CSE 468/568 Assignment 5
A* Path Planning

The objective of this assignment is to plan a path for a robot from a given starting point to a destination. Download the package from UBlearns. For this assignment, you will be using the same husky environment from previous few lab assignments by running `roslaunch husky_gazebo husky_playpen.launch`.

The objective of the assignment is to use A* planning algorithm to find a route from a default start point, e.g. (-8.0, -2.0) to a target goal, e.g. (-5.0, 9.0). Please go through the tutorial on ROS Parameters. The goal should be defined as two parameters `goalx` and `goaly` both of which should be of type `double`. This allows us to set a new goal before launching.

There are a couple of challenges in implementing A* planning as discussed in class. The first challenge is to derive a graph representation of the workspace. This depends on the map representation that the estimation block provides us. Typical examples of such representations are occupancy grids - a grid representation with 1s and 0s with 1 indicating an obstacle in that cell and 0 representing an empty cell. For this assignment, we have provided you such an occupancy grid. It is the file `map.txt`. It grids the husky_playpen world as 0.5m x 0.5m cells. You should import this into your program as the map. You can simply paste the array into your code and read it appropriately as a 2D matrix with dimensions of (42, 42).
The second challenge is the heuristic for the estimated cost between the current node and the goal. Given you know the current location and the goal, you can use Euclidean distance between the current location and the goal as the heuristic cost. ϵ is the coefficient of the heuristic function, used to scale the heuristic cost. You can start with $\epsilon = 1$, and tune only if required, based on the paths returned by your A* algorithm.

But how do you know where you are on the map? This is the problem of localization, which we will work on in the next assignment. For now, you will use the global location of the husky robot. This is provided via the topic:

```
$ /gazebo/link_states
```

which publishes a set of arrays containing the global position of all objects in the world. You will need to find the index of the link named: "/husky/base_link", as the husky's position corresponds to the index in the Pose array.
Take a look at the message documentation for more info:
https://docs.ros.org/en/jade/api/gazebo_msgs/html/msg/LinkStates.html.

Path planning and executing the plan should be performed separately. Modularization is the key to the correct implementation. This is why we suggest you split the work into 3 stages.

1) Path Planning (50 points)

You will implement a path planner, that runs only once after launch. You will need:

- The map
- The start position
- The goal position

If you don't have any of them initially, you should subscribe to the appropriate topic, and wait until a message is received and update the value. Your planner should wait for the update before executing. This problem can be solved using the map cells, rather than absolute positions. You must be able to correspond any absolute position in the world, to a cell position in the map. Implement and test functions that only perform this.

- A function that takes (i, j) index of the map node and returns (X, Y) world coordinates of the center of the cell.
- A function that takes (X, Y) world coordinates and returns (i, j) node indexes of the cell which (X, y) point falls in.

Your A* path planner algorithm should output a global path from the start node to the goal node. The global path essentially contains a list of nodes/checkpoints, such that when the robot moves to each one of them in succession, it will eventually reach the final goal location. Once you solve this part, you can output the results to the console.

- First, output the sequence of the node indexes. e.g.:
  `(3,4) -> (3,5) -> (4,5) ->` ...
- For each node in the sequence, output the absolute position of the center of the cell. Use a single fraction digit format. e.g.:
  `(0.0,0.0) -> (0.0,0.5) -> (-0.5,0.5) ->` ...

You may refer to this video for a walk-through of the A* algorithm:
https://www.youtube.com/watch?v=-L-WgKMFuhE

You should be able to test all the functionality of your path planner. Once a path is generated, you should plot it using `matplotlib`. You should clearly indicate freespace, obstacles, the start and goal positions as well as the planned path between them, on your plot. Feel free to use colors, legend, etc. Test to see if you can launch everything by running the command:

```
$ roslaunch lab5 lab5.launch
```

2) Single Step Execution (20 points)

In order for your robot to follow a sequence of targets/checkpoints, it should first follow a single step. Given any current position, you should command the robot to move towards its current immediate goal. This should be the coordinates of the center of a cell. This will be an important step, as the robot needs to move from center of a cell, to the center of the next cell to avoid crashing into obstacles that are at the edges. This also depends on the map resolution and the safety margin of the robot size. So you may need to observe the performance, and make changes. We suggest that you keep an internal global variable for the current immediate goal. This way, it can be updated asynchronously in the next stage. Anytime you receive an update on the position of the robot, you can publish the appropriate `cmd_vel` message in the callback, to move the robot towards the current immediate goal.

Test this stage using hardcoded coordinates, and observe the performance. Test to see if you can launch everything by running the command:

```
$ roslaunch lab5 lab5.launch
```

3) Full Execution (30 points)

After you have a full path planner and your robot can move towards a goal, you can put them together towards a full execution. Once you have a full sequence of checkpoints/targets, you can pick the first one as the current immediate goal. When you reach a certain **proximity** of the current goal, you can update the goal towards the next target. This should continue until the path is finished and you can announce the completion of the execution, and terminate your controller.

Submission Instructions

You will submit `lab5.zip`, a compressed archive file containing the lab5 package (folder). The package should contain a launch file `lab5.launch`. The package should compile if we drop it into our catkin workspace and call `catkin_make`. Please take care to follow the instructions carefully so we can script our tests. Problems in running will result in loss of points. Test to see if you can launch everything by running the command:

```
$ roslaunch lab5 lab5.launch
```

Please use UB Learns for submission
The assignment is due Friday, Nov 18 before midnight.

Tips

Jump Nodes

As maps become more fine-grained, it becomes tedious navigating from just one cell to the next with a larger robot that might just roll past the cell it meant to go to. Consider looking ahead of your path to see if you can navigate directly toward a more distant cell along a straight line path, going over all the other cells with minimal readjustment. This method borrows from a technique used to speed up A* search called "Jump Point Search".
https://en.wikipedia.org/wiki/Jump_point_search

Resetting the Husky

You can easily move the husky back to a set position by using the provided terminal command, just change the position's x and y values. To do so in python, use the corresponding message type:

```
rostopic pub /gazebo/set_model_state gazebo_msgs/ModelState "model_name: 'husky'
pose:
  position:
    x: 0.0
    y: 0.0
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0
twist:
  linear:
    x: 0.0
    y: 0.0
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
reference_frame: ''"
```

Map

No need to parse the map text file, you can just copy it right into your code. From there you can change the representation of the map to how you see fit.
The square of asphalt you see in Gazebo is 20m x 20m, and the map is 21m x 21m to provide a solid wall on all sides. To better see the grid for positioning, you can select View and mark Transparent. Here is an example path on the map.