Journal of Intelligent and Robotic Systems (2005) 42: 393–413 © Springer 2005

A Hierarchical Extension of the D^* Algorithm

DANIEL CAGIGAS
Department of Computer Architecture and Technology, University of Seville, Spain; e-mail: dcagigas@us.es

JULIO ABASCAL
Laboratory of Human-Computer Interaction for Special Needs, University of the Basque Country, Spain

(Received: 24 February 2003; in final form: 5 January 2005)

Abstract. In this paper a contribution to the practice of path planning using a new hierarchical extension of the D^* algorithm is introduced. A hierarchical graph is stratified into several abstraction levels and used to model environments for path planning. The hierarchical D^* algorithm uses a down-top strategy and a set of pre-calculated trajectories in order to improve performance. This allows optimality and specially lower computational time. It is experimentally proved how hierarchical search algorithms and on-line path planning algorithms based on topological abstractions can be combined successfully.

Key words: efficient path planning, hierarchical search, D^* algorithm, mobile robots.

1. Introduction

Until now, maps managed by mobile robotic systems based on topological maps (graphs) usually involved no more than 100 nodes approximately. Mobile robots are usually prototypes or industrial robots that work in a reduced environment. Metric maps are mainly used in local path planning and obstacle avoidance where environments are not quite big.

However, sometimes the path planner of a mobile robot must face large and structured environments where traditional *branch&bound* or *genetic algorithms* are not efficient enough. To face this problem, hierarchical search is a very effective alternative. Firstly, a hierarchical algorithm needs a hierarchy of abstractions representing different views of an environment where a mobile robot has to work. Highest levels in that hierarchy represent a global view of an environment and deepest levels show the most detailed views. Secondly, refinement processes are applied to the hierarchy in order to obtain a path[⋆] free of obstacles. There are also reconstruction processes that link partial paths obtained after refinement.

Hierarchical search has been applied in large static environments where a path must be re-planned until the mobile robot reaches its goal. Little attention has been given in mobile robotics to systems that need on-line hierarchical path planning

[⋆] It will be used path or trajectory either.

algorithms. This work is also an on-line path planning extension of the hierarchical path search algorithm described in [3].

In Section 2 previous studies made in hierarchical path planning and on-line path planning are briefly reviewed. In Section 3 it is described the abstract world model and the algorithm proposed. A previous description of the fundamentals of the D^* algorithm is also commented. In Section 4 experimental results obtained with the hierarchical algorithm proposed are analysed and compared with other algorithms.

2. Previous Studies

2.1. HIERARCHICAL PATH PLANNING

In *Artificial Inteligence* (AI) hierarchical problem solving consist of some steps that can be summed up into three:

- Abstraction: a space search is modelled by a hierarchy of abstractions divided in several sublevels. In mobile robotics abstractions are called maps.
- Resolution: a problem is solved using partial solutions that are refined during this process. A partial solution is found in one abstraction level and then it is refined in the next abstraction level. In mobile robotics, solutions and/or partial solutions are called paths or trajectories.
- Reconstruction: partial solutions are linked until a global solution is achieved.

Hierarchical mobile robot path planning is not the most used strategy in robot path planning. Traditional path planning techniques (specially Dijkstra's or A^* algorithm) are still the most used techniques in robot path planning. This is a direct consequence of problems faced in mobile robotics. Autonomous mobile robots are just prototypes that move in small areas due to their navigation restrictions. Small robot environments do not need hierarchical path planning. AGV (automated guided vehicles) are the only industrial mobile robots used widely but they are always guided in reduced industrial areas.

However, some special robotic systems that have computational time restrictions (not necessary real-time) and/or large/complex environments, could be benefited from hierarchical path planning. A *TetraNauta wheelchair* is one of these examples. TetraNauta [4] is a controller for standard wheelchairs that permits semi-automatic navigation in close environments such as hospitals. Hospitals are usually large (with many floors, corridors and rooms inside) and every trajectory must be generated on-board by the TetraNauta's wheelchair computer system. Moreover, that wheelchair computer system must attend several real-time tasks (user interface, sensors, communications, ...) and the type of computer load that can be managed on board is not high. An alternative could be a huge list of precalculated trajectories from each pair of points inside the hospital. Again, it does not seem to be a very suitable choice, specially when maps are too big (even thousands of starting/end points) and the path planner module must face on-line path planning (multiple alternative paths between a single pair of points).

Similar reasons (not only in robot path planning) have justified hierarchical search techniques in some other cases. Thus, in [16] the problem of finding good abstraction hierarchies is analysed. In [5] a hierarchical multilevel discretization and a *wave front expansion algorithm* are used together to solve a robot motion planning problem. In [19] a hierarchical abstract map is used to speed up the problem of intercepting a moving object. In [11, 12] the trade off of using abstraction hierarchies is studied and analysed. In [13] a hierarchical path planning algorithm is proposed to plan a collision free path for mobile robots and robot manipulators. In [6, 7, 9] the concept of H-Graph (hierarchical graph) is described and applied to mobile robot path planning.

2.2. ON-LINE PATH PLANNING

On-line path planning refers to problems where robots must replan their initial paths because their abstract environment model has been updated due to a change. This last fact usually happens when an unknown obstacle is found.

Branch&Bound algorithms in dynamic environments are based on *RTA**–*LRTA** [17], Dynora [10] and specially D^* [20] algorithms. The first three are real-time algorithms that combine execution and calculation cycles. They are not path length or time optimal algorithms and are focussed on computational time restrictions.

The D^* algorithm was first introduced by Stentz (see [20]) and represents a dynamic version of the A^* algorithm. A^* algorithms are widely used in off-line path planning and robot motion planning (see [18]). It is a path length optimal algorithm and saves a lot of computational time when comparing with brute-force methods. Genetic algorithms are a relative novel path planning technique applied to metric maps that provide flexibility in dynamic environments. Examples can be found in [2, 1] or [21]. The newest strategies combine genetic and branch&bound algorithms. Thus, branch&bound algorithms are used to generate an initial population of paths for the genetic algorithm or regenerate it when an obstacle is found (see [15]).

3. Description of the Algorithm Proposed

3.1. D^* ALGORITHM

First of all, it is going to be briefly described the D^* algorithm basis. The environment is modelled by a graph. As it was said, the D^* algorithm is a dynamic version of the A^* algorithm. A D^* algorithm restart the general process of a A^* algorithm every time it is found a "broken" arc A_b in a initial path P_i. A_b is connected to a node N_c (current node) which represents the current robot position. This happens when, for example, a mobile robot detects an unknown obstacle in P_i. P_i is an optimal path[⋆] between an starting node N_s and a goal node N_g. Every node connected to the current node N_c is added to the OPEN LIST. The OPEN LIST is the list where A^* (or D^*) takes nodes that are expanded. The only not expanded node is the node connected to node N_c via A_b. The whole process ends when:

1. A node N_x is found, was included in P_i and the rest of partial paths (solutions) have a higher f value. $f = g + h$, where g is the accumulated path cost and h is the heuristic value that estimates path cost to N_g. D^* algorithm (and A^*) is an algorithm that tries to minimize f function.
2. The goal node N_g is found and the rest of partial paths have a higher f value. This is equivalent to a brute-force method and represents the worst case.
3. The OPEN LIST is empty. This means that there are no solutions without A_b.

3.2. ABSTRACT WORLD MODEL

The abstract world model is based on a hierarchical graph (H-Graph). This H-Graph model was also described in [3] when an off-line version of this hierarchical D^* path planning algorithm was described. The H-Graph proposed has a sequence L of hierarchical levels, where $L = \{L_0, L_1, L_2, \dots, L_D\}$. D is the depth of the hierarchy. The “root level” is L_0 and it represents the highest abstract description of an environment. On the contrary, L_D contains the most detailed description of an environment. For example, it may contain the internal structure of a room in a building. In each level L_i ($0 \leqslant i \leqslant D$) there is a graph $G_i = (N_i, A_i, C_i, W_i, T_i)$, where N_i is a set of nodes, A_i is a set of arcs, C_i is a set of Cartesian coordinates for N_i, W_i is a set of weights for A_i and T_i is a set of precalculated paths associated to N_i. The union of graphs $G_0, G_1, G_2, \dots, G_D$ is a graph $G = (N, A, C, W, T)$, where $N = N_0 \cup N_1 \cup \dots \cup N_D$, $A = A_0 \cup A_1 \cup \dots \cup A_D$, $C = C_0 \cup C_1 \cup \dots \cup C_D$, $W = W_0 \cup W_1 \cup \dots \cup W_D$, $T = T_0 \cup T_1 \cup \dots \cup T_D$.

An arc $a(n_J, n_K, w_H) \in A$ is defined by three elements n_J, n_K, w_H, where $n_J, n_K \in N$, $n_J \neq n_K$ and $w_H \in W$. A Cartesian coordinate $c_I \in C$ is defined by (x, y), where x, y are natural numbers ($x, y \in \mathbb{N}$). A weight $w_I \in W$ is real number ($w_I \in \mathbb{R}$).

Some nodes can represent a cluster (subset) of nodes in a deeper abstraction level of the hierarchy. These nodes are called *submap* nodes and the submap node set contained in N is called SN ($SN \subset N$). There are some functions/methods associated to a node $n_J \in G_j$ ($0 \leqslant j \leqslant D$). The dot notation is used:

– map $\to$ $n_J.map = n_K$, where $n_J \in L_j$, $n_K \in L_k$, $j = k + 1$ ($0 < j \leqslant D$, $0 \leqslant k \leqslant D$) and $n_J \subset n_K$ in L_k. Namely, it indicates in which node is n_J included in an upper level of the hierarchy. It is said that n_K *submap* (cluster or subset) of n_J.
– depth $\to$ $n_J.depth = x$, where $n_J \in L_x$ ($0 \leqslant x \leqslant D$). Namely, it returns the level of the hierarchy where n_J belongs.

[⋆] Path length or path time are usually the optimised factors.

Nodes are also classified in four classes: *end nodes*, *cross nodes*, *submap nodes* (cluster nodes) and *bridge nodes*. End nodes are starting or goal points that a robot path planner can select. Cross nodes represent subtargets that indicate turns or crossroads. Bridge nodes are nodes that connect a submap to a "parent" submap. The bridge node subset contained in N is called BN. Formally, $n_I \in G_i$ is a bridge node (i.e. $n_I \in BN \subset N$) if there is a node $n_J \in G_j$ and an arc $a_I(n_I, n_J, w_X) \in A$, where $i = j + 1$. The concept of bridge node leads to a new function/method:

– get_bridge_nodes $\rightarrow$ $n_I.get\_bridge\_nodes = BN_I \subseteq BN$, where $n_I \in N$, $n_I.depth < D$ and BN_I satisfies that $\forall n_x \in BN_I, n_x.map = n_I$. Namely, if n_I is a submap, it obtains its bridge node set included in the next deeper level of the hierarchy.

Arcs (A) are non-directed: a mobile robot can navigate between two points (nodes) in both ways. An important difference from other H-Graph models is that here arcs do not contain other arcs in a deeper abstraction level of the hierarchy. Cartesian coordinates (C) are attributes associated to every node. They are used in the heuristic function of the path planner. Weights (W) are attributes associated to each arc and indicate the cost of traversing an arc. They are used by the cost function of the path planning algorithm and represent a length in metres.

A path is defined as a succession of nodes. The whole set of paths contained in a H-Graph is called P. Formally, a path $P_I \in P$ of length L is defined by $P_I = (n_0, n_1, n_2, \ldots, n_L)$, where $n_0, n_1, \ldots, n_L \in N$ and $\exists a_0(n_0, n_1, w_{(0,1)})$, $a_1(n_1, n_2, w_{(1,2)}), \ldots, a_{L-1}(n_{L-1}, n_L, w_{(L-1,L)}) \in A$. A path P_I has three attributes/methods:

– cost $\rightarrow$ $P_I.cost = x$, where $x \in \mathbb{R}$. It gets or assigns a path cost to P_I.
– length $\rightarrow$ $P_I.length = L + 1$, where $L \in \mathbb{N}$, $P_I \in P$ and $P_I = (n_0, n_1, n_2, \ldots, n_L)$. It returns the path length of P_I.
– index $\rightarrow$ $P_I.index(J) = n_J$, where $n_J \in P_I = (n_0, n_1, \ldots, n_J, \ldots, n_L)$, $n_J \in N$, $P_I \in P$, $L + 1 = P_I.length$, $0 \leqslant J \leqslant L$. Namely, it returns the node of a path in position J.

Each submap node $n_I \in N_i \subset N$ $(0 \leqslant i < D)$ has its own precalculated path set $PPS_{n_I} \in T_i \subset T$ $(0 \leqslant i < D)$. Thus, a new method/function associated to a submap node n_I can be defined:

– *pre_path* $\rightarrow$ $n_I.pre\_path(n_X,\ n_Y) = P_Z$, where $n_X, n_Y \in N$, $P_Z = (n_X, n_{X+1}, n_{X+2}, \ldots, n_{Y-2}, n_{Y-1}, n_Y)$, $P_Z \in PPS_{n_I} \subset P$. Namely, a node n_I returns a path P_Z between nodes n_X and n_Y whether it has an attached precalculated path set PPS_{n_I} that contains P_Z. Nodes n_X and n_Y that define a precalculated path will be usually bridge nodes. If a node n_I is not a submap node or it does not contain the required path, the method/function returns NULL.

Precalculated paths in PPS_{n_I} are optimal-length paths and are off-line calculated. They are grouped in three classes:

1. Paths that link two bridge nodes inside n_I.
2. Paths that link the bridge nodes of n_I (n_I*.get_bridge_nodes*) with the bride nodes of its “parent” submap ((n_I*.map*)*.get_bridge_nodes*).
3. Paths that link “brother” submaps contained in n_I. Two submap nodes $n_X, n_Y \in N$ are “brother” submaps contained in n_I if $n_X.map = n_Y.map = n_I$. Namely, they are “brother” submaps if they have the same “parent” submap in the previous level of the hierarchy.

Precalculated paths avoid recalculating several subpaths in a hierarchical search process. This is called *materialization of costs* [14]. On one hand, materialization of cost requires extra storage space for paths and costs and off-line path calculation [8]. On the other hand, it can guarantee optimality in a classic refinement hierarchical search method.

3.3. PATH PLANNING

The algorithm proposed is not a real-time algorithm. However, depending on the problem, it could be a feasible alternative (or complement) to classic real-time algorithms. The heuristic used (h function in $f = g + h$) is the Euclidean distance.

The general search process is similar to D^*. The “broken” arc A_b connected to the current node N_c is first erased from the H-Graph. Nodes connected to N_c are added to an *OPEN_LIST* and then expanded until the goal node N_g is reached. Nodes N_g and N_c are supposed to be end, bridge or cross nodes (i.e. everything except submap nodes). Node processing in *MAIN_PROCEDURE* is divided into four parts.

First and fourth parts implement the same subprocesses included in D^*. If a new candidate node for expansion N_a is in the initial path P_initial, then current path P_current is completed using the same nodes included in P_initial, starting in N_a and finishing in N_g (lines 12 to 15). Fourth parth just expands N_a, that is to say, generates new partial paths joining P_current and N_a neighbours nodes (lines 37 to 40). This last subprocess is detailed in *D*_NODE_EXPANSION* procedure.

Second and third parts deal with bridge nodes and submap nodes respectively. Here materialization of costs (precalculated paths) are used to link submaps through their bridge nodes or used to substitute submap nodes by their precalculated paths.

MAIN PROCEDURE. D*_HIERARCHICAL_PATH_PLANNING (Node N_c, Node N_g, Arc A_b, Path P_initial):

```
1: {Begin variable declaration:}
2: Node $N_a$, $N_\mathrm{aux}$;
3: Path $P_\mathrm{current}$, $P_\mathrm{new}$;
4: Path_Set Open_List;
```

```
 5: {End variable declaration.}
 6: $P_{\text{current}} = (N_c)$;
 7: $P_{\text{current}}.cost = 0$;
 8: $Open\_List = (P_{\text{current}})$;
 9: while ($Open\_List \neq NULL$) do
10:   $P_{\text{current}} = GET\_BEST\_PATH(Open\_List)$;
11:   $N_a = P_{\text{current}}.index(P_{\text{current}}.length - 1)$;
12:   if ($N_a \in P_{\text{initial}}$) then
13:     {The current partial path has intersected the initial path.}
14:     $P_{\text{new}} = COMPLETE\_PATH(P_{\text{current}}, N_a, P_{\text{initial}})$;
15:     $PROCESS\_SOLUTION(P_{\text{new}})$;
16:   else if ($N_a \in BN \subset N$) then
17:     {Bridge nodes expansion is processed separately.}
18:     if ($N_a.map \neq N_g.map$) then
19:       {Node expansion using precalculated paths (materialization of costs).}
20:       $HIERARCHICAL\_D^*\_NODE\_EXPANSION$ ($P_{\text{current}}, N_a, P_{\text{initial}},$
          $Open\_List$);
21:     else
22:       {Both nodes are included in the same submap.}
23:       $P_{\text{new}} = (N_a.map).pre\_paths(N_a, N_g)$;
24:       if ($P_{\text{new}} \neq$ NULL) then
25:         {There is a precalculated path between $N_a$ and $N_g$.}
26:         $P_{\text{current}} = P_{\text{current}} \cup P_{\text{new}}$;
27:         $P_{\text{current}}.cost = P_{\text{current}}.cost + P_{\text{new}}.cost$;
28:         $PROCESS\_SOLUTION(P_{\text{new}})$;
29:       else
30:         $D^*\_NODE\_EXPANSION$ ($P_{\text{current}}, N_a, A_b, Open\_List$);
31:       end if
32:     end if
33:   else if ($N_a \in SN \subset N$) then
34:     {Submap nodes expansion is processed separately too.}
35:     {$N_a$ represents another subgraph in a deeper abstract level of the hier-
        archy. Precalculated paths (materialization of costs) are used to avoid
        refining paths that cross $N_a$.}
36:     $D^*\_SUBMAP\_NODE\_EXPANSION$ ($P_{\text{current}}, N_a, Open\_List$);
37:   else
38:     {$D^*$ normal node expansion.}
39:     $D^*\_NODE\_EXPANSION$ ($P_{\text{current}}, N_a, A_b, Open\_List$);
40:   end if
41: end while
42: return $BEST\_SOLUTION()$;
```

There are some subprocedures in *MAIN_PROCEDURE* that are not detailed due to their simplicity.

– *GET_BEST_PATH* (line 10). It returns and deletes the *best path* from a list of partial paths (*Open_List*). The best path is the path that has the lower $f = g + h$ value, that is to say, the lower cost.
– *COMPLETE_PATH* (line 14). It has three arguments: a partial solution path P_{current}, a complete solution path P_{initial} and a common node N_a of both paths. Returns a new path composed of nodes from P_{current} and nodes from P_{initial} that start in N_a and finish in goal node N_{g}.
– *PROCESS_SOLUTION* (line 15). It manages a hidden global variable which contains the best current solution P_{solution}. If this global variable was empty, then this subprocedure just assigns to P_{solution} the new solution path. If P_{solution} was not empty and the new solution path has a lower cost than P_{solution}, then it is assigned to P_{solution} the new solution path.
– *BEST_SOLUTION* (line 42). It returns the path contained in P_{solution}. If P_{solution} has not any value assigned, then it returns NULL (there is not any possible solution path). This subprocess expands submap nodes of P_{solution}. It is used a similar process to *D*_SUBMAP_NODE_EXPANSION* subprocedure.

The D*_NODE_EXPANSION subprocedure is a key part in a D^* algorithm. First, it adds to the *Open_List* new paths composed of nodes from the current path P_{current} and nodes connected to the last node N_a of P_{current}. Nodes connected through a “broken” arc A_{b} are avoided. Second, it continues the A^* search three expansion.

```
D*_NODE_EXPANSION (Path $P_{\text{current}}$, Node $N_a$, Arc $A_{\text{b}}$, Path_Set *Open_List*):

{Begin local variable declaration:}
Path $P_{\text{new}}$;
Arc $A_i$;
Node $N_i$;
{End local variable declaration.}
**for all** ($A_i = a(N_a, N_i, w_{(a,i)}) \in A$, $A_i \neq A_{\text{b}}$, $N_i \in N$) **do**
  **if** ($N_i == N_{\text{g}}$) **then**
    {The goal node $N_{\text{g}}$ has been found:}
    $P_{\text{new}} = P_{\text{current}} \cup N_i$;
    $P_{\text{new}}.cost = P_{\text{new}}.cost + w_{(a,i)}$;
    *PROCESS_SOLUTION*($P_{\text{new}}$);
  **else if** ($P_{\text{new}}.cost + w_{(a,i)} < (BEST\_SOLUTION()).cost$ ) **then**
    $P_{\text{new}} = P_{\text{current}} \cup N_i$;
    $P_{\text{new}}.cost = P_{\text{new}}.cost + w_{(a,i)}$;
    $Open\_List = Open\_List \cup P_{\text{new}}$;
  **end if**
**end for**
```

The *D*_SUBMAP_NODE_EXPANSION* subprocedure expands submap nodes. This means that a submap node N_a in a path is substituted by precalculated paths that cross N_a in a deeper abstract level of the hierarchy. This may be viewed as a *node unrolling*. There is the possibility that precalculated paths include submap nodes too. This does not affect the general path search process. However, final solution paths have to “unroll” submap nodes. Subprocedure *BEST_SOLUTION* in *MAIN_PROCEDURE* performs a recursive process similar to *D*_SUBMAP_NODE_EXPANSION* before it returns a solution path.

D*_SUBMAP_NODE_EXPANSION (Path P_{current}, Node N_a, Path_Set *Open_List*):

```
 1: {Begin local variable declaration:}
 2: Node $N_{a-1}$, $N_j$, $N_k$, $N_h$, $N_{\text{last}}$;
 3: Path $P_{\text{aux}}$;
 4: {End local variable declaration.}
 5: $N_{a-1} = P_{\text{current}}.index(P_{\text{current}}.length - 2)$;
 6: $N_k \Leftarrow N_k \in BN \,/\, \exists a(N_{a-1}, N_k, w_{(a-1,k)}) \in A$ and $N_k.map = N_a$;
 7: **for all** $(N_j \in N_a.get\_bridge\_nodes, N_j \neq N_k)$ **do**
 8:    $P_{\text{aux}} = N_a.pre\_paths(N_k, N_j)$;
 9:    $N_{\text{last}} = P_{\text{aux}}.index(P_{\text{aux}}.length - 1)$;
10:    $N_h \Leftarrow N_h \in N \,/\, \exists a(N_{\text{last}}, N_h, w_{(\text{last},h)}) \in A$ and $N_h.map = N_a.map =$
       $N_{a-1}.map$;
11:    {In $P_{\text{current}}$ the last node ($N_a \in SN$) is substituted by a refined path ($P_{\text{aux}}$)
       that crosses that node in a deeper abstract level. It is also added the next
       node ($N_h$) that follows to $N_a$.}
12:    $P_{\text{new}} = (P_{\text{current}} - N_a) \cup P_{\text{aux}} \cup N_h$;
13:    $P_{\text{new}}.cost = P_{\text{current}}.cost + P_{\text{aux}}.cost + w_{(\text{last},h)}$;
14:    $Open\_List = Open\_List \cup P_{\text{new}}$;
15: **end for**
```

The *HIERARCHICAL_D*_NODE_EXPANSION* subprocedure is another key part in the hierarchical D^* algorithm. It implements the linkage process between different submap nodes and abstract levels. Submap nodes are linked through their bridge nodes. The same hierarchical levels (and submap nodes) included in the initial path P_{initial} have to be again traversed again but nodes in between may be different. The subprocedure is divided in three steps. First step finds the next submap node N_{submap} that has to be reached. Second step finds the submap node

$N_{\text{submap\_pre\_paths}}$ that stores the precalculated paths necessaries to make the linkage process. Third step joins current path P_{current} through its last bridge node N_a with bridge nodes of N_{submap}.

HIERARCHICAL_D*_NODE_EXPANSION (Path P_{current}, Node N_a, Path P_{initial}, Path_Set *Open_List*):

```
 1: {Begin local variable declaration:}
 2: Node $N_{\text{submap}}$, $N_{\text{aux}}$, $N_{\text{submap\_pre\_paths}}$;
 3: Path $P_{\text{aux}}$;
 4: int $ind = 0$;
 5: {End local variable declaration.}
 6: {First step: get the next submap ($N_{\text{submap}}$) that the current partial solution path
    (i.e. $P_{\text{current}}$) has to reach.}
 7: $N_{\text{aux}} = P_{\text{initial}}.index(ind)$;
 8: while ($N_{\text{aux}} \notin BN$ and $N_{\text{aux}}.map \neq N_a.map$) do
 9:    $ind = ind + 1$;
10:    $N_{\text{aux}} = P_{\text{initial}}.index(ind)$;
11: end while
12: {$N_{\text{aux}}$ is a bridge node, "brother" of $N_a$ in the original path ($P_{\text{initial}}$). Now it
    must be localized the next bridge node after $N_{\text{aux}}$ included in $P_{\text{initial}}$. It will
    indicate the next submap traversed in $P_{\text{initial}}$:}
13: repeat
14:    $ind = ind + 1$;
15:    $N_{\text{aux}} = P_{\text{initial}}.index(ind)$;
16: until ($N_{\text{aux}} \notin BN$)
17: $N_{\text{submap}} = N_{\text{aux}}.map$;
18: {Second step: get the submap node ($N_{\text{submap\_pre\_paths}}$) that stores the precal-
    culated paths that link $N_a$ (i.e. last node of $P_{\text{current}}$) and the bridge nodes of
    $N_{\text{submap}}$ (i.e. next submap to reach).}
19: if (($N_a.map$).map == $N_{\text{submap}}$) then
20:    {The linkage process is made from $N_a$ to bridge nodes included in the
       "parent" submap of $N_a.map$:}
21:    $N_{\text{submap\_pre\_paths}} = N_a.map$;
22: else if ($N_{\text{submap}}.map$) == $N_a$) then
23:    {Opposite case. The linkage process is made from $N_a$ to bridge nodes
       included in a "children" submap of $N_a.map$:}
24:    $N_{\text{submap\_pre\_paths}} = N_{\text{submap}}$;
25: else
26:    {$N_a.map$ and $N_{\text{submap}}$ are "brother" submaps. Precalculated paths are stored
       in their "parent" submap:}
27:    $N_{\text{submap\_pre\_paths}} = N_{\text{submap}}.map$;
28:    {The sentence $N_{\text{submap\_pre\_paths}} = (N_a.map).map$; is valid too:}
29: end if
```

```
30: {*Third step*: linkage process. New partial solution paths are obtained joining
    $P_{\text{current}}$ to a set of precalculated paths contained in $N_{\text{submap\_pre\_paths}}$.}
31: **for all** ($N_{\text{aux}} \in N_{\text{submap}}.get\_bridge\_nodes$) **do**
32:     $P_{\text{aux}} = N_{\text{submap\_pre\_paths}}.pre\_paths(N_a, N_{\text{aux}})$;
33:     $P_{\text{new}} = P_{\text{current}} \cup P_{\text{aux}}$;
34:     $P_{\text{new}}.cost = P_{\text{current}}.cost + P_{\text{aux}}.cost$;
35:     $Open\_List = Open\_List \cup P_{\text{new}}$;
36: **end for**
```

Some important characteristics of D^* algorithms remain in this hierarchical extension. For example, this algorithm is still time optimal when precalculated trajectories are used. Time optimality instead of length optimality is possible because this D^* version can take into account robot turns. Mobile robots turns imply stop, turn and start again. Therefore, that implies a time cost in a path or trajectory. A robot path planner based on this algorithm associates a fixed length cost to every turn in a trajectory, and therefore ensures time optimality. Fixed length costs associated to turns depends on each specific robot system and must be estimated previously.

3.4. EXAMPLE

Figure 1 contains four schemes and shows an example of different path planning cases. The schemes represent a H-Graph containing three abstract levels: $i-1$, i and $i+1$ (see vertical lines). Submap nodes SM_s and SM_g are contained in hierarchical level i and their "parent" submaps are contained in hierarchical level $i+1$. Abstract levels and submaps are connected through bridge nodes (notice black dots in vertical lines).

An initial path P_i links a starting node N_s and a goal node N_g. Both nodes are included in hierarchical level $i-1$. A robot is positioned in a "current" node N_c. It can not continue because there is an obstacle. As it was said previously, obstacles are modelled by "broken arcs" in a H-Graph. In first scheme in Figure 1, the broken arc in P_i is called A_b.

Scheme A in Figure 1 shows the simplest replanning process. Here no hierarchical model is needed. This is performed by D*_NODE_EXPANSION subprocedure in Section 3.3. A node (bridge node) is found in the same hierarchical level and function $f = g + h$ (h heuristic, g accumulated cost) invalidate the rest of possible solutions.

Schemes B and C show general cases where the replanned new path involves finding new bridge nodes in different abstract levels. This is performed by HIERARCHICAL_D*_NODE_EXPANSION subprocedure in Section 3.3. Precalculated paths (materialization of costs) avoid replanning paths between bridge nodes of different submaps.

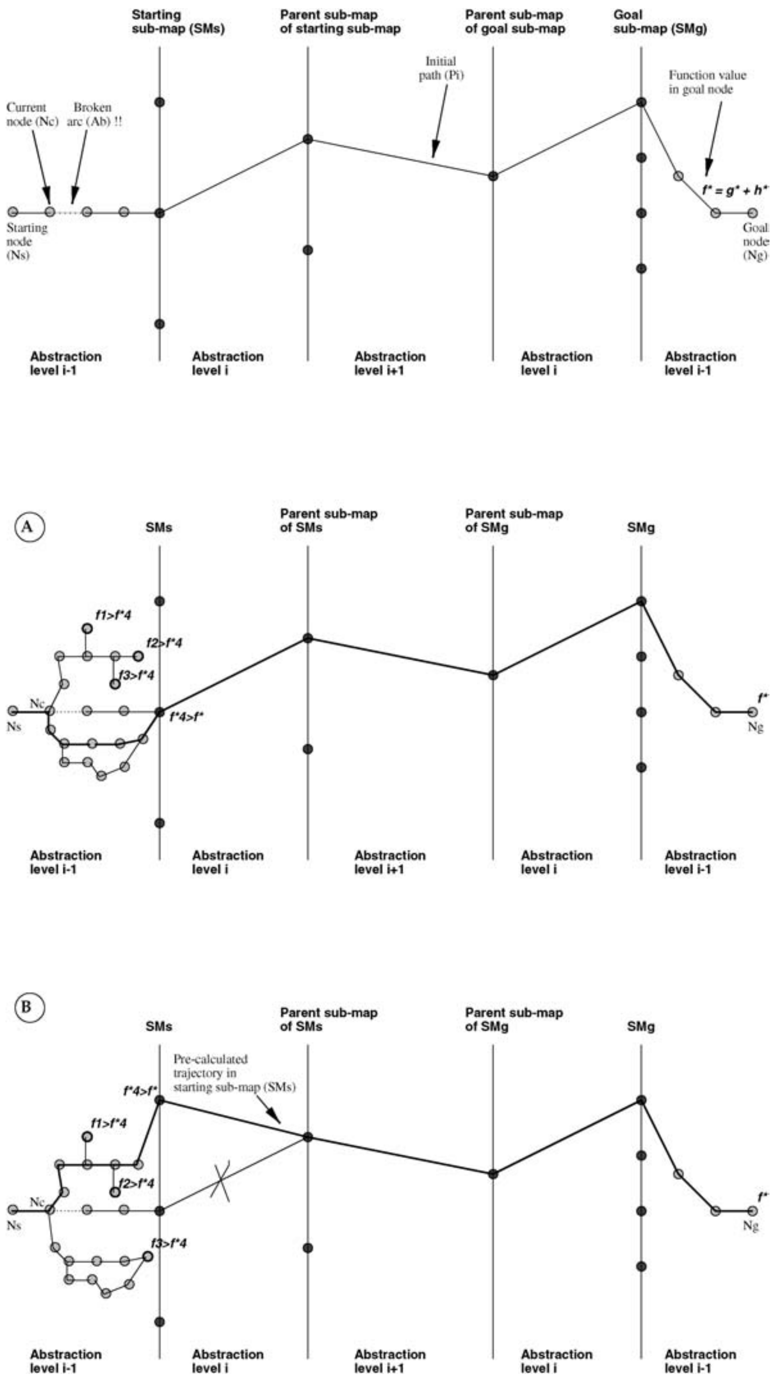

Figure 1. Three different examples of replanning processes using the hierarchical D^* algorithm proposed.

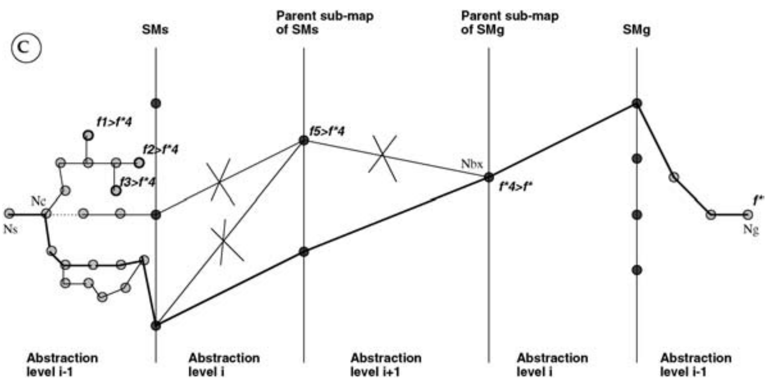

Figure 1. (Continued.)

3.5. VERTICAL PATH PLANNING

In some robot indoor environments, paths may start in a floor but end in another different floor of the same building. This is what we call *vertical path planning*. To adapt A^* or D^* algorithms (plain or hierarchical) to these environments without radical and costly changes, it is necessary to add some new elements to the H-Graph and to the path planning algorithm. These new elements are the *vertical bridge nodes*.

Thus, bridge nodes defined in section are divided in two classes: *horizontal bridge nodes* and *vertical bridge nodes*. Horizontal bridge nodes follow the definition given in Section 3. Vertical bridge nodes are almost equal to horizontal bridge nodes but conceptually they connect two submaps that represent two floors in a building. In fact, elevator entrances are modelled as vertical bridge nodes in a H-Graph. These nodes allow path planning between different floors of a building. This strategy allows to use the Euclidean distance heuristic, and therefore, function $f = g + h$ of D^* algorithm is still valid.

Nevertheless, there is another problem that a path planner has to solve: the elevator entrance selection. There are two possibilities: select again the elevator entrance in the initial path or select an alternative elevator entrance if it exists. The second option implies a total path recalculation process and it does not take advantage of the D^* algorithm. In addition, it does not guarantee completeness (i.e. it does not ensure a solution). However, if no path is found when selecting the elevator entrance in the initial path, an alternative elevator entrance must be selected.

4. Experimental Results

4.1. DESCRIPTION OF EXPERIMENTS

These experiments take into account not only continuous horizontal environments, but also *vertical environments* such as buildings. Path planning experiments are grouped into three categories:

Table I. Characteristics of the abstraction models based on a hospital and two industrial buildings

Number of:	Hospital	Industrial buildings
Buildings	1	2
Floors	4	5
Nodes	2349	417
Arcs	2422	463
Precalculated paths	3165	281
Arcs per node	1.9	2.1
Broken arcs	1	2
Initial calculated trajectories		
HPP paths	100	137
VPP paths	100	288
Replanned paths (solutions found)		
HPP paths	75	137
VPP paths	81	299

1. *Horizontal path planning (HPP)*: paths between nodes connected in a horizontal way.
2. *Vertical path planning (VPP)*: paths between nodes connected in a vertical way. Paths begin in one floor and finish in another floor of the same building.

Four maps have been tested in order to check the performance of the hierarchical D^* algorithm proposed. The first map is an abstract model of a hospital. The second map represents two industrial buildings. The third map is an Airport and the fourth map is the central building of a telephone company.

Characteristics of the maps and the trajectories tested are showed in Tables I and II.

The hierarchical D^* path planning algorithm with and without attached arrays of precalculated paths (materialization of costs) is compared to other path planning algorithms. These algorithms (or similar versions) have been traditionally used by most of the robot path planners. In order to perform these experiments, several initial paths are calculated between pairs of random nodes. Then, one or more arcs are deleted from each path ("broken arcs") to simulate obstacles. Each algorithm must find an alternative path. Path lengths (L) and computational time (T) needed in each algorithm are finally summed. These algorithms and their basic characteristics are as follows:

Table II. Characteristics of the abstraction models based on the central building of a telephone company and an airport

Number of:	Central building	Airport
Buildings:	1	2
Floors	5	2
Nodes	2794	369
Arcs	3188	401
Precalculated paths	12841	123
Arcs per node	2.1	2.1
Broken arcs	1	1
Initial calculated paths		
HPP paths	100	100
VPP paths	100	100
Replanned paths (solutions found)		
HPP paths	79	64
VPP paths	85	70

– D^*: it uses the same heuristic as the rest of algorithms: Euclidean distance.
– *D^* with prunes*: a version of a D^* algorithm. Useful when computational speed-up is needed. The D^* algorithm with prunes (and its A^* brother version) shows here an approximation of the total calculation time needed by another algorithm, widely used in mobile robotics: the *RTA** algorithm.
– *Hill climbing D*: a dynamic version of a hill climbing algorithm. It has the same backtracking process as *D^* with prunes* algorithm. This last process guarantees the property of completeness. That is to say, a solution/path (if it exists) will always be found.
– *Genetic Algorithm D1*: a dynamic version of a genetic algorithm. Initial populations are generated randomly.
– *Genetic Algorithm D2*: same as before but now paths from an initial population are generated using an analogous method based on [15]. It uses branch& bound algorithms to speed-up an initial population generation.

4.2. RESULTS ANALYSIS

In Appendix A are showed the experimental results obtained with the experiments described in Section 4.1.

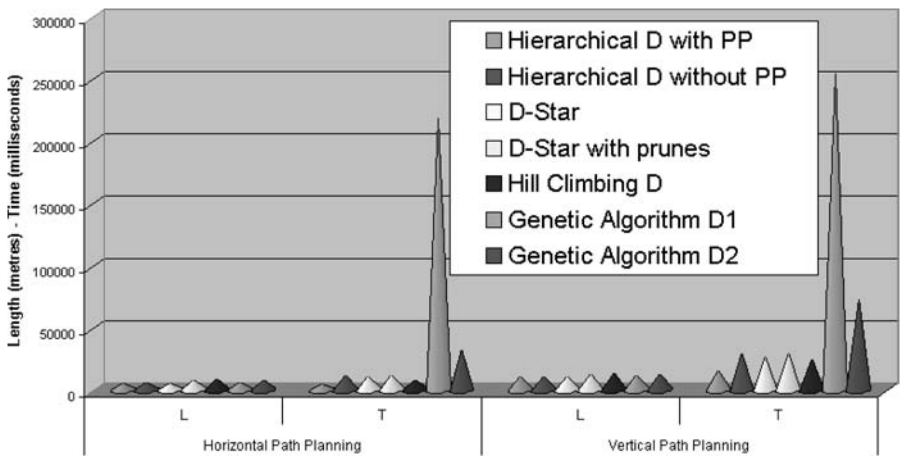

Figure 2. Industrial buildings results. Genetic algorithms are included. It can be noticed that computational time needed by genetic algorithms is high.

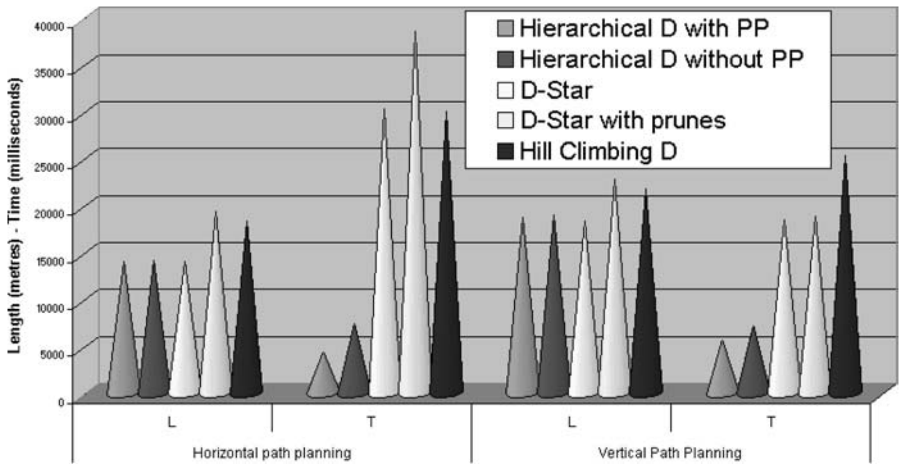

Figure 3. Central building of a telephone company results.

The first significant result that must be pointed out is the high calculation time cost needed by genetic algorithms. See Figure 2. Similar results are obtained using other maps. For simplicity only the results for the industrial buildings are showed. Although length of trajectories achieved are similar to other algorithms, the huge calculation time needed with these algorithms make genetic algorithms a not suitable choice. The main reason for this result is the abstract world model proposed (H-Graphs). Strings or trajectories/paths in this case, have variable length. The design of a genetic algorithm with a variable length coding scheme is usually ad hoc and complicated [21]. That is an important reason why genetic algorithms are mainly used in metric maps and not graphs.

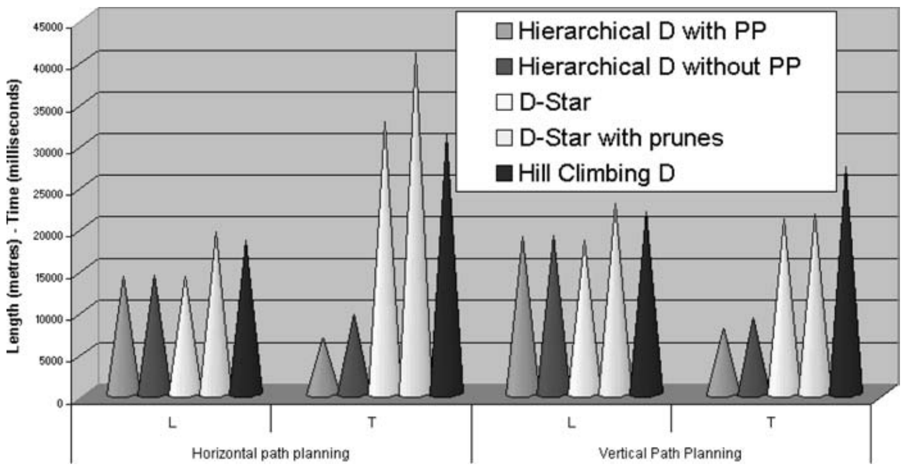

Figure 4. Central building of a telephone company results using only 3290 precalculated trajectories. Here hierarchical D^* uses 74% less of precalculated trajectories than in Figure 3. Length of path solutions (L) increase 6%.

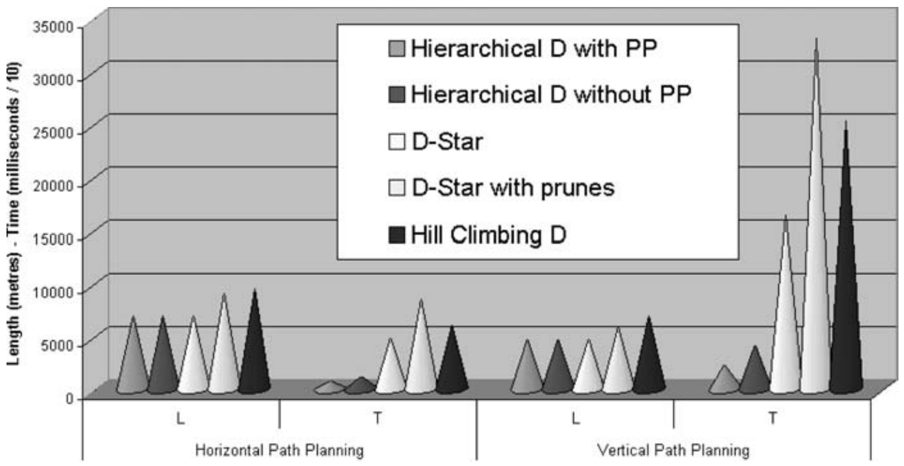

Figure 5. Hospital results.

Hierarchical D^* algorithms have an excellent calculation time (T) performance. Up to 85% calculation time reduction is obtained when comparing with the classic D^* algorithm and the hospital map (see Figure 5). It is also evident that not always increasing precalculated trajectories, solutions can be improved significantly. For example, with the telephone company building map (Figures 3 and 4), when decrementing precalculated trajectories from 12841 to 3290 (74% fewer trajectories), only 6% of computational time increment (T) is obtained. Sum of trajectories length (L) remain similar.

In general, quality of paths/solutions with hierarchical D^* algorithms are very close to optimal in vertical path planning (VPP). It must be pointed out that the hi-

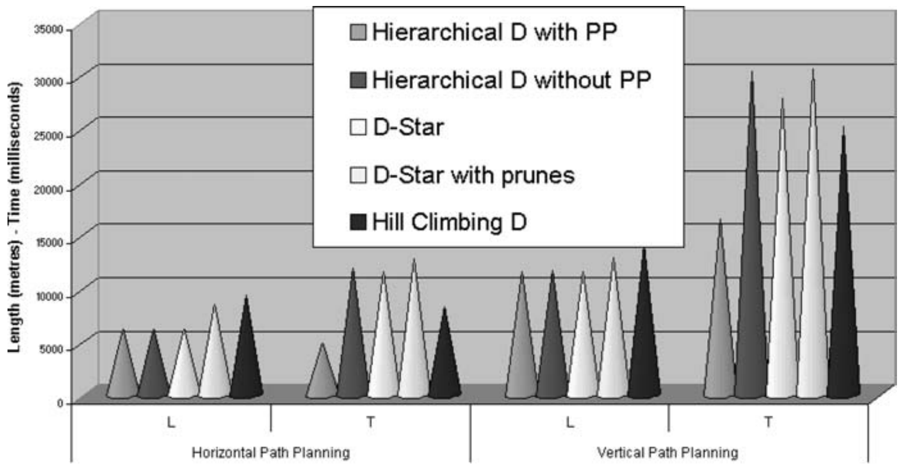

Figure 6. Industrial building results. Vertical path planning results show an overhead in hierarchical D^* algorithm without precalculated paths: computational time (T) is higher than its plain D^* version.

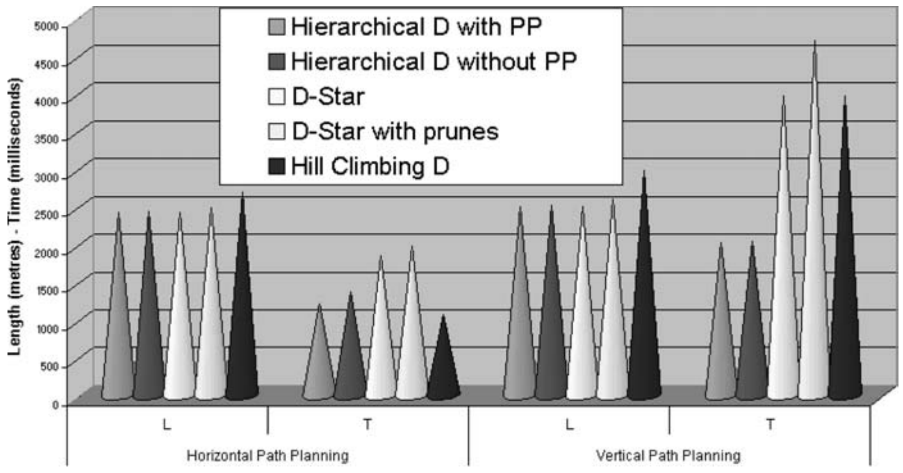

Figure 7. Airport results.

erarchical D^* algorithm with precalculated paths, as well as its plain version, generates time-optimal paths/solutions in HPP. The rest of algorithms can not ensure optimality in HPP.

Results using the hierarchical D^* algorithm without precalculated trajectories are also quite interesting and up to 75% calculation time reduction is achieved when comparing with the D^* algorithm. Sum of trajectories length (L) is very close to trajectories achieved with D^* algorithm. These results depends also on abstract world models (maps) which is now discussed.

In Figure 6 (results achieved with the industrial buildings map) it can be noted some type of overhead. The computational time with hierarchical D^* algorithm without precalculated trajectories is higher than D^* algorithm in HPP experiments. The node density in each floor (bones per floor) in both maps is the lowest among the four maps. Moreover, hierarchical D^* with or without precalculated trajectories must made more operations due to the complex of the abstract world models (H-Graph) it uses. A simple D^* algorithm just work with a plain graph. In this case, it is experimentally proved that the hierarchical D^* algorithm proposed is more effective in environments where a large search space can compensate any overhead caused by working with complex abstract models like H-Graphs. The Hierarchical D^* algorithm with precalculated trajectories is not so easily affected by overheads as its brother algorithm without precalculated trajectories. This happens because precalculated trajectories (materialization of costs) speed up search processes. They also help to avoid finding local minimums in a search process.

On one hand, results using the D^* algorithm with prunes do not show good calculation time reductions (T) as it was expected. On the other hand, sum of trajectories (L) are always above D^*. The dynamic hill climbing algorithm can reach up to 50% computational time reductions (T values) but also up to 35% increments when comparing with the D^* algorithm. Nevertheless, it always obtain longer (worse) trajectories (L values). This is again a direct consequence of the example maps used in these experiments. Algorithms and search strategies are very sensitive to maps and local minimums. Fortunately, as it was said before, hierarchical D^* algorithms can avoid easily this problems because it divides more efficiently any environment information.

5. Conclusions

Hierarchical techniques have been mainly used in static or off-line robot path planning but can be also extended to dynamic or on-line path planning successfully. Thus, D^* algorithm can be adapted to work in complex environment abstractions. In the solution described in this paper, traditional top-down refinement strategies used in hierarchical search are inverted in order to obtain optimal (time or length) solutions.

Moreover, experimental results have proved that the hierarchical D^* algorithm proposed can reduce calculation time significantly (specially if materialization of costs are used). Hierarchical on-line path planning in continuous or 2 dimension environments is also extended to not continuous or 3 dimensions environments where elevators are taken into account. Here no optimal solutions are generated but time calculation reductions and quality of solutions obtained make this hierarchical D^* algorithm a quite interesting choice. Other traditional techniques and algorithms (Hill Climbing or Genetic Algorithms, for example) do not seem to be appropriate solutions in this case because need more calculation time or just return worse solutions. Maybe better results could be achieved using hierarchical metric abstractions instead of using hierarchical topological abstractions (graphs).

In this context, type of abstractions used in path planning are experimentally proved to be a critical performance factor. Some algorithms can be very sensitive to this factor. However, it can be concluded that hierarchical D^* algorithm extensions are a feasible solution in dynamic robot navigation systems. Robot navigation systems that must work in large environments with not high computer requirements can be highly improved.

Appendix A. Experimental Results

L indicates total path length of random experiments. T indicates total computational time needed. Graphics help to compare quality of solutions found (L) with CPU time needed to obtain solutions (T). The lower L or T values, the better algorithm performance.

References

1. Ahuactzin, J. M., Talbi, E.-G., Bessiere, P., and Mazer, E.: Using genetic algorithms for robot motion planning, in: *Proc. of the 10th European Conf. on Artificial Intelligence*, 1992, pp. 671–675.
2. Ashiru, I., Czarnecki, C., and Routen, T.: Characteristics of a genetic based approach to path planning for mobile robots, *J. Network Computer Appl.* **19** (1996), 149–169.
3. Cagigas, D. and Abascal, J.: Hierarchical path search with partial materialization of costs for a smart wheelchair, *J. Intelligent Robotic Systems* **39** (2004), 409–431.
4. Civit-Balcells, A. and Abascal, J.: TetraNauta: A wheelchair controller for users with severe mobility restrictions, in: *3rd TIDE Congress*, Helsinki, Finland, 23–25 June 1998.
5. Conte, G. and Zulli, R.: Hierarchical path planning in a multi-robot environment with a simple navigation function, *IEEE Trans. Systems Man Cybernet.* **25**(4) (1995), 651–654.
6. Fernandez, J. A. and Gonzalez, J.: A general world representation for mobile robot operations, in: *Seventh Conf. of the Spanish Association for the Artificial Intelligence*, Malaga, Spain, 1997.
7. Fernandez, J. A. and Gonzalez, J.: Hierarchical graph search for mobile robot path planning, in: *Internat. IEEE Conf. on Robotics and Automation (ICRA'98)*, Leuven, Belgium, 1998, pp. 656–661.
8. Fernandez, J. A. and Gonzalez, J.: *Multi-Hierarchical Representation of Large-Scale Space*, Kluwer Academic, Dordrecht, 2001.
9. Fernandez, J. A. and Gonzalez, J.: Multihierarchical graph search, *IEEE Trans. Pattern Anal. Machine Intelligence* **24**(1) (2002), 103–113.
10. Hamidzadeh, B. and Shekhar, S.: Dynora II: A real-time planning algorithm, *J. Artificial Intelligence Tools (Special Issue on Real-Time AI)* **2**(1) (1993), 93–115.
11. Holte, R. C., Drummond, C., and Perez, M. B.: Searching with abstractions: A unifying framework and new high-performance algorithm, in: *Proc. of the 10th Canadian Conf. on Artificial Intelligence*, Morgan Kaufman, 1994, pp. 263–270.
12. Holte, R. C., Perez, M. B., Zimmer, R. M. and MacDonald, A. J.: The tradeoff between speed and optimality in hierarchical search, Technical Report, School of Information Technology and Engineering, University of Ottawa, Canada, 1995.
13. Hyun, W. K. and Suh, I. H.: A hierarchical collision-free path planning algorithm for robotics, in: *Intelligent Robots and Systems 95, Proc. 1995 IEEE/RSJ Internat. Conf. on 'Human Robot Interaction and Cooperative Robots'*, Vol. 2, 1995, pp. 448–495.

14. Jing, N., Huang, Y.-W., and Rundensteiner, E. A.: Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation, *IEEE Trans. Knowledge Data Engrg.* **10**(3) (1998), 409–432.
15. Kanoh, H., Kashiwazaki, A., Bui, L. T. H., Nishihara, S., and Kato, N.: Real-time route selection using genetic algorithms for car navigation systems, in: *IEEE Internat. Conf. on Intelligent Vehicles*, 1998, pp. 207–212.
16. Knoblock, C. A.: Learning abstraction hierarchies for problem solving, in: T. Dietterich and W. Swartout (eds), *Proc. of the 8th National Conf. on Artificial Intelligence*, AAAI Press, Menlo Park, CA, 1990, pp. 923–928.
17. Korf, R. E.: Real-time heuristic search, *Artificial Intelligence* **41–42** (1990), 189–211.
18. Latombe, J. C.: *Robot Motion Planning*, Kluwer Academic, Dordrecht, 1990.
19. Sasaki, T., Chimura, F., and Tokoro, M.: The Trailblazer search with a hierarchical abstract map, in: *Proc. of the 14th Internat. Joint Conf. on Artificial Intelligence*, Montreal, Canada, 1995, pp. 259–265.
20. Stentz, A.: Optimal and efficient path planning for partially-known environments, in: *Proc. of the IEEE Internat. Conf. on Robotics and Automation (ICRA'94)*, Vol. 4, 1994, pp. 3310–3317.
21. Sugihara, K. and Smith, J.: Genetic algorithms for adaptative planning of path and trajectory of a mobile robot in 2D terrains, *IEICE Trans. Inform. Systems* **E82-D**(1) (1999), 309–317.