49274 Advanced Robotics (Spring 2019)

Assignment 2: A* Path Planning

- Due date: 02/09/2019 (week 7) at 6 pm
- Total marks: 10 (10% of the final mark for the subject)

Introduction

Path planning is essential for mobile robotics. Given a known map, a start position, and a goal position, the aim of path planning is to find a path from the start to goal. Dijkstra's algorithm and A* are two important optimal path planning algorithms.

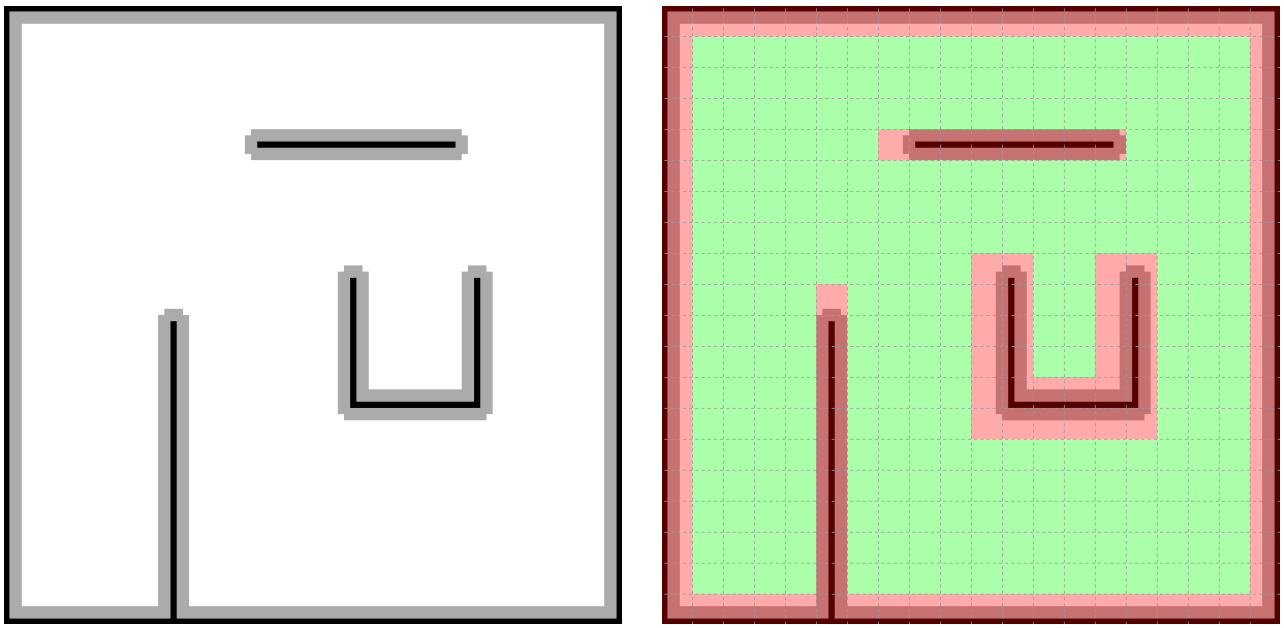
More generally Dijkstra's algorithm and A* are known as graph traversal algorithms. A graph is a set of vertices (also called nodes) and edges that connect them. Dijkstra's algorithm and A* are used to search through the graph to find the optimal (shortest) path between two vertices/nodes.

In mobile robotics, a (2D) map of an environment is typically represented by an occupancy grid. With a holonomic vehicle (i.e. differential drive) we can inflate the occupancy grid so that the robot can be represented by a point (shown in Figure 1a). Finally we can discretise the occupancy grid at a lower resolution to reduce the amount of computation for path planning (shown in Figure 1b).

By assuming that each cell is connected to any unoccupied adjacent cell, we have a graph that we can search with Dijkstra's algorithm or A*. Each unoccupied cell is a node in the graph, and the weight of the edge between connected nodes is the distance between the centres of each cell. We can allow only Manhattan movement (up, down, left, and right), or both Manhattan and diagonal movement.

Dijkstra's algorithm

Dijkstra's algorithm can be implemented using two sets of nodes, an open set and a closed set. Each node has a cost (the distance from the start node), and a parent node. Initially the open set contains only the start node. In each iteration we remove the node with the lowest cost from the open set, put it on the closed set, and put any nodes connected to it onto the open set. This process is referred to as expanding the node.



(a) Initial (black) and inflated (grey) occupancy grid

(b) Discretised at a lower resolution (red: occupied, green unoccupied)

Figure 1: An example occupancy grid

Before placing a new node onto the open set we first check if the node is already on the closed or open sets. If the node is already on either the closed or open set we discard the new node, since the cost of the node already on the closed or open set will be less than the cost of the new node.

The search finishes when we place the goal node onto the closed set. The closed set gives us the shortest path from the start node to the goal node. First we find the goal node on the closed set, then we find it's parent, and then the next parent, and so on until the start node is found. The list of parent nodes is the shortest path from the goal to the start, and reversing this gives us the shortest path from start to goal.

Dijkstra's algorithm is shown in Algorithm 1, and the process of extracting the path from the closed set is shown in Algorithm 2.

A*

A* is a modification of Dijkstra's algorithm that can significantly reduce the number of nodes expanded. It does this using a heuristic, a function that provides an estimated value for the quality of a node. In mobile robotics each node in the graph represents a real position in the map, and the heuristic we use is the real distance between the node and the goal node. If a heuristic is both admissible (it doesn't overestimate the cost of reaching the goal) and consistent (the estimate is always less than the estimate of any neighbour, plus the cost of reaching the neighbour), A* is guaranteed to give an optimal result. Real world distance is

```
Data: Graph, start node, goal node
Result: Set of expanded nodes (ClosedSet)
OpenSet contains start node;
ClosedSet is empty;
while Goal node not found do
    Remove lowest cost node from OpenSet;
    Put lowest cost node onto ClosedSet;
    if Lowest cost node is goal node then
        Goal node found;
    end
    Get neighbours of lowest cost node;
    for Each neighbour do
        if Neighbour node not in ClosedSet or OpenSet then
            Add neighbour node to OpenSet;
        end
    end
end
```

Algorithm 1: Dijkstra's algorithm

```
Data: Set of expanded nodes from Dijkstra's algorithm (ClosedSet), 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;
    end
    CurrentNode is the parent of CurrentNode;
end
Reverse Path;
```

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

both admissible and consistent.

There are two modifications to Dijkstra's algorithm to implement A*. First, when selecting the lowest cost node to remove from the open set, a combined cost is used:

$$\text{cost} + \text{heuristic} \times \text{weight} \tag{1}$$

The weight is used to modify the behaviour of the algorithm. A weight of 0 will behave like Dijkstra's algorithm, while a weight of 1 is a traditional implementation of A*.

The second modification to Dijkstra's algorithm is a change to what happens when checking if a new node is already on the closed or open sets:

- If the node is already on the closed set, we do nothing.
- If the node is already on the open set, we check if the new cost is less than the existing cost. If the new cost is less we replace the cost and the parent node of the node in the open set.
- If the node is not on the closed or open sets, we add it to the open set.

A* is shown in Algorithm 3. As with Dijkstra's algorithm, Algorithm 2 is used to extract the path from the closed set.

Your Task

Your task is to complete the code in 6 methods/functions:

- `pop` in the `OpenSet` class (0.5 marks)
- `update` in the `OpenSet` class (1 mark)
- `getAdjacentCells` in the `OccupancyGrid` class (3 marks)
- `heuristicCost` in the "astar_path_planner.cpp" file (0.5 marks)
- `planPath` in the `PathPlanner` class (3 marks)
- `getPath` in the `ClosedSet` class (1 mark)

The locations where you should insert your code are marked with a comment "YOUR CODE HERE"

There is one final task: describe what happens when the heuristic cost weight is above 1.0. You should experiment with various start and goal positions, and various weight values, and describe what you see. Answer this question in the "QUESTION.txt" file, found in the root folder of the package.

```
Data: Graph, start node, goal node
Result: Set of expanded nodes (ClosedSet)
OpenSet contains start node;
ClosedSet is empty;
while Goal node not found do
    Remove lowest combined cost node from OpenSet;
    Put lowest cost node onto ClosedSet;
    if Lowest cost node is goal node then
        Goal node found;
    end
    Get neighbours of lowest cost node;
    for Each neighbour do
        if Neighbour node is in ClosedSet then
            Do nothing;
        end
        else if Neighbour node is in OpenSet then
            if Neighbour node has lower cost than node already in OpenSet then
                Replace node in OpenSet with neighbour node;
            end
        end
        else
            Add neighbour node to OpenSet;
        end
    end
end
```

Algorithm 3: A* algorithm

Overview of the node

The node starts with the main function in “src/astar_path_planner.cpp”. The main function sets up the node, creates an instance of the `PathPlanner` class, and “spins” to process callbacks.

The `PathPlanner` class makes use of three other classes:

- `OccupancyGrid`
- `OpenSet`
- `ClosedSet`

`OccupancyGrid` class

The `OccupancyGrid` class is declared in “include/astar_path_planner/occupancy_grid.h” and implemented in “src/occupancy_grid.cpp”. The constructor of the class takes a ROS occupancy grid message, inflates it, and stores it.

In the header file (“occupancy_grid.h”) a number of data structures are declared:

- `WorldPosition`
- `GridPosition`
- `Cell`
- `AdjacentCell`

`WorldPosition` and `GridPosition` both contain the member variables `x` and `y`. In `WorldPosition` these variables are *doubles* (real numbers) and refer to a position in the map in metres. In `GridPosition` these variables are *ints* (integers), and refer to a position in the map in cells.

The `Cell` structure is used to return information about a cell in the grid. It contains the member variables `id`, `occupied`, `grid_position`, and `world_position`. `world_position` is the position of the centre of the cell.

The `AdjacentCell` structure is used to return information about cells adjacent to a particular cell. It contains the member variables `id`, `cost`, and `world_position`. `cost` is the cost (distance in metres) of moving from the parent cell to the adjacent cell, and `world_position` is the position of the centre of the cell.

The constructor of the class takes a ROS occupancy grid message and an inflation radius (in metres), and:

- Copies the given occupancy grid message

- Creates an image that is used to access the data in the occupancy grid message
- Creates a structuring element of the correct size for the given inflation radius
- Dilates the image with the structuring element
- Sets variables that store the limits of the map

The class has methods for querying the occupancy grid, such as:

- `isOutOfBounds`
- `isOccupied`
- `getGridPosition`
- `getWorldPosition`
- `getCellId`
- `getCell`

There are a number of different methods with the same name, which differ by type of argument they accept. You should also keep in mind that the private methods (indicated in the header file) are only accessible within the class. You should not need to use these from outside of the class (i.e. in the `planPath` method of the `PathPlanner` class), but they are accessible in the `getAdjacentCells` method.

`getAdjacentCells` method

The `getAdjacentCells` method takes a cell ID and a *bool* indicating whether or not diagonal movement is allowed, and returns a *vector* of `AdjacentCell`. Completing the method is one of the tasks for this assignment.

The cell ID argument has been converted into a grid position for you (named `grid_position`). From this grid position you can find adjacent cells, e.g. the cell to the right is $(x + 1,\ y)$ and the cell to the upper right is $(x + 1,\ y + 1)$.

You need to use `isOutOfBounds` and `isOccupied` to ensure that you return only valid unoccupied cells, and you should only return diagonal cells if `diagonal_movement` is true.

The `AdjacentCell` structure contains the variables `id`, `cost`, and `world_position`. You should use `getCellId` and `getWorldPosition` to get the ID and world position respectively. The cost can be determined by the resolution of the map, given by the `map_.info.resolution` variable. This is the horizontal or vertical distance between cells, remember that the cost value is different for diagonal cells.

`Node` structure

The `Node` structure is declared in the file "include/astar_path_planner/Node.h", and is used by the `OpenSet` and `ClosedSet` classes. It contains the member variables `id`, `parent_id`, `cost`, and `heuristic_cost`. `cost` is the cost (distance in metres) of the node from the start node. `heuristic_cost` is the Euclidean distance (in metres) of the node from the goal node.

`OpenSet` class

The `OpenSet` class stores a vector of `Node` structures in `nodes_`, and provides a number of methods:

- `push`: adds a node to the open set.
- `pop`: removes the lowest combined cost node from the open set.
- `contains`: returns true if the open set contains the given node ID.
- `update`: replaces a node in the open set, if the cost of the given node is less than the node already in the open set.
- `empty`: returns true if the open set is empty.
- `getNodes`: returns a reference to the `nodes_` vector, which is used for publishing the open set markers.

Completing the `pop` and `update` methods are tasks for your assignment.

`pop` method

In the `pop` method you want to find the node with the lowest cost combined cost (cost $+$ heuristic $\times$ weight). Set the value of the `index` variable to be the index of the lowest cost node in the `nodes_` vector, and it will be removed and returned.

`update` method

The `update` method takes a node as an argument. You want to find the same node already in the `nodes_` vector and, if the cost of the new node is less than existing node, replace the existing node.

`ClosedSet` class

The `ClosedSet` class stores a vector of `Node` structures in `nodes_`, and provides a number of methods:

- `size`: returns the number of nodes in the closed set.
- `push`: adds a node to the closed set.
- `contains`: returns true if the closed set contains the given node ID.
- `getPath`: returns a vector of node IDs (integers) from the given start id to the given goal id.
- `getNodes`: returns a reference to the `nodes_` vector, which is used for publishing the closed set markers.

Completing the `getPath` method is a task for your assignment.

`getPath` method

The `getPath` method should return a vector of node IDs which is the shortest path from the start ID to goal ID. You want to fill the `path` vector, which is returned by the method.

To extract the path from the closed set, you find the goal node on the `nodes_` vector, then find it's parent, and then find it's parent, and so on until you find the start node. The path created will be from the goal node to the start node, so it should be reversed before it is returned.

`PathPlanner` class

The constructor of the class:

- Reads parameters.
- Acquires a map (ROS occupancy grid message) from the "static_map" service.
- Creates an instance of the `OccupancyGrid` class with the map.
- Publishes the map created by the `OccupancyGrid` class.
- Advertises topics for a number of markers: start and goal positions, nodes in the open and closed sets, and the path.
- Subscribes to the "initialpose" and "move_base_simple/goal" topics that are published by RViz.
- Advertises the "plan_path" service, which is used to initiate path planning.

The `planPath` method is where the A* path planner is implemented. The `req` variable is the request message that is used to call the method. `req` contains two variables:

- `heuristic_cost_weight`: The weight value for the heuristic cost function
- `diagonal_movement`: a Boolean variable that indicates that diagonal movement should be enabled

The template code provided will:

- Create a `Cell` for the start and goal positions (named `start_cell` and `goal_cell`)
- Create an empty `OpenSet` and `ClosedSet` (named `open_set` and `closed_set`)
- Create a start node (named `start_node`), which is put onto the open set
- Create a Boolean variable that indicates that the goal has been found (named `goal_found`), which is initially false

Your task is to complete the code within the *while* loop to finish the implementation of A* path planning. Your code should:

1. Remove the lowest cost node from the open set with the `pop` method (remember to use `req.heuristic_cost_weight`).
2. Put the lowest cost node onto the closed set.
3. Exit the *while* loop if the lowest cost node is the goal node (remember to set `goal_found` to true).
4. Get the cells adjacent to the lowest cost node (remember to use `req.diagonal_movement`).
5. Then, for each adjacent cell create a new node and:
   (a) If the new node is already on the closed set, do nothing.
   (b) If the new node is already on the open set, update it if it's better.
   (c) If the new node is not on the closed or open sets, add it to the open set.

The code after the part you are expected to write (within the *while* loop) will publish markers for the open and closed sets, and delay to loop to achieve a certain update rate.

After the *while* loop, the code will:

- Display an error if the goal has not been found
- Call the `getPath` method to get the path from the closed set
- Convert the path of node/cell IDs into a vector of `WorldPosition`
- Publish markers for the path
- Set the variables `length_of_path` and `number_of_nodes_in_closed_set` in the response message

Compiling and Running the Node

You will have already created a Catkin workspace for your previous assignment, place the "astar_path_planner" package into the "src" directory.

Compiling the node

Open a terminal.

If you are using the UTS FEIT computer labs, first run:

```
singularity shell /images/singularity_containers/ros-melodic-ar.sif
```

Navigate to the "catkin_ws" folder, e.g.:

```
cd ~/catkin_ws
```

Compile all nodes in the workspace with:

```
catkin_make
```

Running the node

Open a terminal.

If you are using the UTS FEIT computer labs, first run:

```
singularity shell /images/singularity_containers/ros-melodic-ar.sif
```

Set up your environment (you only need to do this once when you open a new terminal):

```
source ~/catkin_ws/devel/setup.bash
```

Run the A* path planner node and other required nodes with:

```
roslaunch astar_path_planner astar_path_planner.launch
```

To set the start and goal positions, use the "2D Pose Estimate" and "2D Nav Goal" buttons in RViz. Click on the button, and then click on somewhere in the map. A green or red sphere will indicate the start and goal positions (shown in Figure 2).

To initiate path planning you need to send a service call. First open a terminal.

If you are using the UTS FEIT computer labs, first run:

```
singularity shell /images/singularity_containers/ros-melodic-ar.sif
```

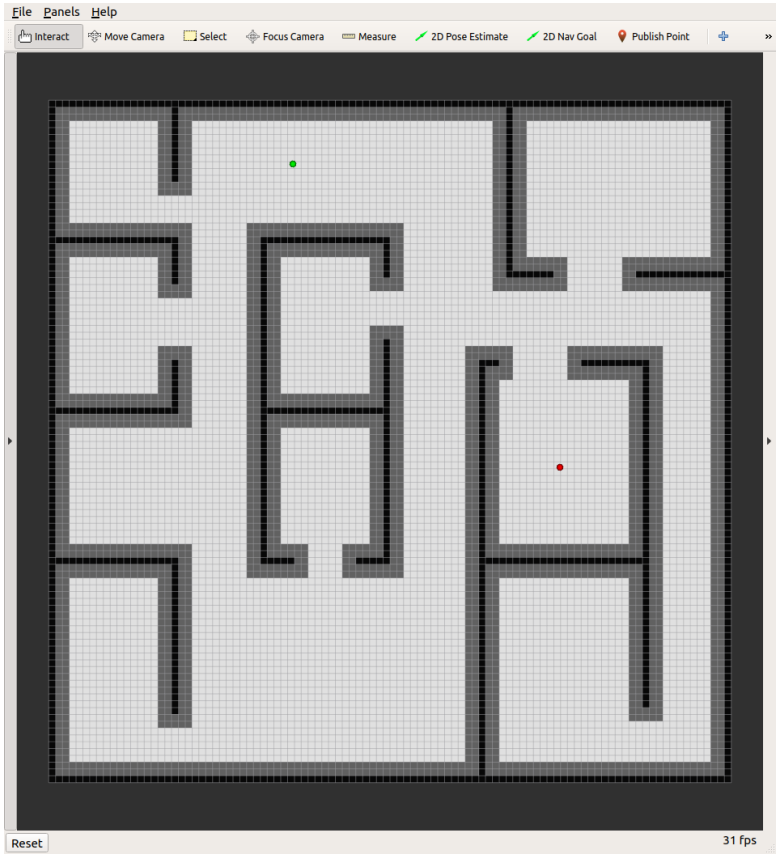


Figure 2: RViz

Set up your environment (you only need to do this once when you open a new terminal):

```
source ~/catkin_ws/devel/setup.bash
```

You need to source the setup script before you can send the service call. To send a service call run:

```
rosservice call plan_path false 0.0
```

“plan_path” is the name of the service. The next argument is diagonal movement: false to disable diagonal movement, true to enable diagonal movement. The last argument is the heuristic cost weight: 0 will behave like Dijkstra’s algorithm, 1 will behave like a conventional A* algorithm.

When the service call finishes it will print out the length of the path, as well as the number of nodes expanded (number of nodes in the closed set). The final task is to describe what happens when the heuristic cost weight is above 1.0. You will need to use the service call with different values for the heuristic cost weight, and observe difference in the length of path and number of nodes expanded. You should also try various start and goal positions.

Tips for developing your code

You can print a node, or the whole open or closed sets at using `ROS_INFO_STREAM`. For example:

```
ROS_INFO_STREAM("\n\nNode: \n" << node);

ROS_INFO_STREAM(open_set);

ROS_INFO_STREAM(closed_set);
```

This will print to the console where “astar_path_planner” is running. Note the `\n` is just a line break to improve readability.

You can also pause execution with:

```
waitForKey();
```

To resume execution press “Enter” in the console where “astar_path_planner” is running.