On Using a Manhattan Distance-like Function for Robot Motion Planning on a Non-Uniform Grid in Configuration Space. *

Carl Van Geem
RISC-Linz
Research Institute for Symbolic Computation
Joh. Kepler University
A-4040 Linz, Austria
Internet: Carl.Van.Geem@risc.uni-linz.ac.at

Abstract

The general Robot Motion Planning Problem, to find a collision-free path from a given start position to a given goal position amidst obstacles, can be extended in several ways, e.g. by taking into account kinematics and dynamic constraints, by allowing obstacles to move slightly, by asking for a path that can be seen as a good pay-off between safest and shortest path, by planning with uncertainty,... All these extensions are usually treated separately from each other in the literature. We will consider a combination of some of these extensions.

We propose an approach using a Manhattan Distance-like Function on a non-uniform grid that models the space of all Free Positions in the Configuration Space of the robot.

The path from start to goal position follows the descent of this potential field and visits the shortest sequence of cells between them.

The non-uniformity of the grid keeps the number of cells low and makes a repulsive potential superfluous. The grid can be represented by a generalisation of the quad-tree data structure.

Keywords: Robotics, Computational Geometry, Representation, Quadtree.

*Submitted to the International Journal of Flexible Automation and Integrated Manufacturing (FAIM), on September 19, 1994.

1 Introduction

The general Robot Motion Planning Problem, to find a collision-free path from a given start position to a given goal position amidst obstacles, can be extended in several ways, e.g. by taking into account kinematics and dynamic constraints, by allowing obstacles to move slightly, by asking for a path that can be seen as a good pay-off between safest and shortest path, by planning with uncertainty,... All these extensions are usually treated separately from each other in the literature. We will consider a combination of some of these extensions. A nice overview of the current state of the art can be found in [5] and [4].

We propose an approach using a discrete potential function on a non-uniform grid that models the space of all Free Positions in the Configuration Space (Cspace) of the robot.

The Cspace of a robot is usually high dimensional. The number of cells of a uniform grid on the Cspace depends exponentially on the dimension. Using a non-uniform grid, fine nearby obstacles and coarse elsewhere, reduces the number of cells as much as possible. Small changes in the position of an obstacle will effect only the cells nearby, which might be divided into smaller cells or joined together.

On this non-uniform grid, we build a potential field that looks similar to the Manhattan Distance on a uniform grid. To each cell c of the grid, we assign a value of the potential field that expresses the number of cells that have to be visited in the shortest sequence of cells between c and the goal position. The path from the start to the goal position follows the descent of this potential field.

This approach seems to be suitable for a combination of possible extensions to the general Robot Motion Planning Problem. The descent of the potential function is steeper along cells of bigger size, and thus the robot tends to move away from the obstacles to 'safe' cells. But the approach also leeds to a 'fast' path through the Configuration Space, due to the definition of the Manhattan Distance-like function. However, this approach does not have the main disadvantage of the Potential Field Approach as known in the literature, since local minima do not occur for this Manhattan distance-like function. Furthermore, this approach delivers a complete solution (i.e. always finds a path if there is one and reports failure otherwise if the space is bounded). Since the representation is done by a non-uniform grid, the size of the cells along the path gives us some information about the allowed error during the execution of the path. So also uncertainty seems to be incorporated in a natural way in this approach.

The potential field approach on a grid leads in a natural way to parallel implementation and the computation of the potential can be done with a technique known as 'wavefront expansion' (see [5], [2]).

In the next section, we introduce the well known notion of 2^k-tree and two representations for it. In Section 3, we describe the problem tackled in this article. In Sections 4 and 5, we explain the way in which the problem is solved. In Section 4 we describe our algorithm to compute the neighbours of a cell in the non-uniform grid. In Section 5 we give our algorithm to compute the Manhattan distance-like potential function. In Section 6, we briefly describe an implementation of these algorithms and discuss some experimental results. Finally we conclude with some considerations for future work on the approach presented here.

2 Representation of a Non-Uniform Grid

In modelling, the 2^k-tree is a well known technique for representing objects in k-dimensional space (called quadtree for $k = 2$ and octree for $k = 3$). Its properties are well studied and discussed in the literature (e.g. in [6]). There are several ways to represent the structure of such a tree by lists of numbers. We are concerned with two of these representations (presented in [3] and [1]).

2.1 2^k-Tree

A quadtree is a tree representation of a partition of an area into smaller subareas, possibly of different sizes. Assume a rectangle divided into smaller rectangles as in Figure 1, where the partitioning is done by cutting a rectangle into 4 smaller rectangles of equal size.

We represent this partitioning by a tree, where each node corresponds to a particular rectangle. The root corresponds to the original rectangle. Each child of a node n corresponds to one of the 4 smaller rectangles in which the rectangle corresponding to n was divided. The quadtree corresponding to the partitioning in Figure 1, is depictured in Figure 2.

An octree is the straightforward generalisation of the quadtree to 3 dimensions. Here one starts with a cube and partitions it into 8 smaller cubes of equal size. In this case, every parent node will have 8 children. A 2^k-tree is the straightforward generalisation of the quadtree to k dimensions.

2.2 IG-tree Representation

In [3], a representation of octrees is described, which was a generalisation of a representation of quadtrees also due to Gargantini. We will refer to this representation in the rest of the paper by the name *IG-tree representation*. With this representation, neighbouring cells are easy to find. Once having the concept of IG-trees in 2D, it is straightforward to see how it generalises to higher dimensions. So, we next give the IG-tree representation for a quadtree.

For the IG-tree representation of a quadtree we assume that the children of a node are always listed from left to right according to the relative position of the corresponding partitioning. We can denote the cells as in Figure 3, where NW, NE, SW, and SE stands for north-west, north-east, south-west, and south-east respectively, and we assume that the first child of a node in the tree always corresponds to the NW-cell, the second child to the NE-cell and so on. The tree in Figure 2 is already in this format.

Let us number the children of each leaf from left to right, with 0 (NW), 1 (NE), 2 (SW), and 3 (SE). We now represent a leaf in the tree by a tuple

$$(a_0, a_1, ..., a_h),$$

where h is the height of the tree and each of the a_i is an element of the set $\{0, 1, 2, 3, X\}$. Roughly stated, an X is used to indicate leaves in the tree with height less than the height of the tree. In the following we will call such a tuple a *code* for the corresponding cell. When we go from the root to a leaf following branches of the tree, we will in general pass by other nodes. On each level i, we set a_i to the value corresponding to this intermediate node. When we reach the leaf, we fill up the tuple with symbols 'X', i.e. when a leaf has height m in the tree and $m < h$, then for all $i > m$, we set a_i to X. Doing so for the tree of Figure 2, we get the tuples as depictured in Figure 4.

The generalisation to octrees is now done by introducing a new direction. A cubic cell will be partitioned in 8 smaller cubes. The 4 cubes on the 'front' can be encoded by NWF, NEF, SWF, and SEF, the 4 cubes on the 'back' by NWB, NEB, SWB, and SEB. We assume that the children of the nodes are always ordered according to the sequence NWF, NEF, SWF, SEF, NWB, NEB, SWB, and SEB, and we number the children from left to right by 0, 1, ..., 7. The IG-tree representation is then obtained by proceeding in the same way as for the quadtree.

The same construction can be done for 2^k-trees, coming from $k-1$ to k dimensions, where we need k directions and 2^k possible values for a_i (besides 'X'). Further we will consider this general case also. The size of the data however, grow very rapidly for increasing k.

The partitioning is then represented by a list of codes, one code for each cell.

2.3 GET Representation

Another, but very compact, representation for quad- and octrees, the so called *GET*, has been invented (see [1]). Starting from the tree, where we assume as in the previous that the children of each node are ordered according to the position of the respective cells, here one constructs two lists with binary entries. The first list, called *DFA*[1], is constructed by running through the tree in a breadth-first transversal order. When passing through a node, a '0' is added to the list when the node is a leaf, and a '1' when the node has children. In the case of the tree in Figure 2, we get the following DFA:

1 1010 0001 0000 0000.

In fact, we can begin with recording on the level (height) in the tree where the first leaf will be encountered, i.e. we can drop the leading '1' in the example above, since it represents the root. In the following, we will assume that the DFA does only start recording on the level where the first leaf is encountered. In particular, we will always drop the first entry, representing the root. We will call an element ('0' or '1') in the DFA-list an *entry*.

As the spaces in the DFA of the example insinuate already, a way of looking at the DFA corresponding to a quadtree is by taking parts of length 4 out of this representation of the quadtree. We will call these parts of length 4 *4-tuples* of entries. Indeed, one sees that the first '1' in the list (say, representing node n) gives rise to a new 4-tuple of entries representing the children of node n. This new 4-tuple is in this case the second 4-tuple in the DFA. The second '1' in the DFA gives rise to the third 4-tuple, the third '1' to the last 4-tuple.

Let us now look to the case of higher dimensions. In the case of an octree, the DFA can be seen as a list of tuples of length 8 and the two previous remarks will also hold in this case. For a 2^k-tree, moreover, we will look for tuples of length 2^k.

Note that the whole partitioning is now represented by this one DFA-list. The second list, called *DFB*[2], is not of any importance when we are only interested in the partitioning of an area. It describes for each leaf, i.e. for each zero in the DFA, whether the cell is coloured black or white, e.g. 'inside' or 'outside' an obstacle when we consider robot environments. The m-th entry in DFB corresponds to the m-th zero in the DFA, and black or white can be represented also with one digit

[1] Stands for 'Datafield A' in [1].
[2] Stands for 'Datafield B' in [1].

(1 or 0). The DFB list does not play a role in the partitioning of the Cspace, but will be needed in the motion planning. The potential field will be kept in a list similar to the DFB list (see below).

Looking more closely to one of the 4-tuples in the DFA of a quadtree and reminding ourselves that the children of a node are ordered always in the same way according to the position of the corresponding cell in the partitioning, we observed that the first entry corresponds to NW, the second to NE, and so on. This observation is the essential link between the IG-tree representation and the GET representation, which enables us to compute the entries of neighbours in a DFA-list also in an easy way.

3 General Description of the Problem

The reason for dealing with the problem of finding neighbours in the GET representation of 2^k-trees is, that we want to plan a collision-free path for a robotarm by going through following steps:

- Convert the path planning problem in the workspace of the robot to the path planning problem for a point in the joint space (or configuration space) of the robot.
- Put a non-uniform grid on the Cspace. This grid is fine grained close to the boundary of the set of free placements in the Cspace (i.e. close to the obstacles in Cspace), and coarse grained elsewhere.
- Assign the value '0' to the cell c_g with the goal position lying in it. Compute a new value for all neighbouring cells c_m (all in the free space, subscript m takes values 0,1,...), based on the value for cell c_g. In this way we compute a discrete potential funtion u on all cells of the free space. As a first attempt, we will just add '1' to the value for cell c_g and give this as the new values for the neighbours (in this case, when the grid would be uniform, we would get the Manhattan distance to the goal postition). Iterate this process for the neighbours of the cells c_m (when we would have a uniform grid, this would be the wavefront expansion approach).
- If we have been given an initial position of the robot, we can now create a path by going from cell c_s in which the initial position lies to a neighbouring cell c_p with value less than the value in c_s. We then iterate this process for c_p until we reach c_g.
- We might want to smoothen the path obtained.

The advantages of this approach are summarised in [7]. Of course, this is just a rough sketch of a general setting, and the aim of this paper is not to cover all details of all these steps. We will, however, look more closely to the third and forth item on the list above.

The main advantage of the GET representation of the 2^k-tree and its underlying partitioning of an area, is the compactness of the representation (for a discussion on this aspect, see [1]). It is however not clear at all how and where to find the neighbouring cells in the GET representation. The IG-tree representation is very well suited for the computation of neighbouring cells, and algorithms to find the neighbours for an octree are given in [3]. These algorithms can easily be generalized for 2^k-trees for arbitrary large k.

Another advantage of the GET representation lies in the second list, the DFB. This list will be replaced by a list of all values assigned by the discrete potential function u to the cells (which are represented by a zero in the DFA).

4 Computing Neighbouring Cells

Looking at the partition in Figure 1 again, we can see that some of the neighbours of a cell may be cells of the same partitioning step (i.e. cells of equal size, nodes with the same parent node). Sometimes however, a big neighbour is partitioned itself and gives rise to some smaller neighbouring cells. This phenomenon can occur even recursively. Also, not all neighbours are cells represented by nodes with the same parent node. In Figure 5, all neighbours of one particular cell are marked.

The central observation on which the algorithms in [3] are based, is that the neighbouring cells in the partition are represented by IG-tree codes equal up to a shift of some of the entries. For IG-tree representations, shifting in a particular direction means adding a fixed number (depending only on the direction) modulo 2^k. E.g. in the example of Figure 1, the neighbour at the East of the cell with IG-tree code '21X' is '20X', the neighbours at the East of the cell with IG-tree code '1XX' are '01X', '031' and '033', i.e. the last digit before the 'X' is decreased by 1 and then the children at the West-side (with '1' or '3') are found in the list of IG-tree representation codes. We observed that an analogous statement holds for the representation as DFA-list, and that the key idea of the algorithms computing neighbours in [3] can also be applied to the GET representation. This results in the algorithms described in the following subsections.

4.1 The Neighbours With Same Parent Node

Let us first look at the neighbours that are children of the same parent node. We look at the neighbours in a given direction, i.e. the neighbour of a cell when looking in a certain given direction. We assume for the moment that all these children are leaves.

In the case of a quadtree, we have, for a given direction, two nodes that have a neighbour with the same parent node. When we look to the West, the nodes in the NE and SE have respective neighbours NW and SW at their West-side. When we look to the North, the nodes SW and SE have respective neighbours NW and NE. Similar statements hold for the East-side and South-side neighbours. In the DFA-list, we just look at one particular 4-tuple, since the cells have the same parent node. All entries in this 4-tuple are '0', since they all represent leaves. Let us number these entries from 0 to 3. For the West-side, the entries with odd number have a neighbour with the same parent, namely the one represented by the even number left of them (i.e. entry 1 has neighbour with entry 0, entry 3 has neighbour with entry 2). For the East-side, entry 0 has neighbour 1, entry 2 has neighbour 3. For the North-side, entry 2 has neighbour 0, entry 3 has neighbour 1. For the South-side, entry 0 has neighbour 2, entry 1 has neighbour 3.

In the case of an octree, a similar thing happens. The relative position of the cells determine the ordering of the entries in the DFA in such a way that we can see the octree as two quadtrees: one quadtree on the front and one at the back. Therefore, the same shift operations as in the case of a quadtree hold, but we now have to use them for 8, that is twice for 4, entries. So, for example, when we look for the neighbours on the West-side, the cells NEF, SEF, NEB, and SEB have respective neigbours NWF, SWF, NWB, and SWB. In the 8-tuple (consisting of '0'-s and let us number them from 0 to 7), the odd numbered entries have the even numbered entries as there West-side neighbours, i.e. 1 and 0, 3 and 2, 5 and 4, 7 and 6. Similarly, for the neighbours on the North-side, we get the shifts from 2 to 0, from 3 to 1, from 6 to 4 and from 7 to 5. Moving East and South is just the other way round. However, for the octree, we have a new dimension. We can also move to the front or to the back. In case of neighbours in front (moving to the front) we have for NWB, NEB, SWB and SEB, the respective neighbours NWF, NEF, SWF, and SEF. In the DFA-list, we get a shift in the 8-tuple from 4 to 0, from 5 to 1, from 6 to 2, and from 7 to 3. In the opposite direction it is moving to the back.

In general, for the k-dimensional case, we have all neighbours like in the $(k-1)$-dimensional case twice (a 2^k-tree can be seen as two 2^{k-1}-trees 'behind eachother', taking into account the convention on the ordering according to the relative position of the cells in the partitioning of the area). These neighbours can be found in the 2^k-tuple by splitting it up in two 2^{k-1}-tuples, by taking the first 2^{k-1} entries as one 2^{k-1}-tuple and the remaining entries as another 2^{k-1}-tuple, and applying the rules for the 2^{k-1} case. However, two additional directions have to be considered as well. Let us denote them as A_k and B_k, and add them to the order as follows: W, E, N, S, F, B, A_4, B_4, A_5, B_5, ..., A_k, B_k (where we have the correspondence $W = A_1$, $E = B_1$, $N = A_2$, $S = B_2$, $F = A_3$, $B = B_3$). For neighbours on the A_k-side (when moving in the direction A_k), the nodes represented by the second part (second 2^{k-1}-tuple) have neighbours in the first part (first 2^{k-1}-tuple). Let us number the entries in the 2^k-tuple from left to right, beginning with 0 up to $2^k - 1$. Then for all $l \geq 2^{k-1}$, entry with number l has A_k-neighbour $l - 2^{k-1}$.

The nice thing about it is, that this iteration also holds for $k = 2$, i.e. starting from a binary tree, representing two cells on an interval, only considering W and E as directions.

4.2 The Neighbours With Other Parent Node

Given a cell c, we now assume that the neighbour of c for which we are looking, is larger in size than c. This means that the neighbour of c cannot be found in the 2^k-tuple in which c is represented, but has to be found in a 2^k-tuple that occurs earlier in the DFA-list. We can apply the following algorithm.

input: The DFA-list, the entry for cell c.

output: The 2^k-tuple in which the entry in the DFA-list of the parent node p of the node representing cell c lies, and the entry of p itself.

step a: Divide the DFA-list into levels of the tree (left to right transversal, counting the encountered '1'-s).

step b: Divide the level on which the entry of c occurs, into 2^k-tuples.

step c: When c is in the i-th 2^k-tuple on level m, then the parent p of c is the i-th '1' in level $m-1$.

step d: Divide level $m-1$ (of parent p) in 2^k-tuples (same procedure as in step b).

step e: Return the i-th entry and the 2^k-tuple (on level $m-1$) in which this entry lies.

This case of the search for neighbours is reduced to the previous case, with the following observation. The larger neighbour of c on the Γ-side, where Γ is an element of a subset of $\{A_1, ..., A_k, B_1, ..., B_k\}$, is the neighbour of p when we neglect the fact that p is not a leaf in the 2^k-tree. (I.e. we forget about the children of p. This can be done, since these children never will be a neighbour for which we are looking in the case under consideration, here.)

A final remark about this case; The procedure described above, might have to be executed recursively on p. I.e. we might look for a neighbour that corresponds to a node several levels higher in the tree than the node corresponding to c. Then the node representing this neighbour lies several levels higher than the representation of c in the DFA.

4.3 Going Down To Smaller Sized Neighbours

Till now, we assumed that neighbours were always of equal size or larger than cell c. This is of course not always the case. However, in the previous, we never had to check whether the entry of a possible neighbour was '0' or not, and all the reasoning of above holds still up to the following small adaptation. Let us call nodes n that are a parent node of neighbours of c and not a parent node of c itself, *parent neighbours* of c. In the previous subsections, we could find such nodes n. So, we reduced the remaining problem of finding neighbouring cells of smaller size than parent neighbours of c represented by leafs, to finding the leafs which have parents who are parent neighbours of c. From now on, we assume that we are looking for these leafs.

Let us first look at quadtrees. If we look for the East-side neighbours of cell c, we first find the parent neighbour of the neighbouring cells in the East. When this parent cell n is represented by a '1', it is divided into subcells, and we have to select the two children of n on the West-side of the partition. This is illustrated in Figure 6. So, in the DFA, we have to look for the first and the third entry in the 4-tuple corresponding to the '1' representing node n. When these two children are leaves, we are done. When one of them, say node w is represented by a '1', we have to look for the West-side children of this cell w, i.e. to the first and the third entry in the 4-tuple corresponding to the '1' representing node w. When these are leaves, we are done, otherwise we have to iterate this procedure again.

When looking for the South-side neighbours of c, we have to look for the North-side children of the South-side parent neighbour, and iterate when necessary. The analogue holds for West- and North-side neighbours.

In the case of an octree, we can apply the same rules for N, S, W and E as in the case of a quadtree, only we have to apply this twice: again, we divide the 8-tuple in two 4-tuples, as described in the previous. For the directions F and B, we have to look for the B-, resp. F-, children of the F-side, resp. B-side, neighbours.

In the case of a general 2^k-tree, the same method of generalisation can be applied. So we divide the 2^k-tuple in two 2^{k-1}-tuples for the directions A_1, ...,A_{k-1} and B_1, ...,B_{k-1}, and we look for the A_k-, resp. B_k-, children of the B_k-side, resp. A_k-side, neighbours.

5 The Discrete Potential Function u

We now come to the computation of the discrete potential function u by a wavefront expansion approach. The '0'-s in the DFA-list represent leaves in the 2^k-tree, and therefore cells in the partitioning of the area. When applying such a partitioning to the robot motion planning problem using the Cspace of a robotarm, this partitioning is a non-uniform grid over the set of feasible configurations in Cspace. So, some cells will be forbidden area for the robot (due to an obstacle in the Workspace or due to some physical limitations on the feasible motions of the robotarm), small cells will occur close to these forbidden areas. The goal position of the robot will ly in some cell c_g, the initial position usually in some other cell c_s. The path through this grid, consisting of a list $c_0, c_1, ..., c_q$, where $c_0 = c_g$, $c_q = c_s$, and c_p is a neighbouring cell of c_{p+1} for all $p = 0, ..., q-1$.

Then the following algorithm computes a discrete potential function u on the non-uniform grid, when calling it for $c = c_g$.

step a Find the entry for cell c in the DFA-list.

step b Find the entries for all neighbours of c.

step c For each of the entries $\mathcal{E}$ of all neighbours $N_{\mathcal{E}}$ of c, do

- If the corresponding cell is not feasible (i.e. if the corresponding entry in the DFB is a '1'), put 'X' on the corresponding position in the u-list,
- Else, $u(N_{\mathcal{E}}) \leftarrow u(c) + 1$, and put this value on its place in the u-list.
- Call this algorithm recursively for $N_{\mathcal{E}}$.

The path starts from c_s, and can be build up by choosing at each step a neighbour c_{p+1} of the cell c_p added one step earlier, for which $u(c_{p+1}) < u(c_p)$ holds, until c_g is reached. In this paper we assume that such a path exists. If such a path does not exist and the connected component of the Cspace in which c_g lies is bounded, the wavefront expansion algorithm will stop at this boundary. In the case when a path does exist, the wavefront algorithm may stop as soon as c_s is reached and $u(c_s)$ is computed, or when the Workspace is bounded, when all cells c (finitely many) are reached and $u(c)$ is computed for all cells. This remark is exactly the same as in the wavefront expansion approach for a uniform grid in [5].

This potential function u has the following meaning. For each feasible cell c, the value $u(c)$ is the minimal number of cells that has to be crossed for reaching cell c_g. In particular, $u(c_g) = 0$, $u(c_{p+1}) - u(c_p) = 1$ for each $p = 0, ..., q-1$.

6 Implementation, Experimental Results

We implemented the computation of neighbours, the construction of the potential field and pathfinding on a Silicon Graphics machine in the programming language $C++$, using extensively some standard $C++$-classes available in the *gnu*-library ([8]). This programme is a sequential algorithm, which already performs quite well for a decomposition of the space into up to 8,000 cells. The computation of the path goes as follows: For cell c_i on the path, found in the previous iterative step, we compute all neighbours of c_i. Then, we run through the list of all these neighbours and select the cell first appearing in the list, with lowest value of u assigned to it. We wanted to have a look at the outcome of the algorithm, by implementing a prototype with few efforts, using extensively tools already available. The goal was not to make a program that runs as fast as possible. A better sequential implementation, in the C programming language, running on a powerful PC, will certainly perform much better than the current version. Therefore, we did not compare computing time with results known in the literature yet. However, this first implementation does have a good performance, as we will describe in the following.

The paths computed as results of the experiments, indeed prefer to go along large cells. The number of cells for which neighbours have to be computed in order to find the path is relatively small, namely equal to the number of cells along the path. Since the time complexity depends mainly on the number of times neighbours have to be computed, the computation of the path goes very fast. We could also observe this when comparing the time needed for computing the potential field and the time needed for computing a path.

Experiments show that the time necessary to compute the potential field increases with the number of cells in the decomposition of the Cspace. Keeping the total number of cells fix, we observed that the height of the tree (i.e. the refinement of the grid) also has a slight influence on execution time.

We also observed that with dimension higher than 6, the computation of all neighbours of a cell c sometimes takes about 16 milliseconds, i.e. some noticeble time. The number of directions in which to look for neighbours seems to become significantly large for high dimensions.

The method described above needed about 15 seconds for 4 dimensional cases with 900 cells, about 1.5 minutes for 6 dimensional cases with 2,000 cells, about 23 minutes for 7 dimensional cases with 8,000 cells, about 2 hours for 6 dimensional cases with 15,000 cells, and almost 7 hours for an 8 dimensional case with 32,896 cells.

Of course, the number of cells depends on the dimension of the space, not only for uniform grids but also here. A speed-up can be expected by parallelisation of the computation of neighbours, e.g. by computing neighbours of a cell in each direction by another processor (and to be expected speed up proportional to the dimension of the space, also depending on the number of available processors). The computation of the path could also be made faster by storing in memory which cells are neighbours of eachother. However, this would increase drastically the amount of memory needed.

In some real applications, carried out over the last years at the Research Institute for Symbolic Computation, in the fields of geometric modelling and obstacle avoidance, the number of 'cells' used was 20 to 50 per object in 3D, and an estimate of 10,000 cells for a quadtree representation. A reasonable amount of cells to be expected seems to be between 5 and 20 cells along each dimension. So, the real cases the sequential program as described above can handle (in less than 2 hours) are limited to 3 dimensions if the number of cells in one dimension is 20 ($20^3 = 8,000$) and to 6 dimensions if the number of cells in one dimension is 5 ($5^6 = 15,625$). However, a speed-up with an estimated factor of 10 to 100 can be expected by implementing the algorithm in C, and by parallelising the algorithm.

In real applications, usually the tasks of a robot results in moving from a given start position to another place and then returning to the start position. So, as long as the environment of the robot does not change from a geometric point of view, it suffices to compute the potential field u only once, building up u from the start position. A path to different goal positions can be computed with the same u. In this way, the computation of u is part of the modelling phase, and might take several minutes or even hours. The computation of a path however is just a matter of seconds, since it only depends on the computation of neighbours of the cells along the path and since the path consists of the lowest number of cells between start and goal position.

7 Conclusion

The approach presented here to solve the general robot motion planning problem, uses a representation of the Cspace which takes very few memory size. Up to the case of a 6 dimensional Cspace, the method presented here is very fast. The resulting path is a pay-off between shortest path in Cspace and safest path.

Some speed-up can be expected from a better sequential implementation, by just using the programming language C and by writing all programming structures by ourselves. One could also look for another - more involved - algorithm to compute neighbours in different branches of the 2^k-tree, using the GET-representation, or for a more involved way to compute the potential field.

The most time consuming is the computation of the neighbours; not so much because it would take long to find a neighbouring cell, but for the fact that so many neighbours have to be computed. We therefore are optimistic that a further speed-up can easily be obtained by parallelising this computation.

We will dedicate some of our time in future to the implementation of a parallel version of this algorithm. Furthermore, we will experiment with assigning other potential fields. When an obstacle has moved slightly, the 2^k-tree representation will also change. Therefore we are especially interested in a potential field that can be adapted in a small environment of such a slightly moved obstacle. We will also look for a potential field that favours a particular direction (i.e. moving a particular joint of a robotarm). It would be interesting to look for a potential function which would result in a path that is a good pay-off between safe and shortest path in the workspace of the robot.

References

[1] **Burger, K. C.** , A Kernel-Based Construction System (KBC), diploma thesis, RISC-Linz, July 1992.

[2] **Conte, G. , Longhi, S. , and Zulli, R.** , CAD-Based Generation of Collision-Free Paths for Robotic Manipulators, (preprint, will be published in Control Engineering Practice, Dec.1993).

[3] **Gargantini, I.** , Linear Octrees for Fast Processing of Three-Dimensional Objects, Computer Graphics and Image Processing 20, pp. 365-374 (1982).

[4] **Hwang, Y. K. , and Ahuja, N.** , Gross Motion Planning - A Survey, ACM Computing Surveys, Vol. 24, N. 3, Sept. 1992, pp. 219-292.

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

[6] **Samet, H.** , Applications of Spacial Data Structures, Addison-Wesley, Reading, MA, 1990.

[7] **Van Geem, C. ,** Towards a Fast Solution Method for the General Robot Motion Planning Problem using a Manhattan-like Distance Function on a Non-Uniform Grid in Configuration Space, Technical Report, RISC-Linz, 93-66.

[8] **Doug Lea,** User's Guide to the GNU C++ Library (Version 2, p75 ff.), ©1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA.

List of Figures

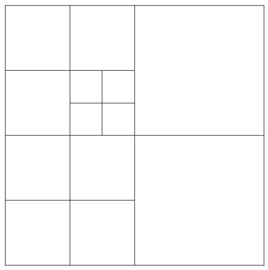

Figure 1: A Partitioning of a Rectangular Area .

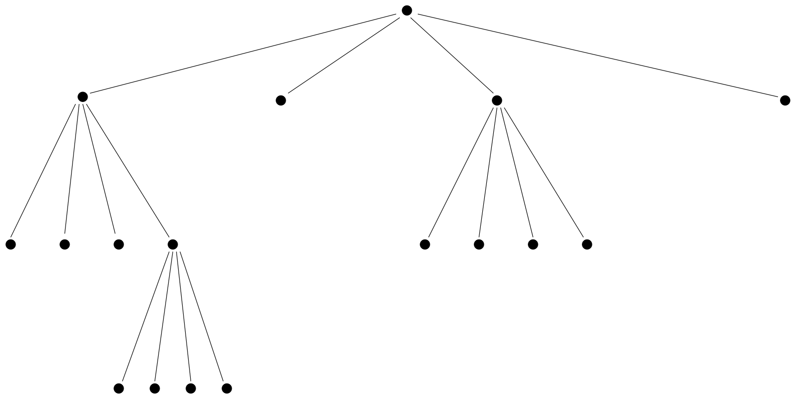

Figure 2: The Quadtree Corresponding to the Partitioning in Figure 1.

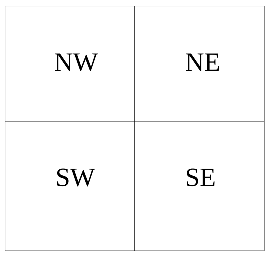

Figure 3: Denotation for Cells.

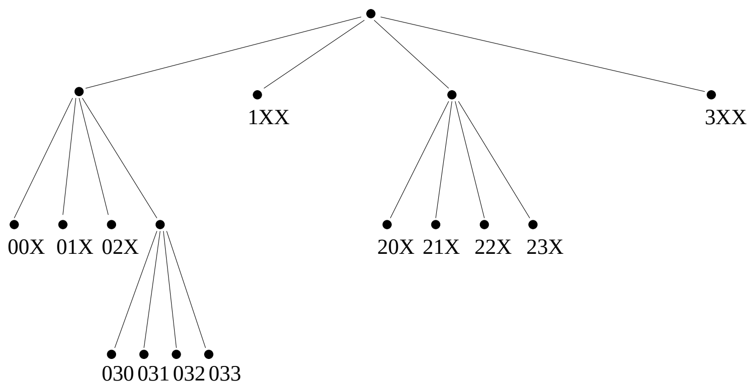

Figure 4: The IG-tree Representations for the Quadtree of Figure 2.

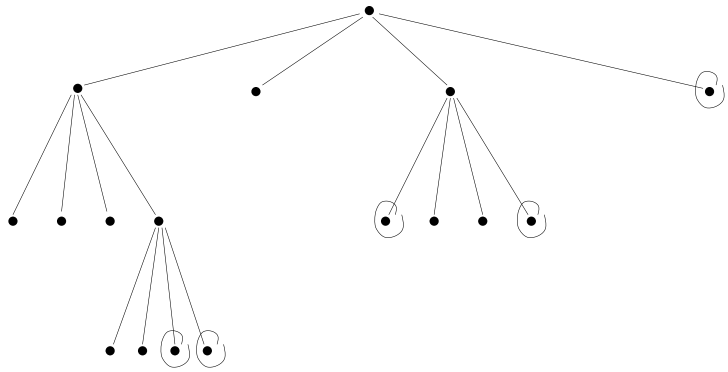

Figure 5: Neighbours of a Cell, Scattered About the Tree.

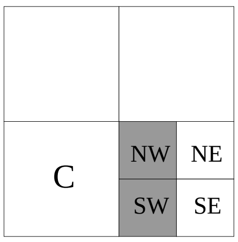

Figure 6: The East-Side Neighbours of Cell c Are West-Side Children.