CS 421 Report

A* Algorithm using Makeblock mBot

Robot Motion Planning

Ashir Judah
Fauzi Mohamed
Jeremiah Erome

UofR Undergraduates

david22a@uregina.ca
mohmed2f@uregina.ca
eromeuto@uregina.ca

Table of Contents

1. Abstract:

The design of the project was to create a path finding algorithm to find the goal route from an initial tile in a defined map, in order to provide the path to the robot to find the goal tile. We planned to implement the Makeblock mBot robot using A* algorithm in order to utilize the path finding method to search for the goal tile from the initial tile. Mblock 3 is the software we used to implement functionalities into the mBot robot hardware and is also used as reference for our program. Arduino C (C & C++ functionalities) is the programming platform we used for our implementation. We decided to use a 10x10 grid to represent our map's structure, because it allows optimal movement for our mBot since the grid is neither too small or too big. We conducted research into which heuristics methods may prove to be the best for our program, and we found that the Manhattan Distance method was not only simple enough to understand but also simple to implement. We decided to create a function that calculates the number of possible moves that can be taken per tile as well. Since the tiles chosen in the path were picked efficiently from our open list, the path finding algorithm we implemented proved to be very useful. We decided to use time as movement rather than distance for the following many reasons, more explained in the report.

2. List of Keywords

- A* Algorithm
- Mbot
- Path finding
- Arduino C
- Heuristics

3. Introduction

The motivation for this project was to implement a robot that would reach a goal. This was chosen because we were interested in exploring the concept of AI in Video Games and Robotics. We thought that an implementation of this concept would be a good experience and beneficial in our education. The design of the project was to create a path finding algorithm to find the goal route in a virtual map in order for the robot to move to the goal.

4. Problem

We planned to implement the Makeblock mBot robot using A* algorithm in order to utilize the path finding method to search for the goal tile from the initial tile, and we wanted to improve the consistency of our search, by making sure that after all the calculations, the

mBot will indeed always find its goal, while taking the best path without backtracking. The search for the goal state is done using a 10 x 10 grid as the map, and using heuristics from the A* algorithm to find the goal state.
Using the Mbot robot and working with Arduino C and Mblock software, we design the pathfinding map and algorithm to get the robot from its initial state to the goal state.

5. Software and Programming Languages

5.1 Working with the Makeblock Software (known as Mblock)

Mblock 3 is a software that is used to implement functionalities into hardware based on Arduino C and Scratch code, and is used for implementing hardware in robotics, specifically Makeblock bots. We initially started with mBlock 5 but as we started to use it, we realized that it didn't fit with what we were trying to use due to incompatible arduino code. So, we switched to mBlock 3 to create parts of our implementation, primarily for functions that were focused on using mBot movements, for example moving forward, turning, etc.. It was also used to initialize the base movements of the mBot's robot wheels. The Mblock software was useful for writing the more basic aspects of our program code, including many of the looping code and general structures of our code.

5.2 Arduino C

Arduino C (C & C++ functionalities) is the programming platform we used for our implementation. Arduino is the IDE that is used by mBot and was helpful for our implementation since all of us have experience in Arduino code. There were benefits with using Arduino C, primarily that it was easy to modify and had libraries/examples available for us to use to aid in developing the program. Arduino C's platform is very compatible with academics, making the learning process smoother, since a major part of our degree composes of learning C++ primarily. Arduino is a flexible option to use when working with controllers such as the mBot and is very useful when working with electronics. Arduino also has compatibility with a wide range of sensors, given the right libraries specific to the Makeblock robot used in the process.

6. Grid Map

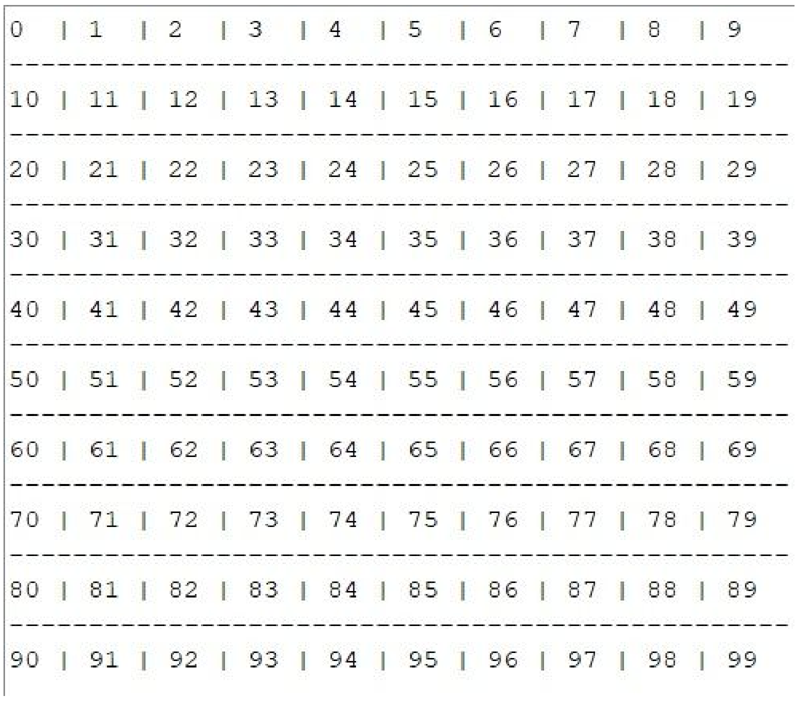
6.1 Choice of Grid Map

We decided to use a 10x10 grid to represent our map's structure, because it allows optimal movement for our mBot since the grid is neither too small or too big. Initially, we started with a 5x5 grid but with the suggestion of Dr. Malek, we decided that a 10x10 grid is preferable. However, As a result of this, any grid larger would require a lot of space in order to perform our robot's movements. Each tile is approximately 18 inches, given that our mBot requires some space to move, in order to properly represent each movement the mBot takes.We first built a virtual map in our Arduino C program to visualize how the grid map is structured, then from there we built a semi-realistic scenario in a lab, in order to view the experimentation of the grid map.

6.2 Structure of Grid Map

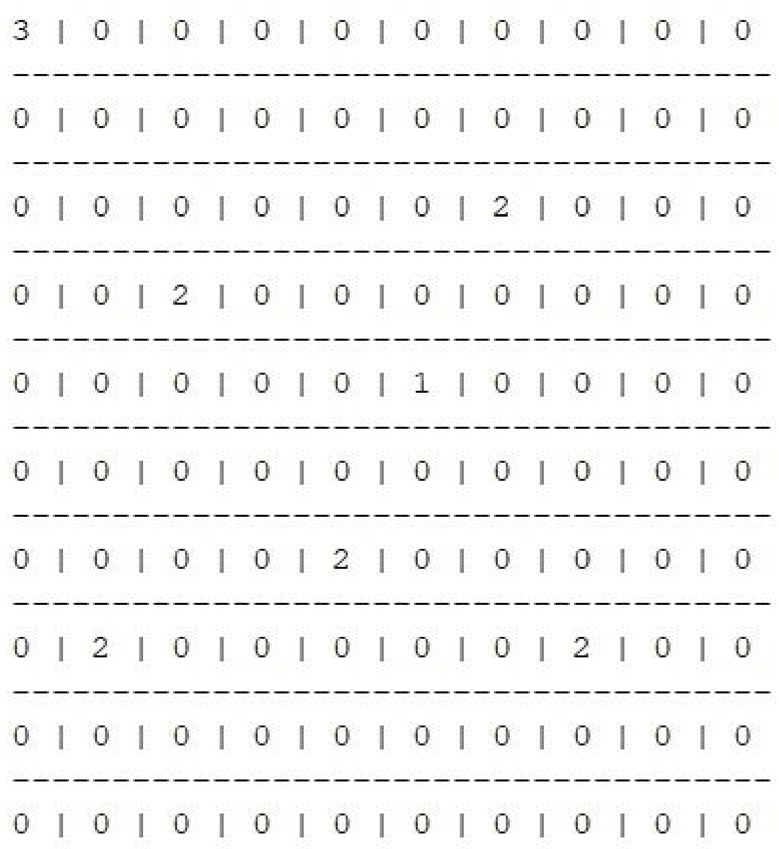
```
struct Node
{
  byte g, h, f; ----------------heuristics variables
  byte parent; ---------------tile's parent tile
  byte index; ----------------tile representation
  byte gridNom; ------------tile's grid number
};


struct Grid
{
  Node Map[row][col];-----tiles at each position contains multiple variables
} PF ; -------------------------path finding object of type Grid
```

6.3 Design & Representation of Grid Map



```
0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9
-------------------------------------------------
10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19
-------------------------------------------------
20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29
-------------------------------------------------
30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39
-------------------------------------------------
40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49
-------------------------------------------------
50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59
-------------------------------------------------
60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69
-------------------------------------------------
70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79
-------------------------------------------------
80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89
-------------------------------------------------
90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99
```

V1: Indices represent each tile's number on the Grid Map & indicates the corresponding path taken on the grid map



```
3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0
-------------------------------------
0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------------
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
```

V2: 0 represents untravelled path, 1 represents initial/travelled path, 2 represents obstacles and 3 represents the goal tile

6.4 Functions

Void buildMap() - essentially, this initializes the Grid Map and sets all values (G, H, F, parents, etc.) to 0

Void printGrid1() - prints the Grid Map with the grid numbers (0 - 99)

Void printGrid2() - prints the Grid Map with the index values representing the path (0 for untravelled, 1 for initial/travelled, 2 for obstacles and 3 for goal)

7. A* Algorithm

7.1 Research of A* Algorithm

We conducted research into which heuristics methods may prove to be the best for our program, and we found that the Manhattan Distance method was not only simple enough to understand and implement, but also one recommended method suggested by experts we found online; and since our mBot only travels in one of four directions, the Manhattan

Distance was the best suited for this. Not only that but we also had an assignment which implements search algorithms, A* Manhattan Distance being one of them, so it made things easier for us since we were familiar with the method.

7.2 Heuristics / Manhattan Distance Method

We decided to implement the Manhattan Distance which finds the amount of tiles needed horizontally and vertically to get the mBot from the initial tile to the goal tile, and this is how we calculated our H value. The G value was calculated based on the “depth” of the “tree” if we were to think about it like a tree. So, essentially the initial tile had a zero G value and its children had one as a G value, and everytime the tile had available children, we incremented the G value by 1. The F value was calculated by adding the tile’s individual G and H values which results in our F value, used in the heuristics function.

7.3 Grid Map Breakdown of Possible Moves

We decided to create a function that calculates the number of possible moves that can be taken per tile, and how we calculated it was quite simple. First, we checked if the mBot is on a corner tile and if it is then there would be two possible moves; second, we checked if the mBot is on an edge of the Grid Map (row 1 - 8, column 10 - 80, column 19 - 89 and row 91 - 98) and if it is then there would be three possible moves; finally we checked for every other tile besides the ones previously mentioned and there would be four possible moves from those tiles. Using this method, it allowed us to simplify the method of calculating the possible moves and allowed us to visualize how the algorithm would be implemented.

7.4 Path Finding

Our closed list fortunately only contained the path needed for the mBot to take to reach its goal tile, since the tiles chosen in the path were picked efficiently from our open list, using a function that retrieves the lowest/best F value from the open list. The path finding algorithm we implemented always succeeded in terms of finding one of the best paths from the initial tile to the goal tile; although it was complicated when incorporating obstacles which could potentially be in the way of our goal tile. We also implemented the ability for the robot to move

7.5 Functions

Void setup() - calls functions to build the Grid Map, prints the two versions of the Grid and then asks user for input concerning the goal tile

Void loop () - this function keeps looping and checks to see if the goal tile has been found, and stops execution of the loop if the goal has been found and checks if the mBot is done moving

Void _delay(float seconds) - when called upon, it will delay the program's execution by the amount of seconds specified; is not used

Void delay(float milliseconds) - when called upon, it will delay the program's execution by the amount of milliseconds specified; we used this one since it worked more efficiently with our program

Void _loop() - the loop function calls this function, and the purpose of this function is to call the possible moves function and then add the appropriate tile to the closed list after the heuristics has calculated the "best" move

Void setGoal() - asks the user for input based on setting the goal tile, and initializes tile 45 as the initial tile(since its approximately in the middle) and initializes fixed tiles (numbers) for the obstacles

Void possMov(byte gridNom) - calculates the number of possible moves from the current grid number (gridNom), explained earlier in this report; we used byte instead of int because of memory space and byte requires less memory usage than an integer (further explained later in this report)

Void AddOpenList(byte aol) - adds the possible moves of a tile grid number (aol) passed to this function, to the openList array, and calls the heuristics function

Void heuristics(byte curIn) - takes in a tile's grid number as a parameter and calls the H, G and F (FV) heuristics calculating functions in order to calculates the current tile's cost

Byte getNextFI() - this functions finds the lowest/best heuristics cost of all the tiles in the openList and returns the grid number associated to the tile grid number chosen

Void AddClosedList() - adds the chosen tile (mentioned previously) to the closed list array, sets the tile's index to 1 (travelled), marks the current position as the mBot's current position and calls the function which removes tiles from the open list

Void removeFOL(byte rfol) - since it is possible that the lowest cost calculated may be earlier in the path chosen, we decided to implement this function to remove the 2 earliest tiles in the open list every time it is called, in order to get the "best" path without interruption; 1 tile removed every time is too small of a number and could not guarantee optimal results, and 3 or more tiles removed every time may cause errors if for example there are less tiles in the openlist than the 3 or more tiles removed

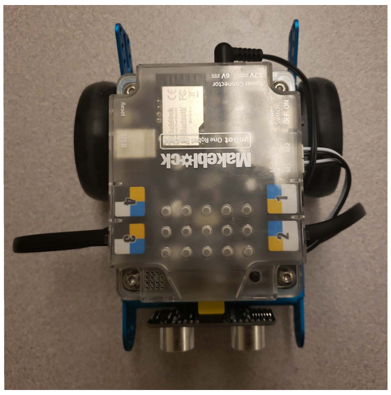
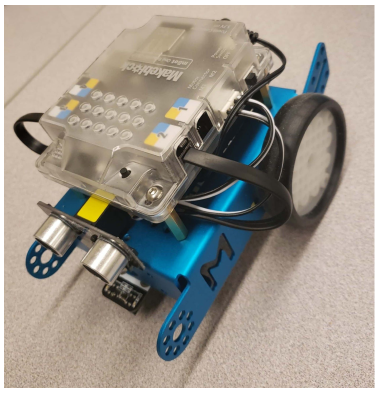
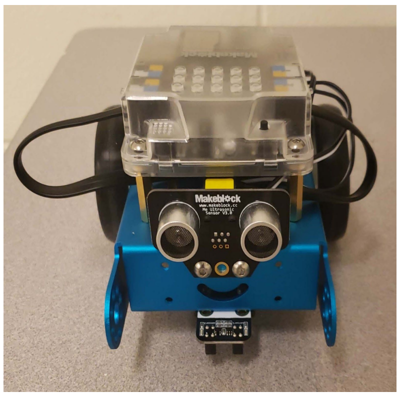
Bool OLE() - this function checks to see if the open list is empty and returns true if it is

Bool isGoal(byte ig) - this function checks to see if the current tile number (ig) is the goal tile and returns true if it is

Bool alreadyOnOL(byte rowaol, byte colaol) - this functions takes in the possible moves row and column number, and checks to see if the tile is already in the open list, so it does not add to it again

8. mBot Movement

8.1 mBot Robot

Our mBot has two sensors, one line following sensor and one ultrasonic sensor, as well as an IR remote. The line following sensor is self explanatory, it makes the mBot to follow black lines as long as those lines are directly under the sensor, and the ultrasonic sensor sends out wave signals which bounce off surfaces back to the mBot to calculate its distance.

8.2 Movement Reasoning

We decided to use time as movement rather than distance for the following reasons: We could not find a room with the same dimensions as our grid and the rooms that came close had objects within them and we could not simply take them all out. If we used the line following sensor, the mBot would continuously follow the black line without stopping, unless we worked with the ultrasonic sensor as well as a variety of extra measures, and it proved to be very difficult to do. We also could have used the ultrasonic sensors but since the room or location we had was not the same dimensions we needed, those calculations based off distance would be incorrect; also, even if we used a grid of our determined size, since the right wheel is a little stronger than the left, it would veer and make the calculations incorrect when turning or moving.

8.3 How it Works

We decided to use time as the distance moved by the mBot, anytime the mBot needed to go forward, it would approximately move forward for a second and a half. Anytime the mBot needed to turn left or right, we set the speed of the movement to be fixed (speed of 126.5) and turned the mBot approximately for a second for both turns. Finally, how we turned the mBot 180 degrees to face the other way, we implemented the mBot to take two right turns to execute this. In order for the mBot to know its location on the grid, after every move it takes, it repositions itself to face the “front” so this would not lead to random and incorrect movements by the mBot.

8.4 Functions

Void movement() - performs the actual movement of the mBot corresponding to the tile path chosen by the algorithm

Void forward() - moves the mBot forward for a second and a half

Void leftturn() - turns the mBot to the left in approximately a second

Void rightturn() - turns the mBot to the right in approximately a second

Void backwards() - performs two rightturns()

9. Experiments & Results

We found that, besides the mistakes made by the mBot itself, our path finding algorithm always found its goal tile and took appropriate steps to reach that tile, given that the surface (we used tables) was large enough to allow the mBot to make its appropriate movements. That being said, the mBot had a lot of trouble moving around obstacles and taking appropriate measures to reach its goal tile; the mBot did indeed avoid obstacles but most or some of the times, it would make different moves from what was needed and would sometimes barely miss the goal tile by one tile. Besides that issue (described more in Issues & Difficulties), most of the time we executed our program, our mBot always found it goal, taking the best path, again given that the surface was large enough for the mBot to move appropriately (otherwise it would fall off the table).

10. Issues & Difficulties

10.1 Algorithm

We experienced issues with the A* algorithm we implemented because by every right, the algorithm is meant to find the lowest F value calculated in the heuristics, but this proved to impact our program negatively. There were occasions where the lowest F value picked was a tile way earlier in the path, instead of the tiles near the current position of the mBot, and this proved to be troublesome because it made our algorithm faulty. So, what we did was that every time we add an element to the closed list, we would take out 2 of the earliest tiles in the open list as well as the tile that was chosen for the lowest F value, in order for our algorithm to run more smoothly. Indeed, it did run incredibly better but there can always be improvement and we can implement an optimal/better solution to this problem in the future.

10.2 Arduino Memory

The Arduino's dynamic memory is quite small in relations to other platforms, but it is meant to be this way because Arduino is meant to build smaller projects than large scale projects. This proved to be an issue for us because the program took up approximately 91% of the dynamic memory, and if the dynamic memory is almost full, this problem may lead to miscalculations, errors and/or bugs, according to Arduino. These issues are further explained in the next subsections. The local memory however was completely fine and we used approximately 25% of the local memory, which did not prove to be an issue.

10.3 mBot Movement

This specific topic proved to be our biggest challenge yet and we will tell you why. The mBot we worked with is not entirely strong, meaning that any dust/grains/etc on the surface we run the mBot's program on, these debris would affect the mBot's movements and slow it down, which interfered with us a lot since the turns and movements were based on time rather than distance. The reason for this is because the mBot has two sensors, and in our current situation, it was very tough to use these to our advantage. Another issue we ran into is the fact that the mBot's left wheel is weaker than its right wheel, so while the mBot is meant to move straight forward, it would veer off its path and make our recordings tougher to execute. Lastly, when avoiding obstacles to reach its goal state, the mBot would make appropriate moves by avoiding the obstacle but would be 1 tile out of place (in any direction) when searching for the goal tile. The reason for this is because the mBot skips a step, which is backtracking (in some cases) to take the appropriate path to the goal tile.

10.4 Bugs

The biggest and arguably the main bug we ran into during our program's execution, is the fact that since most of the dynamic memory was occupied, which could lead to errors and such stated by the Arduino IDE, also the robot's IO board gets confused at times and the mBot's hardware "malfunctions" sometimes. There were occasions where we ran the program and the mBot would take different moves that was not implemented according to our algorithm. We did check and debug our algorithm just to play it safe, and where certain moves were supposed to take place, sometimes different moves would occur, which is not something we could fix. If we had more dynamic memory to work with, then it is possible that this would help in improving our mBot's movement.

11. Desired Improvements & Future Work

11.1 Potential Improvements

Given more time and pondering, we believe that we could fix the issues in our algorithm concerning the open list and closed list, and implement it so that no matter what the situation, the closed list will always retrieve the lowest F value near the current mBot's position on the path, without looking or considering other tiles earlier in the search. We could also fine tune the movements of the mBot so that it runs more smoothly, but as it is now, we believe that it is definitely good enough, but maybe not optimal. Another potential improvement we could make, is to use using all the sensors of the mBot; given more time, we could potentially "perfect" our algorithm and the mBot's movements, despite the fact that the right wheel is stronger than the left.

11.2 Future Work

In the future, some of us would like to work with more robot motion planning, in order to impact the industry and make a change to the AI and computer science world. The first step to take is to fine tune our program and algorithm to make it optimal, and from there experiment with more path finding and search algorithms, along with incorporating robotics. In particular, we want to implement the line sensor into the program and use a drawn 10X10 grid to make sure that the robot does not deviate from the given path. Another idea is to implement the robot's Ultrasonic sensor into program so that the robot can more accurately calculate the distance from the goal.

12. Conclusion & Final Thoughts

This project was a great learning experience for all of us, it gave us a little more insight on the A* algorithm and how it is applied, and gave us a lot more insight in robotics, though the mBot is just the surface when it comes to working with and implementing robotics. It also gave us insight on the limitations of our robot, and helped us to contemplate other ways to further improve the A* algorithm. With the growth of robotics and programming, it was important learning about this area in computer science and the experience helped with giving us something different from working on this project.

13. Appendix (How it works)

Our program for the robot was written in Arduino C/C++ and so it uses the Arduino IDE to move the robot. When the program has been compiled and uploaded, the output is displayed in the COM5 serial port where it will display the grid, which displays the robot's initial position. The user will then be prompted to enter a coordinate in the grid for the robot to traverse to. Once that coordinate has been entered, the program prints the grid multiple times, displaying the grid for each coordinate that the robot will travel to and will also show how many times it goes left,right, up, or down. Once the program has completely uploaded, the robot will begin its movements.

14. Acknowledgements (if any)

We would like to express our deepest gratitude and great appreciation to Dr. Malek Mouhoub and the UofR CS Department for funding this robot for us, and helping us with this process.

15. Video of Program Execution

https://youtu.be/adDFbS26GVg
https://youtu.be/Xjy-V-PrWp8

16. References

https://www.geeksforgeeks.org/a-search-algorithm/

https://www.arduino.cc/reference/en/

http://www.mblock.cc/software-1/mblock/mblock3/

Libraries included in Code