Documentation for the model car driver:

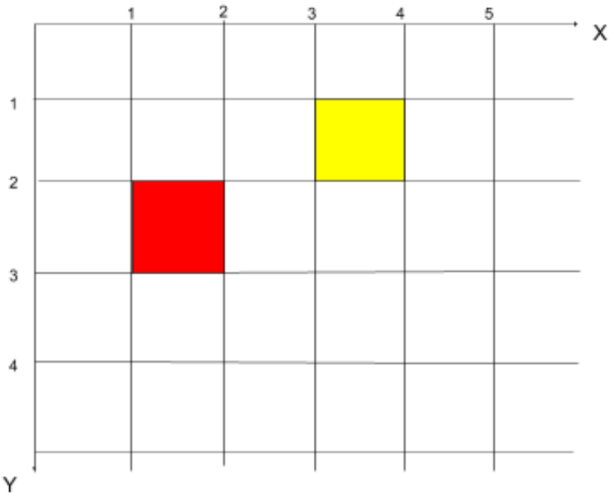
November 16, 2020

1 Introduction:

The path planner algorithm implemented in this project handles the task of pathplanning for a 2D holonomic robot confined in a 2D rectangular grid space

- The car traverses in discrete movements, with its positions limited to the corners of the 1 unit grid cells and its orientation limited to North, West, East and South
- The car can only make 6 moves: FWD, FWD_RIGHT, FWD_LEFT, REV, REV_RIGHT, REV_LEFT
- The car starts from (0,0) facing east(+X direction)
- Obstacles occupy the spaces within grid cells, making the 4 corners around them invalid for car to move into
- Certain cells are marked as rough roads, which doubles the fuel cost of traversing a path around them.

Objective: To solve the path planning problem for any given goal pose(x,y and heading) and return the solution path along with the control list.



Approach:

The solution implemented here is uses A* search algorithm as the base.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 bigger complex maps, where optimization for the shortest path is more effective.
- RRT like sampling algorithms are suited for continuous openworld exploration

- 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 acheived by mainitaing a tree of paths to explore and choosing the best option based on the cost function.

$f(pose) = g(pose) + h(pose)$

Where h(pose) is the heuristic function that estimates the cost of the smallest distance from cell to goal. g(pose) 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.

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 neigbours 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

Customizing A*:

- The 2D grid map is extended to a 3D map with the addition of orientation as the 3rd dimension.
  - 0<=x<mapWidth, 0<=y<mapHeight, orientation = {N,W,E,S}
- The cost from start g(pose) is calculated as g(pose) = g(parentpose) + MoveCost(parentpose -> pose)
  - MoveCost returns unit cost for FWD and REV movements
  - For the other 4 turning movements, it returns a quarter cirumference ($\pi\frac{unit}{2}$) and scales this value by 2 if the path is over a rough road
- The heuristic function calculates a diagonal cost that is a sum of (quarter circumference* diagonal cells) + (unit * orthogonal cells)
  - This is an admissible heuristic function since the real cost to goal will always be greater than or equal to the heuristic estimate.

- The goal heading conditon cannot be included in the heuristic, since the goal heading cannot be arrived at using a monotonous function. But the priority queue implementation of the Openset ensures that the algorithm is driven towards the matching goal pose in an optimal way.

2 Astar_planner class:

The algorithm is implemented in the Astar_planner library under Astar_planner.h and Astar_planner.cpp. This is standalone library that uses only standard c++ libraries and works with c++11 and above

```
class Astar_planner{
private:
....
public:
Astar_planner(bool verbose_flag=false) : verbose(verbose_flag);
~Astar_planner();
List<Pose> search(int mapWidth, int mapHeight,const vector<spair<int,int>>&obstacles, const vector<pair
...
};
```

The member function Astar_planner::search() inputs the map dimensions, obstacles and rough road vectors, robot pose and goal pose and outputs the path from starting position to goal along with the moves list through a reference function input.

The map coordinates are represented by a simple structure xy:

```
struct Pose { int x; int y; Direction orientation}; where Direction is an enumerable type defined by: e
The returned moves list is a list of ControlOptions defined by the enumerable type:
enum ControlOptions {FWD,FWD_RIGHT,FWD_LEFT,REV,REV_RIGHT,REV_LEFT};
```

Internally, the algorithm uses a few key data structures and functions:

- The resulting waypoints and moves are returned as std::lists for fast insertion and removal
- The obstacles are stored in std::unordered_set<pair<int,int>> for O(1) lookup
- Rough roads are stored in std::unordered_map, where each (x,y) coordinate is linked to a set of coordinates that has a rough road between them. Gives near constant look up
- An unordered_map is used to store parent, move command and f,g,h information about cells, keyed by the cell's pose using a custom hash function(HashTriplet()), O(1) look up
- The closedList used to track previously explored cells is implemented using an unordered_set, using a custom hash function (HashPair()), O(1) lookup

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

3 ROS architecture:

The Astar_planner library is called through an AstarPlanner service under the ROS architecture. This is supposed to mimic an implementation of the path planner in a navigation stack. An AstarPlanner Server and Client are implemented to test the algorithm, with the output being published through ROS topics to be visualized in rviz. Custom rosmsgs are built to assist in the transfer of information between server and client.

- The AstarPlannerServer runs the ServiceServer for the algorithm that takes the following service format

```
int32 mapWidth
int32 mapHeight
VectorPair obstacles
VectorPair roughRoads
Pose2D start
Pose2D goal
---
Pose2D[] path
int32[] moves
```

Where, VectorPair is an array of Pair messages

```
Pose2D[] vect
```

Where, Pair is a 2D point on the map with x,y coordinates

```
int32 x
int32 y
```

And Pose2D defines 3D pose in the map with x,y and direction

```
int32 x
int32 y
int32 d
```

- The AstarPlannerClient is the ServiceClient that takes in optional custom user arguments about the map and builds the service message to call AstarPlannerServer. Once it gets a response from the server, it parses the response and publishes the output through three different topics to be visualized in rviz.
- rviz is used to visualize the map and the path planner output. A custom config file is included to setup the rviz window for this. Red cells represent obstacles, yellow cells represent rough roads. The green poses represent the position and orientation of the car along the path.
- A tf publisher node is run in parllel to assist rviz in recognizing the car frame as the map frame.

Comments:

- ROS action is a better implementation of a path planning algorithm than ROS services since they are non-blocking and offer feedback. But, a ROS service demonstration is done in this project due to time constraints.

4 Demo Instructions:

The project requires ROS-kinetic or higher installed on an ubuntu machine with rviz package installed as well. The project is compiled and built using cmake in a catkin workspace.

1. Create a catkin workspace if it doesnt exist and clone the repo into the <catkin_workspace>/src folder.
2. Build the project by calling catkin_make from <catkin_workspace> and then $ source devel/setup.bash for good measure
3. There are two ways to run the demo. Using the car_driver.launch launch file or running the nodes individually.
4. If using the launch file, simply run the following command, which launches the AstarPlanner service and client, and rviz with the required config file with default map parameters.

   ```
   roslaunch car_driver car_driver.launch
   ```

5. If using the launch file with custom parameters, use the following command and modify the args accordingly. Note the client ignores the args unless all 10 args are defined in cmd line.

```
roslaunch car_driver car_driver.launch mapWidth:=20 mapHeight:=20 startX:=0
startY:=0 startD:=2 goalX:=18 goalY:=18 goalD:=2 obstacleCount:=25
roughRoadCount:=25
```

(a) Where, startD and goalD are the orientations of the car at start and goal poses.

```
Key: North = 0; West = 1; East = 2; South = 3;
```

(b) obstacleCount and roughRoadCount are the number of obstacles and rough road gridcells to be randomly inserted into the map respectively.

6. If running the nodes manually, execute the following commands in order

(a) Launch AstarPlannerSever:

```
rosrun car_driver AstarPlannerServer
```

(b) Launch AstarPlannerClient with default parameters

```
rosrun car_driver AstarPlannerClient
```

(c) Alternatively, launch AstarPlannerClient with custom parameters(args ordering follow that of the roslaunch car_driver.launch file. Refer(5) above.

```
rosrun car_driver AstarPlannerClient 20 20 0 0 2 18 18 2 25 25
```

(d) Launch the tf publisher to assist rviz in frame transform

```
rosrun tf static_transform_publisher 0 0 0 0 0 0 map Car_driver 50
```

(e) Launch rviz and load Rviz_config.rviz from <catkin_workspace>/src/car_driver/

```
rosrun rviz rviz
```

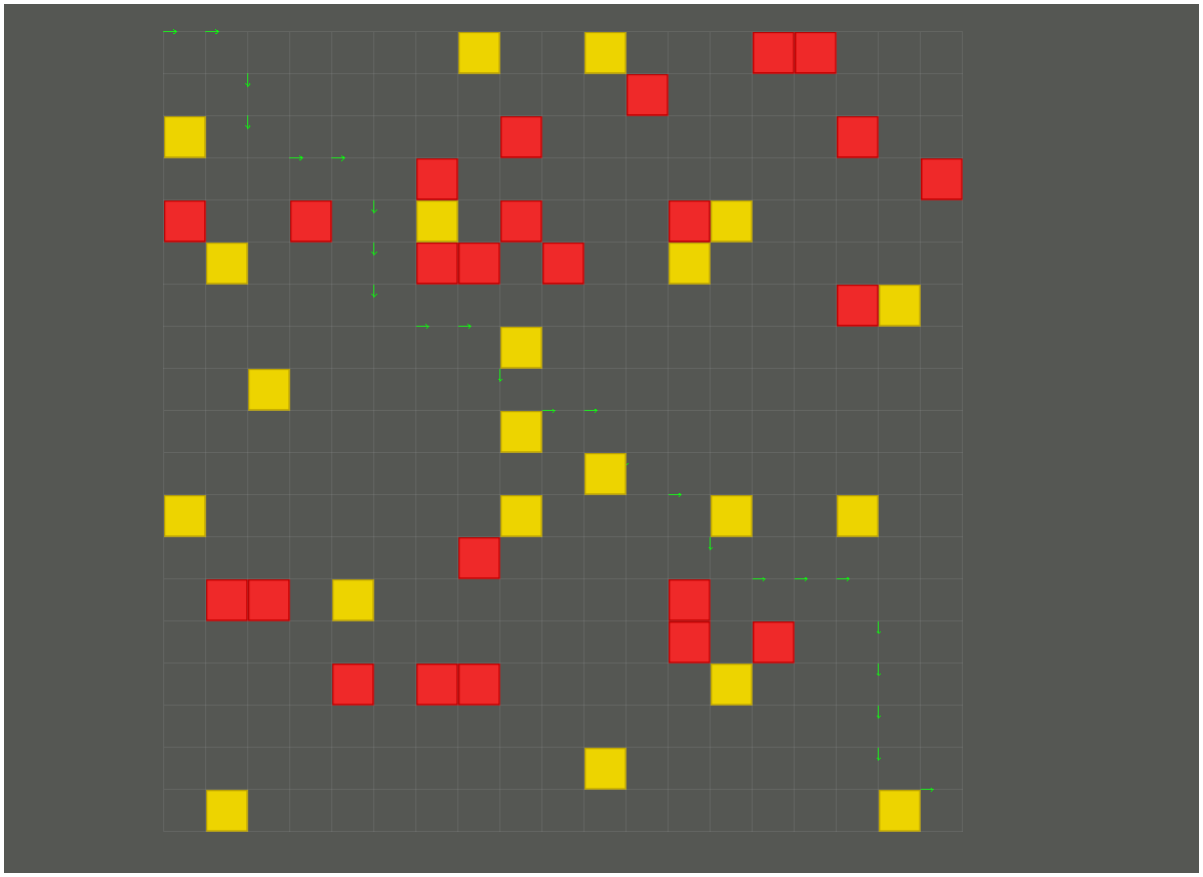
Further customizations:

For testing the algorithm further with custom obstacles and rough roads vectors, edit AstarPlannerClient.cpp at lines **11** and **78** and follow the instructions given there.

5 Understanding the visuals and command line outputs:

Here is a sample out of the algorithm using the following args

```
roslaunch car_driver car_driver.launch mapWidth:=20 mapHeight:=20
startX:=0 startY:=0 startD:=2 goalX:=18 goalY:=18 goalD:=2
obstacleCount:=25 roughRoadCount:=25
```



- The map is oriented with +X direction going East and +Y direction going south. The coordinates (x,y) represent the intersection of grid lines, while the obstacles and rough roads cells occupy the spaces between the lines.
- The green pose array represents the pose of the car at each way point. Please refer to the ROS_INFO way point list output in command line to follow the way points clearly.

```
[ INFO] [1605572343.880192057]: Way point list:
[ INFO] [1605572343.880233870]: Go FWD       -> (0,0,E)
[ INFO] [1605572343.880249966]: Go FWD       -> (1,0,E)
[ INFO] [1605572343.880263566]: Go FWD_RIGHT -> (2,1,S)
[ INFO] [1605572343.880282402]: Go FWD       -> (2,2,S)
[ INFO] [1605572343.880295364]: Go FWD_LEFT  -> (3,3,E)
[ INFO] [1605572343.880307872]: Go FWD       -> (4,3,E)
[ INFO] [1605572343.880320475]: Go FWD_RIGHT -> (5,4,S)
[ INFO] [1605572343.880333048]: Go FWD       -> (5,5,S)
[ INFO] [1605572343.880344929]: Go FWD       -> (5,6,S)
[ INFO] [1605572343.880360081]: Go FWD_LEFT  -> (6,7,E)
[ INFO] [1605572343.880372818]: Go FWD       -> (7,7,E)
[ INFO] [1605572343.880384671]: Go FWD_RIGHT -> (8,8,S)
[ INFO] [1605572343.880396716]: Go FWD_LEFT  -> (9,9,E)
[ INFO] [1605572343.880408554]: Go FWD       -> (10,9,E)
[ INFO] [1605572343.880422758]: Go FWD_RIGHT -> (11,10,S)
[ INFO] [1605572343.880435067]: Go FWD_LEFT  -> (12,11,E)
[ INFO] [1605572343.880446869]: Go FWD_RIGHT -> (13,12,S)
[ INFO] [1605572343.880458787]: Go FWD_LEFT  -> (14,13,E)
[ INFO] [1605572343.880470573]: Go FWD       -> (15,13,E)
[ INFO] [1605572343.880482176]: Go FWD       -> (16,13,E)
[ INFO] [1605572343.880496429]: Go FWD_RIGHT -> (17,14,S)
[ INFO] [1605572343.880508296]: Go FWD       -> (17,15,S)
[ INFO] [1605572343.880519891]: Go FWD       -> (17,16,S)
[ INFO] [1605572343.880531686]: Go FWD       -> (17,17,S)
[ INFO] [1605572343.880543518]: Go FWD_LEFT  -> (18,18,E)
```

- If the server isn't available a ROS_ERROR message is printed acknowledging the same
- If the algorithm detects invalid arguments, std::cout messages are printed explaining the type of error encountered.

6 Performance:

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 h(pose) has to be designed smartly for the algorithm to succeed. For this problem, having h(pose) represent the diagonal distance from cell to goal is sufficient. In the case described in section 5, A star found the optimal path with 24 steps or 30.84 units of fuel cost, by exploring 171 nodes out of 400

Space_complexity: A* by design has exponential space complexity($6^{pathlength}$), since for every node explored from the open list, 6 more potential nodes are added to it. But, in this particular problem of 3D map path planning, space is constrained, as it can never go beyond row*column cells. If the heuristic function is defined optimally, the space needed will only be proprotional 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(row*column).

6.1 Conclusion:

The project was tested with mutiple scenarios to test its performace and check for bugs. Performance wise, A* is deterministic and finds the optimal path in an efficient manner. 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, especially when the map size gets bigger.

Further Improvements: A few functionalities are incomplete because of time constraints.

- Better feedback from the Astar_planner library. Currently the API only outputs a path and a moves list. Other error notifications only appear through commandline messages.
- Perfomance optimizations involving copying objects.
- Unit tests to catch bugs
- Stress testing the algorithm with extreme cases.