Path Planning

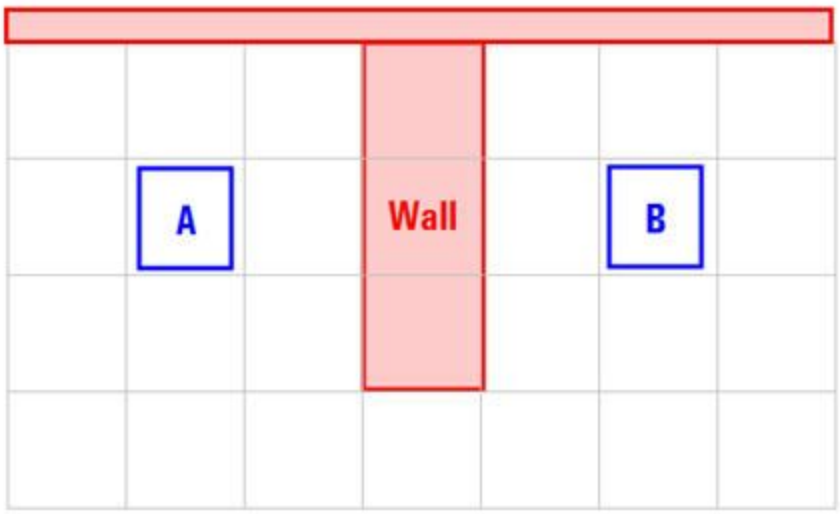
Approach:

The A* path finding algorithm is simple and is able to find an obstacle free path to the goal node if and only if it exists between the start and goal nodes. It is tractable for 2d and small grid sizes.
Given a suitable heuristic function which does not overestimate the minimal cost of reaching the goal from a particular node, it is always optimal exploring the minimum number of nodes necessary to reach the goal state[3]. Since in this particular problem the path planning algorithm runs on a small sized 2d grid, I chose A* as the planning algorithm.

A* is an informed search algorithm in the sense that at every point it takes into consideration the cost of moving to the next cell along the path to the goal. It combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal. In. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

For analysing how these two cost functions g(n) and h(n) affect the performance I read some theory about the algorithm here:
Stanford Heuristics Theory

Let's start with a simple example to demonstrate the concept - suppose the robot wants to move from point A to point B in the most efficient way possible. A wall separates the two points[2].



An open list is maintained in which the non explored neighbours of current grid square are added and eventually evaluated for selection on basis of heuristic for a path towards the goal. The list is sorted as ascending giving the minimum heuristic value as the first member of the list.

The key to determining which nodes to use when figuring out the path is the following equation:

$$F = G + H$$

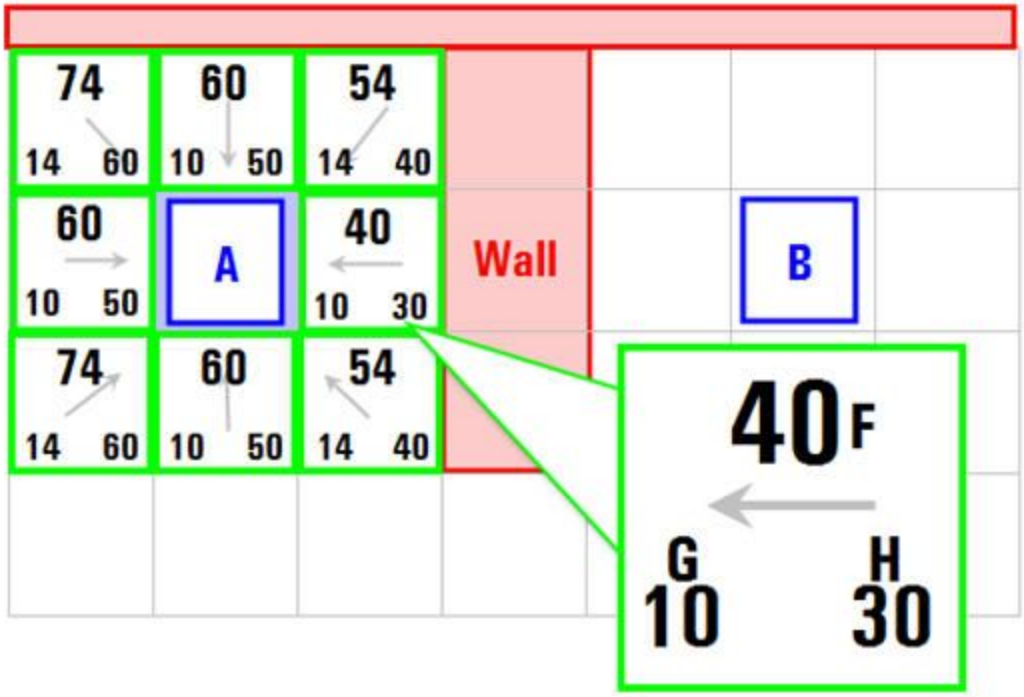
G = the movement cost to move from the starting point A to a given node on the grid
H = the estimated movement cost to move from that given node on the grid to the final destination, point B. This is often referred to as the heuristic

Our path is generated by repeatedly going through our open list and choosing the node with the lowest F score.

G is the movement cost to move from the starting point to the given node using the path generated to get there. In this example, we will assign a cost of 1.0 to each horizontal or vertical node moved, and a cost of 1.4 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 or roughly 1.414 times the cost of moving

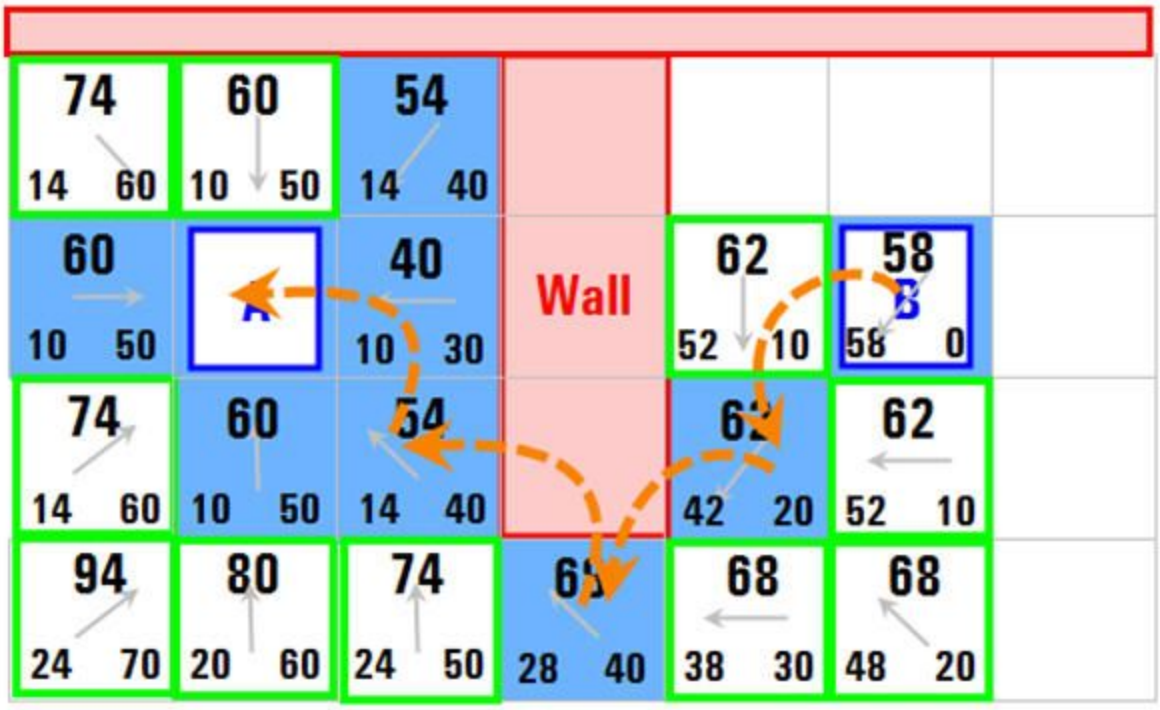
horizontally or vertically, so the ratio is correct and whole numbers are easier to manage (especially in a tutorial).

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of node s moved horizontally and vertically to reach the target node(B) from the current node, ignoring diagonal movement, and ignoring any obstacles that may be in the way.



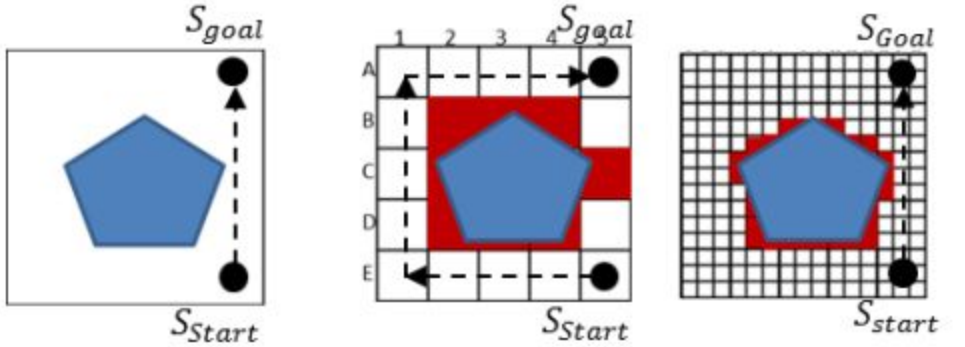
Next, we choose the lowest F score node from all those that are on the open list move it to the closed list. All of the adjacent nodes that are walkable should be added to the open list. The nodes that are on the closed list should be ignored, and nodes that are already on the open list should be evaluated to see if this new path is a better one.

To plan the complete path, you need to continue to repeat this process until you add the target node to the closed list, at which point the map would look like :



Concerns:

A low resolution/coarse grid can be used with fewer nodes to search, and hence permits faster run time. However, coarse discretization of the continuous world can result in the creation of unrealistic paths where an obstacle is close to the boundary



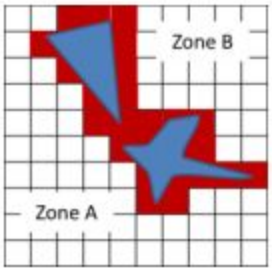
(a) Continuous world

(b) The course gridded world

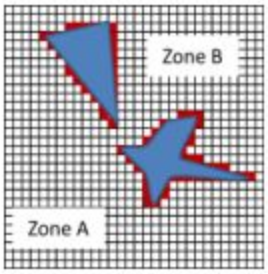
(c)The fine gridded world

It can also place a node beyond reach especially in a cluttered environment

(a) Continuous world



(b) The course gridded world



(c)The fine gridded world

Possible improvements:

The use of a multi-resolution grid representation[4] of the environment is another way to solve this problem in a sparse world. This involves preprocessing the world into a nonuniform grid representation such as quadtrees. Using this approach involves an initially high capital cost in building the world map. However, once that is completed, it could yield good run time performance in terms of search speed.

Simulator

I tried to explore different simulators for the simulation.Looked at V-Rep simulator.Found it to be very versatile.It had support for various languages ,external libraries and plugins.But given the time constraint for the task. I decided not to spend time to figure it out after I invested a day in understanding how would I put all the scripting and simulation together.

I had worked with ros_navigation stack before utilizing its amcl based localization. I researched about custom building planners for path planning and integrating them with the ROS framework. There is a facility provided by ROS where we can implement the BaseGlobalPlanner interface from nav_core package and export that implemented class as a plugin to ROS navigation package.

Implementation

Any new custom built global planner to be added to ROS must adhere to the nav_core::BaseGlobalPlanner C++ interface defined in nav_core package. Once the global path planner is written, it must be added as a plugin to ROS so that it can be used by the move_base package.
Two methods from the interface need to be implemented in the custom planner :

```
void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
 bool makePlan(const geometry_msgs::PoseStamped& start,
                    const geometry_msgs::PoseStamped& goal,
                    std::vector<geometry_msgs::PoseStamped>& plan
                );
```

The first method initializes the costmap and other grid parameters. The second method implements the custom path planning algorithm taking in start and goal as ros::geometry msgs (converted to x,y coordinates and gridSquare indexes on map) and outputting final generated plan to the plan vector.This planned path will then be sent to the move_base global planner module which will publish it through the ROS topic nav_msgs/Path, which will then be received by the local planner module.

The plugin should be registered with BaseGlobalPlanner as:
PLUGINLIB_EXPORT_CLASS(RAstar_planner::RAstarPlannerROS, nav_core::BaseGlobalPlanner).

I followed ROS specific instructions to compile and export the plugin as on the ros wiki:
ROS Writing a Global PlannerPlugin

References

1. Udacity self Driving Car A*
2. A* example in LabView
3. Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Florek, M., Fico, T., & Jurišica, L. (2014). Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, *96*, 59-69.
4. Opoku, D., Homaifar, A., & Tunstel, E. (2013). The Ar-Star (Ar) Pathfinder. *International Journal of Computer Applications*, *67*(8).