
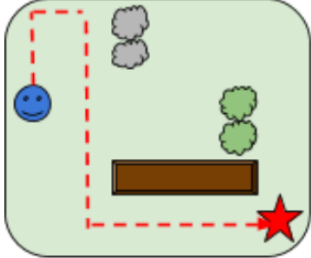


Documentation for Path planner libraries:

October 15, 2018

1 Introduction:

The path planner libraries implemented in this project handle the task of pathplanning for a 2D holonomic robot confined in a 2D rectangular grid space with square grids marked as obstacles. Two different approaches were used here and their performances, space and time complexity are analysed.

<p>Environment</p>  <p>Legend:</p> <ul style="list-style-type: none">Rock (obstacle)Tree (obstacle)Rock (obstacle)RobotGoal	<p>Corresponding world_state, robot_pose and goal_pose</p> <pre>world_state = [[0, 0, 1, 0, 0, 0], [0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0], [0, 0, 1, 1, 1, 0], [0, 0, 0, 0, 0, 0]]</pre> <pre>robot_pose = (2, 0) goal_pose = (5, 5)</pre>
	<p>Example of a valid path shown in red on the left figure</p> <pre>path = [(2, 0), (1, 0), (0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)]</pre>

The two planners implemented are:

- Random Planner: The random planner tries to find a path to the goal by randomly moving in the environment. If the planner cannot find an acceptable solution in less than *max_step_number*, the search fails. The random planner, while being erratic, has a short memory, and it tries to avoid visiting a cell that was visited in the last *sqrt(max_step_number)* steps except if it is the only available option.
- Optimal planner: For an optimal planner approach, A* path planning is used.

The two approaches will be detailed in section 3.

2 Planner classes:

2.1 Random planner:

```
class Random_planner{
private:
....
public:
Random_planner(int n,bool verbose_flag=false) : max_step_number(n),verbose(verbose_flag);
~Random_planner();
vector<xy> search(matrix<char> world_state, xy robot_pose, xy goal_pose);
...
};
```

The member function `Random_planner::search()` inputs the world map, robot pose and goal pose and outputs the path from starting position to goal

2.2 A* planner:

```
class Random_planner{
private:
....
public:
Astar_planner(bool verbose_flag=false) : verbose(verbose_flag);
~Astar_planner();
vector<xy> search(matrix<char> world_state, xy robot_pose, xy goal_pose);
...
};
```

The member function `Astar_planner::search()` inputs the world map, robot pose and goal pose and outputs the path from starting position to goal

The map coordinates are represented by a simple structure `xy`:

```
struct xy { int x; int y; };
```

Refer to `readme.txt` for file layout, testing and modifications.

2.3 Parameters:

2.3.1 In Constructor for Random_planner:

```
Random_planner(int n,bool verbose_flag=false) : max_step_number(n),verbose(verbose_flag);
```

- `max_step_number`: Maximum steps to take before ending the path search.
- `verbose`: Prints detailed information as the algorithm runs. Default is set to false

2.3.2 In Constuctor for Astar_planner

```
Astar_planner(bool verbose_flag=false) : verbose(verbose_flag);
```

- `verbose`: Prints detailed information as the algorithm runs. Default is set to false

2.3.3 In search() function in Random_planner and Astar_planner:

```
vector<xy> search(matrix<char> world_state, xy robot_pose, xy goal_pose);
```

- world_state - char boost matrix that represents the map, where '0' represents movable paths and '1' represents obstacles
- robot_pose: Inputs the initial position of the robot
- goal_pose: Inputs the final destination to be reached

2.3.4 Miscellaneous:

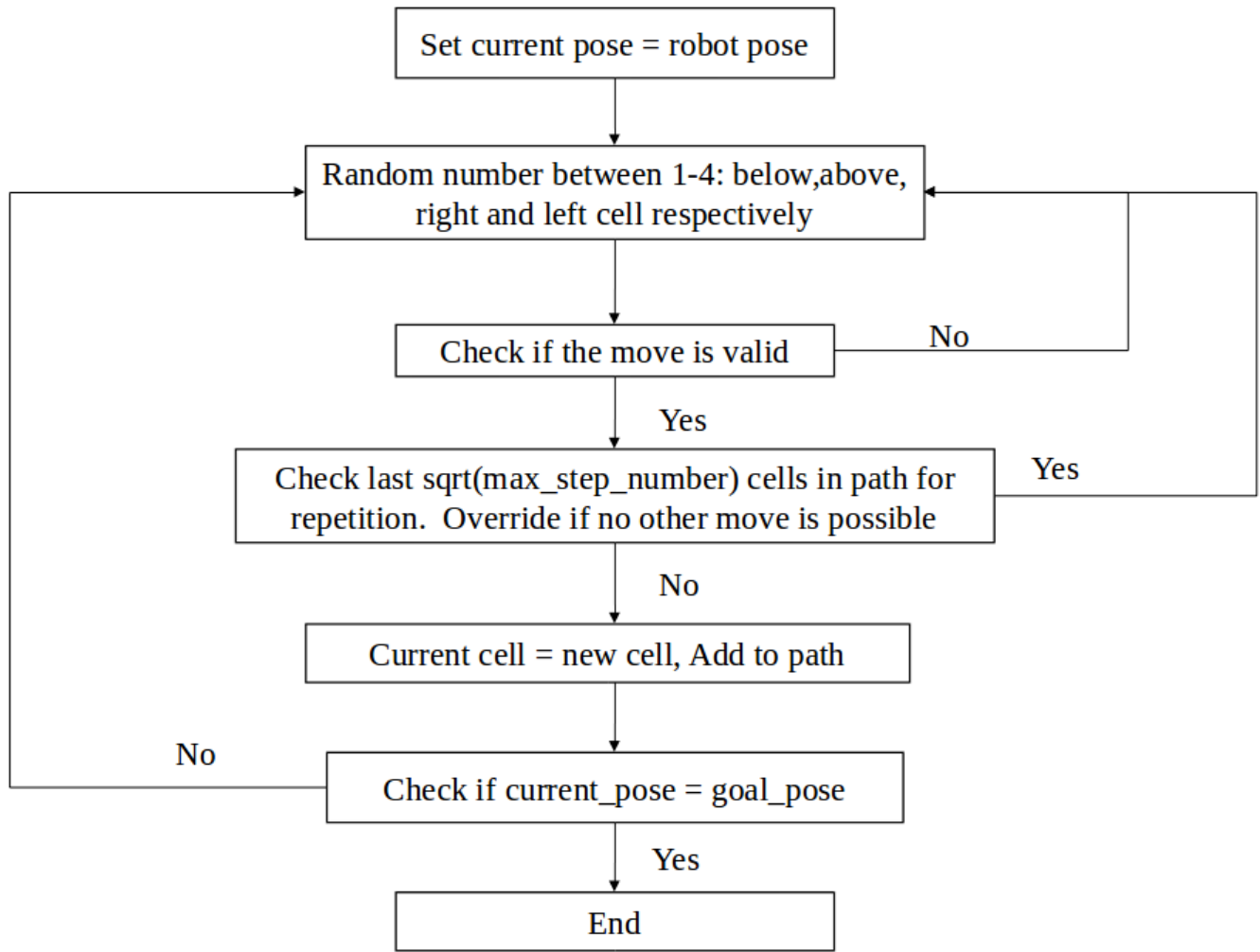
- 'Cell' refers to location marked by coordinates (x,y)
- 'world_state','grid' and 'map' refer to the matrix representation of the world in which the robot moves
- 'start' and 'robot_pose' refer to the starting location of the robot in the grid
- 'goal' and 'goal_pose' refer to the final destination of the robot

3 Algorithm:

3.1 Random Planner:

The random planner tries to find a path to the goal by randomly moving in the environment. To avoid moving randomly between the same set of cell, it uses a short memory to skip recently visited cells.

- Start from the robot_pose and move randomly to the next neighbour cell.
- Random a number between 1-4. Each number corresponds to an intent to move to the cell below, above, to the right and to the left respectively.
- If the move is not valid, check out the next random neighbour cell
- Also, check if the random cell chosen was visited in the last sqrt(max_step_number) moves. If yes, ignore the cell and checkout the next random cell.
- Override the above check, if there are no other options.
- End the search and return path, if goal pose is reached.



3.2 Astar Planner:

To implement the optimal planner, the A* search algorithm is used. Compared to other popular algorithms,

- Dijkstra's algorithm is an exhaustive expansion across the map to find the goal. It makes too many unwanted calculations and needs N^2 space on average.
- Ant colony optimization and Genetic algorithms are better suited for continuous graphs, where optimization of the shortest path is more effective.
- D* is better suited for dynamic maps
- A* is the best option to solve 2D static grid maps accurately and effectively.

A* is an informed search algorithm, where the map exploration is directed towards the goal with an objective function $f()$. This is achieved by maintaining a tree of paths to explore and choosing the best option based on the cost function.

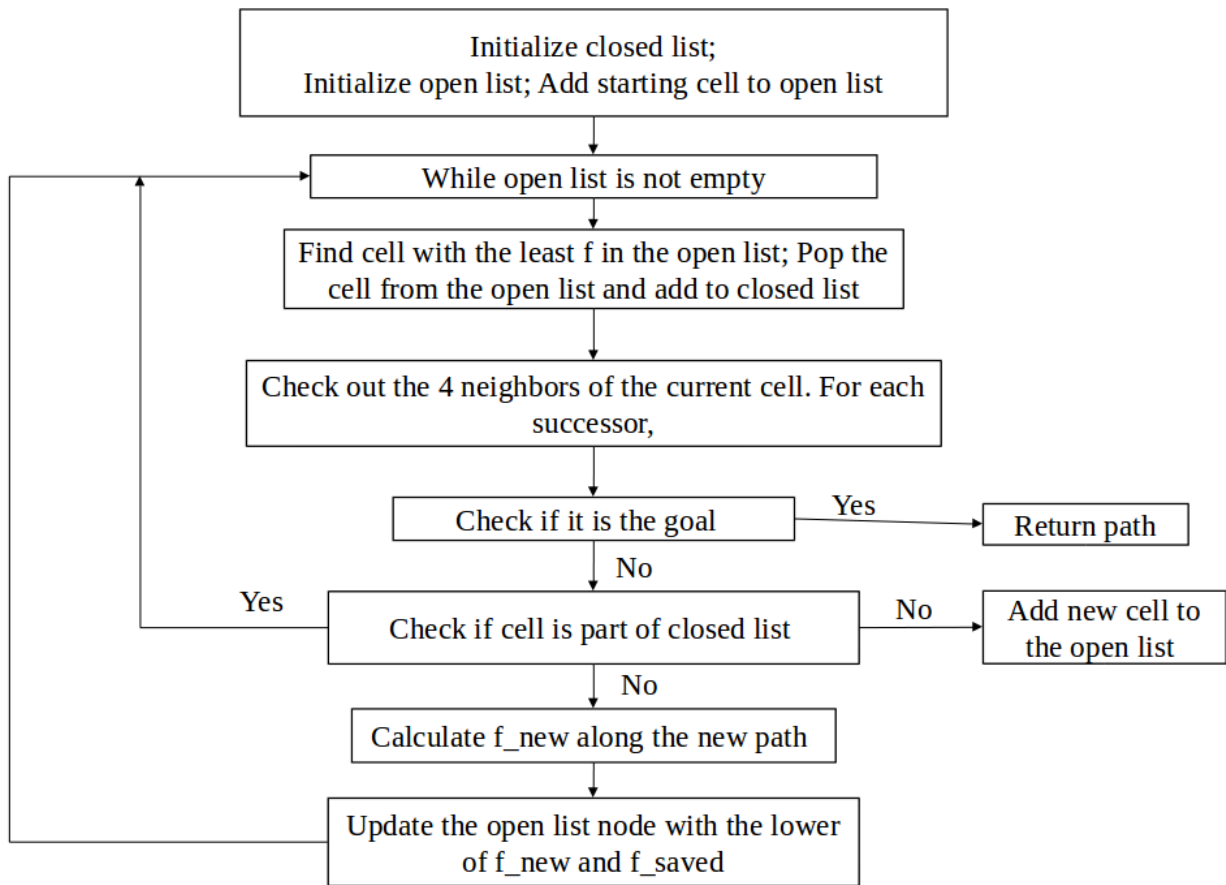
$$f(\text{cell}) = g(\text{cell}) + h(\text{cell})$$

Where $h(\text{cell})$ is the heuristic function that estimates the cost of the smallest distance from cell to goal. $g(\text{cell})$ is the distance of cell from the starting location. The algorithm runs till the goal node is reached or if there are no nodes to explore further.

- $g(\text{cell})$ is updated for each cell as an increment over the $g(\text{parent_cell})$
- $h(\text{cell})$ is the manhattan distance from cell to the goal pose

Summary of the algorithm:

- Maintain two lists of cells:- open list and closed list
- Explore the cell with the lowest cost function value from the open list
- Add the neighbours of each explored cell to the open list
- Update the cost function after each step
- Add explored cells to the closed list
- Keep exploring till the goal is reached



4 Code breakdown:

4.1 Random_planner:

The class is declared in the header file “*Random_planner.h*” and the member functions are defined in “*Random_planner.cpp*”. Matrices and Vectors are handled by boost libraries. The class members are described below:

- Private member `check_wall()` to detect if the cell is out of bounds or if it is occupied by an obstacle

```
bool check_wall(boost::numeric::ublas::matrix<int> map,int x,int y);
```

- Private member function `check_repeat()` to check if the cell was visited in the last 'steps' number of moves

```
bool check_repeat(int steps,int x,int y );
```

- Constructor `Random_planner()` that initializes `max_step_number` and verbose flag

```
Random_planner(int n,bool verbose_flag=false) : max_step_number(n),verbose(verbose_flag)
```

- Public member `search()` that implements the path planning algorithm

```
boost::numeric::ublas::vector<xy> search(boost::numeric::ublas::matrix<char> world_state)
```

- Public member `printpath()` that prints the `world_state` along with the path returned by `search()`

```
void printpath(boost::numeric::ublas::matrix<char> world_state);
```

- Public member `printworld()` that prints the `world_state`

```
void printworld(boost::numeric::ublas::matrix<char> world_state,xy robot_pose, xy destination)
```

4.1.1 `Random_planner::search()`:

- Update `current_pose` to starting location

```
current_pose=robot_pose;
path->insert_element(0,current_pose);
```

- Random a number between 1-4 and enter the loop

```
srand (time(NULL));
rand_num=rand()%4+1;
```

```
while(steps<max_step_number && loop_count<5*max_step_number) {...}
```

- Within the loop switch between the 4 possible neighbour cells using the random number. For each case, Check if the move is valid(no wall or obstacle) and then check if the destination cell was visited in the last `repeat_threshold` steps. If either condition is failed, increase `surround_cells` count by 1 and check the next cell. Move to the cell if both conditions are satisfied. When `surround_cells` ≥ 4 , make any valid move, ignoring repeat cells

```

switch (rand_num) {
case 1:
if(!check_wall(world_state,current_pose.x+1,current_pose.y)) {
if(!check_repeat(steps,current_pose.x+1,current_pose.y) || surround_cells>=4) {
current_pose.x++;
steps++;
path->insert_element(steps,current_pose);
break;
}
else surround_cells++;
}
else surround_cells++;

exitloop = true;
if(verbose) cout<<"Move could not be made: Checking next cell"<<endl;
rand_num++;
break;
}

```

- After each move,check if goal is reached

```

if(current_pose.x==goal_pose.x && current_pose.y==goal_pose.y) {
path->resize(steps+1);
return (*path);
}

```

4.2 Astar_planner:

The class is declared in the header file “*Astar_planner.h*” and the member functions are defined in “*Astar_planner.cpp*”. Matrices and Vectors are handled by boost libraries. The lists are implemented with `std::maps` and `std::priority_queue`. The class members are described below:

- Internal variables to store map properties

```

bool verbose;
int row; int col;
int start_idx; int goal_idx;

```

- Internal structure `cell{}` to store cost function values and index of parent cell for a cell

```

struct cell {
int previous;
int f;
int h;
int g; };

```

- `std::map` `cellinfo` to store struct `cell{}` for explored cells with key as cell index

```

std::map<int,cell> cellinfo;

```

- closedlist boolean vector to mark if a cell has been added to the closedlist

```
boost::numeric::ublas::vector<bool> *closedlist;
```

- std::priority_queue openlist to store pairs of f(cell),cell_index, sorted in ascending order of f(cell)

```
std::priority_queue<std::pair<int,int>,std::vector<std::pair<int,int> >,std::greater<
```

- Private member function Hvalue() to calculate h(cell)

```
int Hvalue(xy a);
```

- Private member function xy2idx() to convert from struct xy{} to 1D index

```
int xy2idx(xy XY);
```

- Private member function idx2xy() to convert from 1D index to struct xy{}

```
xy idx2xy(int idx);
```

- Private member function check_wall() to detect if the move from idx1 to idx2 is out of bounds or if the cell idx2 is occupied by an obstacle

```
bool check_wall(boost::numeric::ublas::matrix<char> map,int idx1,int idx2);
```

- Private member function tracepath() to return the path from start to goal, once goal has been reached

```
boost::numeric::ublas::vector<xy> tracepath(std::map<int,cell> cellinfo,boost::num
```

- Constructor Astar_planner() to initialize verbose flag

```
Astar_planner(bool verbose_flag=false) : verbose(verbose_flag) {}
```

- Public member printpath() that prints the world_state along with the path returned by search()

```
void printpath(boost::numeric::ublas::matrix<char> world_state);
```

- Public member printworld() that prints the world_state

```
void printworld(boost::numeric::ublas::matrix<char> world_state,xy robot_pose, xy
```


4.2.1 Astar_planner::search():

- Initialize map variables and closedlist vector based on the size of the map

```
row=world_state.size1();
col=world_state.size2();
goal_idx=xy2idx(goal_pose);
start_idx=xy2idx(robot_pose);

closedlist = new vector<bool> (row*col,false);
```

- Insert starting cell details into std::map cellinfo and push starting cell into openlist with f(cell)=0

```
h_current=Hvalue(idx2xy(start_idx));
temp=(cell){start_idx,h_current,h_current,0};
cellinfo.insert(pair<int,cell>(start_idx,temp));

openlist.push(make_pair(0,start_idx));
```

- Enter while loop and run it as long as openlist is not empty.

```
while(!openlist.empty()) {...}
```

- Pop the top pair from openlist. Add this cell to closedlist.

```
pair<int,int> current = openlist.top();
openlist.pop();
idx = current.second;
(*closedlist)(idx)=true;
```

- For each possible move from the current cell, check if the move is not blocked. Then, if the next cell is the goal, update cellinfo for the cell and return traced path back to start.

```
if(idx+1==goal_idx) {
temp=(cell){idx,g_current+1,0,g_current+1};
cellinfo.insert(pair<int,cell>(idx+1,temp));
reached_goal=true;
if(verbose) cout<<"Goal reached in "<<g_current+1<<" steps"<<endl;
return tracepath(cellinfo,world_state);
}
```

- Else, check if the new cell belongs to the closedlist. If yes, check out the next cell. If not, calculate their g(cell) and f(cell) along this path.

```

else if ((*closedlist)(idx+1)==false) {
    g_current = cellinfo.find(idx)->second.g;
    h_current = Hvalue(idx2xy(idx+1));
    f_current = g_current+1+h_current;
    ...
}

```

- If the cell isn't added to cellinfo map already, Push this f(cell) value into openlist and add the cell details to cellinfo

```

cell_it=cellinfo.find(idx+1);
if (cell_it==cellinfo.end()) {
    openlist.push(make_pair(f_current,idx+1));
    temp=(cell){idx,f_current,h_current,g_current+1};
    cellinfo.insert(pair<int,cell>(idx+1,temp));
}

```

- If the cell is already in cellinfo, checkout the stored value of f(cell) and update it with the lower of the current value and stored value. Push this f(cell) value into openlist

```

else if (cell_it->second.f > f_current) {
    openlist.push(make_pair(f_current,idx+1));
    cell_it->second.previous=idx;
    cell_it->second.f=f_current;
    cell_it->second.g=g_current+1;
}

```

- Repeat the same process of each neighbouring cell in each loop.

5 Testing the search() functions:

In order to test the two classes, two additional programs have been attached. *test_Random_planner.cpp* and *test_Astar_planner.cpp*. Both the test programs have a very similar code.

If *bool random_map* is set to true, the program generates a random map for the search to be implemented on. The row and column size for the map are set to 20x20 by default. The variable '*ratio*' adjusts the ratio of open spaces to obstacles generated. Robot starting pose is set to (1,1) and goal pose is set to (18,18). Verbose flag is set to False.

Else, The map matrix, robot pose and goal pose can be manually edited in the files.

An instance of the respective classes are created and the search function is called

```

Random_planner obj(max_steps,verbose);
path = obj.search(map,start,end);
//or
Astar_planner obj(verbose);
path = obj.search(map,start,end);

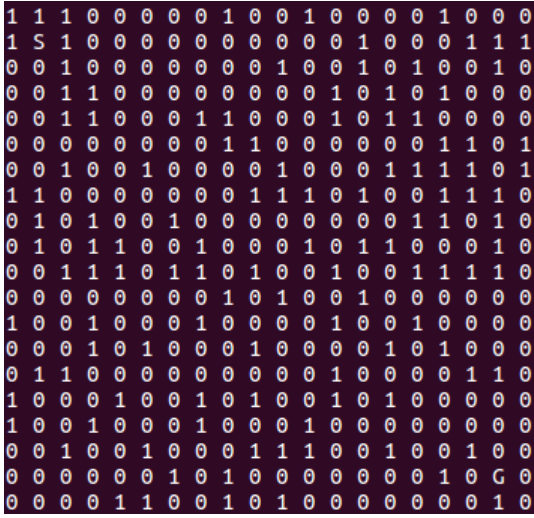
```

5.1 Output paths:

The following paths were obtained from a sample run of the code with the given parameters and a random map generated was used for both cases.

```
row = 20;
column =20;
start = {1,1};
end = {18,18};
verbose = false;
max_step_number = 300;
```

Input map:



5.1.1 Random planner:

The path was found in 246 steps:

```
(1,1)>(2,1)>(3,1)>(4,1)>(5,1)>(5,0)>(4,0)>(3,0)
>(2,0)>(2,1)>(3,1)>(3,0)>(2,0)>(3,0)>(4,0)>(3,0)
>(2,0)>(3,0)>(4,0)>(3,0)>(4,0)>(3,0)>(4,0)>(5,0)
>(6,0)>(6,1)>(5,1)>(5,2)>(5,3)>(6,3)>(6,4)>(5,4)
>(4,4)>(4,5)>(3,5)>(2,5)>(1,5)>(1,6)>(2,6)>(3,6)
>(4,6)>(5,6)>(5,5)>(4,5)>(5,5)>(5,4)>(5,5)>(4,5)
>(4,6)>(4,5)>(4,6)>(3,6)>(3,7)>(3,8)>(2,8)>(1,8)
>(1,9)>(2,9)>(3,9)>(3,10)>(4,10)>(4,11)>(5,11)
>(5,10)>(4,10)>(4,9)>(3,9)>(2,9)>(2,8)>(2,7)
>(1,7)>(1,6)>(1,5)>(0,5)>(0,6)>(0,7)>(1,7)
>(1,8)>(1,9)>(1,10)>(0,10)>(0,9)>(1,9)>(0,9)
>(1,9)>(2,9)>(1,9)>(0,9)>(0,10)>(1,10)>(1,11)
>(1,12)>(2,12)>(2,11)>(3,11)>(3,10)>(4,10)
>(5,10)>(5,11)>(5,12)>(6,12)>(6,13)>(5,13)
>(5,14)>(5,15)>(5,14)>(5,15)>(5,14)>(5,13)
>(4,13)>(3,13)>(4,13)>(5,13)>(5,12)>(5,13)
>(5,12)>(5,11)>(6,11)>(5,11)>(4,11)>(4,10)
```

```

>(5,10)>(4,10)>(3,10)>(3,11)>(2,11)>(1,11)
>(1,12)>(2,12)>(2,11)>(2,12)>(1,12)>(0,12)
>(0,13)>(0,14)>(0,15)>(1,15)>(1,16)>(2,16)
>(2,17)>(3,17)>(4,17)>(4,18)>(5,18)>(6,18)
>(5,18)>(6,18)>(5,18)>(4,18)>(3,18)>(3,19)
>(4,19)>(3,19)>(2,19)>(3,19)>(4,19)>(4,18)
>(3,18)>(4,18)>(5,18)>(6,18)>(5,18)>(4,18)
>(4,17)>(4,16)>(4,17)>(3,17)>(2,17)>(2,16)
>(1,16)>(1,15)>(0,15)>(0,14)>(0,13)>(0,12)
>(1,12)>(1,11)>(2,11)>(3,11)>(3,10)>(4,10)
>(4,9)>(3,9)>(2,9)>(3,8)>(3,7)>(3,6)>(3,5)
>(4,5)>(4,6)>(5,6)>(5,7)>(6,7)>(6,6)>(7,6)
>(7,7)>(7,8)>(8,8)>(8,9)>(9,9)>(9,10)>(10,10)
>(10,11)>(11,11)>(12,11)>(13,11)>(13,12)
>(13,13)>(14,13)>(15,13)>(16,13)>(16,12)
>(17,12)>(18,12)>(18,13)>(18,14)>(19,14)
>(19,15)>(18,15)>(17,15)>(16,15)>(16,14)
>(16,13)>(17,13)>(17,12)>(18,12)>(18,11)
>(18,10)>(18,9)>(19,9)>(18,9)>(19,9)>(18,9)
>(18,10)>(18,11)>(19,11)>(19,12)>(19,13)
>(18,13)>(18,14)>(18,15)>(19,15)>(19,16)
>(19,17)>(18,17)>(18,18)

```

Output path printed on world map: '*' represents cells visited by the path_planner atleast once.

```

1 1 1 0 0 * * * 1 * * 1 * * * * 1 0 0 0
1 5 1 0 0 * * * * * * * 1 0 * * 1 1 1
* * 1 0 0 * * * * * 1 * * 1 0 1 * * 1 *
* * 1 1 0 * * * * * * 1 * 1 0 1 * * *
* * 1 1 * * * 1 1 * * * 1 * 1 1 * * *
* * * * * * 1 1 * * * * * 1 1 * 1
* * 1 * * 1 * * 0 0 1 * * * 1 1 1 1 * 1
1 1 0 0 0 0 * * * 1 1 1 0 1 0 0 1 1 1 0
0 1 0 1 0 0 1 0 * * 0 0 0 0 0 1 1 0 1 0
0 1 0 1 1 0 0 1 0 * * 1 0 1 1 0 0 0 1 0
0 0 1 1 1 0 1 1 0 1 * * 1 0 0 1 1 1 1 0
0 0 0 0 0 0 0 0 1 0 1 * 0 1 0 0 0 0 0 0
1 0 0 1 0 0 0 1 0 0 0 * 1 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0 1 0 * * * 1 0 1 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 1 * 0 0 0 1 1 0
1 0 0 0 1 0 0 1 0 1 0 0 1 * 1 0 0 0 0 0
1 0 0 1 0 0 0 1 0 0 0 1 * * * 0 0 0 0
0 0 1 0 0 1 0 0 0 1 1 1 * * 1 * 0 1 0 0
0 0 0 0 0 0 1 0 1 * * * * * * 1 * G 0
0 0 0 0 1 1 0 0 1 * 1 * * * * * * 1 0

```

5.1.2 Astar planner:

Similarly, for the Astar_planner.

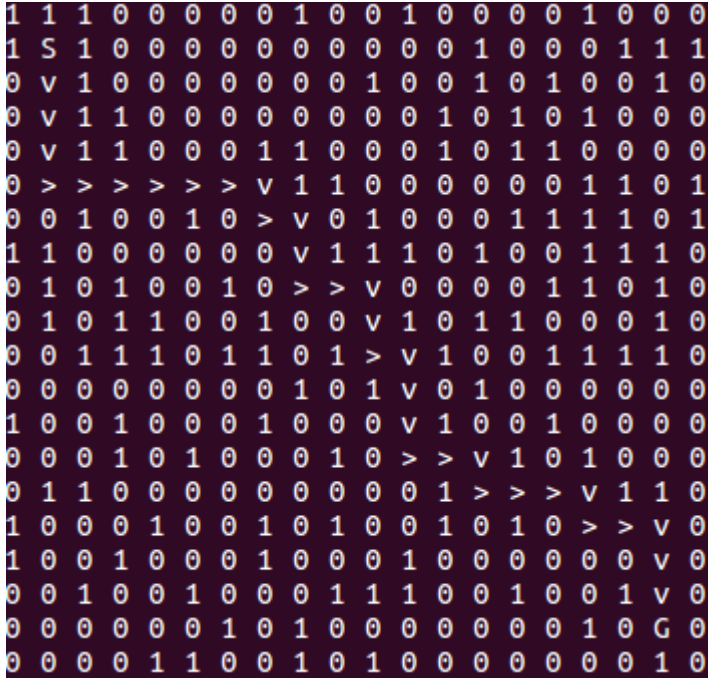
The goal was reached in 34 steps.

```

(1,1)>(2,1)>(3,1)>(4,1)>(5,1)>(5,2)>(5,3)>(5,4)
>(5,5)>(5,6)>(5,7)>(6,7)>(6,8)>(7,8)>(8,8)>(8,9)
>(8,10)>(9,10)>(10,10)>(10,11)>(11,11)>(12,11)
>(13,11)>(13,12)>(13,13)>(14,13)>(14,14)>(14,15)
>(14,16)>(15,16)>(15,17)>(15,18)>(16,18)>(17,18)
>(18,18)

```

Output path is printed on world map: Arrows mark the path from start to goal.



6 Performance:

6.1 Random planner:

Performance: The goal of the algorithm is to explore as many new cells as possible and hopefully end up at the goal. It is a simplistic design that is not reliable in anyway. The path can almost never be optimal. In the case described in section 5, a sample of the code took 246 steps to find the goal in a 20X20 map. The path length went up to 4000+ in some cases.

Time_complexity: Worst case scenario is infinite if the random function `rand()` does not give a true random output. Given, the random path decider is true random, the complexity will be $O(\text{row} \times \text{column})$

Space_complexity: The only non constant space is used up by the path vector. It uses up $\min(\text{max_step_number}, \text{row} \times \text{column})$ space.

6.2 Astar planner:

Performance: The A* algorithm is efficient at exploring the map by moving towards the goal using a cost function. The cost function and in particular, the heuristic function $f(\text{cell})$ has to be designed smartly from the algorithm to succeed. For this problem, having $f(\text{cell})$ represent the manhattan distance from cell to goal is sufficient. In the case described in section 5, A star found the optimal path with 34 steps towards goal.

Space_complexity: A* by design has exponential space complexity ($4^{\text{pathlength}}$), since for every node explored from the open list, 4 more potential nodes are added to it. But, in this particular problem of 2D map path planning, space is constrained, as it can never go beyond $\text{row} \times \text{column}$ cells. If the heuristic function is defined optimally, the space needed will only be proportional to the path size.

Time_complexity: Following the Space_complexity argument, the time complexity also become exponential in the extreme open world problem. In a 2D static map, the algorithm run time is only proportional to the path length. Worst case scenario is $O(\text{row} \times \text{column})$.

6.3 Conclusion:

Performance wise, A* is a much superior algorithm with absolute reliability and finds the path in an efficient manner. Random planner may or maynot find the goal within an acceptable number of steps. As such, the time complexity can be infinite in worst case scenario, whereas the A* algorithm only has $O(\text{row} \times \text{column})$ worst case complexity.

The only drawback of A* is it space complexity. It could potentially take up a lot of space to store the open list, closed list and cellinfo vectors.