Hierarchical D^* algorithm with materialization of costs for robot path planning

Daniel Cagigas

Department of Computer Architecture and Technology, Higher Technical School of Computer Science, University of Seville, Av. Reina Mercedes s/n, 41012 Sevilla, Spain

Abstract

In this paper a new hierarchical extension of the D^* algorithm for robot path planning is introduced. The hierarchical D^* algorithm uses a down-top strategy and a set of precalculated paths (materialization of path costs) in order to improve performance. This on-line path planning algorithm allows optimality and specially lower computational time. H-Graphs (hierarchical graphs) are modified and adapted to support on-line path planning with materialization of costs and multiple hierarchical levels. Traditional on-line robot path planning focused in horizontal spaces is also extended to vertical and interbuilding spaces. Some experimental results are showed and compared to other path planning algorithms.

Keywords: Path planning; Hierarchical search; D^* algorithm; Mobile robots

1. Introduction

Abstractions like graphs are useful to arrange and model information and they are very used by humans. Graphs are also widely used in mobile robotics to abstract environments [1]. Most of the mobile robot systems move into reduced environments where maps usually do not have more than a few hundreds of nodes. However, sometimes the path planner of a mobile robot must face large and structured environments where traditional *branch* & *bound* or *genetic algorithms* may not be efficient enough. Therefore, a plain graph information must be arranged in order to reduce complexity and gain efficiency and clarity. A hierarchical decomposition is necessary and H-Graphs (hierarchical graphs) are a very suitable choice.

Hierarchical search reduces complexity of operations and can lead to important improvements when is applied to mobile robots [2]. In fact, hierarchies of abstraction can reduce exponential complexity problems to linear

complexity [3]. This is important because, for example, A^* algorithm (widely used in robot path planning) has $O(n^2)$ time complexity and $O(b^d)$ space complexity, where "n" is the number of nodes, "b" is the branching factor and "d" depth level reached in a search.

A hierarchical path planner is supported by a hierarchy of abstractions representing different views of a robot environment (i.e. a hierarchical map). Paths are obtained using a refinement process through the hierarchy of abstractions in order to obtain a path[1] free of obstacles. There may be also a reconstruction process that link partial paths obtained after refinement.

Several references can be found in the bibliography. Thus, in [4] 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 [6,7] the trade off of using abstraction hierarchies is studied and analysed. In [8] a hierarchical path planning algorithm is proposed to plan a collision free path for mobile robots and robot manipulators. In [9,10] the concept of H-Graph (hierarchical graph) is described and applied to mobile robot path planning.

On-line path planning refers to problems where robots must replan their initial paths because their abstract model of an environment (map) has been updated. This usually happens when an unknown obstacle is found. Branch & Bound algorithms in dynamic environments are based on RTA^*–$LRTA^*$ [11], Dynora [12] and specially D^* [13] 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 constrained by computational time limitations. Genetic algorithms are an alternative path planning technique applied usually to metric maps. They provide flexibility in dynamic environments. Examples can be found in [14–16]. The newest strategies combine genetic and branch & bound algorithms. Thus, branch & bound algorithms are used to generate an initial population of paths for a genetic algorithm or regenerate it when an obstacle is found (see [17]). There are also some few on-line hierarchical robot path planners. For example, in [18] a two level hierarchical abstract map is used to speed up the problem of intercepting a moving object.

Hierarchical robot path planners almost always use two hierarchical levels to model an environment. Little attention has been given to mobile robot systems with more than two hierarchical abstract levels. Moreover, maps usually represent a continuous horizontal space. Mobile robot path planning may involve two floors of a building or even several floors of different buildings. In this work, H-Graphs and the D^* algorithm are adapted to on-line and *interbuilding* path planning.

Thus, the main contribution of this work is to convert the D^* algorithm into a hierarchical algorithm that uses precalculated paths (materialization of costs). This improves computational time performance of the D^* algorithm in large search spaces (including three dimension spaces), and therefore allows to obtain paths in a faster and accuracy way than traditional hierarchical path planning. The method proposed help for instance, to solve autonomous on-line robot navigation problems in large and/or complex indoor environments such as buildings. In some robot path planning modules, computer time availability may be limited due to real time embedded systems used. Until now on-line robot path planning systems have been usually designed for continuous and not complex environments, where quality and availability of paths obtained, were not a major problem.

In Section 2 the H-Graph model proposed is described. Section 3 describes the hierarchical path planner proposed. A previous description of the D^* algorithm fundamentals are also commented. In Section 4 experimental results obtained are analysed and compared to other algorithms. Finally, in Section 5 some conclusions are pointed out.

2. Map model

The H-Graph proposed is a sequence of hierarchical levels. The sequence is $L = \{L_0, L_1, L_2, \ldots, L_D\}$. The depth of the hierarchy is D. The "root level" is L_0 and it represents the highest abstract description of an environment.

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

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 ($0 \leq i \leq D$) 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 $G_0, G_1, G_2, \ldots, G_D$ is a graph $G = (N, A, C, W, T)$ where $N = N_0 \cup N_1 \cup \ldots \cup N_D$, $A = A_0 \cup A_1 \cup \ldots \cup A_D$, $C = C_0 \cup C_1 \cup \ldots \cup C_D$, $W = W_0 \cup W_1 \cup \ldots \cup W_D$, $T = T_0 \cup T_1 \cup \ldots \cup T_D$.

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 $x, y \in \mathbb{N}$. 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 $n_J \in G_j$ ($0 \leq j \leq D$). The dot notation is used:

- map $\rightarrow n_J.map = n_K$, where $n_J \in L_j$, $n_K \in L_k$, $j = k + 1$ ($0 < j \leq D$, $0 \leq k \leq D$) 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 $\rightarrow n_J.depth = x$, where $n_J \in L_x (0 \leq x \leq D)$. Namely, it returns the level of the hierarchy where n_J belongs.

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.

Bridge nodes are divided in two classes: *horizontal bridge nodes* and *vertical bridge nodes*. Horizontal bridge nodes follow the definition given before. 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 G. These nodes allow path planning between different floors of a building or even between floors of different buildings.

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 an 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 $n_J \in P_I = (n_0, n_1, \ldots, n_J, \ldots, n_L)$, $n_J \in N$, $P_I \in P$, $L + 1 = P_I.length$, $0 \leq J \leq L$. Namely, it returns the node of a path in position J.

Each submap node $n_I \in N_i \subset N (0 \leq i < D)$ has its own precalculated path set $PPS_{n_I} \in T_i \subset T (0 \leq i < D)$. 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 bridge 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* [19]. On one hand, materialization of cost requires extra storage space for paths and costs and off-line path calculation [2]. On the other hand, it can guarantee optimality in a classic refinement hierarchical search method.

Materialization of costs saves computational time too because it avoids refinement of nodes in deeper abstraction levels of the hierarchy. Extra store space and off-line path calculation are not serious drawbacks for a robot computer embedded system when working in static and well known environments such us offices, shops, industrial buildings, etc.

Furthermore, the map model is highly flexible and easily adaptable. If a building map has to be updated, the intrinsic modularity of H-Graphs makes easy a partial path recalculation. Depending on the path planning module requirements, some precalculated paths or path classes may be even added or removed from the H-Graph. This last point implies a variation of the concept of materialization of costs mentioned in [19]. It could be defined as *partial materialization of cost*.

3. Path planning

3.1. D^* algorithm

The D^* algorithm was first introduced by Stentz (see [13]). It defines and represents a dynamic or on-line version of the A^* algorithm. A^* algorithms are widely used in off-line path planning and robot motion planning. The D^* is path length optimal algorithm and saves a lot of computational time when comparing with brute-force methods.

A D^* algorithm restarts the general search process of an A^* algorithm when an obstacle is detected. This is equivalent to find a "broken" arc A_b in an initial path P_i of a graph. P_i is a time or length optimal path between an starting node N_s and a goal node N_g. Arc A_b is defined by two nodes: N_c (current node), which represents the current robot position, and N_n (next node), which represents the next robot's movement/destination. Nodes $\{N_\mathrm{s}, N_\mathrm{g}, N_c$ and $N_\mathrm{n}\} \in N$. Every node connected to the current node N_c, except N_n, is used to generate a new partial solution path set. Each new path P_new from the new path set is added to the D^* OPEN LIST. The OPEN LIST contains a set of partial solution paths that are expanded until N_g is reached. Thus, a first path/node expansion implies to generate new partial solution paths P_new composed of nodes included in $P_i^* = (N_s, \ldots, N_\mathrm{c}) \subset P_i$ and

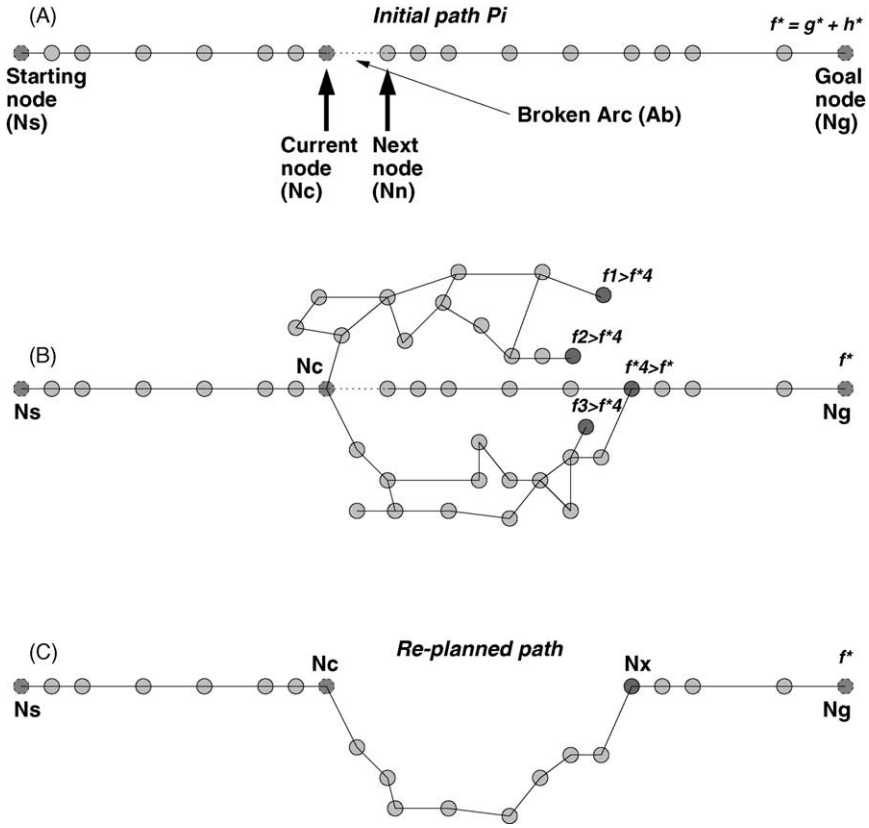

Fig. 1. D^* replanning example: scheme A shows an initial path P_i. N_c is the current node and A_b the 'broken' arc. In Scheme B nodes from OPEN LIST are expanded until another node $N_x \in P_i$ is reached. The rest of partial solution paths have a greater f value. This scheme represents the first algorithm end condition. Scheme C shows the final replanned path.

the set of neighbour nodes connected to N_c. Formally, a path $P_{\text{new}} = P_i^* \cup N_k$, where $\exists A_k = a(N_c, N_k, W_{(c,k)})$, $A_k \neq A_b$, $A_k \in A$ and $N_k \in N$. Arc A_b is ignored and can not be a part of any solution. The whole search process ends when:

(1) N_n or another node N_x included in the original path P_i is found and the rest of partial paths (solutions) included in the OPEN LIST 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 from N_c to N_g (see Fig. 1). Usually, function f implements the Euclidean distance.
(2) The goal node N_g is reached 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. Hierarchical path planning

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 part just expands N_a, that is to say, generates new partial paths joining $P_{current}$ and N_a neighbours nodes (lines 37–40). 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_{aux};
3: Path $P_{current}$, P_{new};
4: Path_Set $Open\_List$;
5: {End variable declaration.}
6: $P_{current} = (N_c)$;
7: $P_{current}.cost = 0$;
8: $Open\_List = (P_{current})$;
9: **while** ($Open\_List \neq NULL$) **do**
10: $P_{current} = GET\_BEST\_PATH(Open\_List)$;
11: $N_a = P_{current}.index(P_{current}.length - 1)$;
12: **if** ($N_a \in P_{initial}$) **then**
13: {The current partial path has intersected the initial path.}
14: $P_{new} = COMPLETE\_PATH(P_{current}, N_a, P_{initial})$;
15: $PROCESS\_SOLUTION(P_{new})$;
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: $HIERARCHICAL\_D^*\_NODE\_EXPANSION$ ($P_{current}$, N_a, $P_{initial}$, $Open\_List$);
21: **else**
22: {Both nodes are included in the same submap.}
23: $P_{new} = (N_a.map).pre\_paths(N_a, N_g)$;
24: **if** ($P_{new} \neq$ NULL) **then**
25: {There is a precalculated path between N_a and N_g.}
26: $P_{current} = P_{current} \cup P_{new}$;
27: $P_{current}.cost = P_{current}.cost + P_{new}.cost$;
28: $PROCESS\_SOLUTION(P_{new})$;
29: **else**
30: $D^*\_NODE\_EXPANSION$ ($P_{current}$, 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 hierarchy. Precalculated paths (materialization of costs) are used to avoid refining paths that cross N_a.}

36: $D^*\_SUBMAP\_NODE\_EXPANSION$ (P_{current}, N_{a}, $Open\_List$);
37: **else**
38: {D^* normal node expansion.}
39: $D^*\_NODE\_EXPANSION$ (P_{current}, N_{a}, A_b, $Open\_List$);
40: **end if**
41: **endwhile**
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 tree expansion.

D*_NODE_EXPANSION (Path P_{current}, Node N_{a}, Arc A_{b}, Path_Set $Open\_List$):

{Begin local variable declaration:}
Path P_{new};
Arc A_i;
Node N_i;
{End local variable declaration.}
for all ($A_i = a(N_{\text{a}}, N_i, w_{(a,i)}) \in A$, $A_i \neq A_{\text{b}}$, $N_i \in N$) **do**
  if ($N_i == N_{\text{g}}$) **then**
    {The goal node N_{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*. See scheme in Fig. 2. 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.

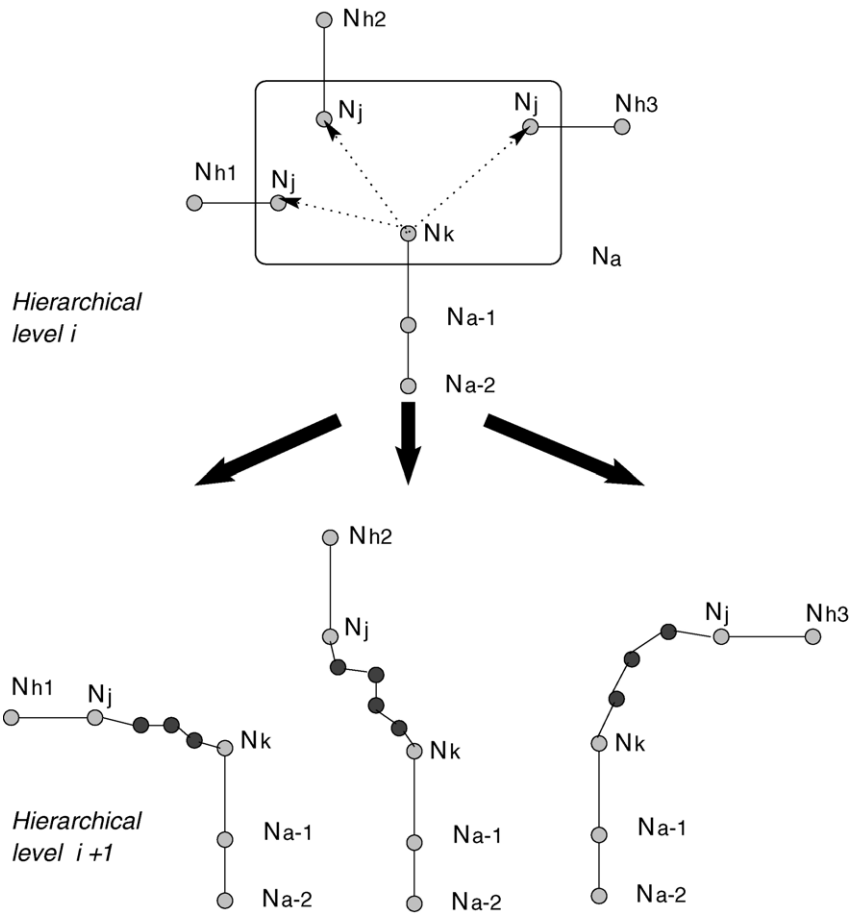

Fig. 2. Submap node expansion. A path $P_{\text{current}} = \{N_{a-2}, N_{a-1}, N_a\}$ generates three new equivalent paths. Each new path P_{new} substitutes submap node N_a by a precalculated path stored in N_a.

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_{last};
3: Path P_{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: $N_{last} = P_{\text{aux}}.index(P_{\text{aux}}.length - 1)$;
10: $N_h \Leftarrow N_h \in N \,/\, \exists a(N_{last}, N_h, w_{(last,h)}) \in A$ and $N_h.map = N_a.map = N_{a-1}.map$;
11: {In P_{current} the last node ($N_a \in SN$) is substituted by a refined path (P_{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: $P_{new}.cost = P_{current}.cost + P_{aux}.cost + w_{(last,h)}$;
14: $Open\_List = Open\_List \cup P_{new}$;
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 $N_{submap\_pre\_paths}$ 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}. See examples in Fig. 3.

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

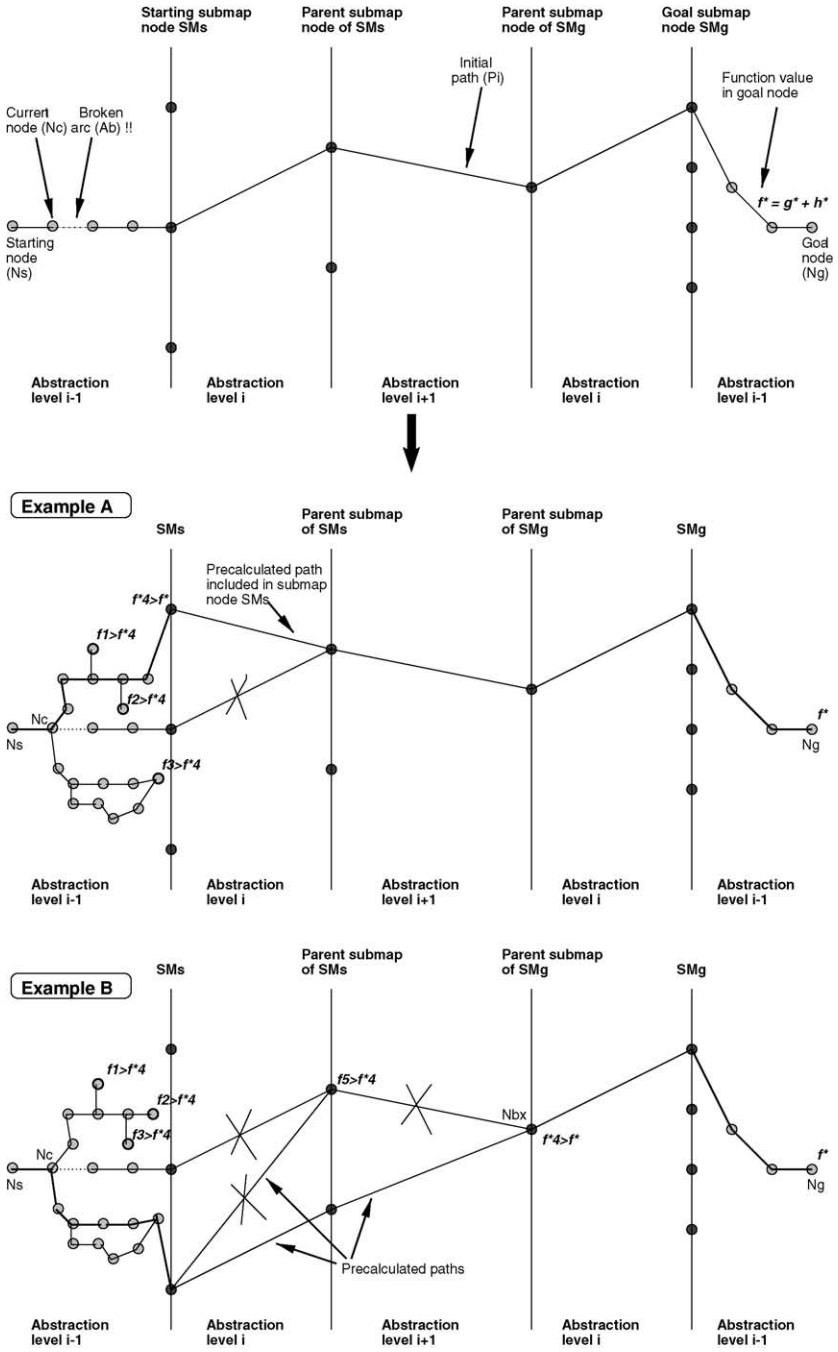

Fig. 3. Hierarchical D^* node expansion: two examples. A bridge node is found during a D^* algorithm search process and is not included in the initial path P_{initial}. Therefore, the search process restarts in another "brother" bridge node. Only precalculated paths are used. Example A: P_{initial} is intersected using one single precalculated path. Example B: linkage process is extended through two abstraction levels. In a general case the search process would continue until the submap node SM_g is reached.

31: **for all** ($N_{aux} \in N_{submap}.get\_bridge\_nodes$) **do**
32: $P_{aux} = N_{submap\_pre\_paths}.pre\_paths(N_a, N_{aux})$;
33: $P_{new} = P_{current} \cup P_{aux}$;
34: $P_{new}.cost = P_{current}.cost + P_{aux}.cost$;
35: $Open\_List = Open\_List \cup P_{new}$;
36: **end for**

Some important characteristics of D^* algorithms remain in this hierarchical extension. For example, this algorithm is still length optimal when precalculated paths (materialization of costs) are used. Time optimality instead of length optimality is also possible when robot turns are taken into account. Mobile robot turns imply stopping, turning and starting again. Therefore, a turn implies a time cost in a trajectory. In order to approximate length cost to time cost, a robot path planner based on this algorithm may associate a fixed length cost to every turn in a path. In expression $f = g + h$, h function does not change and is still Euclidean Distance. However, g function accumulates turns costs. Fixed length costs associated to turns depends on each specific robot system and they have to be estimated previously.

3.3. Vertical and interbuilding hierarchical path planning

Some mobile robots are specially designed for working in buildings. These type of indoor mobile robots can be autonomous office robots but also other type of robotic systems like intelligent electric powered wheelchairs. Buildings have nearly always more than one floor connected through elevators. Thus, elevators (or better said, their entrances) are a fundamental part that must be taken into account when designing the path planning module.

The objective consists in extending hierarchical path planning and H-Graphs but without any serious changes or modifications. For instance, a three to two dimension H-Graph transformation and a more sophisticated heuristic function would need extra computational time and it would imply a low path planner performance. In the proposed H-Graph model only a distinction between horizontal and vertical bridge nodes is necessary. However, the path planning algorithm extension requires an extra analysis.

A simple solution consists in considering *vertical bridge nodes* described in Section 2 as sub starting or sub goal nodes. In this way, the original path planning problem is divided into two parts: first obtain a path from the current node N_c to a vertical bridge node (elevator entrance) and second obtain a path from a vertical bridge node (contained in another "floor submap") to the goal node N_g. This implies that the hierarchical path planning algorithm described in Section 3.2 is executed twice. Notice that an elevator is modelled in an H-Graph as a sequence of vertical bridge nodes. Each vertical bridge node represents an elevator entrance. This solution allows to use Euclidean distance as heuristic, preserves the original H-Graph structure and does not alter the hierarchical search algorithm.

Nevertheless, there is still a problem that a path planner problem 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 (see Fig. 4). The second option implies a total path recalculation process and it does not take advantage of the D^* algorithm. In addition of this, 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.

Elevator entrances in a building floor are usually near. This means that bridge vertical nodes can be easily grouped into a submap node (cluster) in order to speed up search and save computational time. Thus, search is performed from one node (a start node) to n vertical bridge nodes, or better said, from a "start submap" to a "vertical bridge node submap". Depending on the H-Graph structure it is even possible to have several submaps containing vertical bridge nodes in the same "floor submap". This strategy takes advantage of the hierarchical path planner proposed and can give several solutions for the computational cost of one single path.

A more general path planning problem (and an extension of the vertical path planning problem) consists on finding a path through several submap nodes contained in different floors and different buildings. This type of path planning is called *inter building path planning*.

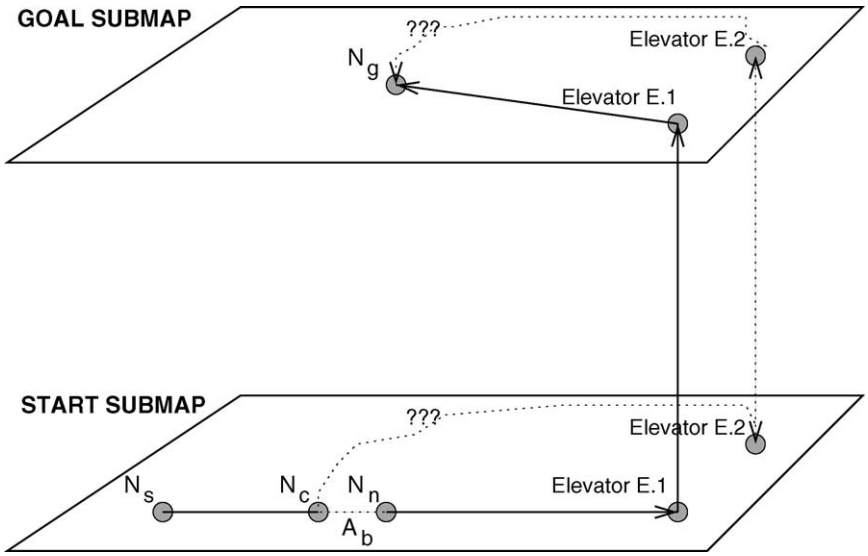

Fig. 4. On-line vertical path planning. A "broken" arc A_b (obstacle) is detected between the current node N_c and the next node N_n in an initial path that joins a start node N_s and a goal node N_g. The path planner module has to select an elevator entrance and there are two possibilities: the elevator entrance in the initial path (Elevator E.1) or an alternative elevator entrance (Elevator E.2). This last selection implies a complete path replanning.

3.4. Algorithm analysis

There are two ways of classifying search algorithms complexity: time complexity and space complexity. Space complexity refers to amount of memory needed by algorithms. Nowadays computer memories cost is reasonable so this is not a serious problem. However, exponential time complexity search problems can not be solved yet by faster microprocessors. Exponential time complexity problems must be transformed or approximated. As it was mentioned in Section 1 a hierarchical decomposition turns a exponential problem into a linear problem.

Exceptionally, and depending on the start and goal nodes selected, the hierarchical algorithm proposed may behave like a pure D^* algorithm. Here no hierarchical path search is performed: only first and fourth parts in $MAIN\_PROCEDURE$ (lines 12 to 15 and 37 to 39 respectively) are executed. That means a $O(2^N)$ time complexity order, where N is number of nodes.

In [2] a expression for the computational cost in hierarchies with more than two hierarchical levels is given and in [3] a similar conclusion is found. The expression that defines the computational cost is:

$$\overline{U_{\mathrm{H}}}(k-1) = U_{k-1} + \frac{1}{2}\frac{\frac{N^{(k+2)/k}}{2^{k-1}} - N^{3/k}}{N^{1/k} - 2} \tag{1}$$

where $\overline{U_{\mathrm{H}}}(k-1)$ is the computational cost of finding a path at the deepest level of the hierarchy, U_{k-1} the computational cost of the most abstract plain path search, N number of nodes and k number of hierarchical levels.

Expression (1) is a valid expression for hierarchical search algorithms when no *materialization of costs* are used. It is also a valid expression for the proposed hierarchical search algorithm because not every cost is *materializated*. In fact, the same hierarchical algorithm can be used without materialization of costs or a more reduced precalculated path set. This is what it has been called *partial materialization of costs*.

Refinement and hierarchical search tries to satisfy a trade-off between optimality and low computational cost. Cluster sizes (nodes per submap) is an important parameter that is strongly associated with optimality and computational efficiency. Small clusters (submaps) imply low computational costs but quality of solutions decrease and vice

versa. A partial materialization of costs has a double positive effect: speed up search (which is always interesting in computational powerless or real-time systems) and it helps to minimize loss of path accuracy (which is increased due to a big amount of small clusters/submaps).

Parameter $\overline{U_{\mathrm{H}}}(k-1)$ in expression (1) (computational cost of finding a path at the deepest level of the hierarchy) is directly determined by the number of nodes (N), the number of hierarchical levels (k) and cluster (submap) structures that are all strongly associated to maps (H-Graphs). Furthermore, expression (1) is also determined now by a partial materialization of costs. As it was said this does not alter time complexity order but improves time efficiency. Thus, the difference respect to other hierarchical search approaches depends strongly on the map structure and the selected materialization of costs (precalculated paths). Next section analyses some practical cases (maps) in order to test the model proposed.

4. Experimental results

4.1. Experiments description

Three types of path planning are considered:

1. *Horizontal path planning* (*HPP*): paths between nodes connected in a horizontal way. Equivalent to traditional robot path planning.
2. *Vertical path planning* (*VPP*): paths between nodes connected in a vertical way. Namely, paths that begin on a floor and finish on another floor of the same building.
3. *Inter-building path planning* (*IPP*): paths between nodes of different buildings. These paths begin on a floor of a building and finish on a floor of another building.

The hierarchical D^* path planner with and without materialization of costs (precalculated paths) is compared to other path planning algorithms. These path planning algorithms are widely used in Artificial Intelligence and robot motion planning. Their basic characteristics are as follows:

- D^*: it uses the same heuristic as the rest of algorithms: Euclidean Distance.
- D^* *with prunes*: a version of a D^* algorithm. The $D^* Open\_List$ is emptied periodically. Useful when computational speed-up is needed and optimality is not a primary goal. The D^* algorithm with prunes represents here an approximation of the calculation time of another widely used algorithm in robot path planning: the RTA^* algorithm [11].
- *Hill climbing D*: a dynamic version of a hill climbing algorithm. It has the same backtracking process as D^* and D^* with prunes. That is to say, if during the replanning process a node of the initial path P_i is found, the current path is completed with nodes of P_i. Backtracking guarantees the completeness property.
- *Genetic Algorithms D*1 *and D*2: in genetic algorithm D1 an initial population is generated randomly. Genetic algorithm D2 uses *branch*&*bound* algorithms to generate an initial population. A detailed description of this method is described in [17].

Algorithms are tested in four maps. These maps are possible environments where an autonomous mobile robot may work. In each map three sets of node pairs (start and goal) are selected randomly. These three node sets define the three path planning types: horizontal path planning (HPP), vertical path planning (VPP) and interbuilding path planning (IPP). Algorithms calculate an initial path for every pair of nodes in each map. Then, one or more arcs are deleted from each path (broken arcs) and algorithms find an alternative or replanned path. Notice that sometimes there is no possible solution. Path costs (length) and computational time consumed in each algorithm are finally summed.

Table 1
Characteristics of maps and experiments performed

Number of:	Hospital	Industrial buildings	Telephone C. Building	Lambert airport
Buildings	1	2	1	2
Floors	4	3	4	2
Hierarchical levels	7	4	5	3
Nodes	2349	417	2794	369
Arcs	2422	463	3188	401
Pre-calculated paths	3165	281	12841	123
Initial calculated trajectories				
HPP paths	100	137	100	100
VPP paths	100	288	100	100
IPP paths	0	588	0	100
Broken arcs per path	1	2	1	1
Replanned paths (solutions found)				
HPP paths	75	137	79	64
VPP paths	81	229	85	70
IPP paths	0	908	0	57

The first map corresponds to a hospital for disabled people in Toledo (Spain). The second map represents two fictitious industrial buildings. The third map is the Lambert Airport in St. Louis (USA) and the fourth map is the headquarters building of a telephone company. See Table 1 for a detailed description of the maps and paths tested.

Speed-up is measured using only CPU time (expressed in milliseconds). A better measurement counts also number of crosses found during a search process and the number of "overhead" operations performed [6]. This is done because algorithms are sensitive to low level programming details. However, this measurement can not be applied to genetic algorithms or to hierarchical algorithms when materialization of costs are used. This last point implies a possible loss of precision but it does not alter general results.

Maps and algorithms were programmed in JAVA (1.2. platform) and tested in a PC with an AMD Atholn 700 MHz processor and 128 MB of SDRAM memory. Both were integrated into an experimental visual tool (see Fig. 5) designed to support the map model described in Section 2. The tool had also the options to calculate partial materialization of cost and to convert H-Graphs into plain graphs. This was necessary to test D^*, D^* with prunes, hill climbing and genetic algorithms, with the same maps in the experiments.

4.2. Results description

Charts or Figs. 6–9 show experimental results achieved. On one hand, *Y*-axes indicate total computational time and total path length. Total path length (L) takes into account turn costs, so path lengths can be considered path costs actually. On the other hand, *X*-axes are divided into three parts whether interbuilding path planning is considered (more than one building) or two parts whether no interbuilding path planning is considered (only one single building). *X*-axes divisions correspond to horizontal path planning (HPP), vertical path planning (VPP) and interbuilding path planning (IPP). Each part or division on the *X* axis is also divided in two subparts that indicate total path cost/length (L) and total computational time (T). Charts help to easily compare relationship between path cost optimality (L) and computational speed (T).

Fig. 10 is a little different. It shows also genetic algorithms results in the airport map. *Z*-axes indicate path cost/length (L). *XY*-plane indicate path planning types (HPP, VPP, IPP) and algorithms. Computational time costs (T) for genetic algorithms are high when comparing with the rest of algorithms. Similar results are obtained in other maps. For a better clarity and legibility, no more genetic algorithm results are showed.

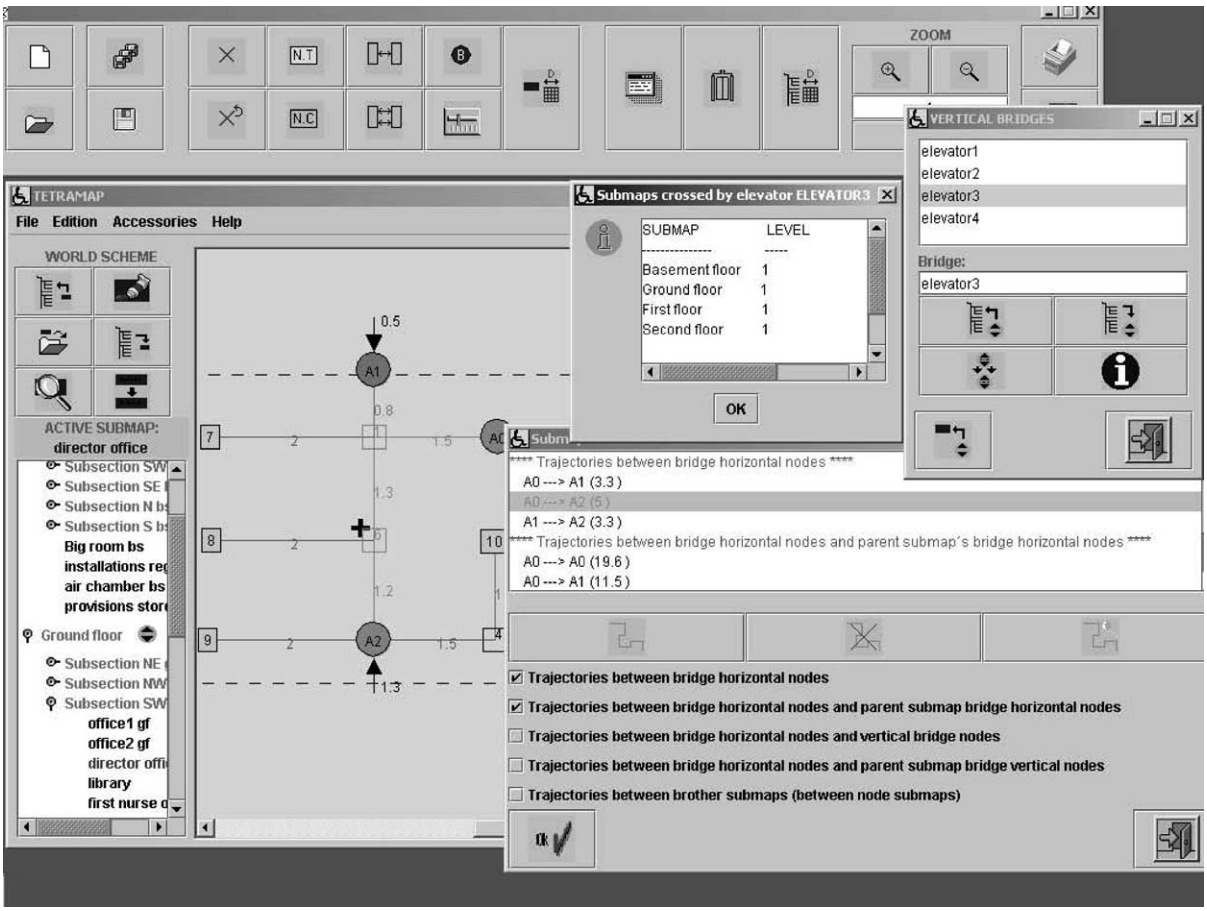

Fig. 5. Visual tool used to implement hierarchical maps and to test algorithms.

4.3. Results analysis

The first significant conclusion is the low efficiency obtained when using genetic algorithms. Although length of trajectories achieved are similar to other algorithms the huge calculation time needed with these algorithms (see Fig. 10) make genetic algorithms a not suitable choice. The main reason for this result is the abstract world model proposed (H-Graphs). In this case trajectories/paths have variable length. The design of a genetic algorithm with a variable length coding scheme is usually ad hoc and complicated [16]. That is an important reason why genetic algorithms are preferable used in metric maps and not graphs.

Computational time performance when using hierarchical D^* algorithms is quite good. Up to 85% calculation time (T) reduction is obtained when comparing with a classic plain D^* algorithm (see Fig. 8). Nevertheless, it must be remembered that unfortunately the hierarchical D^* algorithms proposed (with and without materialization of costs) are not real-time algorithms.

Quality of solutions obtained (path lengths/costs) are close to optimal. In fact, in HPP hierarchical D^* algorithm with materialization of costs and plain D^* algorithm, are length optimal.

Results using the hierarchical D^* algorithm without materialization of costs are also quite interesting. Up to 75% calculation time (T) reduction is obtained when comparing with the D^* algorithm. It is not length optimal but it is

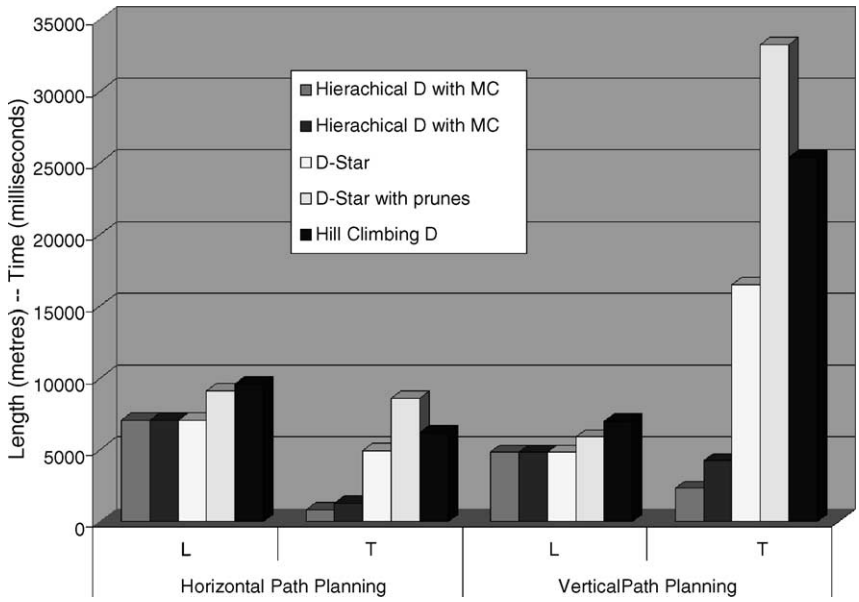

Fig. 6. Hospital results. *L* indicates total path cost (length) and *T* indicates total computational time.

close to results achieved with D^* algorithm in HPP cases. Results vary depending on the map because they depend strongly on graph structures.

In Fig. 7 (results in the industrial buildings map) it can be noted some type of overhead. The computational time (*T*) with hierarchical D^* algorithm without materialization of costs is higher than in D^* and Hill Climbing

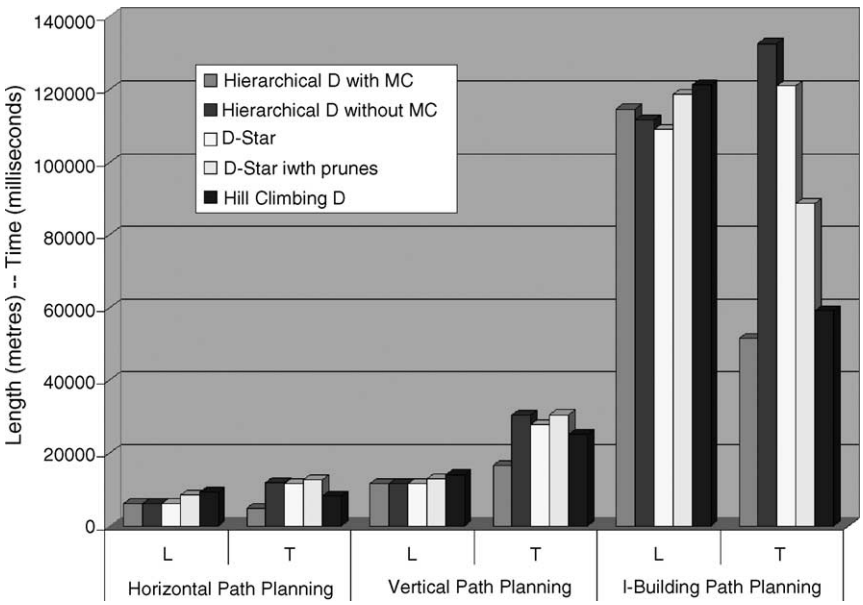

Fig. 7. Industrial Buildings results. *L* indicates total path cost (length) and *T* indicates total computational time.

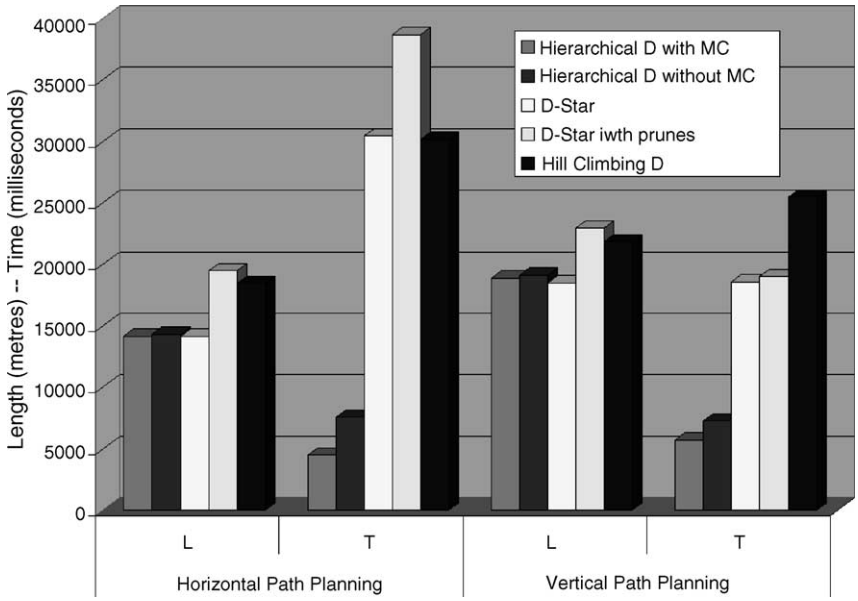

Fig. 8. Headquarters building results. *L* indicates total path cost (length) and *T* indicates total computational time.

algorithms in HPP and VPP experiments. A low node density in each floor (nodes per floor) and a low number of nodes involve in a path are determinant. Hierarchical algorithms perform more operations due to the H-Graphs complexity. Hierarchical path planning is useless and materialization of costs not very effective when replanned paths are contained in a single submap (node cluster) in the deepest hierarchical level of a H-Graph. This is equivalent

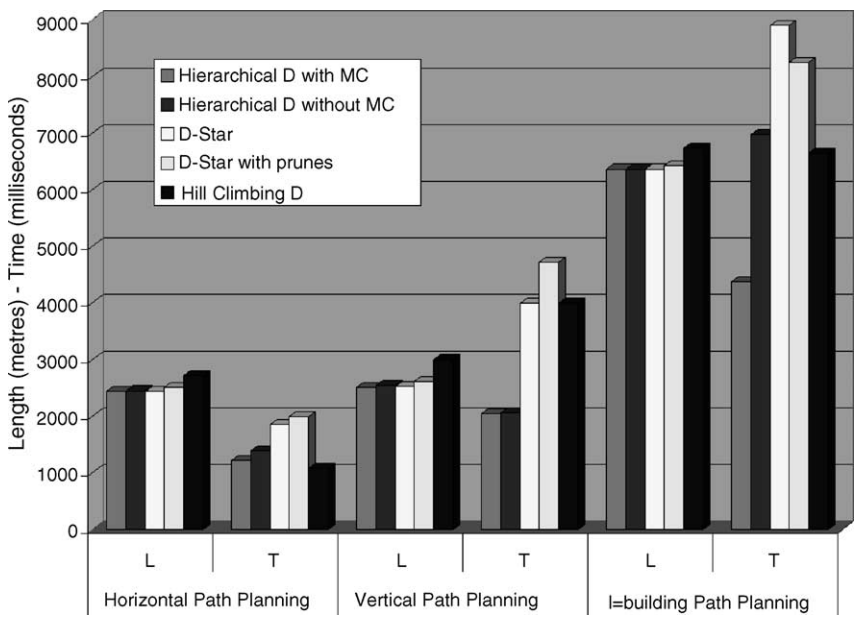

Fig. 9. Airport results. *L* indicates total path cost (length) and *T* indicates total computational time.

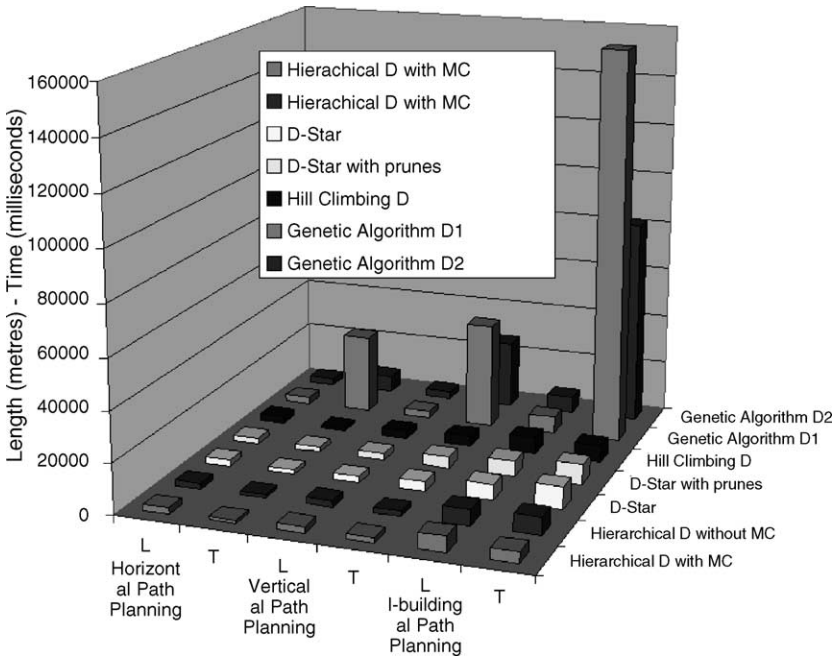

Fig. 10. Airport results. Genetic algorithms are included. Computational time is quite large when comparing with the rest of algorithms.

to find a path in a plain graph and implies to execute always $D^*\_NODE\_EXPANSION$ subprocedure in main procedure $D^*\_HIERARCHICAL\_PATH\_PLANNING$ (line 39) (see Section 3.2).

Path length (L) results using D^* with prunes algorithm are always above D^*. Its computational time is often the highest. On the contrary, using a dynamic hill climbing algorithm, computational time reductions can reach up to 50% when comparing with D^*. However, there are results that show computational time increments of more than 35% and quality of solutions obtained (sum of path lengths) is usually worse. Algorithms and path search strategies are very sensitive to map structures and local minimums. An example can be viewed in Fig. 7 where the hierarchical D^* algorithm without materialization of costs shows worse results than D^* algorithm in VPP and IPP cases. Nevertheless, the hierarchical D^* algorithm with materialization of costs, always has lower computational costs than D^* and path lengths (L) are similar. Thus, materialization of costs can help to prevent some negative side effects of pure hierarchical search.

5. Conclusions

The contribution of this paper is:

(1) A new version of the D^* algorithm for robot path planning that uses a hierarchical map and materialization of costs.
(2) An H-Graph model $G = (N, A, C, W, T)$ suitable for on-line hierarchical path planning. The H-Graph model proposed allows efficient robot on-line path planning and an easy information management.
(3) Extend on-line hierarchical path planning with materialization of costs to H-Graphs with several hierarchical levels (more than two).
(4) Extend traditional on-line hierarchical path planning to *vertical path planning* and *interbuilding path planning*.

Some other conclusions that can be drawn from this work:

(1) Experimental results demonstrate the utility of the model proposed in some practical cases. The method shows better results in large scale graphs and a significant node cluster (submap) density.
(2) The model proposed may be adapted to real-time robot path planning. Calculation and execution cycles can be intercalated in the same manner as the RTA^* algorithm does. Materialization of costs (precalculated paths) may help to improve real-time algorithms performance.

References

[1] J.C. Latombe, Robot Motion Planning, Kluwer Academic Publishers, 1990.
[2] J.A. Fernandez, J. Gonzalez, Multi-Hierarchical Representation of Large-Scale Space, Kluwert Academic Plublishers, 2001.
[3] R.E. Korf, Planning as search: a quantitative approach, Artif. Intell. 1 (33) (1987) 65–88.
[4] C.A. Knoblock, Learning abstraction hierarchies for problem solving, in: T. Dietterich, W. Swartout (Eds.), Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI Press, Menlo Park, California, (1990) 923–928.
[5] G. Conte, R. Zulli, Hierarchical path planning in a multi-robot environment with a simple navigation function, IEEE Trans. Syst., Man Cybernet. 25 (4) (1995) 651–654.
[6] R.C. Holte, C. Drummond, M.B. Perez, Searching with abstractions: a unifying framework and new high-performance algorithm, in: Morgan-Kaufman (Ed.), Proceedings of the 10th Canadian Conference on Artificial Intelligence, (1994) pp. 263–270.
[7] R.C. Holte, M.B. Perez, R.M. Zimmer, A.J. MacDonald, The tradeoff between speed and optimality in hierarchical search, Tech. rep., School of Information Technology and Engineering. University of Ottawa. Canada (1995).
[8] Woong Keun Hyun, Il Hong Suh, A hierarchical collision-free path planning algorithm for robotics, Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems 95, Human Robot Interaction and Cooperative Robots', vol. 2, (1995) pp. 448–495.
[9] J.A. Fernandez, J. Gonzalez, Hierarchical graph search for mobile robot path planning, Int. IEEE Conf. Robot. Automat. (ICRA'98), Leuven, Belgium, (1998) pp. 656–661.
[10] J.A. Fernandez, J. Gonzalez, Multihierarchical graph search, IEEE Trans. Patt. Anal. Mach. Intell. 24 (1) (2002) 103–113.
[11] R.E. Korf, Real-time heuristic search, Artif. Intell. 41–42 (1990) 189–211.
[12] B. Hamidzadeh, S. Shekhar, Dynora II: a real-time planning algorithm, J. Artif. Intell. Tools (Special Issue on Real-Time AI) 2 (1) (1993) 93–115.
[13] A. Stentz, Optimal and efficient path planning for partially-known environments, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'94), vol. 4, (1994) pp. 3310–3317.
[14] I. Ashiru, C. Czarnecki, T. Routen, Characteristics of a genetic based approach to path planning for mobile robots, J. Network Comput. Appl. 19 (1996) 149–169.
[15] J.M. Ahuactzin, E.-G. Talbi, P. Bessiere, E. Mazer, Using genetic algorithms for robot motion planning, in: Proceedings of the 10th European Conference on Artificial Intelligence, (1992) 671–675.
[16] K. Sugihara, J. Smith, Genetic algorithms for adaptative planning of path and trajectory of a mobile robot in 2D terrains, IEICE Trans. Informat. Syst. E82-D (1) (1999) 309–317.
[17] H. Kanoh, A. Kashiwazaki, L.T.H. Bui, S. Nishihara, N. Kato, Real-time route selection using genetic algorithms for car navigation systems, in: Proceedings of the IEEE International Conference on Intelligent Vehicles, (1998) pp. 207–212.
[18] T. Sasaki, F. Chimura, M. Tokoro, The trailblazer search with a hierarchical abstract map, in: Proceedings of the fourteenth International Joint Conference on Artificial Intelligence, Montreal, Canada, (1995) pp. 259–265.
[19] Ning Jing, Yun-Wu Huang, Elke A. Rundensteiner, Hierarchical encoded path views for path query processing: an optimal model and its performance evaluation, IEEE Trans. Knowledge Data Eng. 10 (3) (1998) 409–432.