View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by idUS. Depósito de Investigación Universidad de Sevilla

Hierarchical Path Search with Partial Materialization of Costs for a Smart Wheelchair *

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; e-mail: julio@si.ehu.es

Abstract. In this paper, the off-line path planner module of a smart wheelchair aided navigation system is described. Environmental information is structured into a hierarchical graph (H-graph) and used either by the user interface or the path planner module. This information structure facilitates efficient path search and easier information access and retrieval. Special path planning issues like planning between floors of a building (vertical path planning) are also viewed. The H-graph proposed is modelled by a *tree*. The hierarchy of abstractions contained in the *tree* has several levels of detail. Each abstraction level is a graph whose nodes can represent other graphs in a deeper level of the hierarchy. Path planning is performed using a path *skeleton* which is built from the deepest abstraction levels of the hierarchy to the most upper levels and completed in the last step of the algorithm. In order not to lose accuracy in the path *skeleton* generation and speed up the search, a set of optimal subpaths are previously stored in some nodes of the H-graph (path costs are partially *materialized*). Finally, some experimental results are showed and compared to traditional heuristic search algorithms used in robot path planning.

Key words: hierarchical graph search, path planning, smart wheelchairs.

1. Introduction

The problem presented in this paper focuses on the path planning subprocess of a special robotic system: an aided navigation system of an intelligent wheelchair called TetraNauta.

TetraNauta is a controller for standard electric-powered wheelchairs. Basically, it is an aided navigation system for users who have serious mobility restrictions. The navigation environment is modelled using a hierarchical graph (H-graph). In a TetraNauta system a building corridor represents a non-directed arc and a crossing between corridors represents a node. Thus, the graph contains a sub-set of the free configuration space of a world model and no static obstacles are expected to be

* This work is partially supported by the Spanish CICYT (Comision Interministerial de Ciencia y Tecnologia), Contract TER96-2056-C02-02.

found. The map generation process is made off-line taking into account usability issues of the user interface (see [2]) and path planning efficiency. Although there can be several TetraNauta wheelchairs in a same environment, path planning is performed in each wheelchair embedded computer system. Maps are usually large, complex and computational hardware resources are limited. Several real-time tasks control the aided navigation system and path searching has the lowest priority/importance. Tasks such as guidance, sensors attention or user monitoring, have the highest importance. These factors may delay the aided navigation system response when searching a new path. A more extended description of TetraNauta can be found in [1] and [13]. In this paper only the map model (abstract search space configuration) and the path search tasks are described.

In Section 2 a map model based on a H-graph is specified. Section 3 describes and analyses an hierarchical algorithm for path searching in the map model proposed. Section 4 shows some experimental results. Finally, in Section 5 conclusions are drawn and suggestions for further works are made.

2. Map Model

2.1. REQUIREMENTS AND LIMITATIONS

As was previously mentioned, the types of environments where the TetraNauta wheelchairs navigate (and therefore the maps used) are usually large and complex. Think for example in a hospital which is composed of several floors. On each floor there are different sections (and even subsections) that also contain rooms.

Abstractions are useful to arrange and model information. Graphs are widely used in mobile robotics to abstract environments [10]. However, if plain graphs have a large amount of nodes and arcs, path searching and information management becomes a problem. This last point is specially critical for TetraNauta users that must usually manage a lot of information. Rehabilitation is a key factor for wheelchair users and the whole aided navigation system must be designed according to usability principles. Therefore, graph information must be arranged in order to reduce complexity and gain efficiency and clarity. A hierarchical decomposition is necessary and H-graphs are very suitable in this case.

Moreover, hierarchies of abstraction can reduce exponential complexity problems to linear complexity [9]. Again this is important because path search algorithms like A^* tend to have exponential complexity when searching in large-scale graphs. Remember that computational time in TetraNauta is shared by multiple tasks (including path search). However, A^* guarantee optimality. Optimality is desirable but it is not necessary and critical in this problem. The computer-assisted system just gives aid in the navigation process and initial quasi-optimal paths are acceptable. Other hierarchical path searching approaches in robotics can be found in [3] and [11].

Maps in robotics usually represent a continuous horizontal space. Little attention has been given to applications where robots have to move across floors of a

building and/or between floors of different buildings. However, in TetraNauta path searching may involve two floors of a building (hospital). Therefore, H-graphs must be adapted to this type of path planning without losing their benefits.

2.2. DEFINITION

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 representation of an environment. On the contrary, L_D contains the most detailed representation and 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 pre-calculated 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 $N = N_0 \cup N_1 \cup \cdots \cup N_D$, $A = A_0 \cup A_1 \cup \cdots \cup A_D$, $C = C_0 \cup C_1 \cup \cdots \cup C_D$, $W = W_0 \cup W_1 \cup \cdots \cup W_D$, $T = T_0 \cup T_1 \cup \cdots \cup T_D$.

Elements n_J, n_K, w_H define an arc $a(n_J, n_K, w_H) \in A$, 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 $x, y \in \Re$. A weight $w_I \in W$ is real number $(w_I \in \Re)$.

Nodes can represent a cluster (subset) of nodes in a deeper abstraction level of the hierarchy. There are some functions/methods associated to a node $n_J \in G_j$ $(0 \leqslant j \leqslant D)$. We use the dot notation:

- map $\rightarrow$ $n_J.map = n_K$, 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. We call n_K *submap* (cluster or subset) of n_J.
- depth $\rightarrow$ $n_J.depth = x$, where $n_J \in L_x$ $(0 \leqslant x \leqslant D)$. Namely, the level of the hierarchy where n_J belongs.

Nodes are also classified in four classes: *end nodes*, *cross nodes*, *submap nodes* (node clusters) and *bridge nodes*. End nodes are starting or goal points that a user can select through the navigation user interface menus. Cross nodes represent subtargets that indicate turns or crossroads. Bridge nodes are nodes that connect a submap to a "parent" submap. Formally, $n_I \in G_i$ is a bridge node 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 $\rightarrow$ $n_I.get_bridge_nodes = BNS$, where $n_I \in N$, $n_I.depth < D$ and *BNS* is a bridge node set composed of bridge nodes that satisfies $\forall n_x \in BNS$, $n_x.map = n_I$. 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 given definition. 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 wheelchair 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 search algorithm. Weights (W) are attributes associated to every arc and indicate the cost of traversing an arc. They are used by the cost function of the search 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 two attributes/ methods:

- cost $\rightarrow P_I.cost = x$, where $x \in \Re$. It gets or assigns a path a cost to P_I.
- last_node $\rightarrow P_I.last_node = n_L$, where $P_I = (n_0, n_1, n_2, \ldots, n_L)$ and $n_0, n_1, n_2, \ldots, n_L \in N$.

Each submap node $n_I \in N_i \subset N$ $(0 \leqslant i < D)$ has its own pre-calculated 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 pre-calculated path set PPS_{n_I} that contains P_Z. The nodes n_X and n_Y will be bridge nodes typically. If node n_I is not a submap node or it does not contain the required path, the method/function returns NULL.

Pre-calculated 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.

Pre-calculated paths avoid recalculating several subpaths in a hierarchical search process. This is called *materialization of costs* [7]. On one hand, materialization of cost requires extra storage space for paths and costs and off-line path calculation [5]. 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 in modern computer embedded systems or static environments like buildings, respectively.

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 a partial path recalculation easy. Depending on the path planning module requirements, some pre-calculated paths or path classes may be even added to the H-graph or removed. This last point means a new concept: a *partial materialization of costs*.

2.3. EXAMPLE

In Figure 1 there is an example of how a hospital complex is modelled. The left column shows hierarchical levels L_i and the right column shows graphs G_i ($0 \leqslant i \leqslant 4$). Cross nodes and end nodes are represented by small black filled squares. End nodes are only showed in G_4. Notice that end nodes in L_4 are marked with dotted lines. Submaps are represented in G_j by not filled rectangles and in L_j by ellipses ($0 \leqslant j \leqslant 3$). There can be also seen two horizontal bridge nodes: one in G_3 and the other in G_4. No vertical bridge nodes are showed but they can be included in any graph except in G_0 and G_1.

3. Path Searching

The path planning module of a TetraNauta wheelchair finds paths between two points represented by a pair of nodes in a map. No static obstacles or unknown spaces are expected to be found. On-line path planning and obstacle avoidance are treated separately and are not described in this paper.

3.1. ALGORITHM DESCRIPTION

There are several steps involved in the path search process that can be summed up in two. First, a path *skeleton* set that joins a start node N_s and a goal node N_g is constructed. Second, the best path *skeleton* is completed and returned. A path *skeleton* P_S is a not complete path because it does not contain a continuous node sequence between N_s and N_g. Namely, there is a lack of intermediate nodes in P_S that must be added. Path *skeletons* are constructed as a sequence of subgoals that are reached systematically. These subgoals are special nodes previously called as *bridge nodes* in Section 2.2.

Basically, in a *skeleton* development process the search algorithm tries to find the best trajectories or paths between the submaps where a start node N_s and a goal node N_g are included. These submaps are called “start submap” SM_s and “goal submap” SM_g. SM_s and SM_g are linked to their “parent” submaps in an upper level of the hierarchy of abstractions through their bridge nodes. This process is repeated

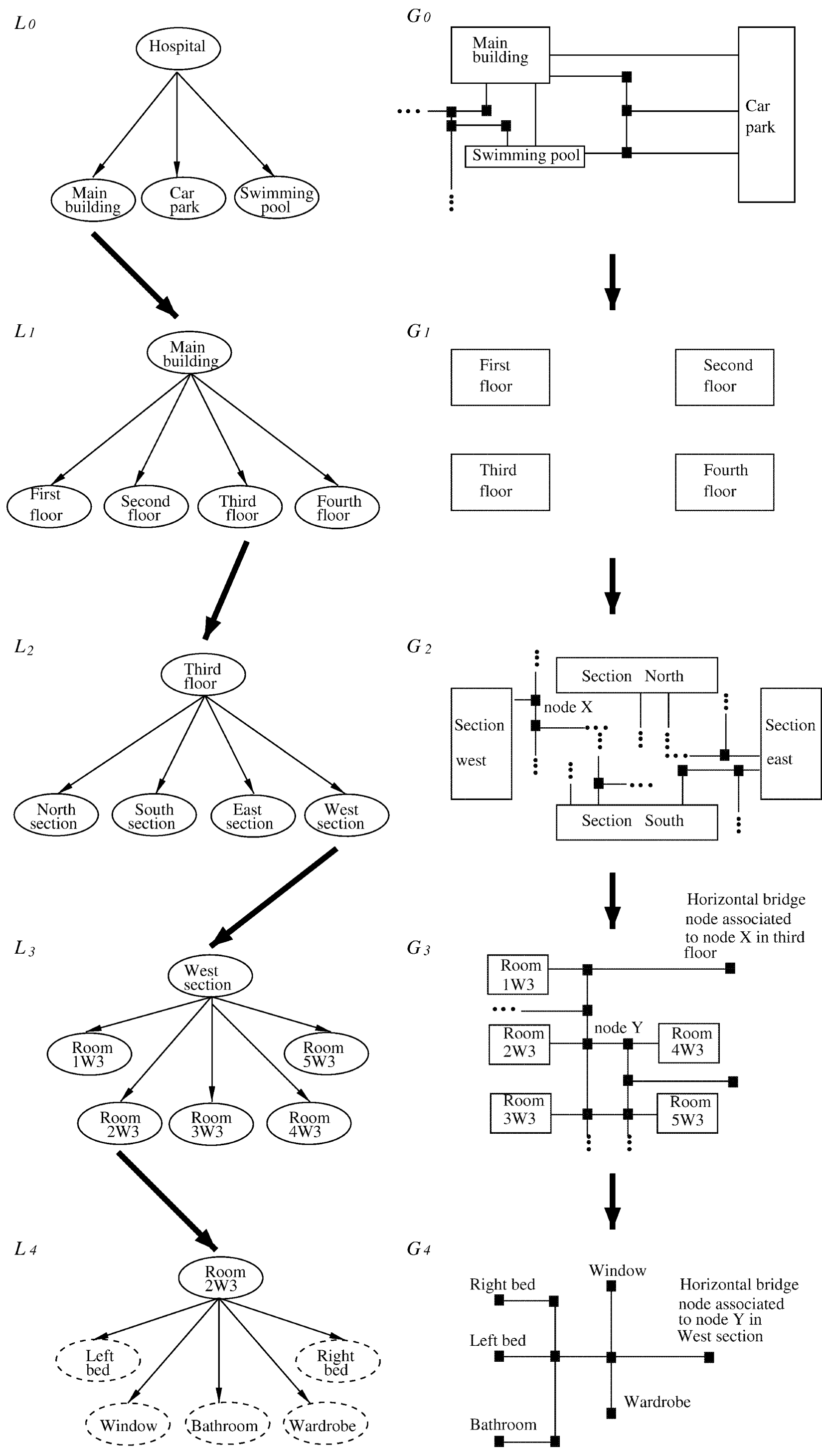

Figure 1. Example of map model.

until both set of partial paths converge into a common submap. This last submap could be even the first abstraction level of the hierarchy ("root" submap at level 0). Every path used to link sub-maps is taken from $T \in G = (N, A, C, W, T)$. The search algorithm defines a bottom-up process instead a traditional down-top refinement process, typically used when working with hierarchical graphs.

The MAIN PROCEDURE consists on four parts that take into account four different cases. Each case depends on the submap (cluster) and the level of hierarchy where N_s and N_g are included.

1. Both nodes are included in the same submap (line 6). This is the simplest case. It is almost equivalent to a plain graph search and an A^* algorithm is used. Notice that if a submap node is found during the search process there is no need to refine it because path costs are *materializated*.
2. N_s is in a deeper level of the hierarchy than N_g (line 14). A path *skeleton* set that joins N_s and N_g is constructed. It begins in $N_s.depth$ and finish in $N_g.depth - 1$ (bottom-up process).
3. Same as before but swapping N_s and N_g (line 20).
4. Both nodes are in the same hierarchical level (line 26). The same previous process is repeated for N_s and N_g until both sets of path *skeletons* converge in a common submap in an upper level of the hierarchy.

There is also a final procedure for cases 2, 3 and 4 (line 32): path *skeleton* sets are connected to N_g (case 2), N_s (case 3) or each other (case 4). Finally, the best path *skeleton* is selected, completed and returned (line 33). Now, the hierarchical path search algorithm is detailed. Algorithm descriptions are based on definitions in Section 2.2. Comments appear into { } symbols.

MAIN PROCEDURE. Hierarchical path search (Node N_s, Node N_g):

```
 1: {Begin variable declaration:}
 2: Integer L_s, L_g, L_sg;
 3: Path Path_aux;
 4: Path_Set PS_1, PS_2, PS_3;
 5: {End variable declaration:}
 6: if (N_s.map == N_g.map) then
 7:    {Case 1: No hierarchical path search is performed.}
 8:    {Pre-calculated paths avoid submap nodes refinement.}
 9:    return (PATH_SEARCH(N_s, N_g));
10: end if
11: {General process:}
12: L_s = N_s.depth;
13: L_g = N_g.depth;
14: if (L_s > L_g) then
15:    {Case 2: N_s is in a deeper abstraction level of the hierarchy.}
16:    PS_1 = GET_PATH_SKELETON_SET(N_s, L_g);
```

17: $Path_{aux} = (N_g)$;
18: $Path_{aux}.cost = 0$;
19: $PS_2 = Path_{aux}$;
20: **else if** $(L_s < L_g)$ **then**
21: {Case 3: N_g is in a deeper abstraction level of the hierarchy.}
22: $PS_1 = GET_PATH_SKELETON_SET(N_g, L_s)$;
23: $Path_{aux} = (N_s)$;
24: $Path_{aux}.cost = 0$;
25: $PS_2 = Path_{aux}$;
26: **else**
27: {Case 4: N_o and N_g are in the same level of the hierarchy.}
28: $L_{sg} = FIRST_COMMON_LEVEL(N_s, N_g)$;
29: $PS_1 = GET_PATH_SKELETON_SET(N_s, L_{sg})$;
30: $PS_2 = GET_PATH_SKELETON_SET(N_g, L_{sg})$;
31: **end if**
32: $PS_3 = LINK_PATH_SKELETON_SETS(PS_1, PS_2)$;
33: **return** $(COMPLETE_BEST_PATH_SKELETON(PS_3))$.

For simplicity there are three subprocedures in MAIN PROCEDURE that will be not detailed:

- *PATH_SEARCH* (line 9).
- *FIRST_COMMON_LEVEL* (line 28).
- *COMPLETE_BEST_PATH_SKELETON* (line 33).

The first subprocedure is similar to an A^* algorithm but with submap (cluster) nodes. The second subprocedure searches the first submap that includes or contains N_s and N_g in an upper level of the hierarchy and returns its depth. The third subprocedure uses the *PATH_SEARCH* subprocedure to convert a path *skeleton* into path.

The *GET_PATH_SKELETON_SET* subprocedure is a key part in the bottom-up hierarchical path search algorithm. It performs the path *skeleton* linkage through the different levels of the hierarchy. Submaps are linked to their "parent" submaps through their bridge nodes. Only pre-calculated paths (materializated costs) contained in $T \in G = (N, A, C, W, T)$ are used. This will mean a significant computational time reduction and a better quality (optimality) of paths returned.

First *for* loop selects a bridge node in a "parent" submap (line 11) and second *for* loop selects a bridge node in a "children" submap (line 14). The pre-calculated path between both nodes that optimises the current *skeleton* path accumulate costs is selected. This submap linkage process is repeated through several abstraction levels recursively. The *while* loop (line 9) stops the process in the previous (deeper) hierarchy level of L_g, where L_g is the hierarchy of the goal node.

GET_PATH_SKELETON_SET (Node N_s, Integer L_g).
1: {Begin variable declaration:}
2: Node *Current_submap*, N_i, N_1;

3: Path P_i, P_j, P_x;
4: Path_Set PS, PS_{new};
5: {End variable declaration:}
6: $NS = (N_s.map).get_bridge_nodes$;
7: $PS = ESTIMATE_PATHS\ (N_s, NS)$;
8: $Current_submap = N_s.map$;
9: **while** ($Current_submap.depth > L_g$) **do**
10: $PS_{new} = \emptyset$;
11: **for all** $N_i \in (Current_submap.map).get_bridge_nodes$ **do**
12: $P_i = \emptyset$;
13: $P_i.cost = MAXIMUN$;
14: **for all** $P_j \in PS$ **do**
15: $N_1 = P_j.last_node$;
16: $P_x = P_j \cup Current_submap.pre_paths(N_1, N_i)$;
17: $P_x.cost = P_j.cost + (Current_submap.pre_paths(N_1, N_i)).cost$;
18: **if** ($P_x.cost \leqslant P_i.cost$) **then**
19: $P_i = P_x$;
20: **end if**
21: **end for**
22: $PS_{new} = PS_{new} \cup P_i$;
23: **end for**
24: $PS = PS_{new}$;
25: $Current_submap = Current_submap.map$;
26: **end while**
27: **return** PS.

The *ESTIMATE_PATHS* subprocedure is included in the previous subprocedure (line 7). It initialises the path *skeleton* set that will join a sequence of submaps through several hierarchical levels. Each pseudo-path included in this initial *skeleton* path set has only two nodes: the start node and a bridge node of the submap where it belongs. Path costs are approximated with a HEURISTIC function (line 8).

The heuristic function is based on the Euclidean distance and initialises the cost (cost function value) of a path *skeleton*. The heuristic function utilizes coordinates $C \in G = (N, A, C, W, T)$ associated to nodes (N) and estimates a path length (cost) between two nodes. The cost function implemented f that the search algorithm tries to minimize is $f = g + h$, where g is accumulated cost and h estimated cost to goal. Function f is the same cost function used in A^* algorithms.

ESTIMATE_PATHS (Node N_s, Node_Set NS).
1: {Begin variable declaration:}
2: Path_Set $PS = \emptyset$;
3: Path P_i;

4: Node N_i;
5: {End variable declaration:}
6: **for all** $N_i \in NS$ **do**
7: $P_i = (N_s, N_i)$;
8: $P_i.cost = HEURISTIC(N_s, N_i)$;
9: $PS = PS \cup P_i$;
10: **end for**
11: **return** PS.

The *LINK_PATH_SKELETON_SETS* subprocess joins two path *skeleton* sets called PS_1 and PS_2. First *for* loop (line 6) takes a path *skeleton* from PS_1 and finds another path *skeleton* in PS_2 that minimize the function cost f of the resulting path. If no pre-calculated paths are found during this process, then costs are estimated using the HEURISTIC function (line 27).

LINK_PATH_SKELETON_SETS (Path_Set PS_1, Path_Set PS_2).
1: {Begin variable declaration:}
2: Path $Path_{pre}$, $Path_{new}$, P_I, P_J;
3: Path_Set $PS_{result} = \emptyset$;
4: Node N_{aux};
5: {End variable declaration:}
6: **for all** $P_I \in PS_1$ **do**
7: **for all** $P_J \in PS_2$ **do**
8: $N_{aux} = (P_I.last_node).map$;
9: $Path_{pre} = N_{aux}.pre_paths(P_I.last_node, P_J.last_node)$;
10: **if** ($Path_{pre} \neq$ NULL) **then**
11: {$PS1_1$ and PS_2 are NOT included in "brother" submaps.}
12: {$Path_{pre}$ is a pre-calculated path of class 2. Namely, it links two bridge nodes of a "parent" and a "children" submap.}
13: $Path_{new} = P_I \cup Path_{pre} \cup P_J$;
14: $Path_{new}.cost = P_I.cost + Path_{pre}.cost + P_J.cost$;
15: **else**
16: $N_{aux} = N_{aux}.map$;
17: $Path_{pre} = N_{aux}.pre_paths(P_I.last_node, P_J.last_node)$;
18: **if** ($Path_{pre} \neq$ NULL) **then**
19: {$PS1_1$ and PS_2 are included in "brother" submaps.}
20: {$Path_{pre}$ is a pre-calculated path of class 3. Namely, it links two bridge nodes of a "brother" submaps.}
21: $Path_{new} = P_I \cup Path_{pre} \cup P_J$;
22: $Path_{new}.cost = P_I.cost + Path_{pre}.cost + P_J.cost$;
23: **else**
24: {No pre-calculated path available.}
25: $Path_{new} = P_I \cup P_J$;

26: $Path_{new}.cost = P_I.cost + P_J.cost +$
$HEURISTIC(P_I.last_node,\ P_J.last_node);$
27: **end if**
28: **end if**
29: $PS_{result} = PS_{result} \cup Path_{new};$
30: **end for**
31: **end for**
32: **return** PS_{result}.

3.2. EXAMPLE

Figure 2 shows an example of a path generation divided in four steps. The H-graph has three hierarchical levels L_0, L_1, L_2 and their corresponding G_0, G_1, G_2 graphs. There are four submap nodes including the submap at L_0 ("root" level composed of only one node). Submaps are represented by rectangles. Two submap nodes at level L_1 are brother submaps (indicated in scheme A). Cross nodes are represented by grey filled circles, bridge nodes by black filled circles and end nodes by a cross. For clarity, no arcs and only two end nodes (start node and goal node) have been drawn. The start node is in a deeper abstraction level of the H-graph (level 2) than goal node (level 1) so this is case 2 (line 14) in MAIN PROCEDURE.

Scheme A shows actions associated to *ESTIMATE_PATHS* subprocedure (called from subprocedure *GET_PATH_SKELETON_SET* at line 7). Initial path costs are estimated using the heuristic and paths have only two nodes: the start/goal node and a bridge node.

Scheme B continues a step further in *GET_PATH_SKELETON_SET*: pre-calculated paths between bridge nodes of G_2 and G_1 are linked to previous paths.

In scheme C path *skeleton* sets are joined. Paths sets are in "brother" submaps and they are joined through their bridge nodes: this case corresponds to lines 18 to 22 in *LINK_PATH_SKELETON_SETS* subprocedure.

Finally, in scheme D the best *skeleton* path is selected and completed (line 32 in MAIN PROCEDURE). Here arcs of the resulting path are marked with thin lines.

3.3. VERTICAL AND INTER-BUILDING PATH PLANNING EXTENSIONS

TetraNauta wheelchairs are specially designed for working in hospitals or day centers. Such institutes have usually 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 search and H-graphs but without any serious changes or modifications. For example, a more sophisticated heuristic function would need extra computational time. In the H-graph model proposed only a distinction between horizontal and vertical bridge nodes is necessary (see

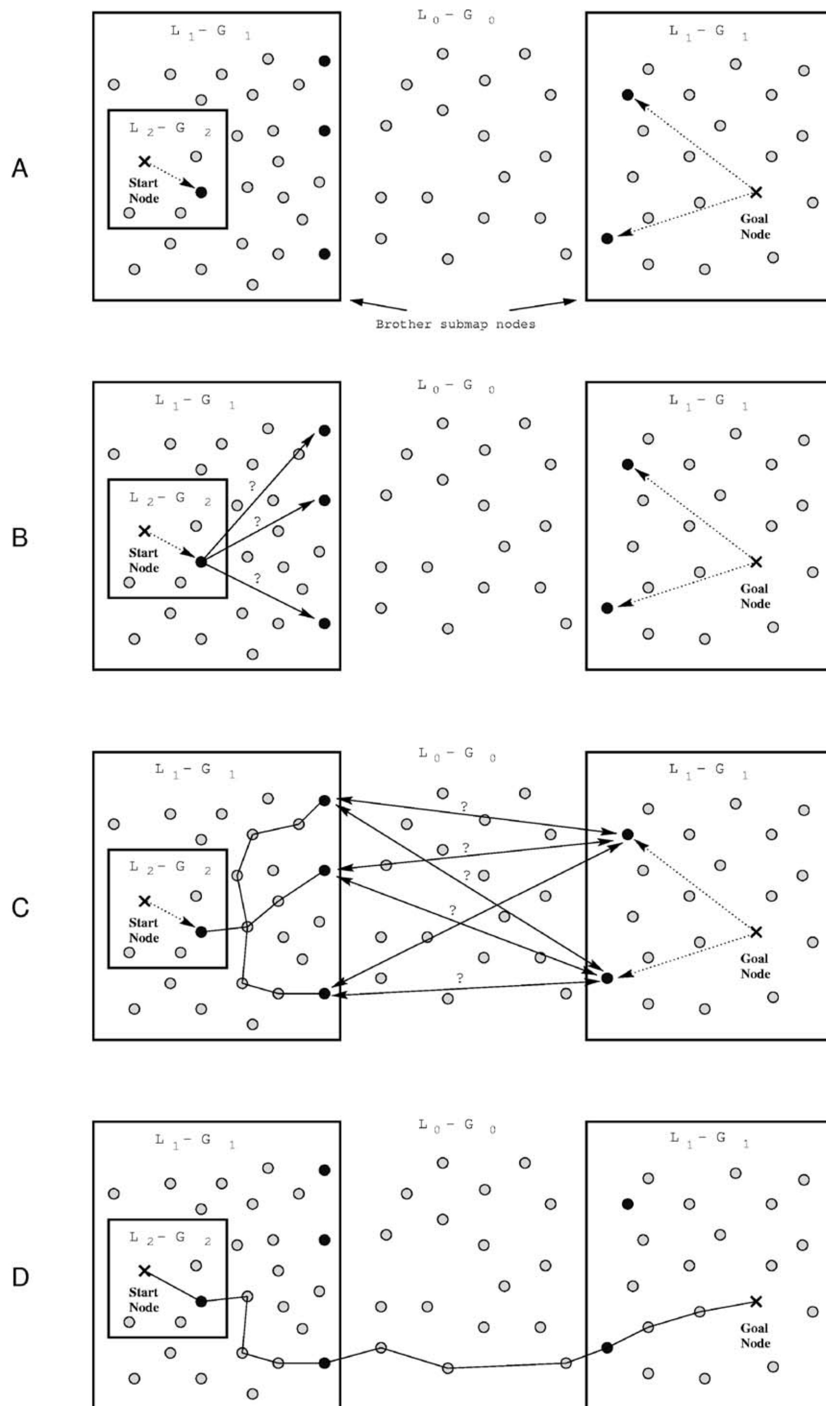

Figure 2. Example of path generation in four steps. For clarity arcs are only shown in step D in the final path.

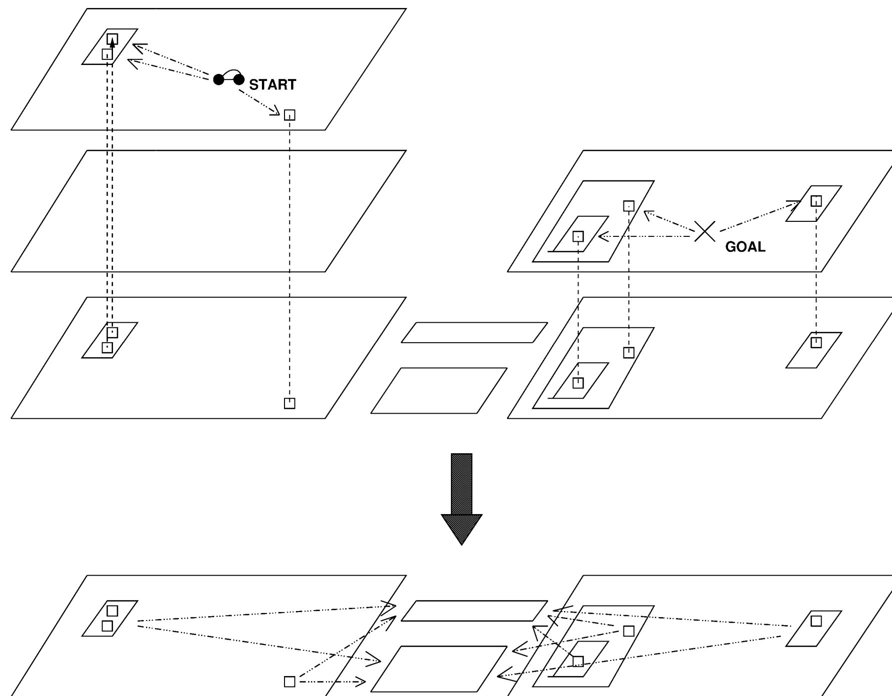

Figure 3. Example of interbuilding path planning: an extension of the general hierarchical path search algorithm. A vehicle on the third floor of a building A plans to reach a goal on a second floor in another building B that is in front of building A. Elevator entrances are treated as subgoals.

example in Section 2.3). However, the path search algorithm extension requires extra analysis.

The solution proposed consists on considering *vertical bridge nodes* described in Section 2.2 as starting or goal nodes. In this way, the original path planning problem is divided in two: first find a path from a start node to a vertical bridge node (elevator entrance) and second find a path from a vertical bridge node (contained in another "floor submap") to the goal node. Notice that an elevator is modelled in an H-graph as a sequence of vertical bridge nodes and each of them represent an elevator entrance.

Elevator entrances in a building floor are usually near to each other. 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". It is even possible to have several submaps containing vertical bridge nodes in the same "floor submap". A more general path search problem consists of finding a path through several submaps containing bridge vertical nodes in different floors and different buildings. See example in Figure 3.

The hierarchical search algorithm that supports "vertical path planning" requires some small modifications. At least one subprocedure must be adapted or completed: *ESTIMATE_PATHS*. A start node N_s is substituted for a start node set NS_s composed of vertical bridge nodes.

```
ESTIMATE_PATHS_2 (Node_Set NS_s, Node_Set NS).
 1: {Begin variable declaration:}
 2: Path_Set PS = Ø;
 3: Path P_i;
 4: Node N_i, N_j, N_submap;
 5: {End variable declaration:}
 6: {NS_s contains the vertical bridge node set.}
 7: {Their submap node is pointed by N_submap.}
 8: N_submap = (N_j ∈ NS_s).map;
 9: for all N_i ∈ NS do
10:   for all N_j ∈ NS_s do
11:     if (N_submap.pre_paths(N_j, N_i) ≠ NULL) then
12:       {There are pre-calculated paths of class 2 between vertical and
13:       horizontal bridge nodes included in submap N_submap.}
14:       PS = PS ∪ (N_submap.pre_paths(N_j, N_i));
15:     else
16:       P_i = (N_j, N_i);
17:       P_i.cost = HEURISTIC(N_j, N_i);
18:       PS = PS ∪ P_i;
19:     end if
20:   end for
21: end for
22: return PS.
```

3.4. ALGORITHM ANALYSIS

There are two ways of classifying search algorithms complexity: time complexity and space complexity. Space complexity refers to the amount of memory needed by algorithms. Nowadays the cost of computer memories is reasonable and this is not a serious problem. However, exponential time complexity search problems cannot yet be solved by faster microprocessors. Exponential time complexity problems must be transformed or approximated. As was mentioned in Section 2.1 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 an A^* algorithm. Here no hierarchical path search is performed (line 9 in *MAIN_PROCEDURE*, Section 3.1 when searching in the deepest hierarchical level). That means a $O(2^N)$ time complexity order, where N is number of nodes.

In [5] a expression for the computational cost in hierarchies with more than two hierarchical levels is given and in [9] a similar conclusion is found. The expression that defines the computational cost is

$$\overline{U_H}(k-1) = U_{k-1} + \frac{1}{2}\frac{(N^{(k+2)/k})/2^{k-1} - N^{3/k}}{N^{1/k} - 2}, \quad (1)$$

where $\overline{U_H}(k-1)$ 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 *material*. In fact, the same hierarchical algorithm can be used without materialization of costs or a more reduced pre-calculated path set.

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 the quality of solutions decreases and vice versa. In a TetraNauta system, cluster sizes are relative low (few nodes in a "room submap", for example) and the number of hierarchical levels (k) can be considerable. Therefore, partial materialization of costs used has a doubly positive effect: speed up search (which is critical due to the real-time TetraNauta restrictions) and help to minimize loss of path accuracy (which is increased due to a large amount of small clusters/submaps).

Parameter $\overline{U_H}(k-1)$ 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 strongly associated to maps (H-graphs). Furthermore, expression (1) is now also determined by a partial materialization of costs. This does not alter time complexity order but improves time efficiency. Thus, the difference with respect to other hierarchical search approaches will depend strongly on the map structure and the selected materialization of costs (pre-calculated paths). Next section analyses some practical cases (maps) in order to test the model proposed.

4. Experimental Evaluation

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 one 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 one floor of a building and finish on a floor of another building.

In the experiments the hierarchical path search algorithm proposed is called TetraNauta. It is compared to other search algorithms widely used in mobile robotics. These algorithms are:

– A^*: it uses the same heuristic as the rest of algorithms: Euclidean distance.
– *A^* with prunes*: a version of the A^* algorithm designed to seep-up search processes. It does not have not the optimality property of A^* (find best path) but can reduce computational time, which is critical and more important in a TetraNauta path planner. The A^* with prunes represents here an approximation of the calculation time needed of another widely used algorithm in path planning: the RTA^* algorithm.
– *Hill climbing*: it has the same backtracking subprocess as A^* with prunes. This subprocess guarantees the completeness property. That is, if there is at least one solution, it will be always found. Under certain circumstances, hill climbing algorithms can be the fastest search algorithms.
– *Genetic Algorithms 1 and 2*: in genetic algorithm 1 an initial population is generated randomly. Genetic algorithm 2 uses *branch & bound* algorithms to generate an initial population. This strategy is based on the method described in [8].

Not only length is a valid measure of optimality in a path or trajectory. Left and right turns are also important issues that imply a time and energy use. A turn implies stopping the wheelchair, changing its direction (rotate) and restarting. Turns costs in a path are modelled as an extra fixed length, which implies extra energy consumption. That is why path costs (values of cost function f) are usually referred to as lengths. The idea is trying to find quasi time optimal paths rather than quasi length optimal paths. Thus, the cost function in a path (or path *skeleton*) can be composed of arc weights (lengths), turn costs and Euclidean distances (estimated lengths).

The TetraNauta algorithm is also tested without pre-calculated paths (no materialization of costs). Computational time represents here an approximation of a typical refinement search algorithm. Unfortunately, the quality (cost) of paths cannot be expected to be equal. As was pointed out, optimality is always desirable but it is not critical. Therefore, the same algorithm without materialization of costs can be a good reference in comparison with the original one.

Speed-up is measured using only CPU time. A better measurement counts also the 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 cannot be applied to

Table I. Characteristics of maps and elements used in experiments

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
HPP pair of nodes	10000	464	100	100
VPP pair of nodes	10000	1196	100	100
IPP pair of nodes	0	1961	0	100

genetic algorithms and to the hierarchical algorithm proposed when it uses pre-calculated trajectories (materialization of costs). This last point implies a possible loss of precision but does not seriously alter general results.

Algorithms are tested in four maps. In each map three sets of node pairs (start and goal) are selected randomly. These three node sets define three path types: horizontal path planning (HPP), vertical path planning (VPP) and interbuilding path planning (IPP). Algorithms have to calculate a path for every pair of nodes in each map. 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 I for a detailed description of the maps characteristics.

4.2. RESULTS DESCRIPTION

Figures (graphics) 4–7 show simulated experimental results achieved using the hospital, industrial buildings, headquarters building and airport maps respectively. On the one hand, Y-axes show total computational time and total path length. On the other, X-axes are divided into three parts when interbuilding path planning is considered (more than one building) or two parts when no interbuilding path planning is considered (only one building). The X-axes divisions correspond to horizontal path planning (HPP), vertical path planning (VPP) and interbuilding path planning (IPP). Each part (division) on the X-axis is also divided in two subparts that indicate total path lengths (L) and total computational time (T) of paths obtained by algorithms. In this way, graphics help to compare the balance between path length optimality (L) and computational speed (T) easily.

Graphic 8 is the same graphic as 5 (industrial buildings) but including genetic algorithms. Computational time costs (T) for genetic algorithms are high when

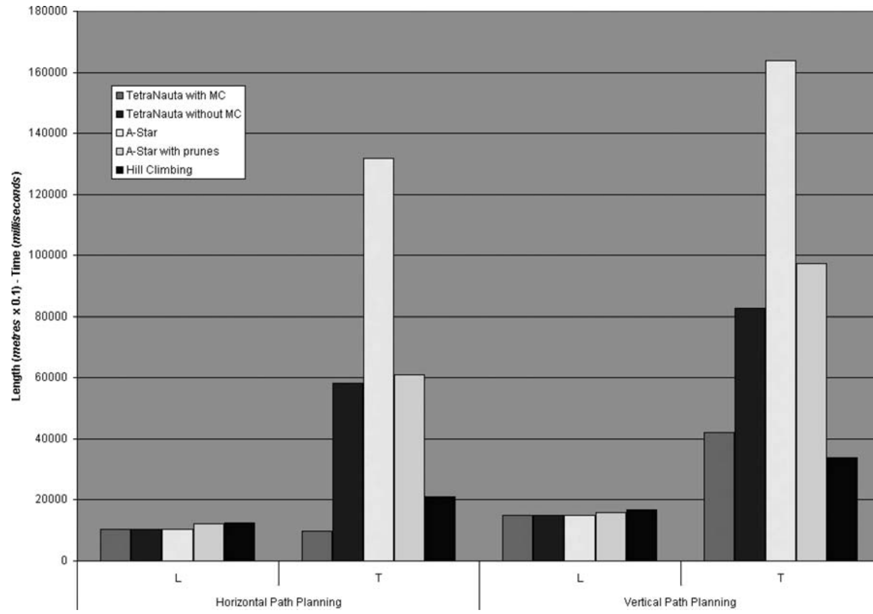

Figure 4. Hospital results.

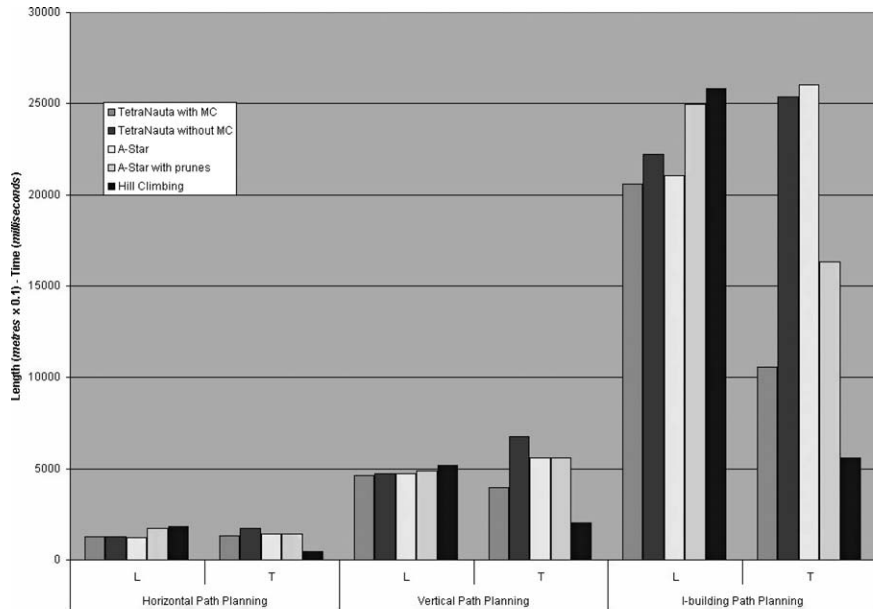

Figure 5. Industrial buildings results.

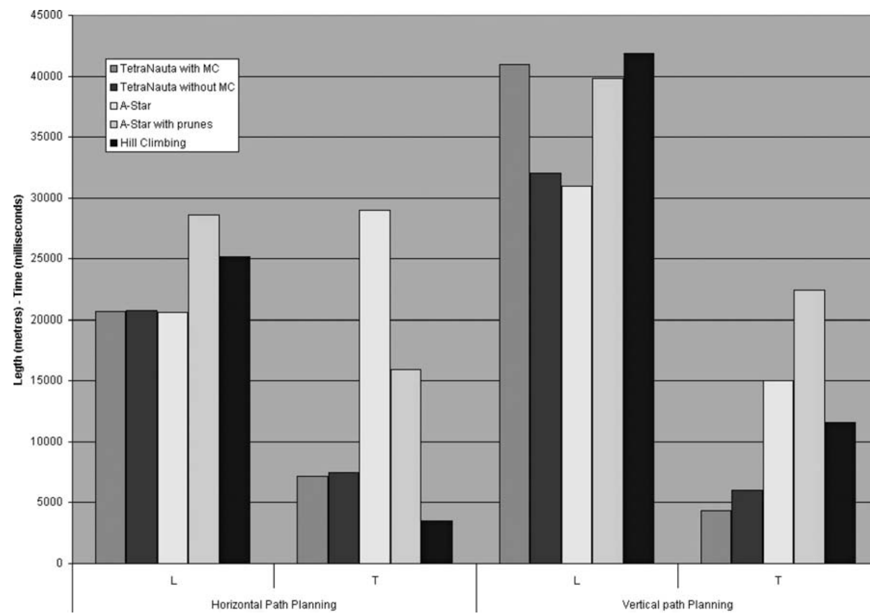

Figure 6. Headquarters building results.

compared with the rest of the algorithms. Similar results are obtained in other maps. For better clarity and legibility, no more genetic algorithm results are shown.

4.3. RESULTS ANALYSIS

As was already mentioned above, the first relevant conclusion is the low efficiency obtained when using genetic algorithms. Length/quality of solutions (L) are even better than other algorithms, but computational time (T) is always too high. This is a direct consequence of the coding scheme used in genetic algorithms when working with topological abstract models (graphs). Codifying strings (paths) of variable length does not work and needs customizing operators [12]. More concretely, the initial population generation process is the key problem here. This can be noticed when comparing computational time (T) in graphic 5. Genetic algorithms design with a variable-length coding scheme is usually ad hoc and complicated.

The best horizontal path planning results appear in graphic 4 (hospital map): total path length (L) is close to A^* (optimality) and computational time (T) is nearly a tenth of A^* when using materialization of costs. On the contrary, horizontal path planning performance in industrial buildings (graphic 5) is closer to other algorithms. This is a direct consequence of the overhead caused by the hierarchical model. Overhead effects caused by hierarchical search can be also observed in graphic 7 (Lambert Airport map) when comparing the TetraNauta algorithm in horizontal and vertical path planning. These maps have a small *node density*, that is to say, cluster sizes (nodes per submap) are lower. Moreover, a small number

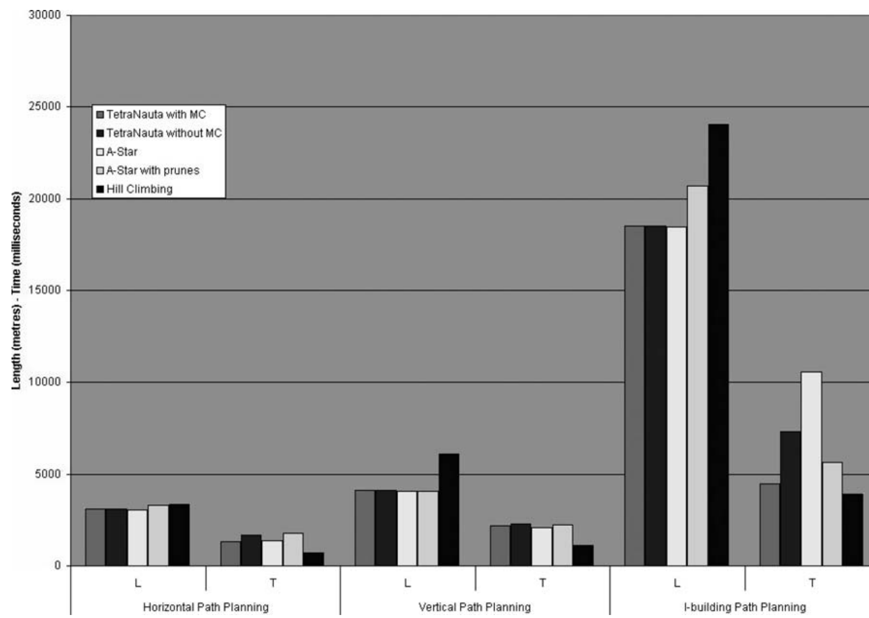

Figure 7. Airport results.

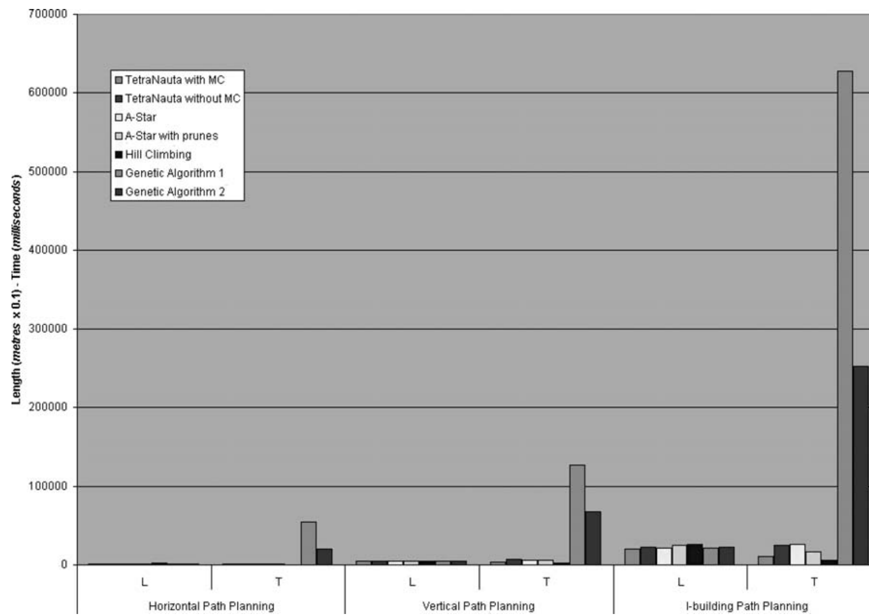

Figure 8. Industrial buildings results. Genetic algorithms 1 and 2 are included (first two columns on the right in HPP, VPP and IPP). Total path length (L) is even better (lower) than other algorithms but total computational cost (T) is worse (bigger). Computational costs differences between Genetic algorithm 1 and 2 indicates the importance (and the problem) of generating an initial population.

of connections (arcs) per node prevents from an exponential search tree expansion and it allows branch and bound algorithms to converge quickly. Therefore, there are situations where for example A^* algorithms are a more suitable choice.

The TetraNauta hierarchical search algorithm provides in general better performance in inter-building and vertical path planning. Here the number of nodes involved is higher and path searching can be considered as a real large-scale problem. However, there can again be some exceptions like in graphic 6 (headquarters building). Total paths length (L) using TetraNauta algorithms and vertical path planning is bigger (worse) than using A^*. Paths achieved with the Hill Climbing algorithm have also an important length difference when compared with A^*. This is a direct consequence of the map structure that forces heuristic search algorithms to fall in local minimums. Heuristic algorithms can be affected by the abstract structures (graphs) used and the chosen heuristic. A better heuristic may prevent this from some local minimums but it implies an extra computational cost. As soon as path lengths increase and therefore graphs get bigger and more complex, hierarchical search with materialization of costs performance increases too. Even so, a Hill Climbing algorithm is sometimes faster than hierarchical algorithms and it may be a good alternative if quality of paths (solutions) is acceptable.

In general, graphics show that as soon as problems get more complex, hierarchical search with materialization of cost and several hierarchical levels, becomes really effective. Computational costs (T) are reduced significantly and quality/length of paths (L) are in some cases close to optimal.

4.4. COMPARISON WITH OTHER RESULTS

In [4] an H-graph with multiples hierarchical levels and a refinement hierarchical search algorithm is analysed. The hierarchical search algorithm uses a top-down strategy. It is pointed out that a 321% computational cost reduction is obtained when compared with Dijkstra's algorithm. Instead of CPU time, computational cost is approximated by number of explored nodes. Using the TetraNauta algorithm without materialization of costs, up to 228% CPU time reduction is obtained when compared with A^* algorithm and horizontal path planning. However, in optimal conditions when using materialization of costs up to 1366% computational time reductions (in hospital map) are achieved. Quality of solutions (path lengths/costs) is important too. In [4] no experimental results are given. In our experiments and horizontal path planning, paths obtained are close to optimal. Differences between path length sums in A^* and the TetraNauta algorithm are always less than 1%. However, exceptions described in the last section must be taken into account. Thus, as was pointed out in Section 3.4, maps structure, cluster sizes or number of hierarchical levels play an important role.

Materialization of costs are for the first time introduced in [7]. Again path retrieval is significantly faster than A^*. It uses only two hierarchical levels, so cluster sizes (nodes per submap) are relative high. On the contrary, the proposed

hierarchical algorithm (TetraNauta) is based on an H-graph that contains several hierarchical levels. Thus, it obtains faster paths/solutions. However, quality of paths (length/cost) is slightly worse. The TetraNauta algorithm and abstract world model proposed does not *materialize* each possible path and it focuses mainly on saving CPU time. Nevertheless, it would retrieve optimal paths if submaps in the deepest level of the hierarchy had only one bridge node. In this case initial path *skeletons* (subprocedure *ESTIMATE_PATHS*) would have only one possibility: estimate a path from a initial/goal node to a single bridge node. In fact, in a TetraNauta system many paths begin or finish in a "room submap" where there is only one bridge node (representing a door). This context helps to avoid a complete materialization of costs (like [7] does), reduce memory storage and speed-up calculations.

5. Conclusions

The contribution of this paper is:

1. Adapt H-graphs for a smart wheelchair aided navigation system. The abstract model allows efficient path planning and easy information management for users with special needs.
2. Propose a new hierarchical graph model $G = (N, A, C, W, T)$ that uses materialization of costs.
3. Extend hierarchical search with materialization of costs to H-graphs with several hierarchical levels (more than two).
4. Extend hierarchical path search 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 on-line path planning (i.e., obstacle avoidance). Pre-calculated paths (materialization of costs) can help to find path deviations faster.

References

1. Abascal, J., Cagigas, D., Garay, N., and Gardeazabal, L.: Interfacing users with severe mobility restrictions with a semi-automatically guided wheelchair, *SIGCAPH* (ACM Press) **63** (1999), 16–20.
2. Abascal, J., Cagigas, D., Garay, N., and Gardeazabal, L.: Mobile interface for a smart wheelchair, in: *4th Internat. Symposium on Human Computer Interaction with Mobile Devices, Mobile HCI 2002*, Pisa, Italy, 18–20 September 2002, pp. 373–377.
3. 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.

4. Fernandez, J. A. and Gonzalez, J.: Hierarchical graph search for mobile robot path planning, in: *Internat. Conf. on Robotics and Automation*, Leuven, Belgium, 1998.
5. Fernandez, J. A. and Gonzalez, J.: *Multi-Hierarchical Representation of Large-Scale Space*, Kluwer Academic Publishers, Dordrecht, 2001.
6. 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*, 1994, pp. 263–270.
7. Huang, Y.-W., Jing, N., and Rundensteiner, E. A.: Hierarchical optimization of optimal path finding for transportation applications, *J. GeoInformatica* **1**(2) (1997), 125–159.
8. 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.
9. Korf, R. E.: Planning as search: A quantitative approach, *Artificial Intelligence* **1**(33) (1987), 65–88.
10. Latombe, J.-C.: *Robot Motion Planning*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1990.
11. 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.
12. 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).
13. Vicente Diaz, S., Amaya Rodriguez, C., Diaz del Rio, F., Civit Balcells, A., and Cagigas Muniz, D.: TetraNauta: An intelligent wheelchair for users with very severe mobility restrictions, in: *Proc. of the 2002 IEEE Internat. Conf. on Control Applications*, 2002, pp. 778–783.