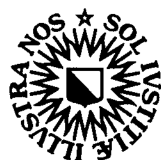
Motion Planning Using A Colored Kohonen Network

J.M. Vleugels, J.N. Kok and M.H. Overmars

RUU-CS-93-38
November 1993



Utrecht University

Department of Computer Science

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

Motion Planning Using A Colored Kohonen Network

J.M. Vleugels, J.N. Kok and M.H. Overmars

Technical Report RUU-CS-93-38
November 1993

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

ISSN: 0924-3275

Motion Planning Using A Colored Kohonen Network*

Jules M. Vleugels Joost N. Kok Mark H. Overmars

Abstract

The motion planning problem asks for determining a collision-free path for a robot moving amidst a fixed set of obstacles. In this paper we present a new approach that combines a neural network and deterministic techniques to solve this problem. We define a colored version of a Kohonen network, which consists of two different classes of nodes. The network is presented random configurations of the robot and, from this information, it constructs a road map of possible motions in the work space. This road map can then be searched to find a motion connecting given source and goal configurations of the robot. The algorithm is simple and general; the only specific computation that is required is an intersection check for two polygons. It has been implemented for planar robots allowing both translation and rotation, and experiments show that compared to conventional techniques it performs well, even for difficult motion planning scenes.

1 Introduction

The design of autonomous robots that are able to perform tasks in an unknown environment without human intervention gives rise to many problems. One of them is the *motion planning problem*:

Definition 1 *Given a robot R and an environment containing a fixed set of obstacles, find a path from some source configuration S to some goal configuration G, such that R can travel freely from S to G without colliding with any of the obstacles. A configuration consists of both a position and an orientation of the robot.*

In Figure 1, a typical planar motion planning problem is shown along with one possible solution. An L-shaped robot has to move from the bottom left to the top right, using both translation and rotation. The obstacles are shown in dark grey; the path from source to goal is indicated by a number of intermediate configurations of the robot.

1.1 Motion planning

The motion planning problem has received considerable attention over the past years. See [8] for an overview of the different techniques that have been developed so far. Existing methods can roughly be divided into three categories: cell decomposition methods, road map methods and potential field methods. These methods can act either on the *work space* of the robot, i.e., the space in which the robot and the obstacles are defined, or on the *configuration space*, the space consisting of all possible configurations of the robot. An advantage of the latter is that the robot reduces to a single point in configuration space. We will now briefly review the mentioned methods. In the following as well as in the rest of this paper, we assume that the robot R is a solid polygon, moving freely between the obstacles, which can also be arbitrary polygons. Furthermore, we assume that the robot allows for both translation and rotation, and that there are no constraints on the kind of motion the robot can perform.

*This research was partially supported by ESPRIT Basic Research Action No. 6546 (project PROMotion) and by the Netherlands Organization for Scientific Research (NWO).

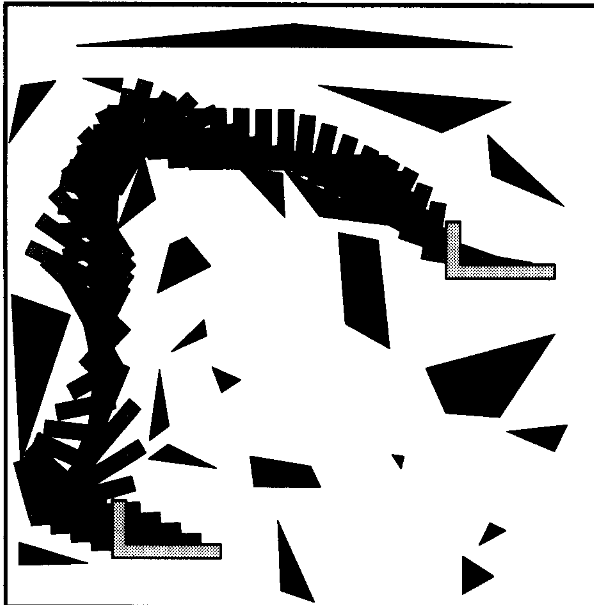


Figure 1: A typical planar motion planning problem.

Cell decomposition methods construct a path in the configuration space by dividing the *free configuration space*, i.e., the part of the configuration space where the robot does not collide with any of the obstacles, into simple cells; this can be done either exact or by approximation. Next, the cells are entered into a graph, in which adjacent cells are connected. Between such cells, a motion is usually easy to compute. A motion connecting given source and goal configurations can then be constructed by determining the cells that contain the source and goal configurations, finding a path between those cells in the graph, and finally computing a motion for each edge on the path.

Road map methods try to construct a graph of highways, along which it is safe to travel, between the obstacles. This can be done in either the work space of the robot or the configuration space. With such a graph available, a motion can be found by moving the robot from the source configuration to some position on a nearby highway, following the different highways to somewhere near the goal configuration, and finally leaving the highway and moving the robot to the goal configuration. An often-used road map is the Voronoi diagram, which can be defined as follows:

Definition 2 *Let $A = \{A_1, A_2, \ldots, A_k\}$ be a set of obstacles. The Voronoi diagram $Vor(A)$ of A is given by*

$$Vor(A) = \{x \mid d(x, A_{m_1}) = d(x, A_{m_2})\} \tag{1.1}$$

where $d(x, A_i)$ is the (Hausdorff) distance $d(x, A_i) = \inf\{d(x,p) \mid p \in A_i\}$ of x to A_i, and A_{m_1} and A_{m_2} are the two obstacles that are nearest to x with respect to this distance.

In other words, the Voronoi diagram consists of all points x for which the distance to the two nearest obstacles is equal. A Voronoi diagram in the configuration space of the robot has the nice property of maximizing the clearance when the robot travels along it. Related work has recently been described by Martinetz [9], who uses a Kohonen network to determine a Delaunay triangulation of a set of points. The Delaunay triangulation is the dual of the Voronoi diagram, in the sense that if for some point p the nearest obstacles are A_i, A_j, then A_i and A_j are connected in the Delaunay triangulation [10].

Both cell decomposition methods and road map methods construct a data structure that is used to find paths between configurations of the robot. An advantage of this is that the data

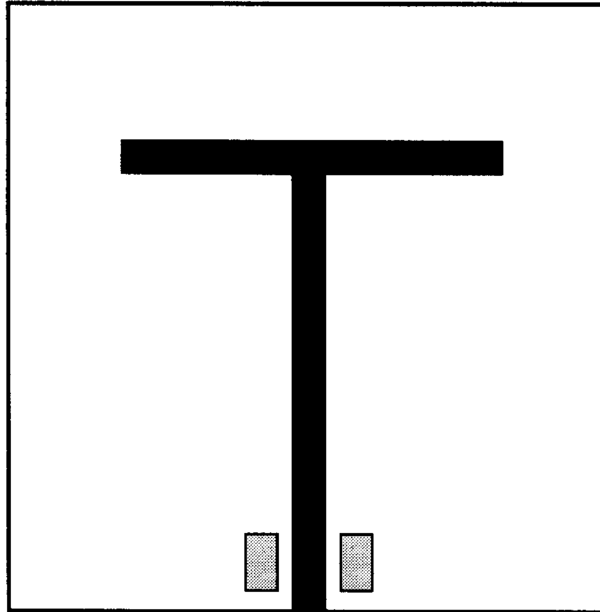


Figure 2: A problematic situation for potential field methods. Both the source and the goal configuration are shown.

structure needs to be computed only once, and can later on be re-used for different source and goal configurations. However, these data structures tend to be very large, and the geometric computations required are often difficult and expensive.

Potential field methods form a completely different approach. They only compute a motion for given source and goal configurations. They start at the source and try to move the robot in small steps towards the goal. The direction of these steps is determined by some force. The goal is assumed to produce an attractive force on the robot; the obstacles produce a repulsive force. The robot is thereby pulled towards the goal while being pushed away from the obstacles, thus avoiding collisions. The main problem with these methods is that for non-trivial problems, they often get stuck in local minima in which the different forces cancel each other. An example of such a situation is shown in Figure 2. A rectangular robot has to move from the left side of a T-shaped obstacle to its right side. The only way to do this is by first having the robot move away from the goal configuration towards the upper left hand corner, from where it can move to the upper right hand corner and from there on to the goal. However, a potential field method will always try to move towards the goal and, hence, will oppose to the only possible path in this particular situation. To overcome this problem, various techniques such as using backtracking, adding random Brownian motion, or modifying the potentials created by the obstacles, have been tried. Unfortunately, such techniques often are either very slow, or only applicable in limited situations.

1.2 Artificial neural networks

Artificial neural networks have recently been applied to various problems in the field of robotics, like the inverse kinematics problem [12] and the grasping problem [13]. However, there has not yet been much research in their use for the motion planning problem. Well-known types of neural networks include Hopfield, back-propagation and Kohonen networks. Because only the latter is of interest for our approach, we refer to [5] and [6] for detailed descriptions of the other types of networks.

A *Kohonen network* as introduced by Kohonen [7] builds up a mapping from an n-dimensional

vector space V to a (usually two-dimensional) graph G of nodes, where each node represents a position in V; this is done by attaching an n-dimensional *position vector* to every node $c \in G$. A common choice for V is the n-dimensional space of real numbers $\mathbb{R}^n$. Any vector $v \in V$ is mapped onto the node in G whose position vector best resembles v. This node is called *bmu* (best matching unit) and satisfies

$$\forall c \in G : |\mathrm{bmu} - v| \leqslant |c - v| \tag{1.2}$$

where $|\ldots|$ denotes a measure of similarity or distance in V. Possible choices for this measure are the *correlation* $C(x, y)$, the *direction cosine*, and the *Eucledian distance* (which is actually a measure of *dis*similarity); see [7] for a discussion of these measures. In our approach the Eucledian distance $\|\ldots\|$ is used, which is given by

$$\|x - y\| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1.3}$$

for two vectors $x = (x_1, x_2, \ldots, x_n), y = (y_1, y_2, \ldots, y_n)$, $x, y \in V$. Since $\|x - y\| = \|x\|^2 + \|y\|^2 - 2(x, y)$, this measure is sensitive to the lengths of the vectors being compared. Therefore often the vectors are normalized to unit length.

Now let P(X) be an unknown probability distribution on V from which we are able to take an arbitrary number of sample vectors. The purpose of a Kohonen network is to find values for the position vectors such that the resulting mapping is *topology-preserving*, meaning that adjacent vectors in V are mapped onto nodes that are adjacent in the graph. The network is trained to find such values for the position vectors in the so-called *learning phase*. It repeatedly receives n-dimensional input vectors (or *inputs* for short) that are chosen randomly from V according to P(X); every input v is mapped onto the bmu according to Equation (1.2). The process of determining the bmu is called a *competition* among the nodes. The position vectors w of the bmu and some nodes in a topological neighborhood of the bmu are then adjusted to the input, according to Kohonen's learning rule [7]:

$$w^{\mathrm{new}} = w^{\mathrm{old}} + \varepsilon (v - w^{\mathrm{old}}) \tag{1.4}$$

ε ($0 < \varepsilon < 1$) is called the *learning coefficient* and often depends on the distance to the bmu. In addition, the size of the neighborhood is often initially large and decreasing in the time.

In the following we will sometimes refer to the position vector of a node as its *position*, and to an adjustment of the position vector as a *movement* of the node. However, one should keep in mind that no actual movement of the node takes place; its position in the graph G remains unchanged throughout the algorithm. After a sufficiently large number of competitions, the position vectors constitute a topology-preserving mapping of V onto G [7]. With some adaptations, the resulting mapping is also distribution-preserving for a function $P'(X)$ that is similar to $P(X)$ ([1, 7]); *distribution-preserving* means that, for a random vector according to P(X), each node has an equal probability of being bmu.

Fritzke [4] described a variant of a growing Kohonen network which is able to map itself onto those parts of the space where the probability of an input is not zero. *Growing* means that the network is extended during the learning phase by adding nodes to its graph, which allows to gradually increase its precision where needed. A possible way to use this for the motion planning problem is by generating random inputs and discarding all inputs that are located in the *forbidden configuration space*, i.e., the space consisting of all configurations in which the robot collides with the obstacles. This would cause the network to be mapped onto the free configuration space. However, when experimenting with this idea we discovered that the number of nodes needed to cover the whole of the free configuration space with a satisfactory precision becomes too large to be practically feasible, especially for problems involving a robot with three degrees of freedom.

A possible way to reduce the number of nodes needed is by having the network approximate only a road map in the configuration space. We used this idea to construct an extension of a

Kohonen network that approximates a Voronoi diagram in configuration space by learning both a representation of the obstacle boundaries and a collection of paths between those boundaries (both in configuration space). The way in which this is done takes advantage of the fact that, unlike most applications of Kohonen networks, in the motion planning problem it is possible to obtain information other than that provided by the random input vectors. The input of the network consists of random configurations of the robot along with the information whether that particular configuration causes the robot to collide with the obstacles. However, this can be determined for any given configuration[1], in other words: the network can explicitly request certain vectors of V to be evaluated. As will be shown, this additional source of information makes it possible to combine the network approach with graph-based algorithms that can be used to improve the network.

The rest of this paper is organized as follows. In Section 2, we describe the approach. For easier understanding, the configuration space will be restricted to $\mathbb{R}^2$ in this section (i.e., planar motion planning problems involving only translation). Then in Section 3 we describe how the algorithm can be generalized to three (and higher) dimensional configuration spaces. Section 4 gives some experimental results on the efficiency of the method and the influence of certain parameters. Finally, in Section 5 we give some concluding remarks and indicate directions of future work.

2 The global approach

As mentioned in the introduction, we will construct a network that approximates a Voronoi diagram in the configuration space. For easier understanding, the configuration space will be restricted to $\mathbb{R}^2$ in this section. In other words, a configuration of the robot consists of a position (x, y) of the robot in the workspace. This is the configuration space for motion planning problems involving only translations of the robot. However, a generalization to higher-dimensional configuration spaces is straightforward and can be found in Section 3.

In the following as well as in the rest of this paper, we assume that the configuration space is bounded and normalized to $[0,1] \times [0,1] \subset \mathbb{R}^2$. The area outside this region is considered an obstacle.

2.1 The network structure

The network is basically a Kohonen network as described in Section 1.2, consisting of a planar graph G of nodes. Following Fritzke [4], the nodes are connected in such a way that the graph consists of triangles with one node in each corner. This is a flexible structure in which it is easy to add nodes without disturbing the connectivity of the network graph (how nodes are added will be discussed later on in this section). A node's position vector consists of a single configuration of the robot.

As an extension to the original Kohonen model, we divide the nodes of the network into two disjunct classes. Nodes whose position vectors are located in the free space are labeled *safe*; the other nodes are labeled *unsafe*. Safe nodes represent configurations that are safe to place the robot in (or *safe configurations* for short); they will be used to build a road map of paths along which the robot can travel. Unsafe nodes represent configurations that cause the robot to collide with at least one of the obstacles. They serve two purposes: at first they will build up an approximation of the obstacle boundaries, while in a later stage they are used to extend the road map given by the safe nodes.

Each node is initially given its correct label. This can be done by determining whether placing the robot at the configuration that is represented by the node's position vector causes it to collide with the obstacles. Assuming that the robot as well as the obstacles are given by a description of their polygons, this is an 'easy' test that can be executed in the two-dimensional workspace using an intersection checking routine for polygons; this also holds for three-dimensional configuration spaces, as will be shown in Section 3.

[1] by means of an intersection routine for planar polygons, in our setting.

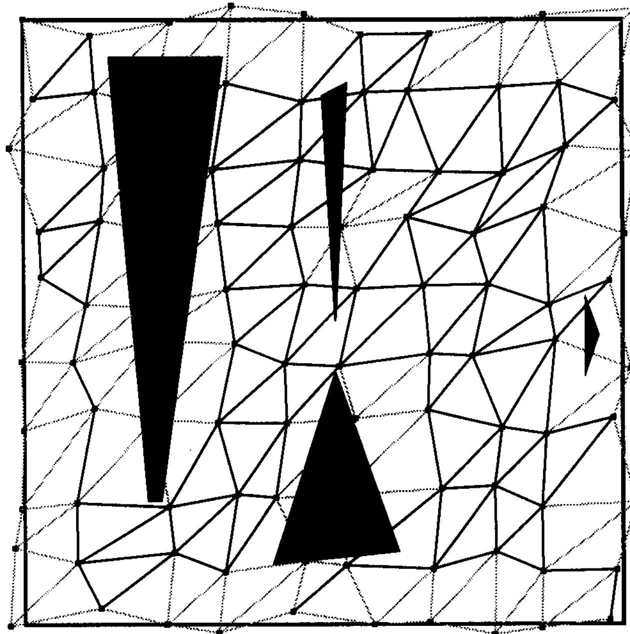


Figure 3: An initial situation of the network. The safe nodes are shown as filled squares, the unsafe nodes as hollow squares. Edges between safe nodes are drawn in solid lines; they represent the road map which will be searched for a path.

Once a node is classified, it is not allowed to violate its classification. E.g., if a safe node's position vector is adjusted to a configuration that causes a collision, we forbid this move and restore the position vector to its previous value. This is enforced to ensure that the label of a node always indicates its status with respect to the free space, so the robot can always safely be placed at a configuration that is represented by a safe node's position vector but never at a configuration represented by an unsafe node.

2.2 Initial position vectors

In the original Kohonen model, the position vectors of the nodes are initialized to random values. The underlying idea is that the learning process is able to find suitable values by itself. However, this does not entirely hold in our approach because the restriction that the nodes are not allowed to violate their classification restricts their position vectors to some connected region of the configuration space near their initial values. In order for the final state of the network to be topology-preserving, its initial state should also satisfy that condition. Furthermore, since no a priori information on the configuration space is known, the nodes should initially be distributed uniformly over the configuration space. The easiest way to satisfy this is by initializing the position vectors of the nodes to a regular grid of size $m \times m$ on the configuration space. To lessen the chance of this grid accidently being a favorable (or unfavorable) initialization, we also add a small amount of random variation to the initial position vectors. See Figure 3 for an example.

2.3 The learning algorithm

After the initialization, the network repeatedly performs so-called learning steps. As in the Kohonen model it receives random inputs, in our case consisting of a random configuration of the robot. With every input configuration (or *input* for short), its correct classification is given. Depending on

this information, only a subset of the nodes are allowed to compete for a given input. If the input is located in the free space, the competition only takes place among the unsafe nodes. Similarly, only the safe nodes compete for an input causing a collision. With this restriction, a bmu satisfying

$$\forall c \in G_{\text{label}} : \|\text{pos}(\text{bmu}) - v\| \leqslant \|\text{pos}(c) - v\| \tag{2.1}$$

is determined for each input v. Here label can be *safe* or *unsafe* (as discussed above), G_{label} is the set of nodes in G with the given label, and pos(c) denotes the position vector of node c. We use the Eucledian distance $\|\ldots\|$ as the measure of distance since this gives a good indication of how far apart two configurations are, in the sense that it represent the distance that the robot has to travel to get from the one configuration to the other.

The resulting bmu and its direct topological neighbors then adjust their position vectors w to the input, according to Kohonen's learning rule:

$$w^{\text{new}} = w^{\text{old}} + \varepsilon(v - w^{\text{old}}) \tag{2.2}$$

To have the unsafe nodes and the safe nodes behave differently, the learning coefficient ε depends on the label of the bmu. In the case of an unsafe node we take $0 < \varepsilon < 1$, where ε can be either ε_u or ε'_u, indicating the amount that the bmu and its neighbors adjust their position vector, respectively. In other words, unsafe nodes are 'pulled towards' inputs that are located in the free space, but do not adjust their position vectors to inputs that cause collisions. The closer a unsafe node is to an obstacle boundary, the more likely it will be bmu for an input on that 'side' of the obstacle, and thus the more likely it will be moved even closer to that boundary. The unsafe nodes will therefore move towards the free space. However, since they are not allowed to violate their classification, their position vectors will come to represent configurations close to the obstacle boundaries.

In the case of a safe node we take $-1 < \varepsilon < 0$, where ε can again be either ε_s or ε'_s (with a similar meaning). As a result, safe nodes are 'pushed away' from inputs that cause a collision, but do not adjust their position vectors to safe inputs.[2] This will move the safe nodes away from the nearby obstacles. Because of the combined influence of the different obstacles, the safe nodes will eventually settle to positions as far away from the obstacles as possible, thereby approximating positions on a Voronoi diagram in the configuration space. However, two problems arise by simply applying this. First, the further a safe node c_s is from the nearest obstacles, the longer the vector $v - w^{\text{old}}$ will be, and the further c_s will be moved by applying the learning rule. Similarly, safe nodes that are near some obstacle will be moved only little. This is just the opposite of the desired behavior; ideally, the distance that safe nodes are moved should be proportional to their distance from the Voronoi diagram. To approximate this, ε is made dependent on $|v - w^{\text{old}}|$ by taking $\varepsilon = \varepsilon_s/(1 + d \cdot |v - w^{\text{old}}|)$, where d is a constant > 0 (and similarly for ε'_s). Secondly, this learning rule would in the long run move all safe nodes to a single point, i.e., the point of the configuration space that is farthest from all obstacles. To prevent this and keep the safe nodes in place, the neighbors of a safe node produce an additional attractive force on the node. The learning rule for safe nodes thus becomes

$$w^{\text{new}} = w^{\text{old}} + \varepsilon(v - w^{\text{old}}) + \sum_{c \in N} \zeta\left(\text{pos}(c) - w^{\text{old}}\right) \tag{2.3}$$

where N denotes the set of all neighbors of the (safe) bmu, and ζ ($0 < \zeta < 1$) is the *learning coefficient* for the node's neighbors (again we take $-1 < \varepsilon < 0$). The attractive force of the safe neighbors tends to smooth the road map, while the force produced by the unsafe neighbors prevents the safe nodes from moving away from between the obstacles.

2.4 The conscience mechanism

A problem that occurs with the described learning algorithm is the following. If there are unsafe nodes on all faces of the boundary of some obstacle A_i, an unsafe node c that is located in A_i but

[2] This behavior of the safe nodes is much alike the Learning Vector Quantification mechanism [7].

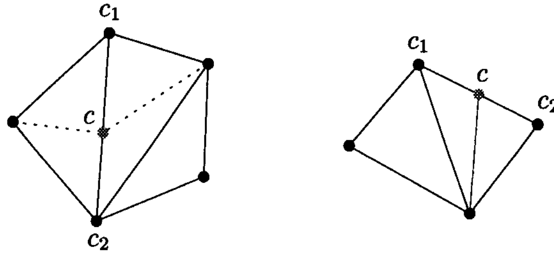


Figure 4: Adding a new node c between c_1 and c_2. (a) In most cases, c gets four direct neighbors. (b) If both c_1 and c_2 are located on the rim of the network graph, c is connected to only three nodes.

not near its boundary will not be bmu for any input v, since there always is another unsafe node in A_i (near its boundary) that is nearer to v than c is. Therefore c's position vector will never change and therefore the node is useless for the network's approximation. Note that this problem will not occur with the safe nodes since they are moved away from the input, thus decreasing the probability of being bmu again.

The sketched phenomenon is well-known with distribution-preserving networks. Among the solutions proposed in the literature is the use of a *conscience mechanism* [1]. In this mechanism a *bias* variable (initially set to zero) is attached to every (unsafe) node. The bmu, given by Equation (1.2), is then redefined for the unsafe nodes to the node which minimizes $\|\mathrm{pos}(c) - v\| +$ bias:

$$\forall c \in G : \|\mathrm{pos}(\mathrm{bmu}) - v\| + \mathrm{bias}_{\mathrm{bmu}} \leqslant \|\mathrm{pos}(c) - v\| + \mathrm{bias}_c \tag{2.4}$$

The bias of each unsafe node c is updated after every competition, according to

$$\mathrm{bias}_c = (1 - \beta)\mathrm{bias}_c + \beta i_c \tag{2.5}$$

where

$$i_c = \begin{cases} \gamma & \text{if } c \text{ is bmu} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

Thus the bias of the node that wins the competition is increased, while the other nodes' biases are decreased. For a large number of competitions, this tends to put a penalty on nodes that win relatively often, while favoring nodes that hardly win at all; β $(0 \leqslant \beta \leqslant 1)$ and $\gamma \geqslant 0$ are parameters indicating how long the bias remains effective and how heavily nodes are punished for being bmu, respectively.

2.5 Adding nodes

In addition to the learning process, nodes are added to the network to locally increase its accuracy. This allows for obtaining a high precision without having to start off with a large number of nodes. In order to maintain the triangular structure of the network graph, nodes are always added on edges in the network graph G. To add a new node c between two other nodes c_1 and c_2, the edge connecting c_1 and c_2 is split, and c is connected with c_1, c_2, and all common neighbors of c_1 and c_2 (see Figure 4); c is then initialized to its correct label. The two criteria for adding nodes (*error-based* and *scene-based*) are discussed below.

2.5.1 Error-based adding

Nodes should be added in places where the network needs improvement. Analogous to Fritzke [3], an *error* value is attached to each node in order to determine how good it approximates either an

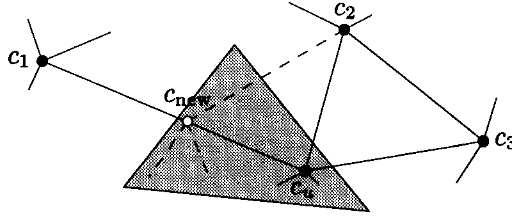


Figure 5: The unsafe node c_u is near the boundary of both safe nodes c_1 and c_2 but not near the boundary of c_3; adding a node c_{new} at the indicated position improves the networks approximation.

obstacle boundary (in case it is labeled unsafe) or the configuration space Voronoi diagram (in case it is safe). Every time a node is moved, we add the distance that it is moved to its error value.[3]

By keeping track of the error that is caused by a node c, we are able to determine where the network needs improvement. A high error value indicates that c is moved a lot, meaning that there are few nodes near c. Therefore we can locally improve the network by adding a new node near c.

This leads to the following *error-based* adding strategy: after every k learning steps we determine the node $\tilde{c}$ with the highest error value and add a node c_{new} between $\tilde{c}$ and one of its direct neighbors (which neighbor is chosen for this is discussed below). Because the two kinds of nodes are by no means equivalent, both a $\tilde{c}_s$ and a $\tilde{c}_u$ are determined, and both a safe and an unsafe node are added to the network.

It is desirable to keep the total error of the involved nodes constant by doing this, yet the error of $\tilde{c}$ should decrease. Also the error of c_{new} should not be initialized to zero but to some average error of the nodes it is connected to. This can be realized in the following way. Let C be the set of nodes that the new node will be connected to. The nodes in C contribute an equal part of their error to the newly added node:

$$\textit{error}_{c_{\text{new}}} = \frac{1}{|C|+1} \sum_{c \in C} \textit{error}_c \tag{2.7}$$

and in order to keep the total error of the involved nodes constant, the error values of the nodes in C are diminished by the appropriate amount:

$$\forall c \in C : \text{error}_c = \frac{|V|}{|V|+1} \text{error}_c \tag{2.8}$$

As a consequence of this error distribution, the nodes in C are less likely to have a new node added next to them in the future.

2.5.1.1 Adding strategy for unsafe nodes As explained in Section 2.1, the unsafe nodes are intended to approximate the obstacle boundaries in configuration space; therefore new unsafe nodes should be added near those boundaries whenever possible. However, we cannot directly test whether a given configuration satisfies this condition. We say that an unsafe node c_u is *near the boundary* of a safe node c_s (with respect to a given obstacle O) if the configuration on the edge $\overline{c_u c_s}$ at one quarter[4] from c_u is located in the free configuration space.

If c_u is not near the boundary of one of its safe neighbors, adding a node on the connecting edge is likely to improve the approximation (see Figure 5). Therefore we determine the farthest safe neighbor c_s of c_u and add a node at the configuration halfway the edge $\overline{c_u c_s}$ if that configuration

[3]Fritzke [3] adds the distance $|\text{pos}(c) - v|$ to c's error value every time it is bmu for an input v. However, in our approach an unsafe node that is located near an obstacle boundary constitutes a good approximation of it, yet it can be bmu for inputs that are far from it. The distance that c is actually moved more accurately describes the notion of the error that it causes in our situation.

[4]This is rather arbitrarily chosen.

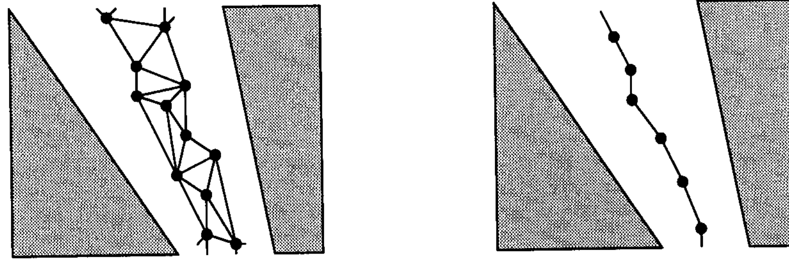


Figure 6: (a) A road map consisting of a structure of safe nodes. (b) A better road map consisting of a chain of safe nodes.

is located in the forbidden configuration space. If however c_u has no such safe neighbor, we try to improve the approximation of the part of the obstacle boundary that c_u is near by adding a node halfway c_u and its farthest unsafe neighbor which is located near the same part of the obstacle boundary. Note that this again can not be tested directly. As a heuristic we enforce that the nodes have a common safe neighbor.

2.5.1.2 Adding strategy for safe nodes Since long edges between safe nodes are not likely to constitute a good approximation of the Voronoi diagram in configuration space, it can be improved by adding a new node on long edges connecting two safe nodes. When doing this, care should be taken not to build up two-dimensional structures of safe nodes between the obstacles, as opposed to a one-dimensional chain of safe nodes (see Figure 6), since this would only complicate the road map without adding any information to it. By adding new safe nodes only on edges between safe nodes that have two common unsafe neighbors, we will build up at most one such chain between each pair of obstacles. If however a safe node has no such safe neighbor, it is not yet part of the road map and therefore a node is added on the edge to its farthest unsafe neighbor to try to connect it to the road map.

2.5.2 Scene-based adding

The error-based adding strategy tries to locally improve the network by keeping track of the error that is caused by a node. However, there are cases in which adding nodes this way is not sufficient. E.g., if there are no unsafe nodes located in a given obstacle O, there will never be added one. Similarly, a passage between two obstacles without any safe nodes inside it will never be included in the road map. In both cases the information given by the network is incomplete, i.e., does not account for possibly important parts of the configuration space. Especially in the case of the road map this poses a real problem because small passages often play an important role in solving motion planning problems, but are not likely to initially have safe nodes added in them.

In order to account for small obstacles and passages in configuration space, we will have to detect them during the learning phase. Again the limited source of information does not allow us to directly detect the occurrence of either problem, therefore we use an heuristic argument to detect an incomplete approximation of the obstacles. If a safe node is pulled into the forbidden configuration space by the attracting force of one of neighbors c, an unsafe node is added at the collision configuration. This can be justified as follows. If c is safe, there obviously is an obstacle located between the nodes, and adding an unsafe node cuts this invalid edge of the road map. If c is unsafe, the safe node must be close to an obstacle. However, the force that is produced by an unsafe goes to zero with decreasing distance to the node, so c is either not close to the obstacle boundary or located in a different obstacle. In both cases adding a node will improve the network.

Similarly, to be able to construct a path the road map of safe nodes should be connected, and should not leave out parts of the free configuration space. If the edge between two unsafe nodes is partly located in the free configuration space, a safe node should be added on that edge.

This is heuristically determined by sampling a small (three in our implementation) number of configurations on the edges from an unsafe bmu to its unsafe neighbors. If any such configuration is located in the free configuration space, we add a node there.

2.6 Removing redundant edges

The nodes of the network remain fully connected in triangles throughout the algorithm. However, only a subset of the edges are useful for finding a path. In particular, the edges connecting unsafe nodes that are located in the same obstacle can never be part of any path and thus contain redundant information. To decrease the complexity of the network graph, these redundant edges are removed in the following way. If two unsafe nodes are connected, we check whether they have a common safe neighbor. If this is the case, the nodes are probably located near the same part of the obstacle boundary and the edge connecting them might contain useful information for the second stage; therefore the edge is kept. If however the unsafe nodes do not have a common safe neighbor, the edge connecting them is removed from the network graph. If this causes an unsafe node to loose all of its edges, it has become redundant and is itself removed from the network.

2.7 The second stage

As already mentioned in Section 2.1, besides approximating the obstacle boundaries, the unsafe nodes are also used to extend the road map. This is done by allowing them to move across the obstacle boundaries and into the free space. A way in which to do this is by prolonging the learning phase with a second stage in which we allow the unsafe nodes to actually move into the free space. Assuming however that they are located near the obstacle boundaries and that the safe nodes are located somewhere near the Voronoi diagram between the obstacles, this can be done in a simpler and much faster way by simply moving every unsafe node to a position in between its direct safe neighbors. To this end, every unsafe node c in the network calculates a local vector $\vec{m_c}$ indicating the direction to a configuration in between its safe neighbors:

$$\vec{m_c} = \sum_{b \in BN(c)} (\mathrm{pos}(b) - \mathrm{pos}(c)) \tag{2.9}$$

where $BN(c)$ indicates the set of direct safe neighbors of node c. It then adds $\vec{m_c}$ to its position vector. If the resulting configuration is located in free space, the node can be considered equivalent to a safe node and can be treated in exactly the same way, thus extending the road map given by the safe nodes.

2.8 Planning a motion

Knowing that the safe nodes approximate a Voronoi diagram in the free configuration space, we can use these nodes to plan a motion between given configurations S and G as shown in Algorithm 1.

Algorithm 1 FindMotion(S,G)

1 add safe nodes v_0 and v_k to the network graph.
2 set the position vector of v_0 to S.
3 set the position vector of v_k to G.
4 find a sequence of safe nodes $S = v_0, v_1, \dots, v_k = G$ such that the robot can travel freely from v_i to v_{i+1} $(0 \leqslant i < k)$ without colliding with any of the obstacles.

A sequence of nodes as in Step 4 of the algorithm can easily be found using an A^* algorithm [2] on the safe nodes; this also allows for determining the *shortest* path with respect to the network. However, when such a sequence has been found it remains to check whether the robot can freely travel along this path, since we can not guarantee that the robot will not collide with the obstacles when travelling along an edge between two safe nodes. This is done by trying to construct a

motion between each pair of successive nodes using a simple motion planning algorithm. There is a trade-off here; a simple algorithm (e.g., a straight move towards the goal node) will be easy and fast to compute, whereas it will often fail to find a motion, especially if the robot has only little freedom to move, e.g. in small passages. On the other hand, a more powerful algorithm will nearly always be able to construct a motion if one exists, but it will be a lot slower. In our implementation, only the linear interpolation type of motion is used. Experimental results can be found in Section 4.

2.9 Establishing a stopping criterion

Typical criteria for determining whether a neural network has been trained well enough are often based on the total summed error that is made by the network. If this error is less than some threshold value, the network is considered to be trained well enough. This could be used in a straightforward way in our algorithm because the error that is caused by a node c has already been defined in Section 2.5.1.

However when experimenting with this criterion, we found that if this threshold is chosen too large, the learning phase is often aborted prematurely, i.e., no motion could be constructed, whereas if the threshold was lowered the time taken to get the total summed error below this threshold by far exceeds the time taken to be able to construct a motion. This can be explained by considering the complexity of the free configuration space, especially for motion planning problems involving rotational degrees of freedom. To be able to construct a path the approximation of the configuration space obstacles and the road map need not be very accurate, since the robot can usually stay relatively far from the obstacles during motions.

Instead we decided to use a simple criterion by trying to find a path as described in the previous section every k learning steps. If a path is successfully found we are done, otherwise the learning phase is continued.

2.10 Summary of the algorithm

Algorithm 2 summarizes the approach described in this section.

Algorithm 2 MOTIONPLANNING(S,G)

```
1  initialize the network to a grid on the configuration space.
2  repeat forever:
3      generate a random input v.
4      adjust the position vectors of the bmu and its topological neighbors to v.
5      every n_add steps:
6          add nodes to the network according to the error-based and scene-based adding strategies.
7          remove redundant edges and nodes from the network graph.
8      every n_stop steps:
9          perform the second stage by moving the unsafe nodes into the free space.
10         FINDMOTION(S,G).
11         if successful return success.
12         else undo the changes made in the second stage.
```

3 Higher-dimensional configuration spaces

The approach described in the previous section can easily be generalized to motion planning problems with more degrees of freedom of the robot, resulting in higher dimensional configuration spaces. Since no assumptions on its dimension were made in the previous sections, the generalization is straightforward. First of all, the n-dimensional configuration space should be normalized to $[0,1]^n$. The position vectors of the nodes as well as the inputs should be made n-dimensional.

Because of the way nodes are added to the network, it will remain two-dimensional if initialized as such. In order to give an approximation in higher dimensions, the network should be initialized to an n-dimensional grid on the configuration space.

Care should be taken with dimensions that represent rotations of the robot, because of the so-called *wrap-around* in these dimensions: two orientations of the robot which differ only by a rotation of $2k\pi$ for $k \in \mathbb{Z}$ are considered equivalent. Therefore, nodes on the different sides of the network graph in this dimension should also be connected.

Finally, a distance measure has to be defined for configurations in a higher-dimensional space. The most obvious choice is the generalized Eucledian distance given by (1.3). However, the role of rotation is more subtle than that. In the case of a 'roundish' robot (e.g., a circle or a square), rotation without translation is normally possible. Therefore the rotational component of the distance should not add too large a factor to the total distance; a configuration at the same position but rotated over an angle of π should be much nearer than a configuration with the same orientation but further away. On the other hand, if the robot is relatively long and thin, the rotational component is important because the sweep volume obtained by rotating such a robot is relatively large. In order to compensate for this, the rotational component is multiplied by the distance that the point that is farthest from the origin of the robot travels. The measure of distance thus becomes

$$\|x - y\| = \sqrt{\sum_{i=1}^{k}(x_i - y_i)^2 + \sum_{i=k+1}^{n} r \cdot \Delta(x_i, y_i)} \tag{3.1}$$

for $x = (x_1, x_2, \dots, x_n), y = (y_1, y_2, \dots, y_n) \in [0,1]^n$, where the first k dimensions represent translational, and the last $n - k$ dimensions rotational degrees of freedom. Furthermore, r is the distance of the point of the robot that is farthest from its origin, and $\Delta(x_i, y_i)$ is the difference in the ith dimension, taking the wrap-around into account.

This generalization has been implemented for three-dimensional configuration spaces (i.e., a planar robot which is allowed to translate and rotate). Experimental results obtained with the implementation are described in the next section.

4 Experimental results

Because the algorithm presented in this paper is highly heuristic, it seems very difficult to give any theoretical bounds on its efficiency. In this section we will present some results that were obtained in some typical settings of the motion planning problem to justify our claims on the performance of the algorithm.

4.1 The implementation

The algorithm was implemented in C/C++ on a SiliconGraphics Indigo workstation, which is based on an R3000 processor running at 33 MHz and is rated on the SpecMarks benchmark with 24.2 SPECfp92 and 22.4 SPECint92. The implementation itself consists of approximately 3700 lines of source code, which includes a nice graphical interface and routines for visualization of the network and the resulting motions of the robot. The interface (Figure 7) is based on the Forms library for SiliconGraphics workstations, which was written by Mark Overmars and is ftp-able from `archive.cs.ruu.nl`. For the visualization we used the PlaGeo library, which provides optimized routines for planar geometry, and the RobSupport library, which contains motion planning-specific routines. Both libraries were written by Geert-Jan Giezeman and are ftp-able from `archive.cs.ruu.nl`. The motion planning scenes depicted in this paper were generated by RobSupport.

The program deals with planar motion planning problems involving both translation and rotation of the robot. The configuration space is therefore three-dimensional and is normalized to $[0,1] \times [0,1] \times [0,2\pi[$. The space outside this area is regarded as an obstacle.

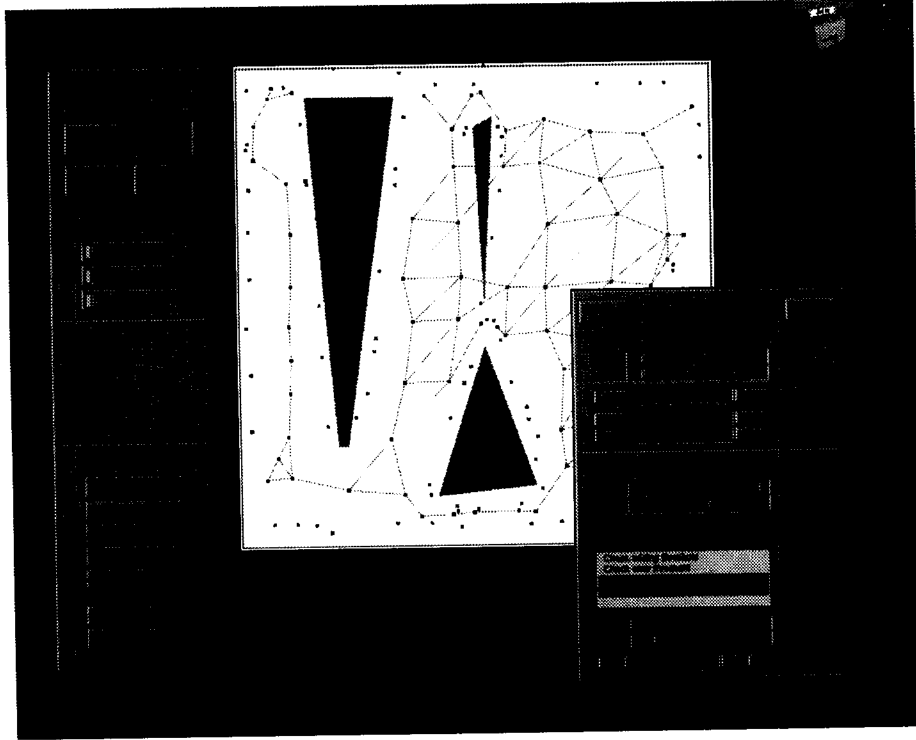


Figure 7: The interface to the program.

A number of parameters control the behavior of the program and can be adjusted for particular scenes:

Planar grid size : the size of the initial grid on the configuration space in both the x- and the y-dimension, denoted m_{xy}. Increasing this value allows for a finer approximation by the network. The default value is 10.

Rotational grid size : the size of the grid in the θ-direction, denoted m_θ; in scenes for which a lot of rotation is required to move the robot this value should be increased. Setting this to 1 disables the rotational degree of freedom, thus restricting the movements of the robot to pure translations. The default value is 4.

Learning parameters : the parameters of the learning algorithm itself (i.e., ε_s, ε'_s, ε_u, and ε'_u) and those of the conscience mechanism (i.e., β and γ) are experimentally set to fixed values for which the program seemed to obtain the best results. Choosing them completely different resulted in worse behavior of the network; small deviations from these default values however have hardly any influence on the network's performance. The fixed values for these parameters are:

parameter	ε_s	ε'_s	ε_u	ε'_u	β	γ
value	0.25	0.15	0.05	0.02	0.001	15.0

4.2 Various scenes

To determine the performance of the program, we tested it on a number of typical motion planning problems. The times shown in the table below are given in seconds and are averaged over 100 runs of the program; they have been optimized in the sense that the parameters were adjusted to good

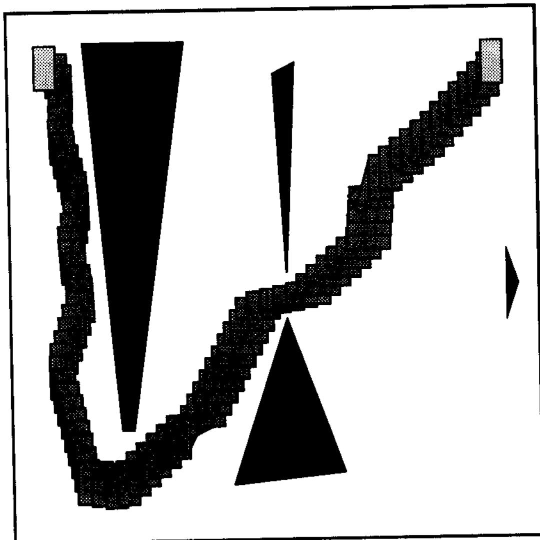


Figure 8: Scene 1.

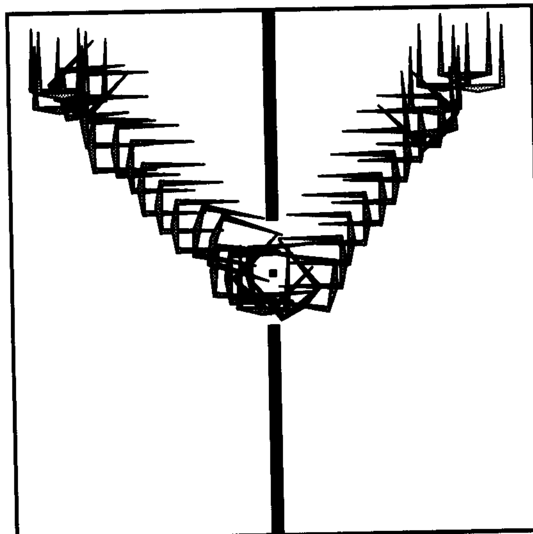


Figure 9: Scene 2.

values for each individual scene. As discussed in Section 2.8, finding a partial motion between two nodes is done only by linear interpolation on their position vectors. we expect that a more powerful algorithm, such as, e.g., the one used by Overmars [11], will provide a significant speed-up.

scene	m_{xy}	m_θ	average	minimum	maximum
1	10	4	4.97	3.45	6.43
2	13	4	6.32	4.75	20.89
3	10	4	4.17	3.33	5.68
4	10	8	11.95	7.95	27.30
5	12	8	16.09	12.37	78.72
6	12	8	20.08	5.71	82.19

The test scenes, together with a typical path computed by the program, are shown in Figures 8 to 13. Each of the test scenes has its own peculiarities which we will now briefly describe.

1. The first scene consists of only few obstacles; still rotation is required to move through any of the three 'holes' in the middle part of the scene. Potential field methods tend to get stuck behind the large obstacle at the left. Cell decomposition and road map methods should have no problems in solving this problem.

2. The second scene leaves much room to move on both sides of the obstacles, but there is only a very small passage in which the robot has to rotate around a small obstacle in order to move from left to right. Again potential field methods will have some trouble here.

3. This scene is quite trivial for cell decomposition and road map methods. No rotation is required to move around the obstacles; because the scene is very simple however, we tested it with rotational freedom anyway. Again potential field methods will have great difficulty in finding a path here.

4. This scene contains a large number of small obstacles and hence is difficult for both cell decomposition and road map methods because of the complexity of the free configuration space. A potential field method however will have no difficulties in finding a path here.

5. This scene is problematic for many approaches because there is little freedom to move the robot and a lot of rotation is required to find a path, which is made even more difficult by the long thin shape of the robot.

Figure 10: Scene 3.

Figure 11: Scene 4.

6. Although the routes along the outside seem more promising at first sight, the robot has to move through the region in the center of the scene in order to get from the top left to the bottom right. Some rotation is required to move around the obstacles in this region.

4.3 Parameter values

The 'optimal' values for the various parameters were obtained by testing the program for different settings. In the first series of tests, the rotational grid size m_θ was fixed for each scene. The table below shows the time (averaged over 100 runs) to find a path for some typical values of m_{xy}:

scene	m_θ	10	12	14	16	18
1	4	4.97	6.45	7.41	9.33	10.94
2	6	92.47	53.47	19.94	15.66	17.58
3	6	4.17	4.94	6.15	8.37	8.60
4	6	15.89	13.59	13.99	15.89	18.18
5	8	15.77	16.09	17.44	19.76	24.41
6	6	27.25	14.74	16.26	13.07	15.07

From these results it can be seen that setting m_{xy} to higher values can provide a significant speed-up, especially for scenes in which a fine precision is required to move the robot. Too large values however slow down the algorithm because of the increasing size of the network.

In the second series of tests, we fixed the planar grid size and tested the program for some values of m_θ; again the times are averaged over 100 runs of the algorithm.

scene	m_{xy}	4	6	8	10
1	10	4.97	6.63	7.17	9.12
2	13	6.32	9.08	11.11	13.20
3	10	4.17	5.90	6.84	7.91
4	10	16.56	15.89	13.41	15.00
5	12	17.14	15.78	16.09	18.02
6	12	20.08	14.74	14.78	19.10

Surprisingly, the value of m_θ does not have a large influence. Only for Scenes 4, 5, and 6, where a lot of rotation is required the performance initially increases for larger values of m_θ. This can

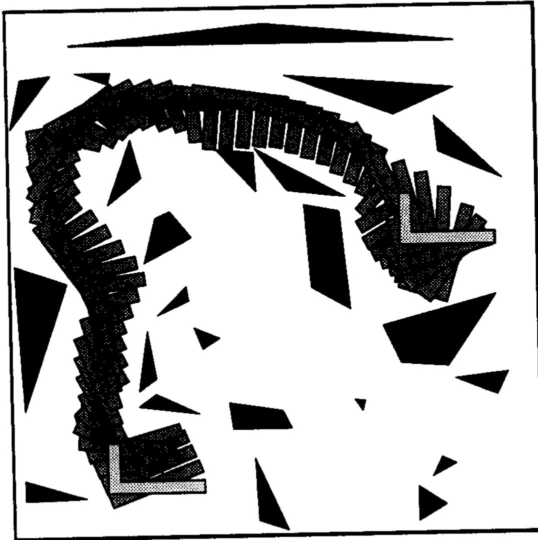


Figure 12: Scene 5.

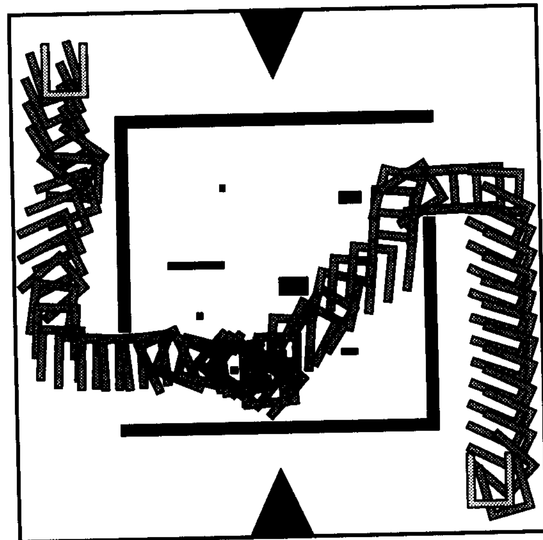


Figure 13: Scene 6.

be explained by considering the size of the rotating robot in terms of the size of the configuration space. A full rotation of an average size robot occupies only a small part of the configuration space when compared to e.g., moving it from $x = 0$ to $x = 1$. Therefore an initial grid size of 4 in the θ dimension already allows for a better precision in that dimension than a grid of size 10 in the x and y dimension.

4.4 Convergence

To test the learning behavior of the network, 100 source and goal configurations were randomly chosen. The *success rate* for a scene is defined as the relative number of these configurations between which the program is able to construct a path. Note that this can only be equal to 1.0 if the configuration space consists of only one connected component.

Figure 14 shows the success rate for Scenes 1, 4, and 5, set against the number of learning steps. The results for Scene 1 are indicated with a dotted line. The networks performance very quickly converges towards a success rate of 1.0, indicating it perfectly 'knows' the whole scene. In the case of Scene 5 (shown with a dashed line), the network converges quickly, but towards a success rate of only 0.95. This can be explained by the fact that the scene contains configurations from which the robot cannot escape (e.g., the small part of free space in the upper right corner). The same applies to Scene 4 (shown with a solid line) where the performance converges towards approximately 0.95. In this case, the high complexity somewhat lessens the convergence speed.

5 Conclusions and future work

In this paper we described a new approach to the motion planning problem. Motivated by the recent success of neural network-based approaches in a number of related problems in the field of robotics, we described a generalization of a Kohonen self-organizing network which is used to approximate a Voronoi diagram in the configuration space for a given robot and a given set of obstacles. This diagram is then used to find a shortest path connecting arbitrary *source* and *goal* configurations of the robot, by combining the network with traditional graph search algorithms.

There are a number of advantages to this approach. First, it is general; the only information that is required is whether the robot in a particular configuration intersects an obstacle. Therefore it could easily be adopted to other kinds of robots. Also the obstacles need not be polygonal, a requirement which is enforced in the majority of motion planning algorithms. Secondly, it is easily

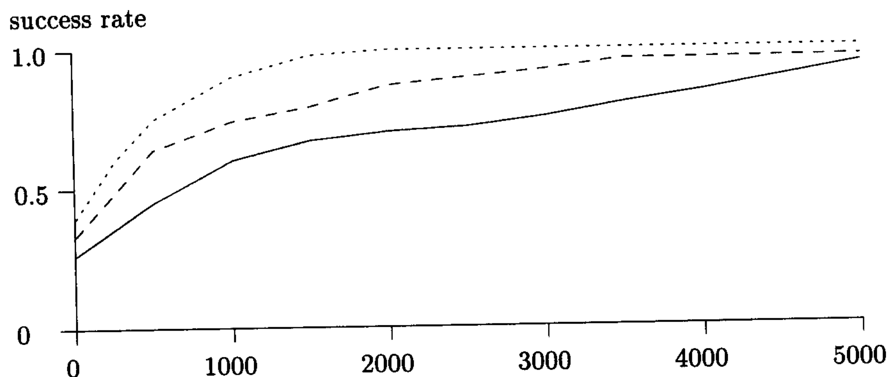


Figure 14: The success rate for three different scenes, set against the number of learning steps.

generalizable to higher-dimensional configuration spaces without major changes of the algorithm. Furthermore it is quite fast; it typically takes only a few seconds to compute a motion, and we expect that a more careful implementation can improve this even further. Finally, the resulting motions are relatively short and smooth, thus eliminating the need for additional (time-consuming) path smoothing.

The generalization of the method to higher dimensions should be straightforward. However, it could still be improved in a number of details, such as a more powerful algorithm for finding a partial motion between two nodes. Also a complexity analysis would be interesting.

References

[1] D. Desieno. Adding a conscience to competitive learning. In *Proc. Int. Conf. Neural Networks*, volume 1, pages 117–124. IEEE Press, New York, July 1988.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] B. Fritzke. Let it grow — self-organizing feature maps with problem dependent cell structure. In T. Kohonen, M. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, volume 1, pages 403–408. North-Holland, Amsterdam, 1991.

[4] B. Fritzke. Unsupervised clustering with growing cell structures. In *Proc. IJCNN-91*. Seattle, 1991.

[5] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1989.

[6] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[7] T. Kohonen. *Self-Organization and Associative Memory*. Springer Verlag Heidelberg, 1984.

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

[9] T. Martinetz. Competitive hebbian learning rule forms perfectly topology preserving maps. In *Proc. Internat. Conf. Artificial Neural Networks (ICANN)*, pages 427–434, 1993.

[10] Atsuyuki Okabe, Barry Boots, and Kokichki Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, 1992.

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

[12] H.J. Ritter, T.M. Martinetz, and K.J. Schulten. *Neural Computation and Self-Organizing Maps.* Addison-Wesley, 1992.

[13] P.P. van der Smagt, B.J.A. Kröse, and F.C.A. Groen. A self-learning controller for monocular grasping. In *Proc. 1992 IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, pages 177–182. Raleigh, N. C., June 1992.