NafisNav: an Indoor Navigation Algorithm for Embedded Systems and based on Grid Maps

Nafiseh Osati Eraghi, Fernando López-Colino, Angel de Castro, Javier Garrido
HCTLab, Dpto de Tecnología Electrónica y de las Comunicaciones
Universidad Autónoma de Madrid, Spain
email: fj.lopez@uam.es

***Abstract*— An important goal in navigation of low cost robots is low memory usage. In this paper, we propose a novel navigation algorithm (NafisNav) suitable for embedded systems with low resources, mainly memory. The proposed path finding algorithm is designed and implemented in grid maps. Unlike existing algorithms, that mainly focus on obtaining the shortest possible path for navigation, the proposed algorithm focuses on reducing memory consumption, even at the cost of not always obtaining the best path. Experimental results show the trade-off between path length and memory consumption that is obtained, comparing it with typical algorithms such as Dijkstra or A*.**

Keywords*— navigation, bresenham algorithm, grid based map, embedded system, backtracking technique, dynamic memory

I. INTRODUCTION

Path planning is one of the most important issues in navigation that it is used in many applications, such as games, robotics or navigators. The objective of path planning is to create a collision free path from the starting point to the given goal in an environment with obstacles. But there are some problems in path planning: finding the shortest path, or amount of system resources such as CPU time and memory. Depending on the application, the trade-off between shortest path, CPU time and memory usage changes considerably. It is not the same problem to navigate using a computer or console than using an embedded system.

Many studies have been carried out in path planning for mobile robots. The most known algorithms for path planning problem are A*[1-2] and Dijkstra's algorithms [3-4]. Kumar Das et al. [5] implemented the heuristic A* algorithm for a mobile robot in an unknown environment. They considered the distance and time metric as the cost function. But memory usage isn't evaluated. In [6] for path planning in mobile robots, they improved real time A* algorithm, again without memory results included in the experiments. Chˆaari et al. [7] designed a tabu search planner algorithm for solving the path planning problem in a grid environment. The path length is evaluated in different map sizes (small, medium and large scales) and compared to A* and GA. The advantage of their algorithm is reducing the execution time. In [8] Cazenave presents some optimizations for A* and IDA* using an array of stacks instead of a priority queue in A*. Results show a higher speed in execution. The HCTNav algorithm [9] is also a path planning algorithm for grid maps implemented for indoor navigation. It has shown to require lower memory usage than Dijkstra's or A* algorithms, so it is more suitable for low cost robots or systems with limited resources. The author in [10] also proposes a new algorithm for finding the shortest path which aims to reduce execution time, comparing his algorithm to Dijkstra's. In [11] a new way for path finding is described which is based on Compressed Path Databases (CPD). It pre-computes and compresses all the shortest path pairs, therefore reducing runtime execution time. In [12] authors use a coarse grid map instead of the original fine grid map for reducing memory usage. Their approach is intended for embedded systems in cleaning robots. The authors in [13] propose a modified A* algorithm to generate safe paths for robots. It considers the robot size and the direction of arrival (DOA) as parameters for avoiding sharp turns around corners and unsafe diagonal moves. In [14], the authors discuss the relation between the performance of different path finding algorithms and obstacles distribution.

This paper presents a navigation algorithm (NafisNav) in grid based maps. It is intended for low cost robots or embedded systems. The main motivation of this work is to minimize the memory consumption. The algorithm is described and compared in terms of path length and dynamic memory to A* and Dijkstra's algorithms. The grid maps are provided in two different sizes and the experiments are performed for each size.

The reminder of this paper is organized as follows. In section II, A* and Dijkstra's algorithms are described briefly. In section III, the proposed algorithm is described in detail. Section IV shows experimental results. The results are discussed in section V and finally conclusions are given in section VI.

II. DIJKSTRA'S AND A* ALGORITHMS

The Dijkstra and A* algorithms are both graph search algorithms which obtain the minimum path length between two nodes in the graph. The Dijkstra algorithm [3,4] is Breadth-first algorithm with a priority queue based on the distance to the initial node. The A* algorithm [1,2] includes a heuristic function which uses both an estimation of the remaining distance to the target and the current distance to the origin for the priority queue.

The application of these algorithms to a grid-based map requires the generation of a graph based on the map description. The lowest memory cost for this procedure is to generate a one-step graph from each cell to every adjacent free neighbor to which the robot may navigate. Fig. 1 shows an example of the generated graph after processing the grid map.

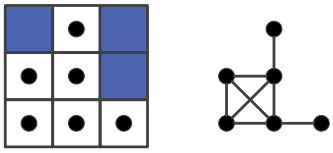

Fig. 1. Graph created from a grid map.

III. NAFISNAV ALGORITHM

In nature, all creatures trying to reach a specified point move towards it, selecting points nearer the target each step. NafisNav algorithm is inspired by this simple rule. On the other hand, most path planning algorithms consider only two parameters for measuring quality: minimum path length and minimum execution time. As memory consumption is very important in low-cost robots and embedded systems, reducing needed memory should be a primary goal for these systems. The performance of the proposed algorithm will be evaluated through experiments and comparison with well known algorithms (Dijkstra and A*).

This algorithm, NafisNav, was designed to be run in low-cost robots for indoor navigation on grid based maps. In contrast to traditional path planning algorithms, this algorithm is rather simple and has the following characteristics.

The main idea is that the approach is similar to the algorithm that a blind person would use. Although the complete grid map is available, the algorithm does not make an a priori analysis of the complete map (does not "see" the complete map, only its immediate surroundings), which is the part that consumes more memory in this kind of maps for other algorithms. Dijkstra algorithm always makes the whole graph of the map, and that is its main memory consumption reason, analyzing parts of the map that are not needed. According to this idea, the algorithm simply moves towards the target from the origin node without considering obstacles until they are found, as a blind person would do. The neighbor cell with the shortest distance to the target is always chosen for obtaining the trajectory, but only among those not already used for the path until now. In this way, loops are avoided. This is true also when obstacles are found. In those cases, some of the neighbor cells are not available (obstacles), but the available neighbor cell with the minimum distance to the target is chosen, even if that distance is greater from the distance of a previous cell in the path (possible when surrounding obstacles). By default only one path from the origin node to the target node is calculated, with an exception: when two neighbors have the same distance to the target node. In this case, one is selected and the other cell is put in a temporary memory, which is called "stack" for future execution. This stack works as a FILO (First Input Last Output), basically as a common stack.

The advantage of the NafisNav algorithm is that it requires less memory, especially in simpler maps where a path can be found without exploring the complete map. The main drawback is that it can lead to non-optimal paths. When reaching an obstacle there is no analysis of other possible solutions, only choosing the available cell with the minimum distance to the target, even if the total path length is finally higher. Therefore, it is expected that the proposed algorithm obtains good memory results at the expense of path length minimization, especially in more difficult maps.

Other important circumstance that must be taken into account is that the idea presented can lead to deadlocks, points in which there is no unused neighbor cell for continuing the search. Therefore, in order to assure that at least one solution is found (even if non-optimal) a backtracking technique is included for exiting from deadlock. In this case, the algorithm "moves" back to an earlier cell with unvisited neighbors. It is important to take into account that a robot would run the complete algorithm before moving (it would need some milliseconds), so there would be no apparent backtracking, as deadlocks are eliminated during the algorithm, not during the movement.

Once the general idea of the algorithm is presented, the details of the algorithm are presented through a pseudo code (Code 1) and some additional explanations.

Before each movement, including the first one, if there is a straight trajectory from the current node to the target node (see navbresenham procedure), the algorithm is finished. Regarding distance calculation, the Euclidean distance is not directly used, but its square. The Euclidean distance from (xi, yi) to (xj, yj) is:

$$D(i,j) = \sqrt{(xi - xj)^2 + (yi - yj)^2}$$

In this work, the square distance, denoted as D2, is used. In this way, no square root is needed, decreasing computation needs and execution time without affecting the obtained path.

When no straight path is available, the origin is included in the path by calling the insert_node procedure. Then the origin node is considered the current node. After that, a loop is executed while there isn't a straight trajectory from the current node to the target node or the stack isn't empty, which means the algorithm must go on. The following step is normally looking for more nodes for the path. However, if there aren't any available neighbors for going on it may be because of two reasons. A deadlock has been reached, so the deadlock_exit procedure is called, or a second path is being looked for but cannot be obtained because nodes cannot be repeated in different paths (see later footprint). We know we are in that case when there is one path already calculated and the stack is empty. That is a break condition, finishing the algorithm with the available path or paths.

The main part of the algorithm is choosing the next current point among unvisited neighbors (those not already included in the path). Only the eight immediate neighbors are considered. The unvisited neighbor with the shortest distance to the target is always chosen. If one node has two neighbors with the same minimum distance to the target node, then the path is divided into two branches from that node. One path is continued and the other will be calculated later, adding the other node to the stack (push).

After each new node is added to a path the algorithm checks if there is a straight line to the target from the new node. In that case, the algorithm would be finished for the current path, and would go on with the stack if not empty. After calculating each path, the footprint from every node will be

included. It means that in the next path, previously selected nodes (footprint) won't be selected again. There is an exception for the footprint: the surrounding nodes of the target node clear their footprints. Therefore, it is possible to reach the target node even if a previous path had used that final step, but the rest of multiple paths will not share any other nodes except the ones adjacent to the target.

Once the path (or paths) has been obtained, it is optimized. Most of the nodes of each path are just one step from each other, as neighbor cells are used during the algorithm. But possibly some not neighbor nodes can be connected directly. This is done in the optimizating_paths procedure, as explained later.

As seen, four procedures are used during the main algorithm: deadlock_exit, navbresenham, optimizing_path and insert_node. In the next sections, each of these procedures is explained in further detail. After those sections, graphical examples of the complete algorithm are included for a better understanding of the algorithm.

Code 1. NafisNav algorithm pseudo code

```
read the origin node (ox,oy), the target node (tx,ty) and the map
 if navbresenham(ox,oy,tx,ty) = clear then
        d² ←(tx-ox)²+(ty-oy)² , path ← (ox,oy), (tx,ty) , length ← d²
 else
   Call Procedure insert_node(ox,oy)
   (cx,cy) ←(ox,oy)
    while   navbresenham(cx,cy,tx,ty) ≠ clear  or  stack isn't empty  do
        if there aren't any neighbors for (cx,cy)  then
            if there is at least one path and stack is empty then
               break
            call procedure deadlock_exit(map, cx,cy)
        min←infinity
        for all neighbors (nx,ny) ϵ (cx,cy) do
            d² ←(tx-nx)²+(ty-ny)² , length ← d²
            if d² < min then
                min←d² , (mx,my)← (nx,ny)
            if d² = min  then
                push (nx,ny)
            Call Procedure insert_node(mx,my)
        (cx,cy) ←(mx,my)
        if navbresenham(cx,cy,tx,ty) = clear  then
            if   stack is empty then
              Call Procedure insert_node(tx,ty)
            else
              Call Procedure insert_node(tx,ty)
               pop(nx,ny) , (cx,cy) ←(nx,ny)
               repeat the before path until before of this node(nx,ny)
        clean footprint the target node and its neighbors
    optimized_paths←Call Procedure optimizing_path(all of paths)
    find shortest path from optimized_paths
```

A. *The deadlock _exit procedure*

When there is no available neighbor cell that has not already been used in the path, it means that a deadlock has been reached. At that time, this procedure starts. It is a backtracking technique used for exiting from the deadlock. This procedure goes back in the sequence of nodes until it finds one that has any unvisited neighbor. That node is called "the promising node". This procedure does two operations: exiting from the deadlock by recognizing the promising node and inserting the new path from the start node to this node. This is illustrated in fig. 2, where an example of a deadlock and a promising node are shown. The backtracking is also shown in fig. 3. The pseudo code of this procedure is shown in Code 2.

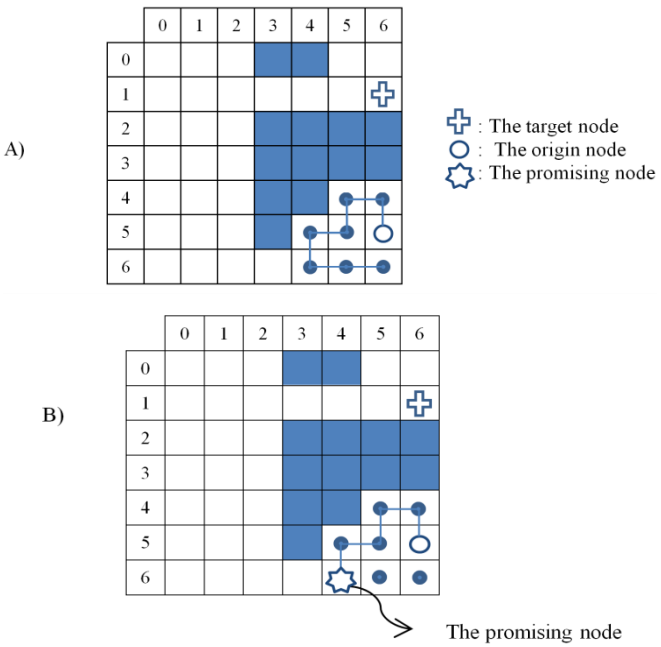

Fig. 2. A) Deadlock example. B) Promising node example.

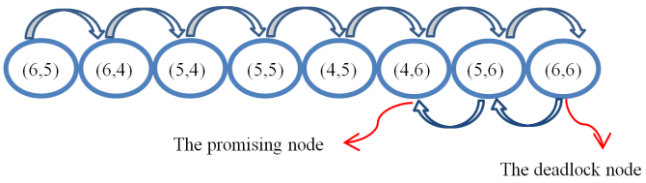

Fig. 3. The path diagram from the origin node (6, 5) to the deadlock node (6, 6) and backtracking to the promising node (4, 6).

Code 2. The deadlock_exit procedure pseudo code

```
for i=node _number-1 to 0 step -1 do
   if  i isn't occupied with obstacle and  isn't filled with footprint     then
       if there is possibility  movement  to one side  then
           for j=0  to i do
              Call Procedure insert_node(xj, yj) // the nodes of previous path
           for all neighbors (nx, ny) ϵ i do
               d² ←(tx-nx)²+(ty-ny) ,  length ← d²
               if d² < min then
                    min←d² , (mx, my) ←  (nx, ny) , (cx, cy)← (mx, my)
               if d² = min  then
                    push (nx, ny)
               Call Procedure insert_node(mx, my)
       Else
           continue
       break
```

B. *The navbresenham procedure*

This is based in the well-known Bresenham algorithm [15], which is commonly used for drawing straight lines in a computer screen with simple arithmetic (not using square roots, for instance). Our procedure calculates all the cells that a robot would touch going in a straight line from one cell to another. The main variation from the classical Bresenham algorithm is that the original algorithm only gives the cells nearer the center of the straight line, but not all cells partially touched by a robot moving through that line. This is solved adding more Bresenham lines from surrounding cells of the origin and

target, as shown in [9]. Therefore, if all the cells returned by this procedure are empty, then a robot can move in straight line without risk of collision.

C. *Optimizing _Path Procedure*

This procedure is used after path or paths are calculated. The original paths include neighbor nodes one step from each other in any of the main eight directions. However, there may be empty straight lines joining some of the nodes of the path in a diagonal not included in the main eight directions. In that case, the final path can be shortened while assuring that a robot can move through the new optimized path. The procedure looks for any possible empty straight line (using the navbresenham procedure) from any pair of nodes of the same path. This is repeated until no more diagonals are found. An example of the procedure is shown in fig. 4, and its pseudo code in Code 3.

Code 3. Optimization algorithm pseudo code

```
i←0                          // the number of paths
For all Pi,j , i ∈ Paths do
    j←0                      // the number of nodes
    For all Pi,j , j∈ Nodes do
        (x0, y0) ← (Pi,j.x , Pi,j.y)
        k← (tx, ty)      // The number of last node or target node
        For all Pi,k , k ∈ Nodes do
            If navbresenham(x0,y0, Pi,k.x , Pi,k.y) = clear then
                Delete all of nodes between (x0, y0), (Pi,k.x , Pi,k.y)
            k←k-1
            If k<j then break
        j←j+1
    i ← i+1
```

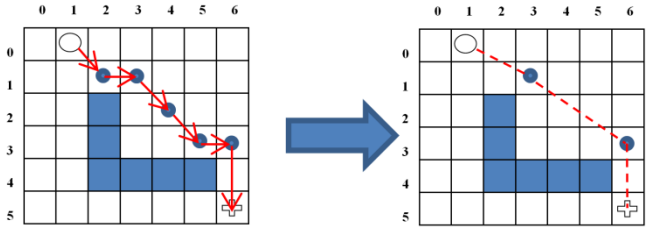

Fig. 4. Initial proposed path and optimized path.

D. *Insert_node Procedure*

In this procedure, a node (xi, yi) is added to the path. If the node to be included is the target point, then it means a new path has been finished, as shown in code 4.

Code 4. Insert_node pseudo code

```
input: (xi, yi)
Add (xi, yi) to the path
if xi = xt and yi = yt then
    create a new path
```

IV. Experimental Results

The NafisNav algorithm is based on a depth-first algorithm which avoids visited nodes and improved with a backtrack procedure. These modifications together with its application to a finite graph ensure its completeness. This section will focus on comparing the proposed algorithm with the A* and the Dijkstra's algorithms. Three aspects may be compared: 1) final path length, 2) memory requirements and 3) execution time. The last aspect, execution time, is the least important as the execution time of the algorithm in the µprocessor is not significant compared to the robot's movement time. Therefore, the following sections will show the results regarding the final path length and the memory requirements of each algorithm.

The three algorithms have been implemented using ANSI-C language. The final code has been compiled using GCC without any optimizing flag. The three algorithms have been executed for the following test set:

- A set of 42 small maps. The size of the maps is 15 columns and 10 rows.
- A set of 6 large maps. These large maps have been created as the scaled and repeated versions of three of the former small maps. The size of the maps is 120 columns and 80 rows, 64 times larger than the equivalent small maps.

Each algorithm has been implemented in a full program that receives the map as a plain text file, the origin and target positions. The result is the sequence of cells for the final path or a message showing that there is no possible solution.

A. *Path length comparison*

The first comparison focuses on comparing the path length provided by the three algorithms. To ensure an equitable comparison, the sequence of nodes provided by Dijkstra and A* is optimized using the "Optimizing_Path procedure" presented before. The optimization procedure makes some of the results obtained by Dijkstra and A* different. The average path lengths for each of the two sets are presented in table 1 and 2.

TABLE I. Average path lengths for the small map set

Algorithm	Average path lengths
A*	7.89
Dijkstra	7.90
NafisNav	8.35

TABLE II. Average path lengths for the big map set

Algorithm	Average path lengths
A*	57.85
Dijkstra	57.57
NafisNav	59.56

A deep study of the results of the three algorithms leads to 13 different occurrences, depending on the comparison of the provided path length of each algorithm. These have been used for presenting the results. For every case the mean percentage of the number of paths is presented for each map set. Table 3 shows the results for the small map set and Table 4 for the big map set.

TABLE III. MEAN PERCENTAGE FOR EACH CASE OBTAINED FOR EACH ALGORITHM IN THE SMALL MAP SET

Case	Result	Average of percentages
1	$PL_{NafisNav}$=PL_{A*}=$PL_{Dijkstra}$	80.591
2	$PL_{NafisNav}$=PL_{A*}<$PL_{Dijkstra}$	5.329
3	$PL_{NafisNav}$=$PL_{Dijkstra}$<PL_{A*}	0.771
4	$PL_{NafisNav}$<PL_{A*}=$PL_{Dijkstra}$	0.295
5	$PL_{Dijkstra}$<$PL_{NafisNav}$=PL_{A*}	0.605
6	$PL_{NafisNav}$<PL_{A*}<$PL_{Dijkstra}$	0.371
7	$PL_{NafisNav}$<$PL_{Dijkstra}$<PL_{A*}	0.166
8	$PL_{NafisNav}$=$PL_{Dijkstra}$>PL_{A*}	0.515
9	$PL_{Dijkstra}$<$PL_{NafisNav}$=PL_{A*}	0.533
10	PL_{A*}<$PL_{NafisNav}$<$PL_{Dijkstra}$	0.194
11	$PL_{NafisNav}$>PL_{A*}=$PL_{Dijkstra}$	9.117
12	$PL_{NafisNav}$>PL_{A*} > $PL_{Dijkstra}$	0.540
13	$PL_{NafisNav}$>$PL_{Dijkstra}$ > PL_{A*}	0.974

TABLE IV. MEAN PERCENTAGE FOR EACH CASE OBTAINED FOR EACH ALGORITHM IN THE BIG MAP SET

Case	Result	Average of percentages
1	$PL_{NafisNav}$=PL_{A*}=$PL_{Dijkstra}$	40.19
2	$PL_{NafisNav}$=PL_{A*}<$PL_{Dijkstra}$	8.04
3	$PL_{NafisNav}$=$PL_{Dijkstra}$<PL_{A*}	4.00
4	$PL_{NafisNav}$<PL_{A*}=$PL_{Dijkstra}$	0.27
5	$PL_{Dijkstra}$<$PL_{NafisNav}$=PL_{A*}	8.53
6	$PL_{NafisNav}$<PL_{A*}<$PL_{Dijkstra}$	1.40
7	$PL_{NafisNav}$<$PL_{Dijkstra}$<PL_{A*}	3.33
8	$PL_{NafisNav}$=$PL_{Dijkstra}$>PL_{A*}	0.80
9	$PL_{Dijkstra}$<$PL_{NafisNav}$=PL_{A*}	3.00
10	PL_{A*}<$PL_{NafisNav}$<$PL_{Dijkstra}$	1.76
11	$PL_{NafisNav}$>PL_{A*}=$PL_{Dijkstra}$	5.66
12	$PL_{NafisNav}$>PL_{A*} > $PL_{Dijkstra}$	12.48
13	$PL_{NafisNav}$>$PL_{Dijkstra}$ > PL_{A*}	10.54

For each of the 13 different possible cases, the path increase penalization is also presented. This penalization is obtained for each case using the following formulas:

P_NafisNav: $((\sum_{i=0}^{n} \frac{(PL_{NafisNav})_i - MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i})/n)$

P _A* : $((\sum_{i=0}^{n} \frac{(PL_{A*})_i - MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i})/n)$

P_Dijkstra: $((\sum_{i=0}^{n} \frac{(PL_{Dijkstra})_i - MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i}{MIN(PL_{NafisNav}, PL_{A*}, PL_{Dijkstra})_i})/n)$

Tables 5 and 6 show the penalization for each algorithm in the small-map and big-map sets respectively.

TABLE V. PATH PENALIZATION FOR EACH ALGORITHM FOR IN THE SMALL MAP SET

Case	NafisNav	A*	Dijkstra
1	0.00%	0.00%	0.00%
2	0.00%	0.00%	7.67%
3	0.00%	2.13%	0.00%
4	0.00%	1.31%	1.26%
5	1.09%	2.39%	0.00%
6	0.00%	0.40%	1.17%
7	0.00%	1.99%	0.62%
8	1.51%	0.00%	1.51%
9	2.86%	2.86%	0.00%
10	0.56%	0.00%	1.23%
11	20.65%	0.00%	0.00%
12	7.51%	1.28%	0.00%
13	18.84%	0.00%	1.33%

TABLE VI. PATH PENALIZATION FOR EACH ALGORITHM FOR IN THE BIG MAP SET

Case	NafisNav	A*	Dijkstra
1	0.00%	0.00%	0.00%
2	0.00%	0.00%	1.38%
3	0.00%	1.725%	0.00%
4	0.00%	0.79%	0.79%
5	1.23%	2.36%	0.00%
6	0.00%	0.54%	1.48%
7	0.00%	2.52%	0.745%
8	0.575%	0.00%	0.575%
9	4.24%	4.24%	0.00%
10	0.365%	0.00%	1.05%
11	13.27%	0.00%	0.00%
12	7.10%	1.33%	0.00%
13	16.60%	0.00%	1.11%

B. Dynamic memory requirements

The second comparison focuses on comparing the memory that each program requires for solving the problem. All three algorithms have been implemented using dynamic memory for storing the map, generating the graph (when required) and generating the solution. The measurement of the allocated memory is done using the Valgrind application. The results are processed with Massif-msprint tool. Similar to the previous experiment, we have measured the memory for every possible path in each map of the set. Fig. 5 shows the maximum memory in each map for the small-map set. Fig. 6 shows the maximum memory in each map for the big-map set.

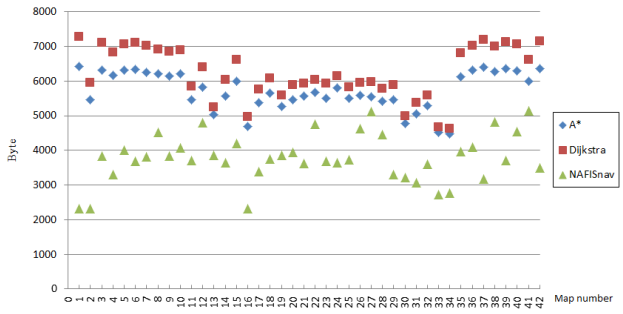

Fig. 5. Comparison of the maximum memory requirement for each small map.

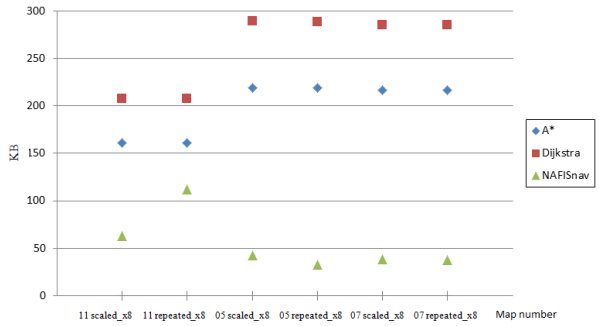

Fig. 6. Comparison of the maximum memory requirement for each big map.

V. Experiment Discussion

As expected, the depth-first approach of the NafisNav algorithm provides, for some cases, longer paths than Dijkstra and A*. This is meaningful observing cases 11, 12 and 13. The normalized penalization in these three cases is 19.25 % for small maps and 13.51 % for big maps. However, the global normalized penalization of the solution provided by the NafisNav algorithm is just 2.08 % in the small-map set and 3.63 % in the big-map set.

The former analysis on its own would not justify the use of NafisNav over the existing A* and Dijkstra's algorithms. The advantage of using NafisNav over the former is the decrease of memory requirement for solving robot navigation. The results for the small-map set shows that NafisNav requires around the 70 % of dynamic memory when compared to Dijkstra and the 80% of dynamic memory when compared to A*. The difference of this memory requirement is more significant for the big-map set. This big-map set resembles the size of real environments. For this set the differences are even more significant as depicted in Fig. 6. These differences are very relevant for choosing a µController when developing a robot.

VI. Conclusions

This paper presents a navigation algorithm (NafisNav) that works with grid maps and that is designed for embedded systems with low resources. The algorithm is similar to what a blind person would do: it just uses the information of the immediate surroundings, advancing towards the goal when there are no obstacles, and otherwise surrounding them. Using only the information of the immediate surroundings greatly minimizes memory consumption, especially in easier and bigger maps. However, the obtained path is not always the shortest possible one, reaching a trade-off between path length and memory consumption. The algorithm is compared using experimental results to A* and Dijkstra's algorithms.

Future research can investigate the performance of the NafisNav algorithm in terms of execution time.

Acknowledgment

This work has been supported by the Spanish Ministerio de Ciencia e Innovacion under project TEC2009-09871.

References

[1] S. Rabin, AI Game Programming Wisdom, Charles River Media, 2002.

[2] M. Buckland, Programming Game AI by Example, Jones & Bartlett Learning, 2005.

[3] R. Diestel, Graph Theory, Graduate Texts in Mathematics. Springer-Verlag, vol. 173, pp. 6–9, 2005.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 658-664, 2009.

[5] P. Kumar Das, A. Konar, R. Laishram, Path planning of mobile robot in unknown environment, Special Issue of IJCCT Vol.1 Issue 2, 3, 4, for International Conference [ACCTA-2010], 2010.

[6] D. Ananya, M. priyadarsini, M. Prakruti, K.D. Pradipta, C.M. Subash, Improved real time A* algorithm for path planning of mobile robot in quadrant based environment, International Journal on Advanced Computer Theory and Engineering (IJACTE), 1, 2319-2526, 2012.

[7] I. Chˆaari, A. Koubˆaa, H.Bennaceur, A. Ammar, S. Trigui, M. Tounsi, E. Shakshuki, H. Youssef, On the Adequacy of Tabu Search for Globa Robot Path Planning Problem in Grid Environments, 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014) , 2014.

[8] T. Cazenave, Optimizations of data structures, heuristics and algorithms for path-finding on maps, CIG'06 (May 22-24 2006), 2006 IEEE, 2006.

[9] M. Pala, N. Osati Eraghi, F. Lopez-Colino, A. Sanchez, A. de Castro, J. Garrido, HCTNav: A Path Planning Algorithm for Low-Cost Autonomous Robot Navigation in Indoor Environments, International Journal of Geo-Information, vol.1, 729-748, 2013.

[10] N. A. El-Sherbeny, A new algorithm of A* algorithm for the time-dependent shortest path problem with time windows, Scientific Research and Impact (Science Park Journals), 2(1): pp. 1-8, 2013.

[11] A. Botea, Ultra-Fast Optimal Pathfinding without Runtime Search, Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2011.

[12] H. K.Lee, W.Y. Jeong, S. Lee, J. Won, A Hierarchical Path Planning of cleaning robot Based on Grid Map, IEEE International Conference on Consumers Electronics (ICCE), 2013.

[13] B. M. ElHalawany, H. M.Abdel-Kader, A. TagEldeen, A.E.Elsayed,Z. B.Nossair, Modified A* Algorithm for Safer Mobile Robot Navigation, International Conference on Modelling, Identification & Control (ICMIC), 2013.

[14] Y. Li, Z. Zhou, W. Zhao, Performance Analysis of Pathfinding Algorithms Based on Map Distribution, TELKOMNIKA Indonesian Journal of Electrical Engineering,Vol.12, No.7, pp.5537-5545, 2014.

[15] J.E. Bresenham, Algorithm for computer control of a digital plotter. IBM Syst. J., 4, 25–30, 1965.