Robot Path Planning using A* Algorithm

Author: **Wael Farag**

1. Objectives

The objective of this project is to develop (code) the A* algorithm [1] in C++. The following points have to be taken into considerations while using the implemented code:

1. The code is implemented using Code::Blocks IDE [2], MinGW-64bit C++ compiler [3] and Windows 10 platform.
2. The hardware used is a Lenovo ThinkPad CORE i5 2.3GHz 8GB RAM laptop.
3. The project uses only the standard C++ libraries. Many comprehensive libraries have been avoided like "OpenCV" [5] for graphics or "Eigen" [6] or "Boost" [7] for matrix manipulations which I usually use for machine learning projects. The idea is to make it as light-weight portable as possible.
4. The software architecture is designed to be simple and easy to understand and follow without over use of complex data structures. However, I believe there are a wide room for improvement and efficiency enhancements, which will be discussed at the end of this report.
5. The variables and data structures names are selected to be self-explanatory (even though comments are added all-over the code files for better code comprehension). The names are sometimes long but meaningful.

2. Structure of the code

The A* code consists of only 4 files (1 ".cpp" and 3 header files) and can be described as follows:

1. "main.cpp": it includes only the pipeline of the algorithm (Set-up the map => Initialize the search grid => Define the starting and end points of the robot => Search the shortest and less costly path (core of the A*) => Display results. The functions implementations and data structure definitions are not done within this file.
2. "Cell.hpp": it includes the main data structure of this implementation which is the "Cell" class with all the parameters and methods (including the constructor and destructor) are defined and coded in this file.
3. "Path_Finding.hpp": this file includes all the functions that are used for setting up the grid and search the optimal path within this grid.
4. "MAP.hpp": this file includes all the details of the search space map. The search space is completely defined in this file with its dimensions and obstacles (walls). You can define several work spaces and their associated topologies in this file and select one of them at each time of the algorithm execution.

3. Setting up the Search Space

Setting up the search space is very simple. It is only done within the “MAP.hpp” file. All you need to do is to define two sets of parameters as follows:

1. The word size: which is simply defined by the constants “N_COLS” and “N_RAWS” in the “Construct_MAP()” function. Every map has its own N_COLS/N_RAWS values. Simply comment or uncomment the associated code lines to unselect or select a certain map.
2. The Obstacles Coordinates: The obstacles are called walls in the code. Define each obstacle by its coordinates (defined in the structure “Wall”). All the obstacles must be defined in the function for example “Set_Map2_Walls()”. The program will assume that all undefined grid cells are open (free) cells.

4. Some Specifics of the Implementation

The following are some specifics of the implementation:

1. G calculation: It can be calculated using either the Euclidean distance or the Manhattan distance. Both methods are implemented in the code, however the calculation of G is always using the “Euclidean distance” as it fulfills the requirements of making the diagonal movement as “$\sqrt{2}$” instead of “1” in case of lateral movements.
2. H calculation: It can be calculated using either the Euclidean distance or the Manhattan distance. There is no restrictions on the method used, it is just a matter of preference and performance. I have tested both methods and the results are shown in the testing and evaluation section of this report.
3. Path Steps: the implementation used a linked list to determine the optimal path steps where each cell memorizes from where the robot to it came from (the previous cell).
4. The algorithm terminates searching in two cases: the first one when the robot reaches the final destination “EndPos” in which case it has reached a “Solution” successfully and the second one when the cell candidate in the “openList” vanishes and moved to the “closedList” (means no more valid cells to search) in which case no solution is found.

5. Testing and Evaluation Results

The algorithm implementations have been tested intensively using several world maps and test cases. However, the code includes only three world maps for convenience. Moreover, the following sample test cases will be presented to shed the light on the performance of the implementation.

Test Case #1:

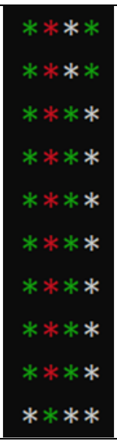
Using ‘Map1’, Euclidean distances for both G & H. The following are the results:

Map	1
World Size	4 x 10

Start Position	(0,0)
End Position	(3,0)
G	Euclidean distance
H	Euclidean distance
Solution Found	Yes
Total Path Cost	19.2426
No. of Searched Cells	25
Computation Time	0.188 Sec

<u>Legend:</u>

Red: Walls
Green: Path Cells
White: Normal Grid Cells



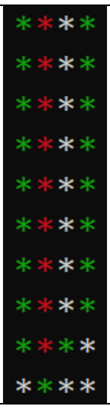
<u>Test Case #2:</u>

The same like the above except changed the calculation of ‘H’ to be Manhattan Distance.

Map	1
World Size	4 x 10
Start Position	(0,0)
End Position	(3,0)
G	Euclidean distance
H	Manhattan distance
Solution Found	Yes
Total Path Cost	19.2426
No. of Searched Cells	18
Computation Time	0.25 Sec

Legend:

Red: Walls
Green: Path Cells
White: Normal Grid Cells



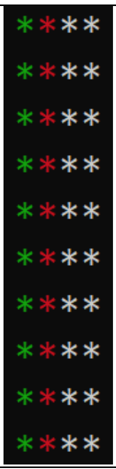
Test Case #3:

The same like the above except changed the Walls positions in the map.

Map	1
World Size	4 x 10
Start Position	(0,0)
End Position	(3,0)
G	Euclidean distance
H	Manhattan distance
Solution Found	No
Total Path Cost	9.0
No. of Searched Cells	10
Computation Time	0.141 Sec

Legend:

Red: Walls
Green: Path Cells
White: Normal Grid Cells



Test Case #4:

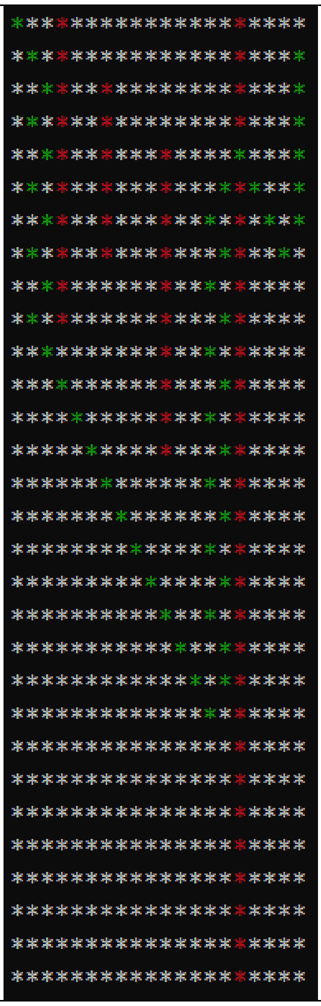
Using 'Map 2':-

Map	2
World Size	20 x 30
Start Position	(0,0)

End Position	(19,1)
G	Euclidean distance
H	Manhattan distance
Solution Found	Yes
Total Path Cost	63.9828
No. of Searched Cells	232
Computation Time	0.858 Sec

Legend:

Red: Walls
Green: Path Cells
White: Normal Grid Cells



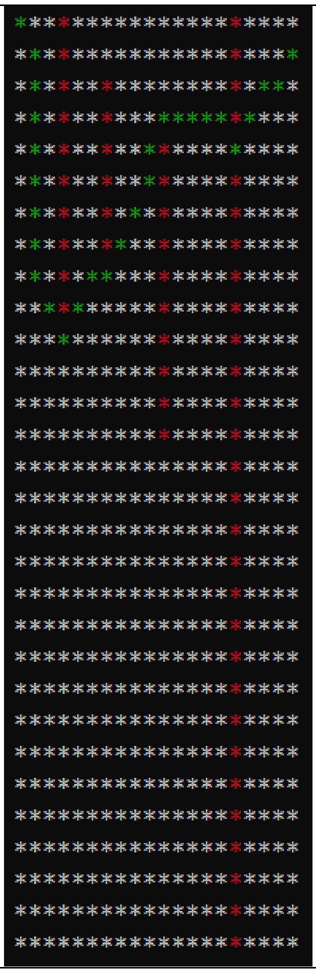
```
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
```

Test Case #5:

The same like the above except changed the calculation of ‘H’ to be Euclidean Distance.

Map	2
World Size	20 x 30
Start Position	(0,0)
End Position	(19,1)
G	Euclidean distance
H	Euclidean distance
Solution Found	Yes

Total Path Cost	32.3848
No. of Searched Cells	113
Computation Time	0.453 Sec

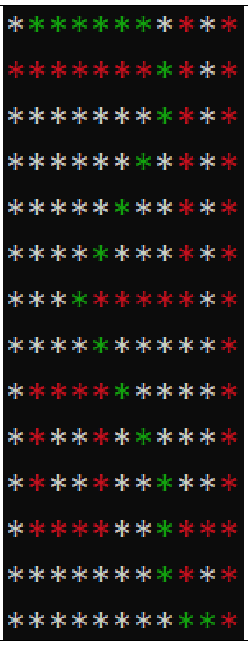
<u>Legend:</u> Red: Walls Green: Path Cells White: Normal Grid Cells	

```
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
```

<u>Test Case #6:</u>

Using ‘Map 3’:-

Map	3
World Size	11 x 14
Start Position	(1,0)
End Position	(9,13)
G	Euclidean distance
H	Euclidean distance
Solution Found	Yes
Total Path Cost	23.1421
No. of Searched Cells	45
Computation Time	0.344 Sec

Legend: Red: Walls Green: Path Cells White: Normal Grid Cells	

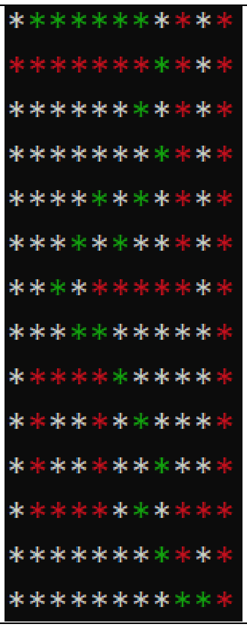
Test Case #7:

The same like the above except changed the calculation of ‘H’ to be Manhattan Distance.

Map	3
World Size	11 x 14
Start Position	(1,0)
End Position	(9,13)
G	Euclidean distance
H	Manhattan distance
Solution Found	Yes
Total Path Cost	28.2132
No. of Searched Cells	24
Computation Time	0.266 Sec

Legend:

Red: Walls
Green: Path Cells
White: Normal Grid Cells


```
***********
***********
***********
***********
***********
***********
***********
***********
***********
***********
***********
***********
***********
***********
```

6. Conclusion

The implementation of the A* path finding algorithm has passed all the extensive carried out tests. Using the Euclidian distance for calculating both G and H showed that it has achieved the optimal path with the lowest cost in all the executed test cases. In the other hand, using Manhattan distance for calculating H managed to find a path but not necessarily the optimal one as shown clearly in the comparison between test cases 4 and 5 as well as test cases 6 and 7. The previous results makes sense, since the diagonal movement is allowed with a cost of 1.414 which saves ~30% of the cost if the robot is allowed only horizontal and vertical movements.

Moreover, the algorithm is fast enough to be used in many serious applications. The execution time for all the test cases never exceeds 1 Sec which is remarkable given the pretty low profile computational power used.

7. Suggestions for Improvements

Improvements for any piece of software never ends. There are always room to do a better things. Here, I am proposing here some improvements that can be done to enhance the code quality:-

1. The code includes many global variables and data structures like the matrix ‘Grid’ and the vectors ‘openList’, ‘closedList’ and ‘pathList’. It is better for these variables to be encapsulated in a structure or a class. I believe the program needs another class like “PathFinding” which groups together most of these global variables and the functions written in the file “Path_Finding.hpp”.
2. The initialization of the grid, initializes all the cells in the grid and their neighbors in advance, which means it allocates memory for all that. However, not all these cells got visited or checked by

the algorithm. I believe there is a better way to dynamically initializes (and consequently allocates memory) these cells based on need.

3. I didn’t do much work to investigate the implementation efficiency in terms of speed. This point needs more and more focus especially if this is going to be used with huge search spaces and may be millions of cells.

8. References

[1] Wikipedia, “A* search algorithm”, https://en.wikipedia.org/wiki/A*_search_algorithm
[2] Code::Blocks, http://www.codeblocks.org/
[3] MinGW, http://mingw-w64.org/doku.php
[4] Windows 10, https://www.microsoft.com/en-us/windows/features
[5] OpenCV, https://opencv.org/
[6] Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page
[7] Boost, http://www.boost.org/