: Scene 1.

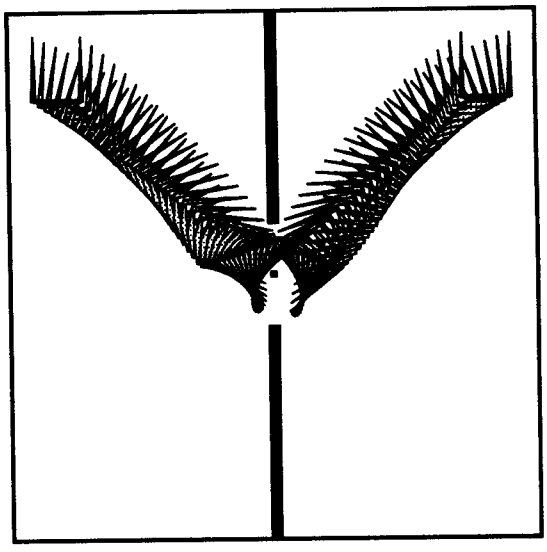

Figure 12: Scene 2.

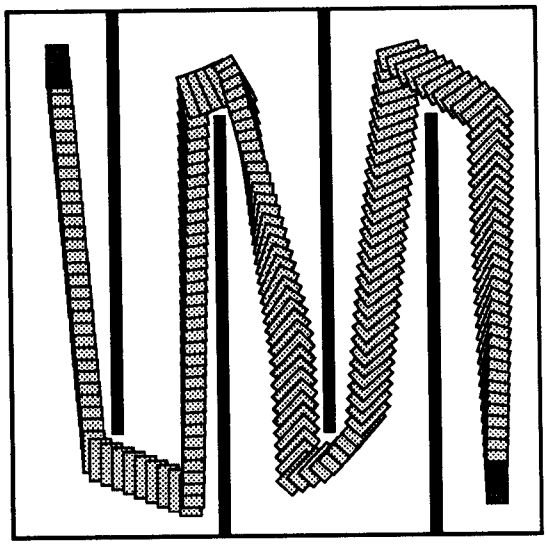

Figure 13: Scene 3.

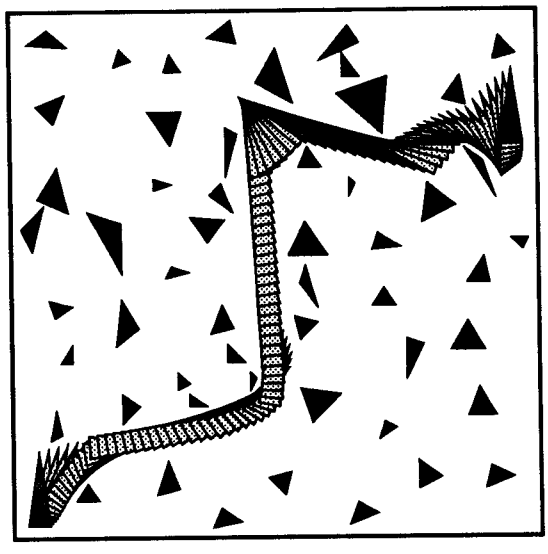

Figure 14: Scene 4.

1. The first test case is a very simple scene consisting of only a few obstacles. Still rotation is required to move from left to right through any of the three

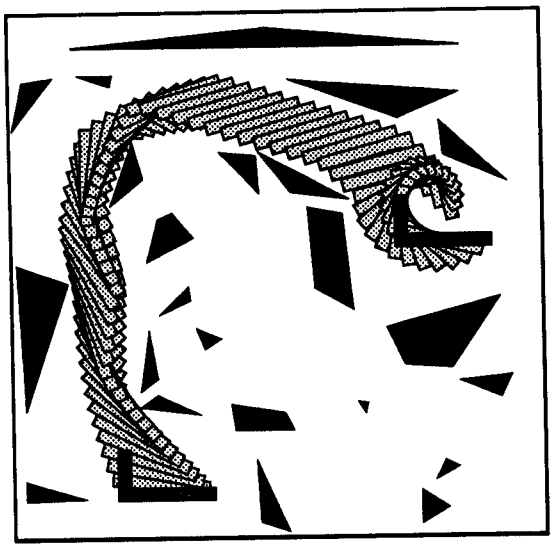

Figure 15: Scene 5.

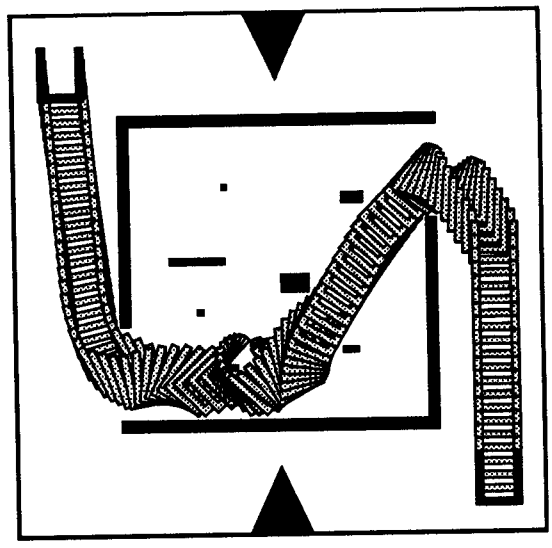

Figure 16: Scene 6.

possible holes. Simple potential field methods tend to get blocked behind the large obstacle at the left. The gridsize used is $100 \times 100$.

2. The second scene has much room for motion to the left and to the right but only a small passage near the center where it has to rotate around the small obstacle. Again this is not easy for potential field methods. It is a typical case where one would expect the use of forbidden configurations to help. Also one would expect the use of the adaptive adding strategy to be good here because soon the graph will consist of only two components, one to the left and one to the right and one would prefer to add configurations in the center. The gridsize used is $100 \times 100$.

3. This scene is trivial for human beings and also quite simple for approximate cell decomposition techniques. It does not really require rotation but we tested it with rotation anyway. For potential field methods this is again a difficult case because the robot has to move away from the goal a number of times. The gridsize used is $100 \times 100$.

4. The fourth scene looks more difficult than it is. Especially potential field approaches tend to find solutions easily. Approximate cell decomposition techniques though tend to have difficulty with such scenes because they require quite a fine precision in the approximation. Because of the large number of obstacles a fast intersection testing routine is needed that efficiently discards obstacles that are far away from the robot. Bounding box tests, as used in our implementation, provide this efficiency. The gridsize used is again $100 \times 100$.

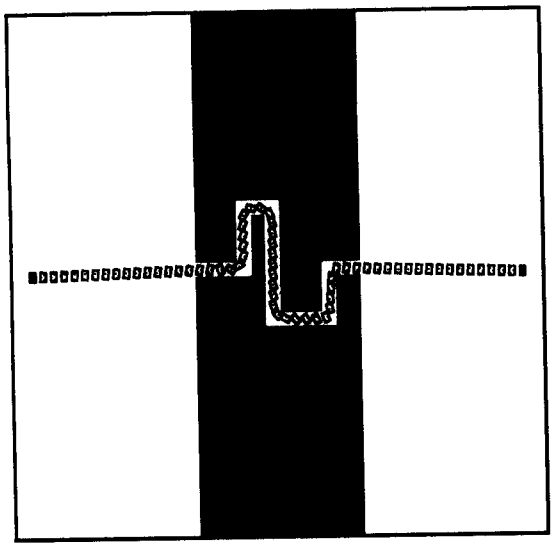

Figure 17: Scene 7.

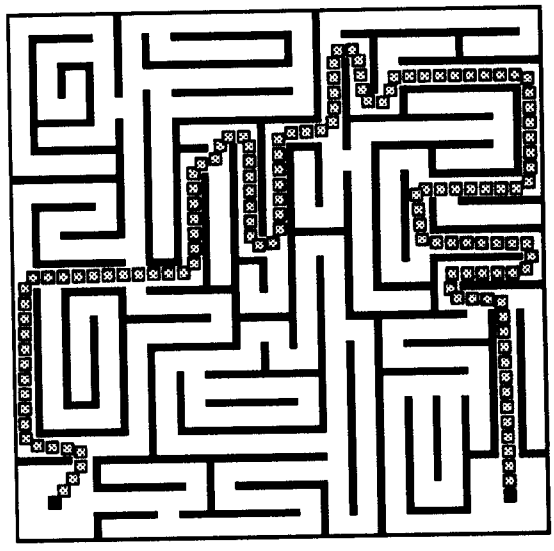

Figure 18: Scene 8.

5. Again a scene that is difficult for cell decomposition approaches. Simple potential field methods will have quite some trouble here as well. Even though the depicted motion looks like being the obvious one there are a number of other ways to get from source to goal. Because the robot is long and thin many rotation steps are required. The gridsize used is again $100 \times 100$.

6. In this scene the robot has to move through the square. The routes along the outside look more promising though at first glance. Also some rotation is required to move around the obstacles in the center. Cell decomposition techniques again would have quite some difficulty with this, also because of the non-convexity of the robot. The gridsize used is again $100 \times 100$.

7. As indicated in subsection 3.3, this scene with a very narrow passage at the center is a difficult scene for our method. The only way to get through the passage is to place a number of nodes in it. But the chance that random nodes will fall inside it is very small. Using forbidden configurations though will help quite a bit. This scene is also difficult for potential field methods. Approximate cell decomposition might be quite successful here because only near the center high precision is required and not many rotational steps are required. Also roadmap methods might be successful here. Due to the narrowness of the passage (and the fact that the robot is small) we need a gridsize of $200 \times 200$.

8. This final scene is a clear maze. Very difficult for potential field methods but doable for cell decomposition techniques cause no rotation is used here. Our method manages to solve this problem in a reasonable amount of time. Realize that, once our method has found a path, it basically knows the whole maze. So

any further motion required goes extremely fast. Also here we need a gridsize of $200 \times 200$.

## 4.3 Default setting

Below we give a table of the time required to find the motions for the different test scenes, using the default settings, i.e., a maximum distance of 0.25, using 26 directions, not using forbidden configurations, and using adaptive adding. (The bounds do not include the time for smoothing.) The timebounds given are in seconds (elapse time). They are the average over 500 runs of the algorithm (except for scenes 7 and 8 where only 50 runs were made).

| scene | average | minimum | maximum |
|---|---|---|---|
| 1 | 0.3 | 0.1 | 0.9 |
| 2 | 2.2 | 0.2 | 19.7 |
| 3 | 0.9 | 0.4 | 2.0 |
| 4 | 1.8 | 0.4 | 4.9 |
| 5 | 1.6 | 0.4 | 4.4 |
| 6 | 2.7 | 0.4 | 35.2 |
| 7 | 17.0 | 1.0 | 87.0 |
| 8 | 60.0 | 35.0 | 110.0 |

Note from the minimum and maximum timebounds that the time per run can vary quite a bit. Also note that the average tends to lie nearer to the minimum than to the maximum time. This means that only rarely the algorithm takes a lot of time. The reason for this seems to be that sometimes an unlucky configuration is added which kind of blocks connections. Such a configuration lies on a crucial place, is often the chosen neighbor but connections to it are often not found. There are two ways in which this might be avoided. One way is to give configurations a kind of conscience that remembers how often we tried to connect nodes to it (like in neural networks, see e.g. [3]) and change their likeliness of being a neighbor accordingly. A second solution might be to restart the process from time to time. E.g., we start letting the process run for at most one second. If there is no success we restart it and let it run for two seconds. If still not successful we let it run for four seconds, etc. We tried both approaches and they indeed reduce the maximum times, but the averages did increase. We did not work this out in further detail.

## 4.4 Number of directions

As indicated in subsection 3.1 we tried some different possibilities in our simple motion planning algorithm. In particular we varied the number of directions in which the algorithm takes a step, namely 26 (a step of $-\epsilon$, 0 or $\epsilon$ in any of the $x$-, $y$- and $\theta$-direction), 6 (a step $-\epsilon$ or $\epsilon$ in only one direction) and 1 (an $\epsilon$ step in the

direction of the goal). We also tested the approach of first trying a straight step and, only if this fails, trying the other 26 directions, which gives nicer smoothed paths (see subsection 3.4). The following table gives the results for the different choices. In each line the best value is given in boldface.

| scene | 26 dirs | 6 dirs | 1 dir | 1+26 |
|---|---|---|---|---|
| 1 | 0.3 | 0.3 | 0.4 | **0.2** |
| 2 | **2.2** | 2.8 | 5.6 | 2.8 |
| 3 | 0.9 | 0.5 | **0.3** | 1.0 |
| 4 | 1.8 | 5.4 | 8.0 | **1.7** |
| 5 | **1.6** | 2.2 | 2.7 | 1.6 |
| 6 | **2.7** | 4.1 | 9.0 | 3.3 |
| 7 | **17.0** | 22.0 | 452.0 | 20.0 |
| 8 | **60.0** | 60.0 | 105.0 | 70.0 |

So for almost all scenes it is best to consider 26 directions (or 1 plus 26 to get smoother paths). This is to be expected because, as explained in subsection 3.1, looking in many directions does not cost us much more time but it increases the flexibility of the simple path finding algorithm and, hence, the chance of success. Only in scene 3 where most paths are straight anyway it is more efficient to consider straight paths only.

## 4.5 Maximal distance

As explained in subsection 3.2, whenever we add a node, we look at all its neighbors within a particular distance. The choice of this distance has a large impact on the efficiency of the method. The following table gives the effect for our test scenes.

| scene | 0.10 | 0.15 | 0.20 | 0.25 | 0.30 | 0.35 | 0.40 | 0.45 | 0.50 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.5 | 0.6 | 0.4 | **0.3** | 0.3 | 0.3 | 0.4 | 0.4 | 0.4 |
| 2 | 7.7 | 2.6 | **1.8** | 2.2 | 3.0 | 4.2 | 4.9 | 6.6 | 6.6 |
| 3 | 1.1 | **0.6** | 0.6 | 0.9 | 1.0 | 1.2 | 1.4 | 1.6 | 1.7 |
| 4 | 4.8 | 2.6 | 2.1 | 1.8 | **1.7** | 1.8 | 2.0 | 2.2 | 2.3 |
| 5 | 4.5 | 2.4 | 1.7 | **1.6** | 1.8 | 2.0 | 2.2 | 2.6 | 2.7 |
| 6 | 6.2 | 2.7 | **2.3** | 2.7 | 3.6 | 5.0 | 6.1 | 6.8 | 8.7 |
| 7 | 28.0 | 14.0 | **14.0** | 17.0 | 22.0 | 50.0 | 54.0 | 93.0 | 112.0 |
| 8 | **25.0** | 30.0 | 40.0 | 60.0 | 80.0 | 100.0 | 125.0 | 150.0 | 185.0 |

It immediately follows that one should not choose the value too large. This results in trying too many impossible paths. The best value depends on two aspects of the scene: the size of the robot and the amount of space between the obstacles. When the robot is small compared to the amount of space motions tend to be easy and, hence, we should choose a larger value because longer paths are still feasible.

On the other hand, when the passages are narrow compared to the size of the robot (like e.g. in scene 8) one better takes small distance.

It would be nice if the algorithm would itself decide on the distance. Such a choice though seems to be difficult. We are working further on this.

## 4.6 Adding method

As indicated in subsection 3.3 we implemented two techniques for adding new configurations that sometimes improve the performance of the method. The first technique was the use of forbidden configurations by moving them towards the free space. As argued this is expected to improve the time bounds when dealing with small passages. The second technique uses an adaptive adding strategy that prefers to add nodes where components can be connected. The following table gives the results for applying these techniques.

| scene | adapt. | forbid. | both | none |
|---|---|---|---|---|
| 1 | **0.3** | 0.4 | 0.3 | 0.4 |
| 2 | 2.2 | 1.4 | **1.3** | 3.0 |
| 3 | **0.9** | 1.0 | 1.0 | 0.9 |
| 4 | **1.8** | 2.1 | 2.1 | 2.0 |
| 5 | **1.6** | 2.0 | 2.0 | 1.8 |
| 6 | **2.7** | 3.0 | 2.8 | 2.9 |
| 7 | 17.0 | 3.0 | **2.5** | 67.0 |
| 8 | 60.0 | 62.0 | 60.0 | **53.0** |

As can be seen adaptive adding is almost always good, except for scene 8. The effect in scene 8 can be explained as follows. In this scene it will often be the case that two components of the graph lie near to each other but there is still a wall between them. Hence, knowledge about the number of components that lie near to a new configuration is not very useful.

Using forbidden configurations is, as expected, only useful for scenes 2 and 7 where there is a narrow passage. For all other scenes it eats up some (but only a small amount of) extra time.

# 5 Conclusions and further work

In this paper we have presented a new technique for motion planning. The technique combines simple potential fields with a roadmap method, constructing a random network of possible motions. The method turns out to work very fast in a planar situation for many different types of scenes. It is simple and, hence, easy to implement.

One of the nice properties of our method is that it uses a learning approach. Gradually it learns the possible motions of the robot among the obstacles. This

knowledge can be reused for each further motion (with the same robot in the same scene). The new start and goal configuration will simply be added to the graph and the graph is extended further until they are connected. After a few tries the graph normally has almost complete knowledge about the connectivity structure of free space and, hence, further motions are computed instantaneously (except for the smoothing of the path). The graph is a very small representation of the possible motions (normally a few hundred nodes and a similar number of edges) which proves very favorable with other data structures that represent free space (as e.g. used in approximate cell decomposition).

We expect that the method can be parallellised in an easy way. The time used by the method is almost completely spend on computing simple motions. Such motions can easily be computed in parallel. This would lead to a setup in which one processor is responsible for maintaining the graph and creating new random nodes while it passes the task of computing simple motions to other processors.

A disadvantage of the method is that it is uncomplete, i.e., the method can never determine that a path does not exist. It will simply go on forever adding new random configurations. Of course, one could argue that the longer the method takes the smaller the chance that a path exists but there is no way to be sure.

At the moment we are considering various extension of the method. First of all we work on extending the method to work in a 3-dimensional workspace with free flying or articulated robots. Although the basic approach simply carries over there are some important details to fill in. For example, in the simple motion planning algorithm required it is unclear in how many directions to search. Looking in all $3^d$ directions, where $d$ is the number of degrees of freedom (e.g., 6) seems too expensive. Also we have to define in a better way what the distance between two configurations is.

Another direction of research we are working on is to extend the method to deal with non-holonomic motion planning. Again it seems that the basic approach can carry over and we mainly have to work on a good algorithm to compute simple non-holonomic paths.

We also work on dynamic version of the motion planning problem and versions where no complete information about the scene is available. The idea here is that we build up a random network based on our current knowledge and revalidate (and change) it when new information becomes available.

# Acknowledgments

I would like to thank Geert-Jan Giezeman for helping me with part of the implementation of this method. All pictures in the paper were created with a general motion viewer MotView, developed also by Geert-Jan Giezeman.

# References


[1] J. Barraquand and J.C. Latombe, Robot motion planning with many degrees of freedom and dynamic constraints, *Proc. 5th Intern. Symp. on Robotics Research*, Tokyo, 1989, pp. 74–83.

[2] R.A. Brooks and T. Lozano-Pérez, A subdivision algorithm in configuration space for findpath with rotation, *Proc. 8th Intern. Conf. on Artificial Intelligence*, Karlsruhe, 1983, pp. 799–806.

[3] D. Desieno, Adding a conscience to competitive learning, *Proc. Int. Conf. on Neural Networks, I*, IEEE Press, New York, 19988, pp. 117–124.

[4] C. Ó'Dúnlaing and C.K. Yap, A "retraction" method for planning the motion of a disk, *J. Algorithms* **6** (1985), pp. 104–111.

[5] B. Faverjon, Obstacle avoidance using an octree in the configuration space of a manipulator, *Proc. IEEE Intern. Conf. on Robotics and Automation*, Atlanta, 1984, pp. 504–512.

[6] B. Faverjon and P. Tournassoud, A practical approach to motion planning for manipulators with many degrees of freedom, *Proc. 5th Intern. Symp. on Robotics Research*, Tokyo, 1989, pp. 65–73.

[7] O. Khatib, Real-time obstacle avoidance for manipulators and mobile robots, *Intern. Journal of Robotics Research* **5** (1986), pp. 90–98.

[8] J.-C. Latombe, *Robot motion planning,* Kluwer Academic Publishers, Boston, 1991.

[9] T. Lozano-Pérez, Automatic planning of manipulator transfer movements, *IEEE Trans. Systems, Man and Cybernetics* **SMC-11** (1981), pp. 681–698.

[10] P. Mastwijk, *Motion planning using potential field methods*, Master thesis, Department of Computer Science, Utrecht University, 1992.

[11] J.T. Schwartz and M. Sharir, On the piano movers' problem I: The case of a two-dimensional rigid polygonal body moving amidst polygonal boundaries, *Comm. Pure Appl. Marh.* **36** (1983), pp. 345–398.

[12] J.T. Schwartz, M. Sharir and J. Hopcroft, *Planning, geometry, and complexity of robot motion*, Ablex, Norwood, 1987.

[13] S. Sifrony and M. Sharir, A new efficient motion planning algorithm for a rod in two-dimensional polygonal space, *Algorithmica* **2** (1987), pp. 367–402.

[14] R. Volpe and P. Khosla, Artificial potentials with elliptical isopotential contours for obstacle avoidance, *Proc. 26th IEEE Conf. on Decision and Control*, Los Angeles, 1987, pp. 180–185.