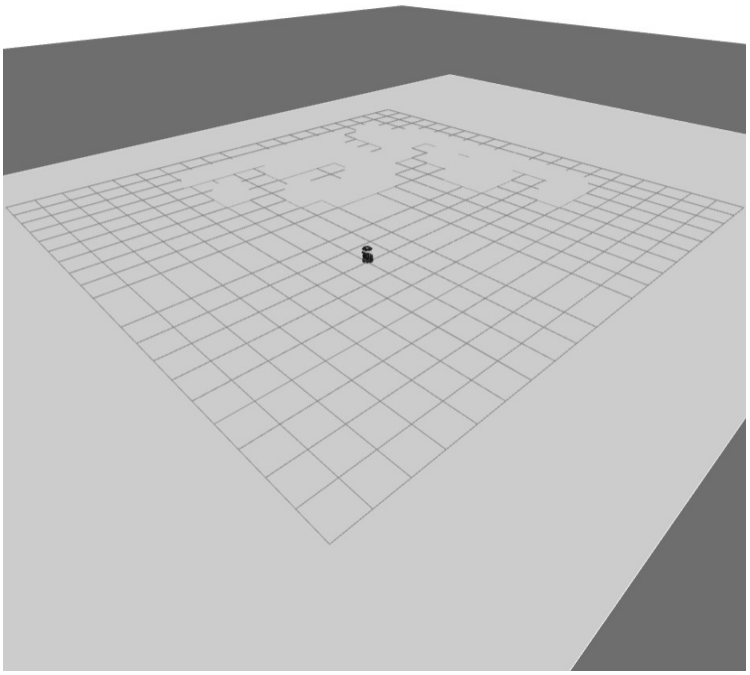
A*-ROS : A Global Path Planning Platform Based on Robot Operating System

[a]Kuldeep Gurjar

[a] School of Robotics, Defence Institute of Advanced Technology,Pune
Mtech: Automation & Robotics
Reg. Number- 21-06-02



Abstract

In this project, Global path planning is done using the Dijkstra algorith & A* search algorithm. ROS and Gazebo simulator is used for simulation.

Introduction

It's no surprise, that navigation apps are popular and widely used in our life.To solve the problem of finding the best route to a destination, all navigation apps rely on a path planning algorithms. Furthermore, path planning algorithms have a broad range of applications in many other significant fields, for example, computer networking, business analytics, video games, gen sequencing, and, no surprise: robotics!
In robotics, path planning is a key component required to solve the larger problem of “autonomous robot navigation”. The main feature of autonomous mobile robots is the unique ability to move independently from a starting point to any point of choice, through a changing environment, without having any collisions.

Robot Navigation can be broken down into the following interrelated subproblems:

- Localization: The robot needs to know where it is
- Mapping: The robot should be able to build a virtual representation of its environment
- Path Planning: The robot needs to be able to plan a route
- Motion Control: The robot has to able to follow a planned route correctly.

A common way of satisfying the requirements of a robust autonomous navigation system is to use a two-level planning architecture. In such systems, a global path planner is paired with a local path planner and both work in a complementary manner.

The *global path planner* is concerned with long range planning and uses the available map information, which can be slow, but is key to finding the most efficient path to a distant goal. It is not concerned with the robot's dynamics or how to avoid unexpected obstacles, which are left to the *local path planner*.

Path planning algorithms:

- Dijkstra algorithm
- Greedy Best-First-Search
- A* search algorithm

I. Dijkstra algorithm

Here's a short description of the process:

Mark your initial node with a g_cost of 0, and add your initial node to open_list

Phase I:
Repeat the following while open_list is not empty:

1. Extract the node with the smallest g_cost from open_list and call it current_node
2. Mark it as visited by adding it into closed_list
3. Check if current_node is the target node, and if so, go to phase II; otherwise, continue with step 5
4. Find the neighbours of the current_node

For each node in the list of neighbours of current_node:

- If a neighbour is inside closed_list, skip it and pick the next neighbour
- If a neighbour is inside open_list:
    - If the new g_cost value is smaller than the current g_cost value:

        - Update its g_cost
        - Update its parent node
- If a neighbour is not inside open_list:
    - Set its g_cost
    - Set its parent node
    - Add it to open_list

When we are done considering all neighbours of current_node, go to step 2.
Phase II:
Build the path from start to end.

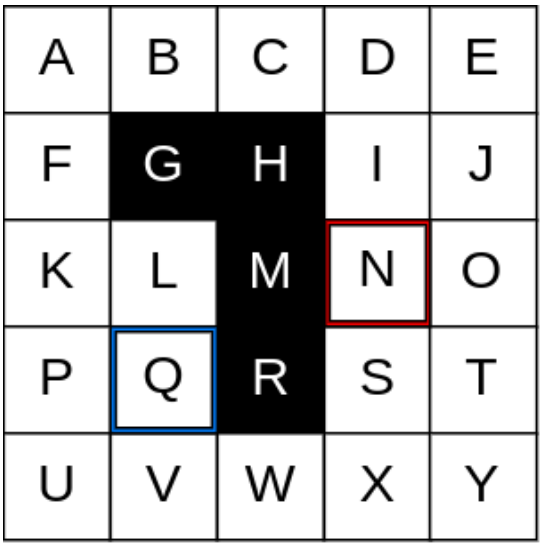
- Trace back from the target node to the start node, using each node's parent node.

To start, assume our robot has a occupancy grid map in advance and is able to self-localize with no error. Recall that an occupancy grid map stores data on free and occupied areas of the robot's environment. Additionally, the robot's start position (where the robot is situated) is known and a goal location (where the robot has to go) is also given.

The image below shows an occupancy grid's white cells that represent free regions, and dark obstacles.From any given grid cell, the robot can only move to adjacent free grid cells and cannot go outside of the map boundaries.

The question we are facing is, what is the sequence of connected waypoints a robot has to follow in order to reach a particular goal location? As shown in grid map, There are many ways a robot can reach the red cell from the blue cell in this 14 × 14 grid map. We want the robot to figure out the best way, that is, the shortest path to move to a chosen destination.

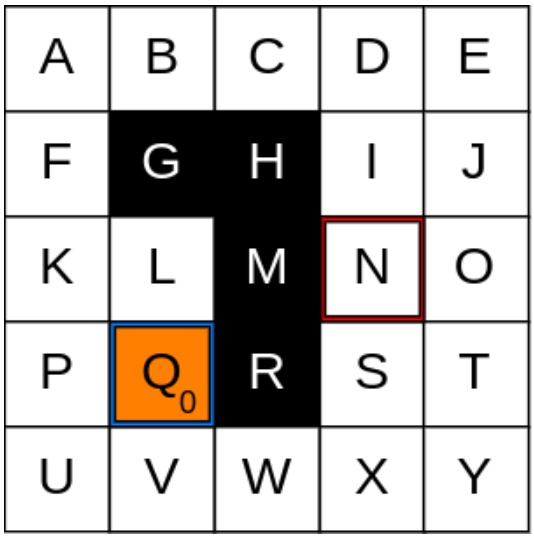
Step by step proceedure-



Suppose we have 5x5 grid map, with letters that identify each of its cells. We are interested in finding the shortest path that starts from the starting point Q, marked with a blue outline, to the goal N, marked with a red outline. Black cells are considered to be occupied cells and cannot be traversed.

A key component of Dijkstra's algorithm is that it always keeps track of the shortest distance from the start node to each individual cell. We will refer to this value as the the g_cost of a node and display it on each grid cell's lower right corner. Before the search process starts, the starting point (node Q) gets a g_cost of 0 (as its distance to itself is 0). Later, as the algorithm progresses, these values will be updated to the actual shortest distance from the start node.Aditionally, we put node Q into a so-called open_list. Nodes that are inside the open_list are the color orange.

Iteration steps :



- Pick a current node

  The iterative search process starts by picking the node inside open list with the lowest g_cost. We call this node current node. At this moment, open list only contains node Q. We will use a bold black outline to identify the current node in the diagrams.

- Neighbours of the current node

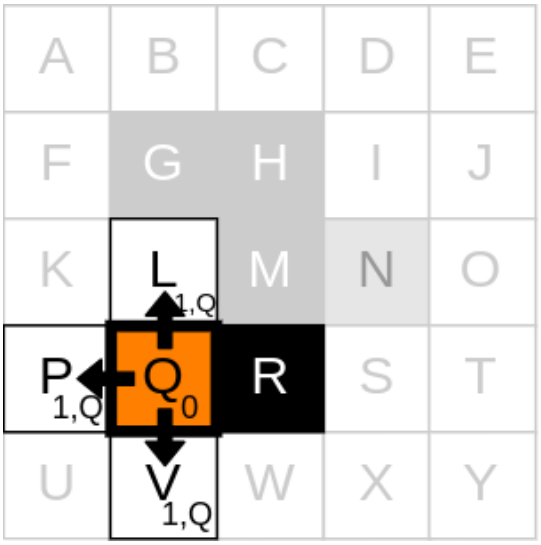
  We continue by examining adjacent nodes that are direct neighbours of the current node. We will only consider traversable grid cells above, below, to the right, and to the left of the current node to be neigbours. Neighbours of our current node can be identified by the arrows pointing from the current node towards neighbour grid cells that are not an obstacle (grid cells L, V, and P).

- Update travel distance values, store parent node

Once we have identified all neighbour nodes in free space, we update their g_cost. Then, we set the parent node of each neighbour. The parent is the node from which the update of the g_costvcame from (which is always the current current node). On all diagrams, we will show each grid cell's parent node at the bottom of a grid cell, to the right of its g_cost. To conclude, we put each neighbour inside open list.

This is a detailed description of how we would proceed in the case of the neighbours of node Q:

Let's begin with neighbour L, as there is no reason to prefer one neighbour node over another



- We add the *g_cost* of the current node (in this case, 0) with the step-cost. The step-cost is the cost incurred when moving to a neighbor cell. We assume that each grid cells side length is 1, therefore, all step-costs will be equal to 1. We set 1 as the g_cost of L.
- We set its parent node, which is Q.
- Now we can put this neighbour inside open list.

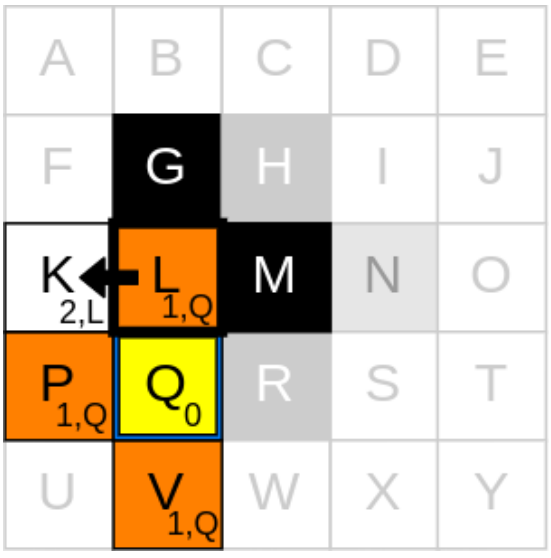
Now, We continue with adjacent cell V

- We add 0 (the *g_cost* of Q, our current node) with 1 to obtain 1. Therefore, we set the *g_cost* of V to 1.
- We set as parent node Q.
- Finally, we add V to open list.

Apply the same steps again for P

We are done considering all neighbours of Q, therefore, we can mark node Q as visited. Visited nodes will be kept in a separate list called closed_list and will be shown in yellow. After the above steps, Dijkstra's algorithm has completed one full iteration cycle.

New cycle, repeat -pick a new current node, check neighbours, update g_cost and parent node



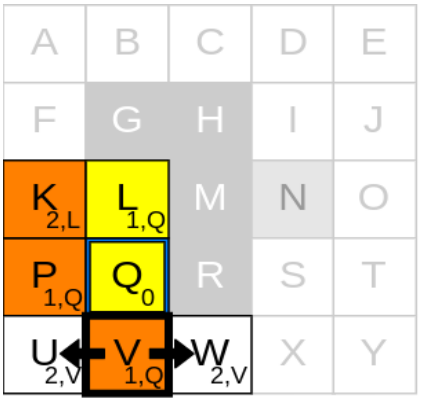
Now we are at the beginning of a new iteration cycle, so we need to select a new current nodeto explore. We pick a node from inside open list. It has to be one of the nodes with the smallest g_cost value. That could be node L, V, or P since all have a *g_cost* value of 1. We can pick any of them, so let's pick L and mark it with a bold black outline to identify it as the current node.
Now we repeat the process by updating the neighbours of L. We ignore nodes inside closed list (in this case, we ignore node Q) and occupied cells (G and M), therefore, we can only process neighbour node K.
This is how we update g_cost and parent node of node K:

- Add 1 (the *g_cost* of L, our current node) with 1 (the step-cost when moving from L to K) to obtain 2.
- Set the *g_cost* of K to 2.
- Set L as parent node of K.
- We add K to open_list.

To wrap up this iteration cycle and prepare for the next one, add L to closed list. In the next iteration cycle, L shows up yellow to represent that it has been visited.

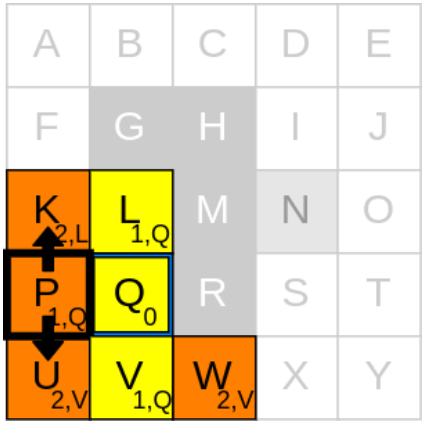
Iteration cycle 3

Pick a new current node to analyze its neighbours nodes. We can take either P or V; both nodes are inside open list and have the smallest *g_cost* (1). Take V, for instance, and set the *g_cost* of its neighbours U and W. Both get a *g_cost* of 2 and V as a parent node. Add both U and W to open list (they show up orange on the next iteration cycle). Add V to closed list (it will show up yellow on the next iteration).



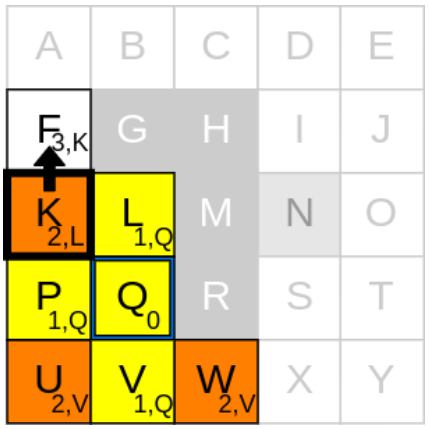
Iteration cycle 4

- Select a new current node, in this case, only P is eligible (node in open list with the smallest g_cost value).
- Notice that both neigbours, K and U, already have a g_cost value. In this case, we have to compare if the new g_cost is lower or not. As the new *g_cost i*s also 2, it is not neccesary to updates K and U's previous *g_cost*. Likewise, the parent node is not updated and remains as it was.
- Finally, mark P as visited by adding it to closed list.



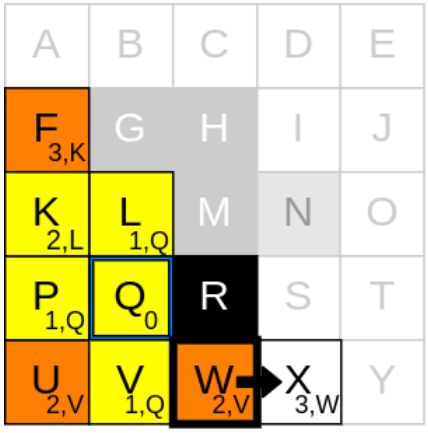
Iteration cycle 5

- Get a new current node, in this case K.
- Get the neighbour nodes of K, F is the only non-visited neigbour cell.
- Set F's g_cost value, Which we get by adding 2 + 1 = 3.
- Set F's parent node (K).
- Add F to open list
- Add K to closed list.



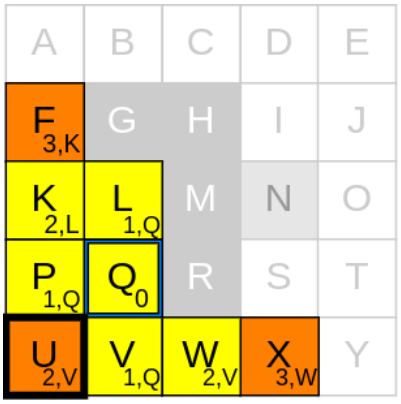
Iteration cycle 6

- Now, the current node is W.
- Neighbours: From W we can only visit X, as V has already been marked as visited and, therefore, there is no need to visit it ever again.
- Update the current g_cost value of X.
- Set the parent node of X, which is W.
- Add X to open list.
- Put W into closed list.



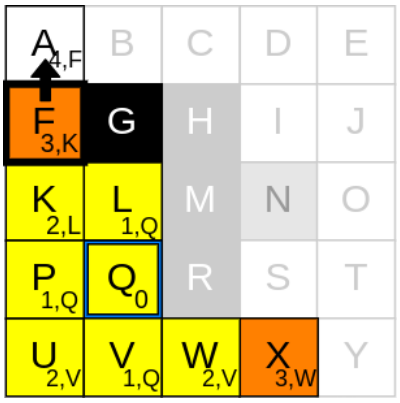
Iteration cycle 7

- Pick a new current node,node U is the only candidate node eligible.
- Since U doesn't have any non-visited neighbours, there are no further steps other than adding it to closed list.



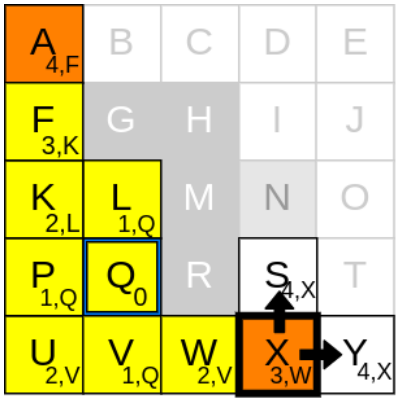
Iteration cycle 8

- Select a new current node from among the candidate nodes F and X.
- Say we pick F, from which we can only access neighbor A.
- Update A's current g_cost to 4.
- Set the parent of A, which is F.
- Add A to open list and F to closed_list.



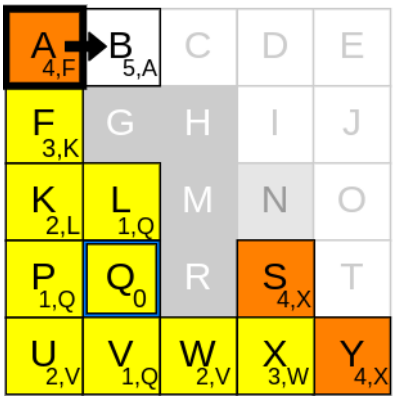
Iteration cycle 9

- Pick a new current node: we are now at X.
- Find the neigbours (S and Y) and set their corresponding g_cost and parent.
- Mark X as visited and put S and Y into open list



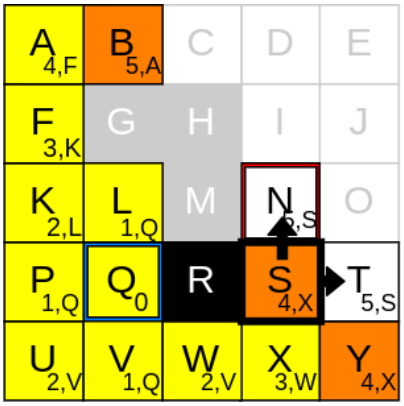
Iteration cycle 10
Almost there. We could pick for the next node A, S, or Y.

- Take A and set B's g_cost to 5 and its parent node A.
- Then mark A as visited and put B into open list



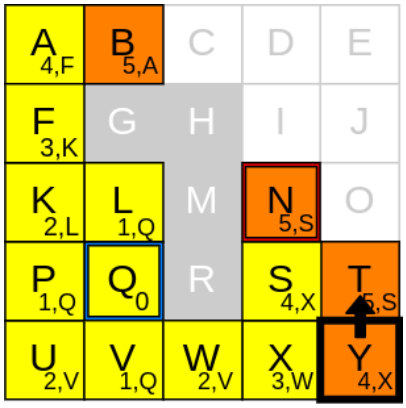
Iteration cycle 11

- Set S as the current node.
- Check each neighbour of S, and for each one, set the g_cost and parent.
- Add S to closed list and its neigbours into open list.



Iteration cycle 12

- Set Y as the current node.
- We compare the new g_cost of T with its previous value, and since the new value is not lower, there is nothing to update other than putting Y into closed list.



Iteration cycle 13

- Set N as the current node.
- We have found node N and our goal!

We are not doned yet, Dijkstra's algorithm is divided into two phases:

- Explore the map to collect data
- Utilize the collected data to build the shortest path

At the moment, we have finished phase 1.

By examining neighbors and deciding which nodes are parent to which other nodes, we build a representation of the connectivity of the free space. We also confirmed that a path to the goal location exists. However, we have not yet built the sequence of nodes that a robot would need to visit to move from start to goal, so let's do that now.

Once we have reached the target node, we can extract the shortest route by following every single node's parent node until we reach the start node.

We first take the target node N and add it to a new list. Then we look up the parent for that node. We add node's N parent to the list, which is S. In the same way, we pick the parent's parent node and store it, too. In this case, the parent of S is X, and we add X to the list. We keep iterating this process until we get the start node Q, which has no parent node. Once the backtracking is complete, the newly created list will contain the shortest path as a sequence of grid cells to visit, but in the wrong order,

To have the path from start to end, so all we need to do is reverse it

Results: Video Dijkstra.mp4

Limitations of Dijkstra:

- Dijkstra's algorithm is an uninformed search algorithm, also known as blind search. It means that during the expansion process, it doesn't know if one grid cell is better than the other. This causes the algorithm to search in all directions in a radial pattern originating from the starting node. This makes it generally slower than informed search strategies, especially as the map size goes up.

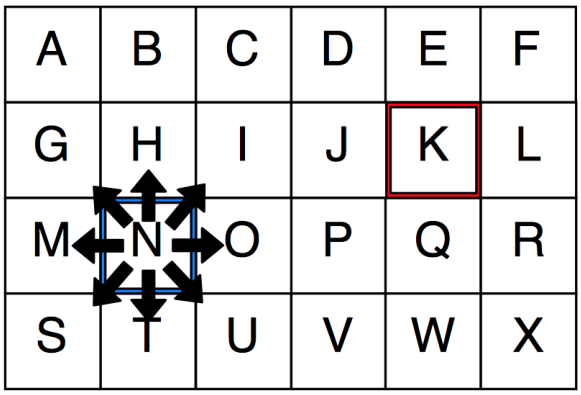
Video Dijkstra.mp4

- Because Dijkstra expands to a very large number of nodes, it is a very cpu-intensive algorithm. This can result in a path search process that is slow or fails to find a path in time.
  The further the search expands, the more memory is needed. This can sometimes be a problem since memory requirements increase exponentially as the map size or space dimensionality increases. One important source of memory usage is the open and closed list.
- Dijkstra assumes that the environment is static and not dynamic. This means that Dijkstra produces a rigid path. As a result, if the environment changes, the robot could be moving along a path that is potentially obsolete. One possible solution to this issue is to continuously recalculate the path with an updated map.

II. Informed search algorithms:

- Greedy Best-First Search
- A* Search algorithm

Informed Search algorithms utilize the information about the goal location to guide the search towards the target and produce a more efficient exploration of the map. For instance, we know that if we want to drive to a location that is east of us, we should search a road that also goes to the east because roads going to the west will most likely take us further away from our target. Have a look at this small grid map below



In this image, say that we know that we want to go to node 'K'. Traversing through node 'M' is not a good idea because it will take us away from node 'K'. We prefer to explore node 'I' or 'O' over 'M' because we intuitively understand that we will be moving closer to 'K' if we do so. Actually, what we are doing is using the information about the the location of 'K' to steer the direction of the search. This is called informed search.

In the example, we can easily look at the best nodes to expand our search. However, what if our map is much bigger - let’s say 10,000 nodes? It wouldn’t be as easy for us to decide which node to explore next. How can we quickly find which nodes are the ones that will lead us faster to the goal? And how do we express this in code? Here is where the concept of Heuristic comes in picture.

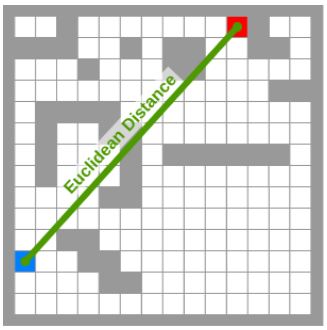
Heuristics:

A heuristic is a method that serves us to solve a problem with an approximate solution. This solution is not optimal, but has the advantage that it can be calculated very quickly and helps to solve the main problem at hand.

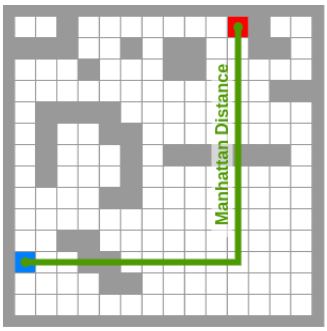
- **Euclidean distance**

  The most common heuristic method for approximating the travel distance from one point to another is the Euclidean distance.

  So this is how we calculate the Euclidean distance between two points (x1,y1) and (x2,y2):



- **Manhattan Distance**



The Manhattan Distance is the distance between two points measured along the axes at right angles. It derives its name from the fact that the distance is calculated allowing only left/right and up/down movements, just as one would travel through the streets of Manhattan, New York.

In a 2-d space, the Manhattan Distance is defined as the sum of absolute differences between two points (x1,y1) and (x2,y2):

Manhattan Distance is a way to get very, very fast distance approximations without requiring square root calculations.

Greedy Best-First Search

The Greedy Best-First Search (Greedy BFS) algorithm is almost same as Dijkstra's shortest path algorithm, except that it expands its way by selecting the node closest to the goal. Greedy BFS takes into account the goal position to guide its search efforts towards the goal and find a path very quickly.

Limitations:

The problem with Greedy BFS is that it always expands to the nodes that *appear* to be closest to the goal. This behavoir can mislead the algorithm to expand to nodes that look promising, but ultimately are not due to obstacles further down the road. As a result, paths generated by Greedy BFS can become unnecessarily long.

Demo gbfs_concave, Gbfs.mp4 , Gbfs1.mp4

A* Search Algorithm:

A* is arguably the most popular choice for pathfinding, A* has proven to be both efficient and effective as it is quick and also finds the shortest paths.

The key factor to the success of A* is that it combines the distance from the start node to the current node, like Dijkstra does, and the estimated distance from the current node to the goal node, like Greedy Best-First Search does.

These two pieces of information are combined to create a value known as the total cost of the node, which we will denote as f_cost = g_cost + h_cost

Algorithm-

- Start by creating: open_list, closed_list, parents , g_cost, f_cost and shortest_path
- Set the g_cost value for the start node, and determine the heuristic value h_cost for start node.
- Put the starting node into open_list with its corresponding f_cost value.

  Enter a condition-controlled loop that runs the instructions below and exits when open_list is empty-

- Pick the node with the lowest f_cost from open_list. We refer to it as current_node.
- current_node gets deleted from open_list and added to closed_list
- If current_node is the goal, exit the search loop
- Get all neighbour cells of current_node and put them inside a list called neighbors.

  Iterate over each neighbors-

- If neighbor is inside closed_list, ignore it and get the next neighbour. Otherwise, do the following:
  - If it isn’t part of open_list:
    - Add it to open_list.

- Make the current_node the parent of this neighbour.
- Keep a record of the f_cost and g_cost of the node.

- If it is inside open_list:
  - Check to see if this new path is better, using g_cost as the measure.
  - If so, change the parent of the node to be current_node, and recalculate the g_cost and f_cost of the node.

When the algorithm exits the main loop:

- Check if the target was found.
- If so, reconstruct the path. Work backwards from the target node, and go from each node to its parent until you reach the starting node.

Python code:

```
#! /usr/bin/env python

import rospy

def find_neighbors(index, width, height, costmap, orthogonal_step_cost):
 neighbors = []
 # length of diagonal = length of one side by the square root of 2 (1.41421)
 diagonal_step_cost = orthogonal_step_cost * 1.41421
 # threshold value used to reject neighbor nodes as they are considered as obstacles [1-254]
 lethal_cost = 1

 upper = index - width
 if upper > 0:
  if costmap[upper] < lethal_cost:
   step_cost = orthogonal_step_cost + costmap[upper]/255
   neighbors.append([upper, step_cost])

 left = index - 1
 if left % width > 0:
  if costmap[left] < lethal_cost:
   step_cost = orthogonal_step_cost + costmap[left]/255
   neighbors.append([left, step_cost])

 upper_left = index - width - 1
 if upper_left > 0 and upper_left % width > 0:
  if costmap[upper_left] < lethal_cost:
   step_cost = diagonal_step_cost + costmap[upper_left]/255
```

```
      neighbors.append([index - width - 1, step_cost])

  upper_right = index - width + 1
  if upper_right > 0 and (upper_right) % width != (width - 1):
    if costmap[upper_right] < lethal_cost:
      step_cost = diagonal_step_cost + costmap[upper_right]/255
      neighbors.append([upper_right, step_cost])
  right = index + 1
  if right % width != (width + 1):
    if costmap[right] < lethal_cost:
      step_cost = orthogonal_step_cost + costmap[right]/255
      neighbors.append([right, step_cost])
  lower_left = index + width - 1
  if lower_left < height * width and lower_left % width != 0:
    if costmap[lower_left] < lethal_cost:
      step_cost = diagonal_step_cost + costmap[lower_left]/255
      neighbors.append([lower_left, step_cost])
  lower = index + width
  if lower <= height * width:
    if costmap[lower] < lethal_cost:
      step_cost = orthogonal_step_cost + costmap[lower]/255
      neighbors.append([lower, step_cost])
  lower_right = index + width + 1
  if (lower_right) <= height * width and lower_right % width != (width - 1):
    if costmap[lower_right] < lethal_cost:
      step_cost = diagonal_step_cost + costmap[lower_right]/255
      neighbors.append([lower_right, step_cost]
  return neighbors
def indexToWorld(flatmap_index, map_width, map_resolution, map_origin = [0,0]):
    # convert to x,y grid cell/pixel coordinates
    grid_cell_map_x = flatmap_index % map_width
    grid_cell_map_y = flatmap_index // map_width
    # convert to world coordinates
    x = map_resolution * grid_cell_map_x + map_origin[0]
    y = map_resolution * grid_cell_map_y + map_origin[1]
    return [x,y]
```

```
def euclidean_distance(a, b):
    distance = 0
    for i in range(len(a)):
        distance += (a[i] - b[i]) ** 2
    return distance ** 0.5
def manhattan_distance(a, b):
    return (abs(a[0] - b[0]) + abs(a[1] - b[1]))
def a_star(start_index, goal_index, width, height, costmap, resolution, origin, grid_viz):
  # create an open_list
  open_list = []
  # set to hold already processed nodes
  closed_list = set()
  # dict for mapping children to parent
  parents = dict()
  # dict for mapping g costs (travel costs) to nodes
  g_costs = dict()
  # dict for mapping f costs (total costs) to nodes
  f_costs = dict()
  # determine g_cost for start node
  g_costs[start_index] = 0
  # determine the h cost (heuristic cost) for the start node
  from_xy = indexToWorld(start_index, width, resolution, origin)
  to_xy = indexToWorld(goal_index, width, resolution, origin)
  h_cost = euclidean_distance(from_xy, to_xy)
  # set the start's node f_cost (note: g_cost for start node = 0)
  f_costs[start_index] = h_cost
    # add start node to open list (note: g_cost for start node = 0)
  open_list.append([start_index, h_cost])
  shortest_path = []
  path_found = False
  rospy.loginfo('A-Star: Done with initialization')
  # Main loop, executes as long as there are still nodes inside open_list
  while open_list:
    # sort open_list according to the lowest 'f_cost' value (second element of each sublist)
    open_list.sort(key = lambda x: x[1])
    # extract the first element (the one with the lowest 'f_cost' value)
```

```
current_node = open_list.pop(0)[0]
# Close current_node to prevent from visting it again
closed_list.add(current_node)
# Optional: visualize closed nodes
grid_viz.set_color(current_node,"pale yellow")
# If current_node is the goal, exit the main loop
if current_node == goal_index:
  path_found = True
  break
# Get neighbors of current_node
neighbors = find_neighbors(current_node, width, height, costmap, resolution)
# Loop neighbors
for neighbor_index, step_cost in neighbors:
  # Check if the neighbor has already been visited
  if neighbor_index in closed_list:
    continue

  # calculate g value of neighbour if movement passes through current_node
  g_cost = g_costs[current_node] + step_cost
  # determine the h cost for the current neigbour
  from_xy = indexToWorld(neighbor_index, width, resolution, origin)
  to_xy = indexToWorld(goal_index, width, resolution, origin)
  h_cost = euclidean_distance(from_xy, to_xy)
  #h_cost = manhattan_distance(from_xy, to_xy) # uncomment to use manhattan distance instead
  # calculate A-Star's total cost for the current neigbour
  f_cost = g_cost + h_cost
  # Check if the neighbor is in open_list
  in_open_list = False
  for idx, element in enumerate(open_list):
    if element[0] == neighbor_index:
      in_open_list = True
      break
  # CASE 1: neighbor already in open_list
  if in_open_list:
    if f_cost < f_costs[neighbor_index]:
      # Update the node's g_cost (travel cost)
```

```
            g_costs[neighbor_index] = g_cost
            # Update the node's f_cost (A-Star's total cost)
            f_costs[neighbor_index] = f_cost
            parents[neighbor_index] = current_node
            # Update the node's f_cost inside open_list
            open_list[idx] = [neighbor_index, f_cost]
        # CASE 2: neighbor not in open_list
        else:
          # Set the node's g_cost (travel cost)
          g_costs[neighbor_index] = g_cost
          # Set the node's f_cost (A-Star total cost)
          f_costs[neighbor_index] = f_cost
          parents[neighbor_index] = current_node
          # Add neighbor to open_list
          open_list.append([neighbor_index, f_cost])
          # Optional: visualize frontier
          grid_viz.set_color(neighbor_index,'orange')
    rospy.loginfo('A-Star: Done traversing nodes in open_list')
    if not path_found:
      rospy.logwarn('A-Star: No path found!')
      return shortest_path
    # Reconstruct path by working backwards from target
    if path_found:
        node = goal_index
        shortest_path.append(goal_index)
        while node != start_index:
            shortest_path.append(node)
            node = parents[node]
    # reverse list
    shortest_path = shortest_path[::-1]
    rospy.loginfo('A-Star: Done reconstructing path')
    return shortest_path
```

Demo Astar.mp4

A*’s Limitation:

A* is a complete algorithm, which is good because it means that it will always find that solution if a solution exists. A* is also an optimal algorithm, good again, because this means that it will always find the shortest path if one exists. But these features come at a cost. To find a complete and optimal solution, a so-called deterministic algorithm is required. A* is such a deterministic path planning algorithm, which means that it always produces (on a given start, goal and map input) the exact same path, following the exact same computation steps. Unfortunately, deterministic algorithms do not scale well with the map size. The more nodes to process, the more difficult it becomes to keep up with the planning time requirements

So, is it possible to create an algorithm that is faster and more memory-efficient than A*?

Yes, but we will have to give up on optimality and completeness to win in computational efficiency. In some cases, it can be convenient, in other cases we have no choice but to make that sacrifice. Rapidly-Exploring Random Tree (RRT) is used in such cases, which belongs to a famlily of algorithms called probabilistic algorithms.

Demo empty dijkstra, gbfs, a*

Thank
you