49274: Advanced Robotics **Spring 2020**

Assignment 2: Path Planning

University of Technology Sydney (CAS) Due date: **18:00 Tuesday 8th of September**

Introduction

The basic motion planning algorithms consist of two steps. Path planning and Control. In this assignment emphasis will be primarily on the first step, where you will be implementing an A* planner and a waypoint smoothing algorithm.

Path Planning

The path planning problem states: Given a starting and ending configurations, how can our robot reach the goal with the lowest "cost". Simply put, the path planning problem is to find the cheapest (e.g., shortest) sequence of actions that will lead our robot from the starting location to the ending location.

Example

First we simplify our environment into discrete grid cells and we define our starting location (S) and goal location (G). Obstacles are represented by (x). Our robot is only allowed to move in 4 different directions: *right*, *left*, *up* and *down*. Visually we can see that the shortest path with lowest

0	0	0	0	0	0	0
0	0	x	x	0	0	0
0	0	x	0	0	0	0
S	0	x	G	0	0	0

cost to reach the goal would be in 11 steps. Given that the cost of each move (g) is 1, the total cost is 11. Algorithmically, searching for the goal can be done by a method called *Dijkstra*'s algorithm.

3	4	5	6	7	8	9
2	3	x	x	8	9	7
1	2	x	10	9	10	11
0(S)	1	x	11(G)	10	11	12

From the starting cell (S) we look in the directions **top**, **right**, **down** and **left** in any order and increment the cost by 1 every time we expand. To keep track of our frontier nodes (i.e., nodes that can be expanded from), we use a list called `openList`. `openList` is a list of nodes: it consists of a x, y position and the cost associated with that node.

`openList`: **Step 1**

x	y	`cost`
1	1	0

To begin the open list will only consist of the start position. Then we expand:

`openList`: **Step 2**

x	y	`cost`
1	2	1
2	1	1

The node that was expanded from is first taken out of our open list, this should be done every time a new node is expanded from. Next we search and add all the new nodes (i.e., neighbours) to our open list. A good practice is to check the nodes being expanded from as *closed*, this means that when performing new searches, closed nodes will be avoided.

`openList`: **Step 3**

x	y	`cost`
2	1	1
2	2	2

It is of utmost importance to always select the node with the lowest cost to expand from, as you can see the next node above will be $\begin{bmatrix}\mathbf{2} & \mathbf{1} & \mathbf{1}\end{bmatrix}$.

Expand Grid

If you continue to do the search until the goal is reached while tracking each expansion, by incrementing a counter by 1, then an expand grid can be obtained.

6	7	8	9	10	11	13
4	5	x	x	12	14	16
2	3	x	19	15	17	20
0(S)	1	x	22(G)	18	21	0

Optimum Policy

From this expand grid, the **optimum policy** or best path, indicated by *, can be found by tracing backwards from the goal back to the start. Search by following the lowest expand number.

When searching for the optimum path incrementally, you can see that some search directions will take you away from the goal.

6	7*	8*	9*	10*	11	13
4	5*	x	x	12*	14	16
2	3*	x	19	15*	17	20
0(S)*	1*	x	22(G)	18*	21	0

This can be considered wasted effort and time consuming especially if the map is complex.

Heuristic Grid

*A** search solves this problem by introducing a heuristic cost (h) to the cost of each step. A simple (admissible) heuristic would be how far away each grid cell is from the goal, ignoring the obstacles (Manhattan distance).

6	5	4	3	4	5	6
5	4	3	2	3	4	5
4	3	2	1	2	3	4
3	2	1	0(G)	1	2	3

Now our cost becomes $f(i,j) = g(i,j) + h(i,j)$, where $g(i,j)$ is our original cost from *Dijkstra*'s algorithm and $h(i,j)$ is the new heuristic cost.

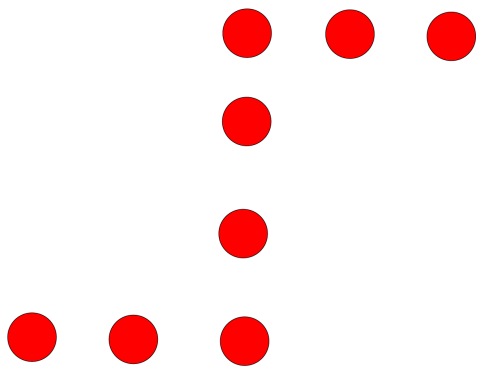
9=(3+6)	9=(4+5)	9=(5+4)	9=(6+3)	11=(7+4)	13=(8+5)	15=(9+6)
7=(2+5)	7=(3+4)	x	x	11=(8+3)	13=(9+4)	15=(10+5)
5=(1+4)	5=(2+3)	x	11=(10+1)	11=(9+2)	13=(10+3)	15=(11+4)
(S)	3=(1+2)	x	(G)	11=(10+1)	13=(11+2)	15=(12+3)

If we only expand to the node with the lowest cost, then the shortest path to the goal can be found without exhaustively searching. You can see that grids with costs values above 11 are never expanded from if the above searching method is applied.

Note that it is possible to insert a coefficient λ in the total cost calculation to select the planning algorithm used.

$$f(i,j) = g(i,j) + \lambda \times h(i,j)$$

With $\lambda = 0$ choosing *Dijkstra*; and $\lambda = 1$ for *A**.


(a) Original Path

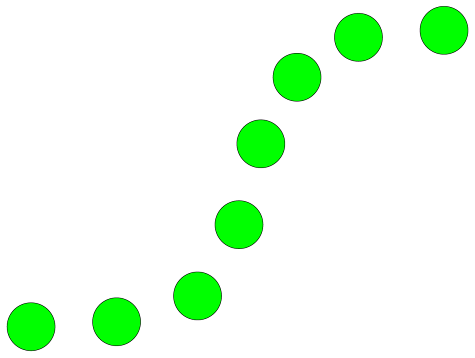

(b) Smooth Path

Figure 1: Path Smoothing

Detailed description of the *Dijkstra* and *A** algorithms are given in the **Appendices** section.

Path Smoothing

Now that we have an optimum path, we want to find the best way the robot can follow. However, our current path has many disadvantages. For example, the current set of way points would require our robot to take 90 degree turn, which is impossible for some robots. Another disadvantage is that the robot would need to stop before making turns resulting in abnormal motion for the robots. By smoothing all our way-points, the robot motion to its goal location will become much more fluid and natural. To perform this smoothing we define and then solve an optimization problem. Suppose S_i is the $i-th$ smooth way-point (the x and y location of the point), and p_i is the $i-th$ point in our original way-points. We want to minimize a **weighted** sum of:

1. (squared) distance between smooth and original points (i.e., p_i and s_i)
2. (squared) distance between consecutive smooth points (i.e., s_i and s_{i+1}).

It is important to realize that, if we only minimize (1) then we will get the original path, and if we only minimize (2) then we will get the straight line linking the start and goal. Therefore for our algorithm to work, we must simultaneously minimize both (a weighted sum of them) and apply weight α and β to each criteria. These values will affect the smoothness of the new path. You need to tune the values of weights (α and β).

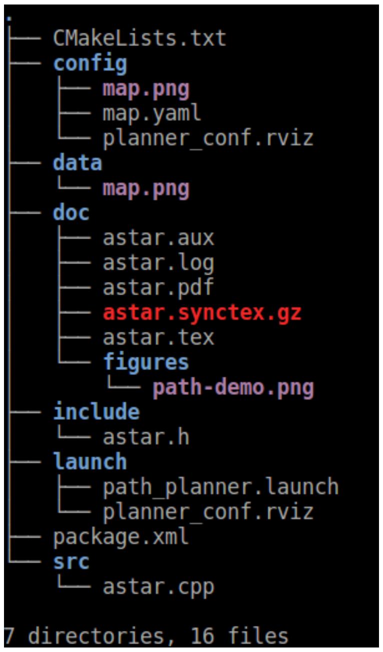
Compiling and Running in ROS

Compilation

Run following commands in a terminal window.

- Download `path_planner.zip` package, extract the folder `path_planner` into ~/catkin_ws/src/.
- The file structure under `path_planner` is shown below
- Verify installation successful by running command: `rospack find path_planner`
- The system should respond with `~/catkin_ws/src/path_planner`
- Change directory: `cd ~/catkin_ws`
- Build catkin_ws: `catkin build`

Figure 2: package content



```
.
├── CMakeLists.txt
├── config
│   ├── map.png
│   ├── map.yaml
│   └── planner_conf.rviz
├── data
│   └── map.png
├── doc
│   ├── astar.aux
│   ├── astar.log
│   ├── astar.pdf
│   ├── astar.synctex.gz
│   ├── astar.tex
│   └── figures
│       └── path-demo.png
├── include
│   └── astar.h
├── launch
│   ├── path_planner.launch
│   └── planner_conf.rviz
├── package.xml
└── src
    └── astar.cpp

7 directories, 16 files
```

Running ROS

Here a simplified way (one liner) of loading the astar planner package, this is possible in the most recent download package, it contains a new version of launch file that loads everything in one step.

- `roslaunch path_planner path_planner.launch`

Alternatively you may want to load the system step by step, in this case, make sure the last two lines in the file astar planner.launch are commented out, as in previous version of your download package.

- `roscore`
- `cd ~/catkin_ws/src/path_planner`: to change to the right directory
- `rosrun map_server map_server data/map.yaml`
- `rosrun rviz rviz -d data/planner_conf.rviz`: to start rviz with the configuration file loaded
- `roslaunch path_planner path_planner.launch`: to start the planner with certain parameters

Parameters

You are required to change the start position and target position in the launch file when you test your program. The basic explanation of the launch file is as below:

- **startx** and **starty** denote the start position in map coordinate system
- **goalx** and **goaly** denote the goal position in map coordinate system
- **lambda** denotes selection of *Dijkstra* or *A**

Data Structure in Brief

Before you begin coding, it is very important to understand some of the functions and structures given to you in the `Astar` class.

Structures

- **Grid**: This structure represents a grid cell, there are four variables that can be used for each grid. These variables are all set to 0 initially.
  - `occupied`
  - `closed`
  - `expand`
  - `heuristic`
- **Node**: A node is represented by an index x position, index y position and a cost value.
  - `x`

  - `y`
  - `cost`

- **Waypoint**: A waypoint is the pose which may belongs to the shortest path.
  - `x`
  - `y`

Variables

- `gridmap_` : 2-dimensional grid map, with index rule `[y][x]`, the related variables are `grid_resolution_`, `grid_height_`, `grid_width_`
- `optimum_policy_` : the optimum policy searched from the goal to the start
- `waypoints_` : the way points published to the correct topics and visualised, needs smoothed, related variables are `poses_` and `waypoints_done_`
- `start_[2]` and `goal_[2]`: define the start position and goal position

Functions

Debugging Functions

- `void debug_break()`: pauses execution, resumes with a hit key
- `void print_list(vector<Node> nodelist)`: prints out an array of nodes
- `void print_grid_props(const char *c)`: print node grids properties according to argument c, 4 attributes available: c closed h heuristic e expand o occupied
- `void print_waypoints(vector<Waypoint> waypoints)`: print way-points

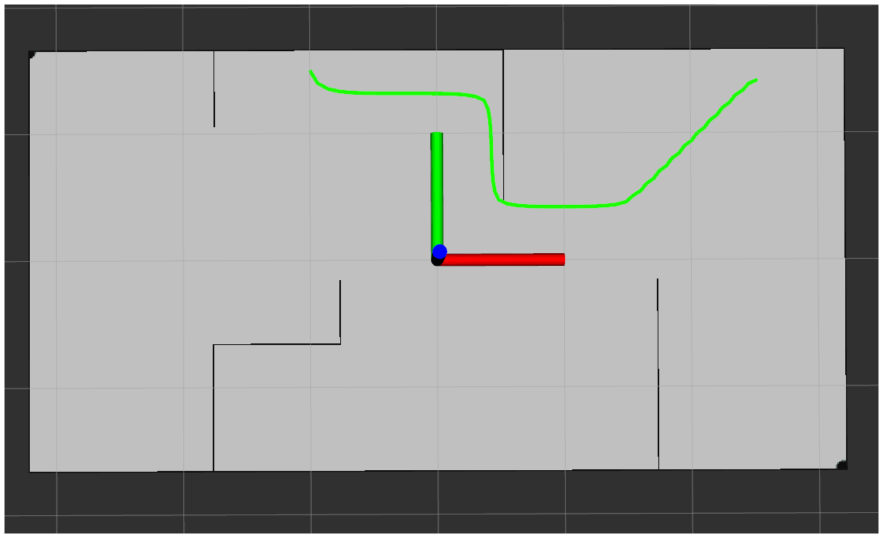
Conversion Functions

- `double grid2meterX(int x)`: convert from grid index x to position in meter
- `double grid2meterY(int y)`: convert from grid index y to position in meter
- `int meterX2grid(double x)`: convert from x coordinate to grid index x
- `int meterY2grid(double y)`: convert from y coordinate to grid index y

Other Functions

- `vector<Node> descending sort(vector<Node> nodelist)`: sort the openlist in descend ing order, needs to be called in `path_search()` function
- `int contains(const vector<Node>& nodelist, const Node& q_node)`: checks if `nodelist` contains an element at same (x, y) coordinates as `q_node`; returnd index in `nodelist` if found, `-1` otherwise
- `void setup_gridmap()`: initialise the grid map, called at the beginning of `path_search()` function
- `void update_waypoints( double * robot pose )`: given robot pose, compute the updated way-points, needs to call `smooth_path()` function

Figure 3: An example result of A*



Key Functions

You are to implement following functions:

- `void init_heuristic(Node goal_node)`: initialise each grid cell's heuristic value
- `bool path_search()`: implement Dijkstra and *A** algorithms, based on λ value
- `void policy( Node start_node, Node goal_node )`: retrieve optimum policy
- `void smooth_path( double weight_data, double weight_smooth )`: smooth the path

Some Assumptions you can make

- Cost for moving between grid cells is 1
- The robot cannot move diagonally through grid cells
- The heuristic value is the number of grid cells to the goal

Hints

- Read **Appendices** carefully to have a high-level understanding of planning algorithms.
- Setup the heuristic values for you grid cells first.
- Use the structure `node` for creating your open list. Keep track of this array variable when debugging
- Try using the `closed` variable to keep track of grids that have already been expanded.
- Try using `expand` variable to keep track of how your algorithm expanded. This will be important to obtain the optimal policy.
- Debugging is very important, the debug functions may prove to be very beneficial

If done correctly, you will be able to: Show how your algorithm searched from start to goal. Show the correct heuristic values. Show that the optimal policy is correct by updating the optimal policy variable.

Smoothing Using the Gradient Descent Algorithm

`void smooth_path( double weight_data, double weight_smooth )`

- Initialise the smooth waypoints s_i with the original waypoints p_i: $s_i = p_i$ (note that s_i and p_i are vectors and have two fields: x and y).
- At each iteration, loop through all the waypoints, except the **first** and the **last**, and update your smooth path, using the equation below:

  $$s_i^{new} = s_i - (\alpha + 2\beta)s_i + \alpha p_i + \beta s_{i-1} + \beta s_{i+1}$$

  where α is `weight_data` and β is `weight_smooth`.
- Finally, keep iterating this loop until your new smooth path changes very little from you previous smooth path:

  $$\sum_i (s_i^{new}(x) - s_i(x))^2 + (s_i^{new}(y) - s_i(y))^2 < \epsilon$$

  where ϵ is our tolerance (e.g., 0.001).

- Try changing variables and to see what kind of paths you are able to get.

- The smoothing function should update the variable `vector<waypoint> smooth_waypoints` with the new smooth path. If you run `updateGUI` from `main.cpp` and turn on waypoints, you can visualise both the normal (red) and the smooth path (green).

Appendices

Algorithm 1: Dijkstra & A* algorithm

```
Data: Gripmap, start node, goal node
Result: Set of expanded nodes (closed grid cells)
openList contains start node;
All cells un-closed;
while Goal node not found do
    Remove lowest cost node from OpenSet;
    Mark cell at node position as closed;
    if Lowest cost node is goal node then
        Goal node found;
    Get neighbours of lowest cost node;
    for Each neighbour do
        if Neighbour node cell already closed then
            Do nothing;
        else
            if Neighbour node is in openList then
                if Neighbour node has lower cost than node already in openList then
                    Replace node in openList with neighbour node;
            else
                Add neighbour node to openList;
                Record expansion order;
```

Algorithm 2: Exracting the path from the set of expanded nodes (closed grid cells)

```
Data: Set of expanded nodes from Dijkstras algorithm (close grid cells), start node, goal node
Result: Path from start to goal
CurrentNode is goal node;
Path is empty;
while Start node not found do
    Add CurrentNode to Path;
    if CurrentNode is start node then
        Start node found;
    CurrentNode ⟵ node with smallest expansion order out of current neighbours;
```