AN IMPLEMENTATION OF PATH PLANNING ALGORITHMS FOR MOBILE ROBOTS ON A GRID BASED MAP

Tolga YÜKSEL
e-mail : tyuksel@omu.edu.tr

Abdullah SEZGİN
e-mail : asezgin@omu.edu.tr

Ondokuz Mayıs University , Electrical & Electronics Engineering Department , 55139 Kurupelit-SAMSUN-TURKEY

Key words : path planning , Breadth-first , Dijkstra , A , grid based map*

ABSTRACT

One of the most known problems for mobile robots on a grid based map is to find the shortest and the lowest cost path from one starting cell to one goal cell and one starting cell to multi-goal cells. In this paper, three algorithms solving the shortest path problem on a grid based map are presented. In addition, the concept of path planning for multi-goal cells is explained and a comparison among the algorithms is performed according to the experimental results.

I. INTRODUCTION

Path planning problem is the fundamental problem for mobile robots. The graph search algorithms are the most known solutions for this problem. These algorithms use the directed or undirected graph trees. In addition these algorithms were used in most computer games and GPS systems for finding the shortest and the lowest cost path [1],[4].

The most known algorithms for the shortest path problem are Breadth-first , Dijkstra and A* algorithms. Most studies focus on one of these algorithms and examines various types of selected algorithm [5],[6],[8],[10],[12]. In this paper, these algorithms are summarized, advantages and disadvantages are defined and experiments for the comparison of the algorithms on various type grid based maps are performed.

The graph search algorithms are based on node-edge notation but this notation lacks when a system like GPS gets an image frame , converts it to a map matrix and uses this map matrix as the grid based map. In these situations using matrix notation gives the advantage of simplicity and comprehension.

Some assumptions are taken into account before starting.

- The map created is same as the real environment. There is no need for re-planning.
- The map is divided into same size square cells.
- The ability of traversing is accepted 90^0 and 4-adjacent traversable neighbours is considered [5],[7]. 4- and 8-adjacency definition is shown in Figure-1.

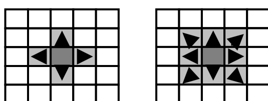

Figure -1 a)4-adjacency b)8-adjacency definition

II. BREADTH-FIRST ALGORITHM

Breadth-first algorithm works with the method branching from the starting cell to the neighbour cells (just traversable cells), (untraversable cells and cells out of boundaries are discarded) until the goal cell is found [3],[4]. Every traversable neighbour cell is added to an array which is called OPEN LIST. OPEN LIST is the array of neighbour cells which must be reviewed in order to find the goal cell. OPEN LIST elements are reviewed if one is the goal cell or not. Then OPEN LIST grows up with the new neighbour cells of the old neighbour cells and this procedure goes on until the goal cell is added to the OPEN LIST. The cost of the starting cell is 0. The cost of each neighbour cell is +1(or +defined constant cost) of the cell which added it to the OPEN LIST. The costs of the cells are stored in a matrix(cost-matrix) with the same dimensions of the map. Then after adding the new neighbour cells , old reviewed cells are pulled out of OPEN LIST. This prevents reviewing the reviewed cells again. When the goal cell is added to the OPEN LIST , to find the shortest path you just follow from the goal cell to the starting cell step by step by the decreasing cost of the cells from the cost-matrix If the OPEN LIST is empty anytime , this means there is no possible paths.

The algorithm is as follows :

Breadth-first Algorithm :
1. Define the starting and goal cells.
2. Load the map matrix.
3. Add the starting cell to OPEN LIST.
4. Add the neighbour cells to OPEN LIST.
5. If OPEN LIST is empty , no possible path.
6. If goal cell is added to OPEN LIST, define the PATH using map matrix. Else compute the cost of neighbour cells.
7. Pull out the reviewed cells from OPEN LIST.
8. Go to step 4.

This algorithm is simple to implement, doesn't need too much matrix operations and also doesn't need to use the location of the goal cell (an advantage if the location of the goal isn't defined) but it has two major and important disadvantages:

1.You have to search the whole traversable cells until the goal cell is found. In large maps it needs very large

computational space. It ignores the knowledge of the location of the goal cell.

2. It is impossible to define cells with different costs (just traversable and untraversable).

III. DIJKSTRA ALGORITHM

This algorithm is like Breadth-first algorithm but adds the computation of different cost cells (not only the shortest path but also the lowest cost path) [4],[6],[9]. In this algorithm, again the neighbour cell array OPEN LIST exists. First the neighbours of the starting cell are added to the OPEN LIST. Then the costs of the neighbours are computed. These costs are the costs of moving from starting cell to these cells and it is the cost function. Neighbour cells are reviewed according to their computed costs. The lowest cost cell is found and first the neighbours of this cell is added to OPEN LIST(lowest cost becomes comparison criterion) For these new cells the cost from starting cell to these cells are computed and again the neighbours of the lowest cost cell is added to the OPEN LIST. This lowest cost criterion obtains the shortest path but it has two problems. First OPEN LIST has to be sorted according to the costs and the new neighbour cells have to be located in the right place in the OPEN LIST. The parents of the neighbour cells have to stored in PARENTS array in order to locate the new cells and in order to find the shortest and the lowest cost path. Furthermore if the cost of the neighbor cell is lower than its parent cell the neighbour becomes the parent and the costs have to re-computed. This procedure goes on until the goal cell is added to the OPEN LIST. When the goal cell is added to the OPEN LIST , following the parents of the cells from the goal cell to the starting cell gives the shortest and the lowest cost path. If the OPEN LIST is empty anytime , it means that there is no possible paths.
Using the lowest cost criterion and the ability of computing different cost cells makes this algorithm efficient in large and different cost terrain maps. But it is lack of searching towards the direction of goal cell.
The algorithm is as follows :

Dijkstra Algorithm :
1. Define the starting and goal cells.
2. Load map matrix.
3. Add the starting cell to OPEN LIST.
4. Add the neighbour cells to OPEN LIST ,compute the costs , record their parent cell to PARENTS.
5. If OPEN LIST is empty , no possible path.
6. If goal cell is added to OPEN LIST define the PATH using PARENTS matrix.Else go on.
7. If neighbour cell is added OPEN LIST before find its new cost and compare to its old cost. If it is lower , update the cost and PARENTS matrix.
8. Pull out the reviewed cells from OPEN LIST.
9. Go to step 4.

IV. A* ALGORITHM

This is the most common and efficient used algorithm in shortest path finding problems [3],[4],[5],[8],[10],[11],[12]. This algorithm has two list arrays OPEN LIST and CLOSED LIST. OPEN LIST does the same work and CLOSED LIST holds the cells that have to be saved. Again first the neighbours of the starting cell are added to the OPEN LIST. And again these cells are reviewed according to their costs. But this time two cost functions exist. First the G cost function is the cost of moving from the starting cell to the current cell and the H cost function is the cost of moving from the current cell to the goal cell. The G cost function can be computed but the H cost function can just be estimated. That's why this cost function is called heuristic cost function. There are several methods for this estimation. For 4-adjacent traversable cells Manhattan method is the most used method. Other methods can be found in the literature [1], [2].

H(current_cell)=abs(currentX–goalX)+
abs (currentY-goalY)

This method directs the search to the goal cell. The total cost function F = G + H is the comparison criterion for the cells. OPEN LIST has to be sorted and in addition as the comparison criterion the F cost array has to be sorted. The parents of the neighbour cells are stored in PARENTS array. Again in this algorithm if the cell exists in OPEN LIST its new cost must be compared to the old cost. If it is lower the cell becomes the parent and G and F costs must be re-computed. The reviewed cells are placed in the CLOSED LIST. Again after the goal cell is added to OPEN LIST, following the parent cells gives the shortest path. If the OPEN LIST is empty at anytime, it means that there is no possible path.
The algorithm is as follows :

A* Algorithm :
1. Define the starting and goal cell.
2. Load the map matrix.
3. Add the starting cell to OPEN LIST.
4. Add the staring cell to CLOSED LIST.
5. Add the neighbour cells to OPEN LIST
- If traversable ;
- If not in OPEN LIST before ;
- If not in CLOSED LIST ;

With the order compute G , H and F cost function values. Record the parent to PARENTS matrix. Locate the F cost function value in the right place.
- If in OPEN LIST before ;

compute the G cost function value. If it is better than the old value , chance the parent with this parent in PARENTS matrix. Update G and F cost functions.
6. If OPEN LIST is empty , no possible path.
7. If the goal cell is added to OPEN LIST define the PATH using PARENTS matrix.
8. Find the lowest cost neighbour cell. Add it to CLOSED LIST and continue the search on this cell.
9. Pull out the reviewed cells from OPEN LIST. Go to step 5.

This algorithm is the most efficient algorithm because it uses both the shortest path information from starting cell and the shortest path information to the goal cell. But if the location of the goal cell is not known, this algorithm can't be used.

V. PATH PLANNING FOR MULTI-GOAL CELLS

The algorithms presented above define the algorithms for one starting cell – one goal cell. But most applications use one starting cell-multiple goal cells without the importance of goal cells order. Think that a mobile robot has to collect all the trashes in an environment and has to turn back to the starting cell. There is no order between the trashes and from one trash point you can reach all the other trash points. In such cases all possible paths have to computed.(Figure-2) First n! paths between the points (n = number of goal points , |SG1| , |SG2 |, |SG3| , |G1G2| , |G1G3| , |G2G3|) have to computed and then the shortest path from starting point to multi-goal points (all points

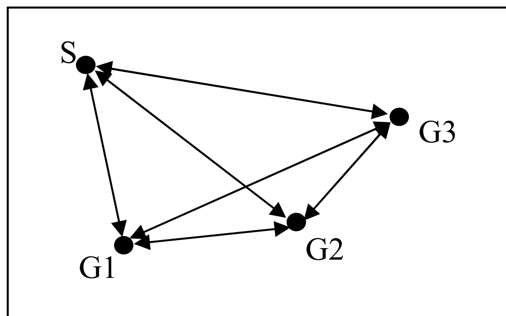

Figure-2 One starting - multi-goal points
(S : starting point , G1-G2-G3 : goal points)

have to be visited once)have to computed. This a most known problem Traveling Salesman Problem(TSP) in graph theory [13],[14]. As like a traveler salesman has to visit a number of towns once and has to turn back to the starting town. The number of entire paths that has to be computed is (n-1)!/2 (n : number of goal points+1). This number is 3 for 3 goal points but for 10 goal points this number becomes 1.814.400. Total computation time for this number is not acceptable and some techniques is used to decrease the number of computation. This subject is not goal of our paper and some useful resources can be found in [13], [14].
The algorithms presented in this paper give some computational advantages in path planning for multi-goal cells. The algorithms seem to find the shortest path between cell to cell but in fact these algorithms can find all the shortest paths from the starting cell to all visited cells. At this point Breadth-first , Dijkstra and A* have to be examined discreetly. Breadth-first and Dijkstra don't use the location of the goal point in the computations that's why they can find all the shortest paths for all visited cells. A* can find paths for all visited cells but doesn't guarantee the shortest path because A* uses the location of the goal point in its computation. This benefit gives Breadth-first and Dijkstra one computation-all use advantage. If the goal cell is not in the visited cells, the computation for the shortest path between cell to cell has to be repeated. In these type of computations, starting the computation between the most far goal cells can give an advantage.

VI. EXPERIMENTAL RESULTS

All the experiments presented below are done with a computer with AMD Athlon 2000+ CPU and 384 MB RAM. All the algorithms are implemented under MATLAB 6.5. The MATLAB Profiler is used to compare the computations of the algorithms.
Experiments are divided into two main groups. One starting – one goal and one starting – multi goal cells experiments.
In the first group the algorithms are compared for one starting-one goal cells. As mentioned in Section II, the Breadth-first algorithm is lack of computing different cost cells. So two maps are used for this group. One with same cost cells (Breadth-first, Dijkstra, A*) and one for different cost cells (Dijkstra , A*).
Figure-3 shows the map with the same cost cells and the paths for the algorithms. The map is a photo from MATLAB *Image Processing Toolbox*. First it is converted to a map matrix then the path is computed from this matrix. The dimension of the matrix is 256*256 cells. The white cells are assumed as the obstacles and the grey cells are assumed as traversable cells. The left top side is assumed (0,0). The coordinates of the starting cell and goal cell are (2,2) and (255,255). The black curve shows the path found. Table-1 lists the comparison of algorithms for CPU time, the sum of the cells, the cells visited, the path cells. It can be seen that although the Breadth-first algorithm visits more cells , its CPU time is better than Dijkstra and A*. The cause of this efficiency is the simplicity. Dijkstra and A* algorithm need a lot of matrix operations and in a map with same cost cells, the costs of the cells must be updated very frequently.
Figure-4 shows the map with different cost cells and the paths for the algorithms. Again the dimensions are 256*256 cell and the coordinates of starting cell and goal cell are (3,3) and (255,255).This map is created by hand. Two layers with costs 60 and 20 surround the big obstacles and the empty spaces in the map are filled with randomly generated different cost cells. They were shown with cells with white and tones of gray according to the cost.(white:10 ,white-like grey: 20, grey:30 , dark grey:40 , black-like grey: 60) The black curve shows the path found. Table-2 lists the comparison of algorithms for CPU time, sum of the cells, the cells visited, the path cells and cost sum of the path cells. It can be seen from the table that A* algorithm doesn't give the shortest and the lowest cost path. The quality of A* algorithm depends on the quality of the heuristic cost function H. If H is close to the true cost of the remaining path , A* algorithm guarantees finding the shortest and lowest cost path. In other condition A* gives no guarantee but it is still efficient. Table-2 shows that the cost sum of the path

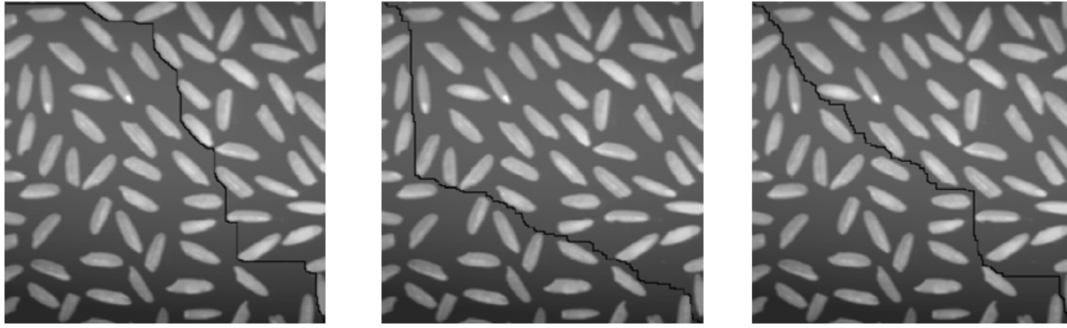

Figure-3 Same cost cell-map a) Breadth-first b) Dijkstra c) A*

Table-1 Comparison of algorithms for same cost cell-map

Algorithm	CPU time(s)	sum of the cells visited	sum of the path cells
Breadth-first	1.078	43002	506
Dijkstra	2.625	43004	506
A*	2.297	25134	506

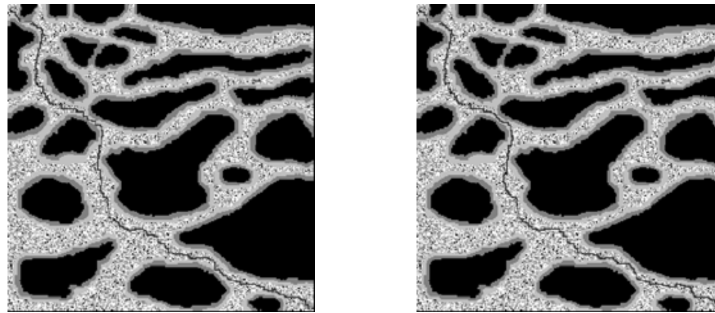

Figure-4 Different cost cell-map a) Dijkstra b) A*

Table-2 Comparison of algorithms for different cost-cell map

Algorithm	CPU time(s)	sum of the cells visited	sum of the path cells	cost sum of the path cells
Dijkstra	2.094	35280	537	6970
A*	1.718	26990	545	7000

cells found by A* is % 0.4 higher than Dijkstra's but it is % 21.9 faster and it needs %30.7 less memory according to the sum of the cells visited.

In the second group algorithms are compared for one starting-multi goal cells. Again the map with different cost cells is used. Three goal cells is defined on the map. The coordinates of starting cell and goal cells are given below.

S : (3,3)

G1 : (120,5)   G2 : (190,140)   G3 : (70,185)

Figure-5 shows the map and the path for the algorithms. Table-3 lists the comparison of algorithms for different start-goal points , CPU times , sum of the path cells cost sum of the path cells and selected path ,total CPU time , sum of the path cells and the cost sum of the path cells. This time A* gives the shortest path. It can be seen that the total CPU times are very close. This result comes from the advantage of computing paths using visited cells. In Dijkstra |SG1|, |SG3| and |G1G2| cells are visited in the previous path and there is no need to

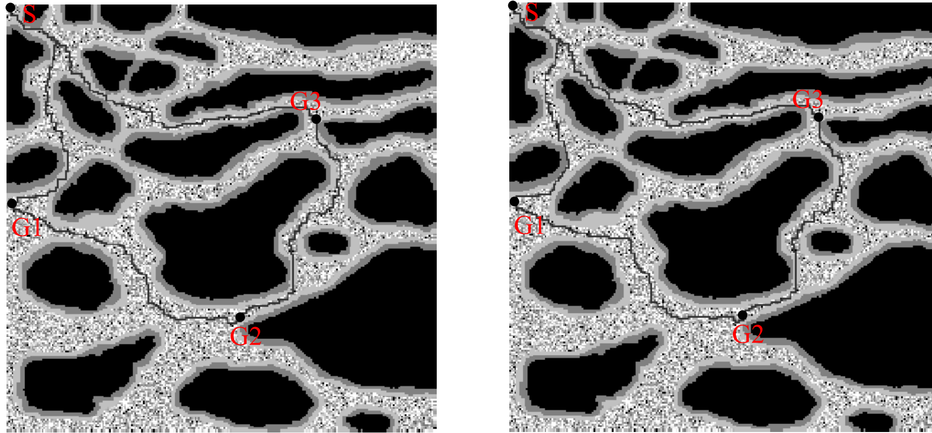

Figure-5 Different cost cell-map with multi-goal points a) Dijkstra b) A*

Table –3 Comparison of algorithms for different cost-cell map with multi-goal points

Algorithm	Start-goal points (most far goal first)	CPU time(s)	sum of the path cells	cost sum of the path cells
Dijkstra	|SG2|	1.453	359	4770
	|SG1|*	0.078	198	2900
	|SG3|*	0.094	276	4220
	|G1G3|	1.594	265	3380
	|G1G2|**	0.078	210	2490
	|G2G3|	1.875	198	2820

* : visited in |SG2| ** : visited in |G1G3|

Selected path

Algorithm	Selected path	sum of the path cells	cost sum of the path cells	Total CPU time
Dijkstra	|SG1|+|G1G2|+|G2G3|+|G3S|	882	12430	5.172

Algorithm	Start-goal points (most far goal first)	CPU time(s)	sum of the path cells	cost sum of the path cells
A*	|SG2|	1.156	359	4780
	|SG1|	0.672	200	2900
	|SG3|	0.953	272	4220
	|G1G3|	0.891	265	3380
	|G1G2|	0.704	210	2490
	|G2G3|	0.781	198	2820

Selected path

Algorithm	Selected path	sum of the path cells	cost sum of the path cells	Total CPU time
A*	|SG1|+|G1G2|+|G2G3|+|G3S|	880	12430	5.157

re-compute the cells. A* is lack of this advantage but it is still more efficient in computing operations.

VII. CONCLUSIONS

This paper presents three path planning algorithms for a mobile robot on grip based map for one starting-one goal cell and one starting-multi goal cells. From the results of the experiments and the inferences from the algorithms some suggestions can be done for path planning for maps with same cost cells, different cost cells and with one starting-one goal and one starting-multi goal cells.
For maps with same cost cells, with one starting-one goal cell and multi goal cells, using Breadth-first algorithm is the best if the computational time is the first desire criteria. But if the size of memory is the first criteria using A* can be a better alternative.
For maps with different cost cells and with one starting - one goal cell A* is best in computational time and size of memory. But the heuristic function H for A* must be chosen carefully in order to make sure of the shortest and lowest cost path.
For maps with different cost cells and with one starting-multi goal cells A* is best in computational time with no guarantee for the shortest path. But it must be noted that Dijkstra, using visited cells advantage especially in enormous multi-goal cells and shortest path guarantee, can be a good choice in these maps.
The algorithms use 4-adjacent traversable cells related to the mobile robot. If a mobile robot with more movement abilities is accepted , using 8- and 16- adjacent traversable cells give better results.
In the experiments A* uses Manhattan method as the heuristic function. Using other functions can give better results.
Choosing the shortest path for multi goal cells using the TSP solving methods will be the next step of this study and it is planned to use real geographical maps instead of the imaginary generated maps.

VIII. ACKNOWLEDGEMENT

This study is a part of project MF104 at O.M.U. and will be used in a GPS-like system with LEGO MINDSTORM based tank-like robot for finding shortest paths on grid based maps.

IX. REFERENCES

1. P.Lester , "A* Pathfinding for Beginners", 2004 , www.policyalmanac.org/games/aStarTutorial.htm
2. A.J. Patel , "Heuristics", 2004 http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
3. K. Manley, "Pathfinding : From A* to LPA", seminar , 21 Apr 2003 , http://csci.mrs.umn.edu/UMMCSciWiki/pub/CSci3903s03/KellysPaper/seminar.pdf
4. B. Stout, " Smart Moves :Intelligent Pathfinding ", Game Developer , October 1996 www.gamasutra.com/features/19970801/pathfinding.htm
5. D.R. Wichmann , B. C. Wünsche , "Automated Route Finding on Digital Terrains", Proceedings of IVCNZ '04, Akaroa, New Zealand, 21-23 November 2004, pp. 107-112.
6. M. Noto, H. Sato,"A method for the Shortest Path Search by Extended Dijkstra Algorithm", IEEE International Conference on Systems, Man, and Cybernetics, Volume: 3, 8-11 Oct. 2000 Pages:2316 - 2320
7. T. Ersson, X. Hu, "Path Planning and Navigation of Mobile Robots in Unknown Environments" , , 2001. Proceedings. 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Volume: 2,29 Oct.-3 Nov.2001 Pages : 858 - 864
8. A. V. Goldberg, C. Harrelson , "Computing the Shortest Path : A* Search Meets Graph Theory" , Technical Report MSR-TR-2004-24, Microsoft Research, March 2004.
9. J. Huh , H. Park , Y. Huh , H. Kim , " Path Planning and Navigation for autonomous Mobile Robot", IECON 02 , Volume: 2 , 5-8 Nov. 2002 Pages:1538 - 1542
10. C. Wurll, D. Henrich , " Point-to-point and Multi-Goal Path Planning for Industrial Robots" , Special Issue on "Motion Planning" of the Journal of Robotic Systems, 2001
11. C. Wurl , D. Henrich , H. Wörn , " Multi-goal Path Planning for Industrial Robots", International Conference on Robotics and Application (RA'99), Santa Barbara, USA, Oct. 28-30, 1999
12. T. Goto, T. Kosaka, H. Noborio , " On the heuristics of A* or A Algorithm in ITS and Robot Path-Planning", Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on , Volume: 2 , 27-31 Oct. 2003 Pages:1159 - 1166 vol.2
13. D. Appplegate , R Bixby ,C. Chvatal ,W. Cook , " Solving Traveling Salesman Problem " , www.tsp.gatech.edu
14. K. Hoffman " Traveling Salesman Problem" , http://iris.gmu.edu/~khoffman/papers/trav_salesman.html