Hierarchical Shortest Pathfinding Applied to Route-Planning for Wheelchair Users

Suling Yang and Alan K. Mackworth

Department of Computer Science, University of British Columbia
Vancouver, British Columbia, Canada V6T 1Z4
{sulingy, mack}@cs.ubc.ca

Abstract. Pathfinding on large maps is time-consuming. Classical search algorithms such as Dijkstra's and A* algorithms may solve difficult problems in polynomial time. However, in real-world pathfinding examples where the search space increases dramatically, these algorithms are not appropriate. Hierarchical pathfinding algorithms that provide abstract plans of future routing, such as HPA* and PRA*, have been explored by previous researchers based on classical ones. Although the two hierarchical algorithms show improvement in efficiency, they only obtain near optimal solutions. In this paper, we introduce the Hierarchical Shortest Path algorithm (HSP) and a hybrid of the HSP and A* (HSPA*) algorithms, which find optimal solutions in logarithmic time for numerous examples. Our empirical study shows that HSP and HSPA* are superior to the classical algorithms on realistic examples, and our experimental results illustrate the efficiency of the two algorithms. We also demonstrate their applicability by providing an overview of our Route Planner project that applies the two algorithms proposed in this paper.

1 Introduction

The population of wheelchair users is very significant and increasing dramatically [13]. Therefore, finding accessible wheelchair routes is an important problem. In developed countries, most buildings and public transportation are accessible, making the lives of people in wheelchairs easier. However, the routes to the closest elevator in a new building, temporary road conditions and bus schedules may be unknown to wheelchair users. Therefore, we were motivated to create route-planning software that can be installed on a small device to give wheelchair users route accessibility information while they are travelling. Besides a route planner, our software contains a simple scheduler that synchronizes with the route planner to provide more accurate commuting information for clients. After obtaining the destination from the scheduler, the route planner establishes some possible paths and displays the best one to the client.

The first stage of our project is to implement an algorithm to hierarchically find paths and obtain multiple levels of detail. Since an abstract high-level path

can provide a general plan, the client can have an impression of future routing. Instead of being presented with a cumbersome and lengthy low-level path, people, especially the elderly, would prefer a cognitively visible path. In the next stage, the software accommodates our scheduler and real-world maps, including indoor and outdoor applications.

1.1 Problem Statement

The notation used in this section is described here. A map is represented by a graph $G = (N, E, l)$, where N is the set of nodes, E is the set of edges, and l is the number of hierarchical levels. Hierarchical levels denote inclusion (containing relation). For example, a building is an ancestor of rooms in it. The cardinalities of N and E are denoted by n and e. Each $node \in N$ contains a set of neighbours $neigh(node)$ and is assigned to a level. Each edge $(i, j) \in E$ is associated with another pair of nodes $(exit, entrance)$ where $exit$ and $entrance$ are nodes of one level lower, or they may be $null$. The weight on an edge (i, j) is denoted by $w(i, j)$, and the weight of a path P is $w(P) = \sum_{(i,j)\in P} w(i, j)$. We want to find the *shortest path* P that has the minimum weight $w(P)$ from a node s to another node d in a graph G.

Definition 1: **Search** *on a graph G: The process of finding a path* $P = \{s, i, ..., d\}$*, a sequence of nodes on G, from the start s to the destination d.*

Human beings may approach complex problems by dividing them into easier sub-problems, each of which can be further divided into smaller problems or solved by a quicker search. More than forty years ago, Minsky realized that a successful division of a complex problem will greatly reduce the search time by a *fractional exponent* [1]. Such divisions can be considered as "islands" in the search space, where these islands can be abstracted from groups of nodes. In this section, an abstraction of a group of nodes is found when these nodes are at the same level and within the same enclosure. The abstraction then is their enclosure. For example, a building is an abstraction of the rooms in it. Conversely, details at the room level constitute the refinement of the building.

Definition 2: **Hierarchical Pathfinding***: The process of finding a sequence of paths* $\{P_m, P_{m-1}, ..., P_0\}, m \leq l$*, where* P_i *contains nodes of* i*-th level and that are an abstraction of nodes on* P_{i-1}*.*

The maximum length of an abstract path then is defined as $maxlength(P_i) = w(P_i) + \sum_{node \in P_i} upperbound(node)$, and the minimum length of the path is $minlength(P_i) = w(P_i) + \sum_{node \in P_i} lowerbound(node)$, where $upperbound(node)$ is the maximum of shortest distances between any pair of places within the node and $lowerbound(node)$ is the minimum distance from the entrance to the exit. The upper bound of a node can be overestimated, whereas the lower bound can be underestimated. Hence, $upperbound(node)$ can be the total of low-level edge lengths in an abstracted node, $node$, and $lowerbound(node)$ can be zero.

1.2 Previous Work

One of the well-known search algorithms is Dijkstra's, a non-heuristic version of A*. However, it is not as efficient as some A* algorithms with good heuristics. A* [5] and IDA* [6] are both heuristic and complete algorithms on locally finite graphs (graphs with a finite branching factor), but neither of these two algorithms is well-suited into a dynamic environment.

Hierarchical pathfinding has been explored since the mid-nineties, and several excellent algorithms have been developed. In [4], Rabin provides a high-level description on path-finding using a two-level hierarchy. This algorithm uses only two levels of abstraction, but real world maps are divided into numerous levels. In [3], Holte *et al.* explain how abstraction could lead to speedup on finding a solution for a graph-oriented problem. Similar to their work, Hierarchical Pathfinding A* (HPA*) [9] and Partial-Refinement A* (PRA*) [8] both construct multi-level abstractions using grid-based representation, and greatly reduce the amount of time required to find a near-optimal solution. Instead of estimating a near-optimal solution, our algorithm finds all possible candidates and keeps them until some are proven to be non-optimal.

If the number of sub-level nodes of each node is fixed, the size of the map increases exponentially as the number of levels increases. The running times of existing search algorithms grow exponentially as well, so they may not be practical in reality. In this section, we propose the Hierarchical Shortest Path (HSP) algorithm, as well as a hybrid of the HSP and A* (HSPA*) algorithms. HSP finds a threshold that is the least upper-bound of the length of a high-level path, and refines the paths that may be shorter than this threshold. It stops refining a path when the path length exceeds the threshold, and finally returns the shortest path among those that remain. HSPA* works in a similar way, except that it uses A* for refinement as soon as it reaches a specific level. The experimental results show that HSP and HSPA* are suitable for various applications.

2 Hierarchical Shortest Path

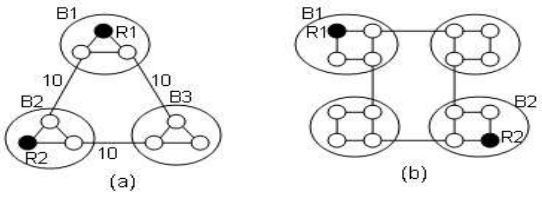

Fig. 1. Examples: (a) and (b) are maps of campuses comprising three and four buildings, respectively, where each building contains a number of rooms

The hierarchical approach taken in this section involves multi-level representation of a real-life map. For example, these levels can be campuses, buildings, floors and rooms. Assume that a person wants to travel from room R_1 in building B_1 to room R_2 in building B_2 in the world of Fig. 1 (a). If the person begins his pathfinding from B_1 to B_2, then he could easily find the shortest path from B_1 to B_2 with one edge. If the walking distances within these buildings are negligible, then the person can take the shortest path between B_1 and B_2, and needs only to refine the path from R_1 to the exit of B_1 and the path from the entrance of B_2 to R_2. Hence, the person does not need to search paths within B_3.

Therefore, in order to save time in planning, we can search for short paths between high-level sites first, and then refine the paths within the selected sites for more details. However, the high-level shortest path does not guarantee the shortest overall path, because the paths within each site can vary in length. Consider the previous example, where the walking distance within buildings B_1 and B_2 is significant. Then, a path from R_1 to the exit of B_1, and a path from the entrance of B_2 to R_2, may increase the length of the overall path by a large amount. As a result, we can prune away a path P only when it is guaranteed to be non-optimal, i.e., the minimum length of the path exceeds the maximum length of another path:

$$\exists P' : minlength(P) > maxlength(P'). \tag{1}$$

2.1 The Algorithm

The main algorithm, 1.1, consists of four major steps: first, find the topmost level abstract nodes wherein the start node and the destination node are different, and abstract the graph G to that level (line:1,2); second, find a high-level shortest path between ancestors of s and d at the $\hat{l}$-th level, based on which a threshold is obtained (line:3,4); third, find all possible high-level paths with minimum lengths less than the threshold (line:5); finally, refine each high-level path to the lowest level and return the one with minimum length (line:6,7). This algorithm is complete since it maintains all possible paths and ignores a path only when it is definitely longer than another possible path. The following will explain the algorithm in further detail.

Abstract of a Graph. As mentioned above, the desired level in the first step is needed for abstracting the graph. For example, if a person travels from a room in a city to another room in another city within a country, the two cities are the topmost places he should consider. In other words, we seek out the first same ancestor of the two places to find the one level below that ancestor that is desired. Algorithm 1.2 presents the procedure for finding the first same ancestor of two places and returning the level of that ancestor. Note that pathfinding between different level architectures is also allowed. After the desired level $\hat{l}$ is found, the graph G is abstracted to that level:

$$\hat{G} := (\hat{N}, \hat{E}), \tag{2}$$

Algorithm 1.1 : main (G, s, d)

Input: A graph ***G = (N, E, l)***, the start node s and the destination node d
Output: A shortest path $\boldsymbol{P}$ from s to d

1: $\hat{l} := sameAncestorLevel(s, d) - 1$
2: $\hat{G} := abstract(G, \hat{l})$
3: $\hat{P} := A^*(\hat{G}, ancestor(s, \hat{l}), ancestor(d, \hat{l}))$
4: $threshold := maxlength(\hat{P})$
5: $S := possiblePaths(\hat{G}, ancestor(s, \hat{l}), ancestor(d, \hat{l}), threshold)$
6: $\hat{S} := refine(S, l, threshold)$
7: **Return** $P := min\{\hat{S}\}$

Algorithm 1.2 : sameAncestorLevel (s, d)

Input: The start node s and the destination node d
Output: The level l' of the first same ancestor of s and d

1: $lower = lowerLevelNode(s, d)$
2: $higher = higherLevelNode(s, d)$
3: **while** *lower* is not at the same level as *higher* **do**
4: $lower = parent(lower)$
5: **end while**
6: **while** *lower* is not a sibling of *higher* **do**
7: $lower = parent(lower)$
8: $higher = parent(higher)$
9: **end while**
10: **Return** $l' := level(lower)$

where $\hat{N}$ and $\hat{E}$ are subsets of N and E, respectively, and the nodes $node \in \hat{N}$ and $i, j : (i, j) \in \hat{E}$ are only those at level $\hat{l}$.

A High-level Shortest Path and a Threshold. Either Dijkstra's or the A* algorithm is used on $\hat{G}$ to find the shortest path $\hat{P}$ between the ancestors of s and d at the $\hat{l}$-th level. Then, $maxlength(\hat{P})$ is the threshold that we are looking for. The time required to find the high-level path and the threshold is insignificant if the number of high-level nodes is relatively much smaller than that of low-level nodes.

High-level Possible Paths. After a threshold is obtained, we could search on $\hat{G}$ for all possible high-level paths and prune away those with minimum lengths exceeding the threshold. This process is shown in Algorithm 1.3.

Hierarchical Refinement on High-level Short Paths. After high-level paths are found, a refinement step is executed, as shown in Algorithm 1.4. It refines each possible high-level path to one level lower each time by recursively finding the shortest path from the entrance to the exit of each node on the high-level path. Then, the lowest level path is constructed by concatenating all partially refined paths. The refinement step ceases if the length of the current path exceeds the threshold.

Algorithm 1.3 : possiblePaths ($\hat{G}$, $ancestor(s, \hat{l})$, $ancestor(d, \hat{l})$)

Input: The abstract graph $\hat{G}$, the ancestors of s and d at level $\hat{l}$, and the threshold t
Output: A set of paths at level $\hat{l}$ with minimum lengths less than t

```
1: CurrentPaths ← { {ancestor(s, l̂)} }
2: Output ← { }
3: while CurrentPaths changes do
4:   for each P ∈ CurrentPaths do
5:     if last node ln of P is ancestor(d, l̂) then
6:       Output ← Output ∪ {P}
7:       PossiblePaths ← PossiblePaths ∪ {P}
8:     else
9:       for each nb ∈ neigh(ln) do
10:        P̂ = P ∪ {nb}
11:        if minlength(P̂) is smaller than t then
12:          PossiblePaths ← PossiblePaths ∪ {P̂}
13:        end if
14:      end for
15:    end if
16:  end for
17:  CurrentPaths ← PossiblePaths
18: end while
19: Return Output
```

2.2 A Hybrid of HSP and A* (HSPA*)

Note that there may be more than one high-level path of shorter length than the threshold, but that is unusual in many realistic cases. The refinement step is fast, if the number of possible short paths is small. Otherwise, the time spent on refining these paths may be excessive. Hence, we could refine our algorithm to select between HSP and A* depending on the number (α) of these paths. Furthermore, from the experimental results shown in the next section, it is evident that HSP is not as fast as A* if the number of levels is small and the number of sub-nodes is not significantly large. Therefore, we could use A* algorithm when a high-level path is refined to a specific low-level (β).

2.3 Analysis of Running Time

Let b be the number of sub-nodes within an abstract node. If b is fixed, then the total number of lowest-level nodes is $n = b^{\,l}$, where l is the number of levels. A* and IDA* with good heuristics can solve difficult problems in polynomial time [2, 7], i.e., $\mathcal{O}(n^c) = \mathcal{O}(b^{\,cl})$ where c is a constant. Therefore, the algorithms are exponential to the order of l. On the other hand, HSP only executes searches on $\mathcal{O}(b)$ nodes on each level if there is only one abstract path found. In this case, HSP has a running time of $\mathcal{O}(b^c\, l)$. Even if there is more than one abstract path, say d paths, the running time of HSP is $\mathcal{O}(b^c\, d^{\,l})$. As long as $d \ll b$, HSP is still

Algorithm 1.4 : refinement (S, l, t)

Input: A set S of high-level paths, the lowest level l and the threshold t
Output: A set $\hat{S}$ of low-level paths at level l

```
1: Ŝ = { }
2: for each hierarchical high-level path P ∈ S do
3:    P̂ = {s}
4:    for each node ∈ P do
5:       find the edge from last node of P̂ to the entrance of node
6:       if level(entrance) ≥ l then
7:          P' = HSP(entrance, exit) where entrance and exit are within node
8:          P̂ = P̂ ∪ P'
9:       end if
10:      if minlength(P̂) > t then
11:         ignore P̂ and continue the outer loop
12:      end if
13:   end for
14:   Ŝ = Ŝ ∪ {P̂}
15: end for
16: Return Ŝ
```

much more efficient than A* or IDA*. In reality, where distances within a site are usually considered insignificant, one high-level abstract path is expected to appear. Hence, the running time is $\mathcal{O}(b^c\, l)$ for the HSP algorithm in real-world examples versus $\mathcal{O}(b^{\,cl})$ for A* and IDA*, i.e., linear in l versus exponential in l. Moreover, l is usually small while b is large. Thus, $\mathcal{O}(b^c\, l)$ or $\mathcal{O}(b^c\, d^{\,l})$ should be relative smaller than $\mathcal{O}(b^{\,cl})$.

2.4 Experiments and Results

In real life, each site is connected to its neighbours from exits to entrances. For example, if two buildings are adjacent, then we can exit from a door of one building, and enter through a door of the other building. Hierarchical maps in our experiments are constructed with this connection in mind. Weights of the edges and the upper bound of the shortest distances between any pair of places within an site are included in the map, which is created in XML form. One of the reasons that we use XML form is due to its consistency and flexibility; i.e., you have to define each element with an opening and an ending tag, while you can use an arbitrary tag name. Another reason is that nested tags in XML can represent hierarchical relations easily.

All experiments described in this section were run on a Linux Intel(R) Pentium 4 machine running at 3.20GHz with 2 GB of RAM, using Eclipse version 3.1.

Examples First, we analyze the algorithms using the example shown in Fig. 1 (a). Each building contains three floors, while each floor contains three rooms,

and so on. In reality, the number of nodes at one level lower is usually much more than three, where the time saved from pruning away unnecessary high-level paths is more obvious. Here we use this example because of its simplicity in building a XML file and its simulation of a simple world. The total number of lowest level nodes is $n_0 = 3^l$, and since it grows exponentially to the order of l, Dijkstra's and A* algorithms are inefficient here. The A* heuristic used in our experiments is the Euclidean distance, which is popular to use in real-world route-finding problems. However, this heuristic may not be useful for a multilevel structure, since the heuristic values for low-level nodes may be similar. For example, the Euclidean distances from nearby rooms in one city to a room in another city may be indistinguishable. In contrast, HSP, which prunes away most insignificant high-level paths, runs well in these examples.

As shown in Fig. 2 (a) and (b), both multi-level representation running times of Dijkstra's and A* algorithms blow up quickly, while the HSP running time grows slowly. A semi-log graph of the three algorithms' running times is drawn in Fig. 1 (c). We can observe that the slope of the HSP curve is significantly less steep than that of the other two curves. A further observation from Fig. 2 (c) is that HSP is not superior among the three algorithms all the time. When the number of levels is small, HSP wastes time on recursive calls. Therefore, it is better to use a hybrid of HSP and A*, the HSPA* (as described in Sect. 2.2). The running-time results for HSPA* are compared with those of the other three algorithms in Fig. 2 (d). Note that HSP and HSPA* have similar performances in these examples, although HSPA* is more stable.

Table 1 shows the running times of the four algorithms in the example where every abstract node has four sub-nodes, as shown in Fig. 2 (b). In this example, more than one abstract high-level path is found at each level, but the running times of HSP and HSPA* are still promisingly good. Examples with larger number of sub-nodes (b) are also explored, and the outcomes are shown in Table 1 as well.

Table 1. Running times: The running times (in milliseconds) of four algorithms using examples where every abstract node has four or more sub-nodes (b)

(b) :	(4)				(5)			
Level:	*1*	*2*	*3*	*4*	*1*	*2*	*3*	*4*
Dijkstra	2	6	35	295	2	11	94	1278
A*	1	4	22	149	1	10	56	717
HSP	1	8	23	130	2	12	42	97
HSPA*	1	8	24	174	1	11	35	88
(b) :	(6)				(7)			
Level:	*1*	*2*	*3*	*4*	*1*	*2*	*3*	*4*
Dijkstra	3	24	350	7817	3	25	466	15601
A*	2	23	218	3462	2	21	231	8368
HSP	2	28	131	354	2	31	137	368
HSPA*	2	24	99	296	2	28	116	316

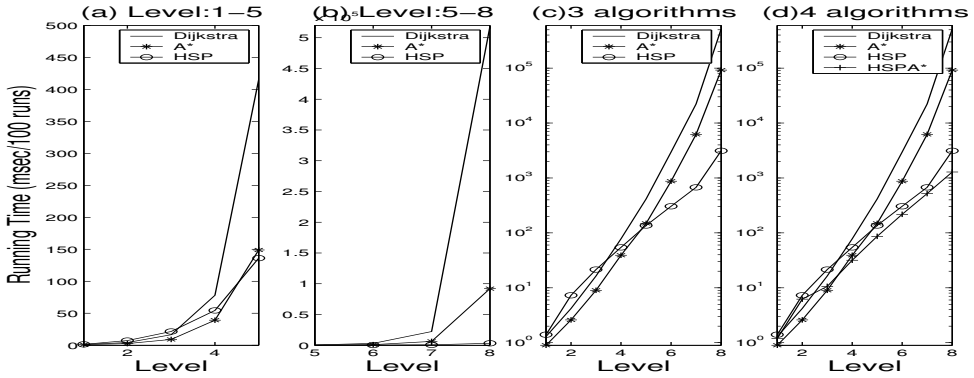

Fig. 2. Running time plots: (a), (b): the running times of three algorithms; (c): the running times of three algorithms in log-scale; (d): the running times of four algorithms in log-scale

2.5 Application

Several existing projects, such as the Assistant Cognition and the Aphasia projects, have been helping physically challenged people to better perform daily activities [10, 12]. The usefulness of these projects motivates us to build a cognitive assisted system for people with disabilities.

Route Planner for People with Disabilities. Our route planner, which shows accessible routes for wheelchair users contains the data of several buildings at the University of British Columbia (UBC): ICICS, the X building and Dempster, the three main buildings in our department. This system is based on a scheduler system, which can be Microsoft Outlook, ESI Planner II [11], or others. We used Microsoft Outlook for our system because of its ease of use and compatibility with other systems. Based on an accurate schedule, the destination can be easily estimated. Then, the route planner computes the possible paths to the destination using HSPA* and shows both high-level path and low-level paths to the client. The system is installed on a small device, such as a tablet PC, so that users can carry the device and find a path to destination while they are travelling. For example, if a client is heading to class, then Fig. 3 shows a high-level path from ICICS to Dempster, where most classes are held. After the user acquires an overview of the whole path, he/she clicks the "Detail" button. Then, the low-level path corresponding to the high-level path is shown, as in Fig. 4. (See Appendix A for more details.)

3 Conclusion and Future Work

In this paper, the Hierarchical Shortest Pathfinding (HSP) algorithm is presented and its running time analyzed. The speedup from eliminating unnecessary high-

level paths is remarkable, and good performances of HSP is expected in real-world pathfinding problems. There are a few directions that this research could be extended beyond the work shown in this paper. First, more examples are anticipated to analyze the performances of the four algorithms. We analyzed the running times of the HSP and HSPA* algorithms based on artificial examples. We shall apply the algorithms on more actual examples, such as the whole UBC campus. Second, as described in Sect. 2.2, the HSPA* algorithm contains two parameters, namely α and β. The best-fit values of these two parameters may vary depending on different problems. Therefore, it may be worthwhile learning well-suited values for these two parameters using stochastic local search methods. Third, comparison of HSP, HSPA* and other hierarchical pathfinding algorithms could be further investigated in terms of their running times and performance.

References

1. Minsky, M.: Steps toward artificial intelligence. (1961)
2. Russell, S., Norvig, P.: Solving problems by searching. Artificial Intelligence: A Modern Approach. (1995)
3. Holte, R., Mkadmi, T., Zimmer, R. M., MacDonald, A. J.: Speeding up problem solving by abstraction: A graph oriented approach. *Artifical Intelligence.* (1996) 85(1-2):321-361
4. Rabin, S.: A* aesthetic optimizations. Game Programming Gems. (2000) 264-271
5. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. on Systems Science and Cybern. (1968) 4:100-107
6. Korf, R.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence. (1985) 27(1):97-109
7. Korf, R., Reid, M., Edelkamp, S.: Time complexity of iterative deepening-A*. Artificial Intelligence. (2001) 199-218
8. Sturtevant, N., Buro, M.: Partial pathfinding using map abstraction and refinement. AAAI-05. (2005) 1392-1397
9. Botea, A., Müller, M., Schaeffer, J.: Near optimal hierarchical path-finding. J. of Game Develop. (2004) 7-28
10. Kautz, H., Fox, D., Etzioni, O., Borriello, G., Arnstein, L.: An overview of the assisted cognition project. American Association for Artificial Intelligence. (2002)
11. Moffatt, K., McGrenere, J., Purves, B., Klawe, M.: The partricipatory design of a sound and image enhanced daily planner for people with aphasia. Proceedings of ACM CHI. (2005) 501-510
12. McGrenere, J., Davies, R., Findlater, L., Graf, P., Klawe, M., Moffatt, K., Purves, B., Yang, S.: Insights from the aphasia project. Proceedings of ACM Conference on Universal Usability. (2003) 112-118
13. Wheelchair Foundation: http://wheelchairfoundation.ca/ (2002)

A Screen Shots

Users can select one of the events from the event list on the left hand side of the main frame to view the event information. If a path is found for that event, then the high-level path is shown, as in Fig. 4. Note that the buttons that represent

the corresponding (sub)destinations are located where the (sub)destinations are. The "YOU" button represents the current position of the client. Clicking on one of the buttons that represents (sub)destinations will display useful information regarding the desired (sub)destination on the left-hand pane. The whole path with specific steps (nodes on the path) is shown on the left-hand as well. Since nodes are not able to show on the same picture, buttons for each picture should be added so that users can view any portion or the whole path. These buttons are placed on the left bottom corner, as shown in Fig. 3 to 4. To view the detailed path, the user can click the "Detail" button, and then Fig. 4 will show.

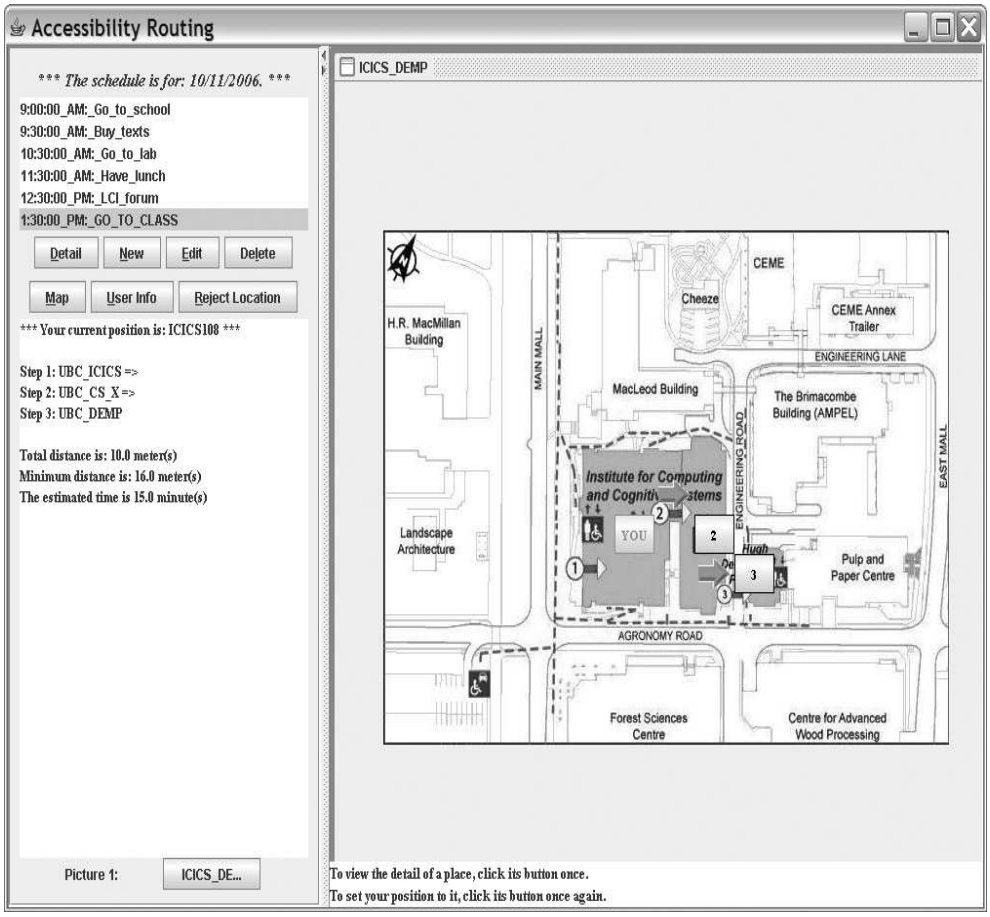

Fig. 3. A screen shot of the Route Planner when a high level path is shown

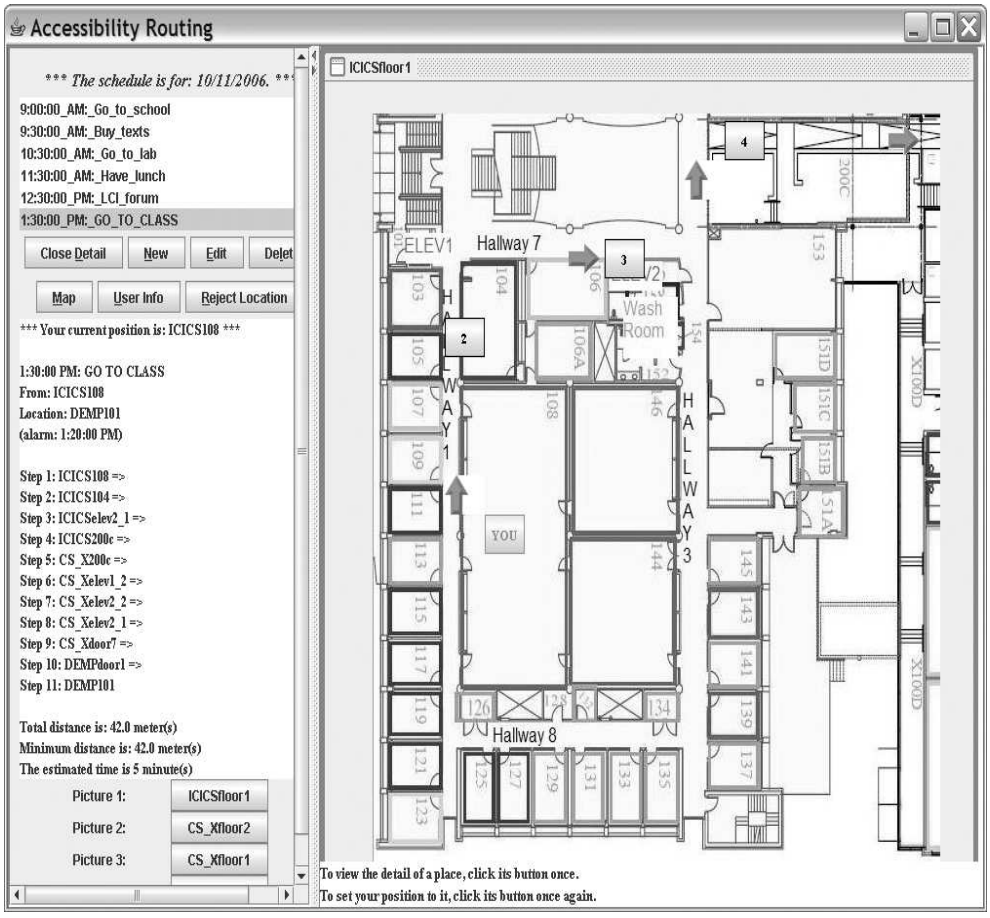

Fig. 4. A screen shot of the Route Planner when a low level path is shown