**Algorithm**

For the MDP challenge task, two algorithms are required: an efficient exploration strategy, and an optimal path finding algorithm. Then, we need to have a simulator to test the algorithms and adapt them appropriately to real-run situation.

During our learning process, we have read through and understood some well-known algorithms to appreciate their problem-solving ideas. However, due to constraints and assumptions regarding to this project, we have decided that our algorithms should be uniquely created on the basis of existing ones.

*(how about start with paragraph instead?)*

*The PC computation consists of exploration and fastest path finding. There are many ways to explore an unknown map, some of which completes the exploration in a short time. However, the exploration is expected to explore as many area as possible to provide optimal solution in the fastest path finding process with limited but yet sufficient exploration time.*

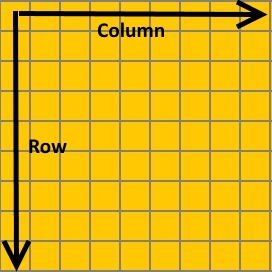
*With the constraints of robot movement, it takes longer to make a turn compare to straight line movement. Factoring in this constraints, the fastest path finding produces path with least turns to allow the robot complete in less duration.*

*Simulator is developed to test the correctness of the program that separates from physical robot. It allows the development to conduct in parallel to Arduino and Rpi and to test with unit testing in the process.*

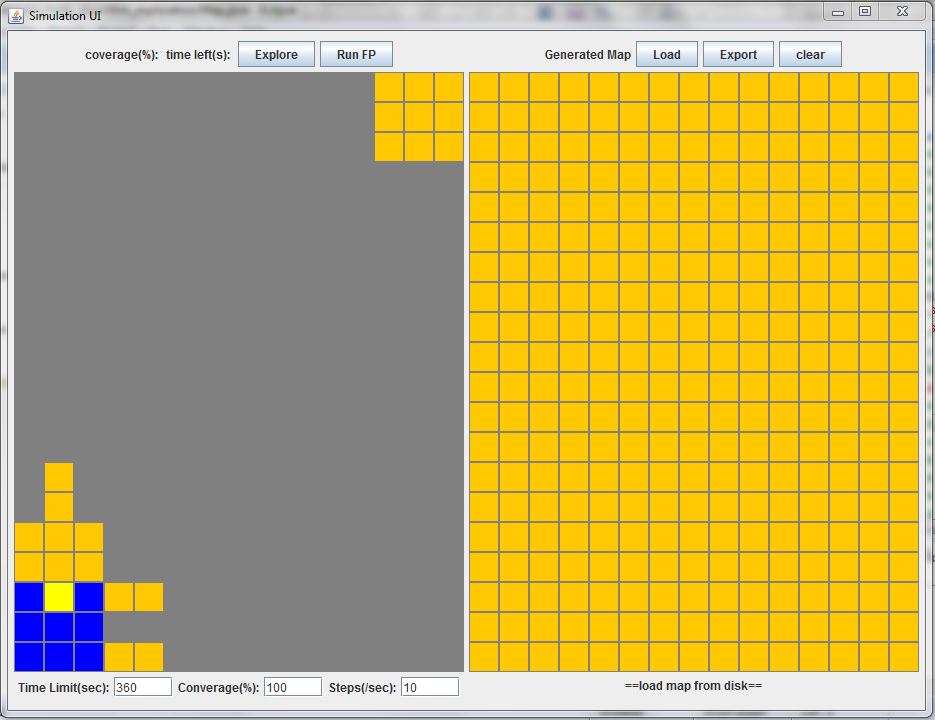
**Exploration**

Map representation

The map is defined by a 2-D array, as a matrix representation of the arena. Each cell corresponds to a 10cm x 10cm square and has a status variable to indicate whether the cell is unexplored, an obstacle or a free space. ~~Hence, the size of the 2-D array will be 20 rows x 15 columns.~~ The map representation as shown in the diagram below and the coordinates of each cell is represented by (a,b) where a represents row and b represents column.



~~To take into consideration of blind spots and accuracy of the sensors, the robot is set to occupy 3 x 3 square instead of 2 x 2.~~The robot is buffered with 5cm safe distance from the possible obstacles with 3 by 3 representation so that if the robot moves more than required, it won’t hit the obstacle directly. Robot calibration is done periodically to align robot to the correct position, ~~The robot will calibrate itself with reference~~ where reference is made to the surrounding obstacle/wall (left, right and front) by the installed ~~two short distance~~ sensors ~~installed at the front, which applies to the right calibration as well.~~ The current position of robot is represented in blue and the yellow cell indicates the orientation of the robot. In the simulator, the robot exploration is displayed in the left panel with the reference map on the right as shown below.  ~~Map will keep track of the robot’s current position and orientation and display it on the left side of the simulator. On the right side, reference map can be created and exported.~~



~~The content of each map array cell will represent its current status. As mentioned earlier, each cell has a status variable to indicate whether it is unexplored, an obstacle or a free space.~~

~~If the content is 0, the cell is unknown. The whole map array will be initialised as 0, to represent nothing has been explored. To distinguish explored areas, 1 is used for obstacles, 2 for empty squares.~~

Sensor representation(I swapped the paragraph order)

A total of 6 sensors are used by the robot as depicted in the diagram below, ‘S’ represents short-range sensors and ‘L’ represents long-range sensors. The short-range sensor is able to detect up to a distance of 2 cells while the long-range sensor detects up to a distance of 4 cells. However, there is a minimum distance that the long distance sensor can sense. If the obstacle is put below the minimum sensing range, the sensor reading is not stable. The two short distance sensors are placed at the right side in order to calibrate the robot in the wall follower algorithm. The long distance sensor of the left side is to explore as many cells as possible.

|  |  |  |
| --- | --- | --- |
| **S**  **L** | **L** | **S**  **S** |
|  |  |  |
|  |  | **S** |

Arduino communicates with PC by sending the 6 sensor readings in this format, a:b:c:d:e:f, ‘a’ corresponds to the long range sensor on the left and the rest of the sensors are read in a clockwise manner respectively. The readings are sent in integer format, for the long-range sensors, the readings ranges from 0-6, whereas for the short-range sensors, the readings range from 0-2.

The sensing range is represented by a form of offset with taking into consideration of current position and orientation of robot. The center cell of the 3x3 robot representation is used to represent the robot offset to update the sensing range dynamically. The offset works in this way, it records the difference in the sensors’ coordinates to the center cell. For example, if the center cell is (0,0) and the middle long-range sensor detects an obstacle in (-5,0), then the algorithm will sub the center cell value with its current position and add (-5,0) to it to get the value to be sent to PC.

~~The orientation of the robot is recorded and taken into consideration as well. When the robot is facing another direction, all that needs to be done is to change the offset.~~

The table below gives an illustration of a few instances to show how the algorithm works with the sensor readings received from Arduino.

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading** | **Interpretation** | **Action** | **Illustration** |
| 0:0:0:0:0:0 | Areas up to the sensor’s range limit are empty. | Mark squares as empty, starting from that sensor position, up to its range limit. |  |
| 4:0:3:0:1:2 | The 1st sensor detects an obstacle 4 cells away; 2nd and 4th sensor detects no obstacles; 3rd sensor detects an obstacle 3 cells away, 5th sensor detects an obstacle in the first cell and 6th sensor detects an obstacle in the second cell. | Mark cells that are free space as yellow; Mark cells with obstacle as black; The rest of the cells within the sensor range remains grey if there is an obstacle. |  |
| 1:1:1:1:1:1 | All sensors detect an obstacle in the first cell. | Mark obstacles detected as black cells. |  |

**~~Exploration~~**

There are two main strategies employed for exploration; namely *Dijsktra’s Algorithm* and *StickToTheWall*.

**Dijkstra’s Algorithm**

Dijsktra’s algorithm assumes the unexplored region is walkable and computes the path to goal initially. When there is obstacle blocks the robot’s way to goal, the path is re-computed with current state of the map. This process is repeated until the robot goes to goal.

The cost is defined as the distance to the start cell. The cost of the cell in the map is marked as infinity initially. At the each iteration, neighbor cell cost is updated with reference to the cost of current cell and is pushed to a sorted list with least cost cell as the next cell to visit, after which marks the current cell as visited and visits next cell. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

However, one thing to note is that this algorithm makes no attempt to direct “exploration” towards the goal as one might expect. Rather, the sole consideration in determining the next “current” cell is its distance from the starting point. This algorithm, therefore “expands outward” from the starting point iteractively until it reaches the goal.

In the Dijkstra’s Algorithm, the robot will move as follows:

Bottom left corner 🡪 Top right corner 🡪 Top left corner 🡪 Check for unvisited nodes and go to unvisited nodes 🡪 Then go back to start

Advantage:

* Adaptive to the ever-changing map. The path is re-computed if there is obstacle blocks robot’s way to goal.
* Flexibility to fit purpose of the application. By controlling the priority of next-neighbour-to-visit, the path is computed with preferred criterion.

Disadvantage:

* Consumes computation power since the path is computed more than one time.
* Requires to pre-compute the path before hand
* Difficult to understand compared to StickToTheWall

**StickToTheWall**

*StickToTheWall*, also known as either the left-hand or right-hand rule~~, is one of the best-known rules for traversing mazes. There are many ways to explore the map, however,~~ *~~StickToTheWall~~* is one of the algorithm that allows the robot to calibrate as many as it can to compensate the inaccuracy of the robot motion. ~~The robot is guaranteed to explore the region near the wall.~~

The robot will carry out the *StickToTheWall* algorithm in an anti-clockwise direction, using the following priority rule, right, front then left. The robot will move one round around the walls and go back to the start position as required.

~~Using the~~ *~~StickToTheWall~~* ~~algorithm has its advantages and disadvantages.~~

**Advantage**:

* Simplified the implementation process with intuitive approach
* Provide more opportunities to calibrate the robot.

**Disadvantage:**

* Prone to getting stuck in a loop. (For instance, should the right obstacle disappear, the robot will be stuck in a loop due to the priority rule.)
* Have less chances to explore the middle region

Solution (-> as one section)

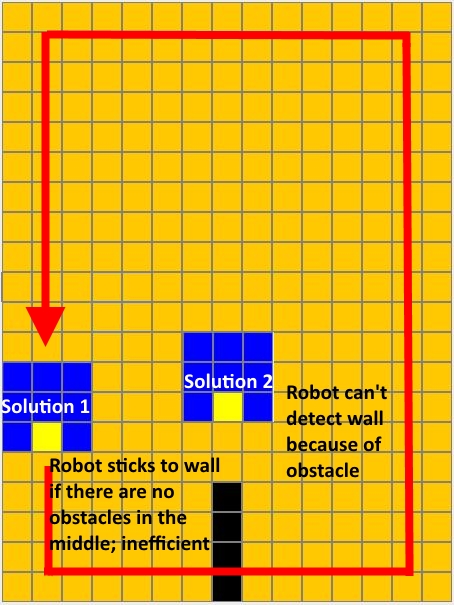
To tackle the problems, the group came up with 3 different solutions as follows:

Solution 1: The robot motion and sensor reading are assumed to work in best condition initially. ~~In the simulator, it is assumed that the robot and its sensors are working in the best condition and that we are required to explore 100% of the map fully.~~ With goal to achieve 100% exploration, we choose Dijkstra’s for simulation.

**Solution 2:** However, the robot is not guarantee to move straight. Hence, stickToTheWall is selected to increase the chances to align the robot. And the sensor reading keeps producing fault results due to insufficient turning or straight movement. When right obstacle of robot disappear, the robot get itself to run in a cycle.

**Solution 3:** When the right sensor detects the disappearance of the obstacle, *findRightWall* algorithm is triggered. However, this eats up exploration time as shown in Solution1 in diagram.

**Solution 4:** Find the nearest wall by computing the distance of the nearest wall. However, the robot can stuck in bigger loop when it detects the condition at the same position. Besides, if robot can not enter region A due to obstacle, the robot will not be able to find the wall as seen Solution2 in diagram.

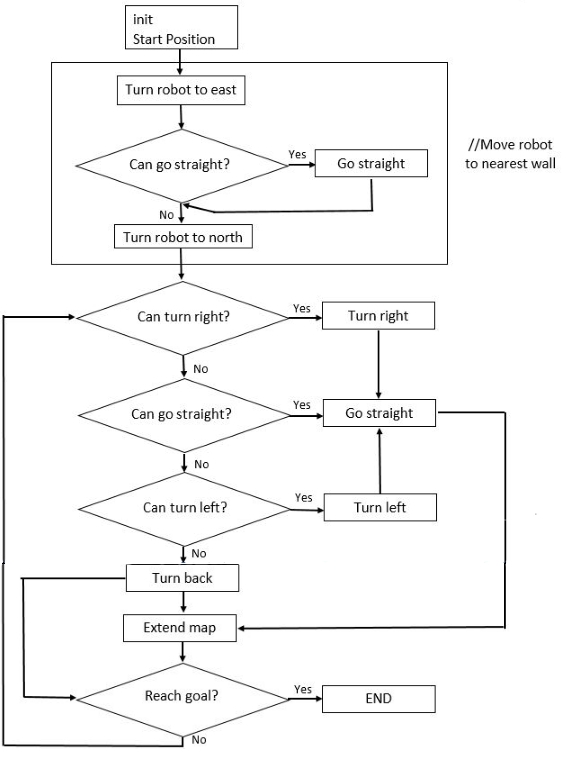


Region A

**Solution 5:** Combine *Dijkstra’s algorithm* with *StickToTheWall. When obstacle at the right of robot disappears, Dijkstra’s is triggered to go to current goal, from which the StickToTheWall is executed. This solution prevents the robot to fall in infinite loop and yet allow the robot to reach goal.*

Hence, the group decided on solution 3, a hybrid of *Dijkstra’s algorithm* and *StickToTheWall.*

~~When doing exploration in the real run, we choose to use a hybrid of both algorithms. This is because the 2 algorithms complement each other and the arena resembles a maze~~.



**Shortest/Fastest Path Algorithm**

~~For a given cell in the map, the algorithm finds the path with the lowest cost between that cell and every other cell. We modified the algorithm to keep track of the number of turns and the shortest/fastest path is determined by the path that uses the least amount of turns.~~

The fastest path is defined as the path with least amount of turns. The robot takes more time on turning compare to straight line movement. When the previous visited cell, current cell and neighbor cell form a turn, more cost will be added to visit that cell. The neighbor cells are pushed to the list with sorting criterion of the cost. And the cell with least cost will be visited next and the cost of neighbor cells is updated according. The process repeats until the goal is visited.

Before running the fastest path, the robot is at the start position. The fastest path will traverse the neighbor of the current position and calculate the movement cost respectively. The movement cost updates the cost of the neighbor cell with higher value if the neighbor, current position and previous position form a corner as shown in the following figure, which means the robot will make a turn if it goes through these three points.

<shift the content here >

By controlling the cost function, the fastest path is able to generate the fastest path. After obtaining the map in exploration, the robot will proceed with fastest path based on the map and calibration will be added to ensure that the robot moves in a straight line.

**Conclusion**

It always wise to weigh the pros and cons of the algorithm to select the most suitable one to fit the purpose of the application. Algorithm is the implementation from the software side. However, when integrating with the robot, there are many more factors to consider to overcome the hardware limitation to improve the degree of intelligence of robot.

~~At the end of the whole process, we learnt that coding for the algorithm is not as simple as it appears. The reason behind this is the need to integrate the code with hardware components. As an actual robot is involved, more considerations have to be taken in when writing the code. For instance, blind spots, limitations and accuracy of the hardware sensors.~~

init

Start position;

Go to goal

Go to bottom right reach time/coverage limit?

Go to top left

Visit cells

Yes

Yes

Is there unvisited cell/square?

No

Go to start

Go straight

Can go straight?

Turn robot to east

init

Start Position

Yes

//Move robot to nearest wall

No

Turn robot to north

Can turn right?

Yes

Turn right

No

Can go straight?

Yes

Go straight

No

Yes

Can turn left?

Turn left

No

Turn back

Extend map

Yes

Reach goal?

END

No