Navigation

So far you have learned how some planning algorithms work. ROS implements some of these algorithms, so we do not have to worry about implementing them from scratch. But where is the fun in that, so let's go through some of the algorithms, and understand what would be the steps to implement a planning algorithm on a "real environment".

The following notebook is aimed explain:

1. The reading process of the *.pgm files generated during the mapping stage in ROS.
2. The concept of Node, Tree and how that is related to the the planning algorithms.
3. The implementation fo the planning algorithms

Map Loading

In order to create a plan (at least a global one), it is necessary to have a map to work on. For that, we will make use of the map we obtained during the navigation course. We can use the information in the pgm and yaml files as follows:

```
from google.colab import files
uploaded = files.upload()

from PIL import Image, ImageOps

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm


import yaml
import pandas as pd

from copy import copy, deepcopy
import time

class Map():
    def __init__(self, map_name):
        self.map_im, self.map_df, self.limits = self.__open_map(map_name)
        self.image_array = self.__get_obstacle_map(self.map_im, self.map_df)

    def __repr__(self):
        fig, ax = plt.subplots(dpi=150)
        ax.imshow(self.image_array,extent=self.limits, cmap=cm.gray)
        ax.plot()
```

```
        return ""

    def __open_map(self,map_name):
        # Open the YAML file which contains the map name and other
        # configuration parameters
        f = open(map_name + '.yaml', 'r')
        map_df = pd.json_normalize(yaml.safe_load(f))
        # Open the map image
        map_name = map_df.image[0]
        im = Image.open(map_name)
        size = 200, 200
        im.thumbnail(size)
        im = ImageOps.grayscale(im)
        # Get the limits of the map. This will help to display the map
        # with the correct axis ticks.
        xmin = map_df.origin[0][0]
        xmax = map_df.origin[0][0] + im.size[0] * map_df.resolution[0]
        ymin = map_df.origin[0][1]
        ymax = map_df.origin[0][1] + im.size[1] * map_df.resolution[0]

        return im, map_df, [xmin,xmax,ymin,ymax]

    def __get_obstacle_map(self,map_im, map_df):
        img_array = np.reshape(list(self.map_im.getdata()),(self.map_im.size[1],self.
        up_thresh = self.map_df.occupied_thresh[0]*255
        low_thresh = self.map_df.free_thresh[0]*255

········for·j·in·range(self.map_im.size[0]):
············for·i·in·range(self.map_im.size[1]):
················if·img_array[i,j]·>·up_thresh:
····················img_array[i,j]·=·255
················else:
····················img_array[i,j]·=·0
········return·img_array


print(Map('my_map'))
```

Graph Representation and Map Conversion

In order to make use of any of the path planning algorithms, it is important to convert the obtained map into a more useful notation. In general, the path planning algorithms work on the basis of a data structure called Graph. A graph is an abstract data type that consists of a finite set of vertices (nodes) and edges that connects them. The edges can have or not direction, which makes the graph directed or undirected respectively. Additionally, it is common to see values associated to the edges, which brings the concept of "cost" or "weight" to the graph.

The main objective is then to implement a processing tool capable of converting the image map representation into a graph, and then solve the planning problem from the map graph representation.

Node/Tree Visualization

A useful tool for graph visualization is Graphviz, which can be used through the `edge` and `node` APIs to create a graph structure as the one shown below.

```
from graphviz import Graph
g = Graph('G')
```

```
g.node('a','a',style='filled')
g.node('b','b')
g.node('c','c')
g.node('d','d')
g.node('e','e')

g.edge('a','b',shape='none')
g.edge('a','c')
g.edge('c','d')
g.edge('c','e')

g
```

```
class Queue():
    def __init__(self, init_queue = []):
        self.queue = copy(init_queue)
        self.start = 0
        self.end = len(self.queue)-1

    def __len__(self):
        numel = len(self.queue)
        return numel

    def __repr__(self):
        q = self.queue
        tmpstr = ""
        for i in range(len(self.queue)):
            flag = False
            if(i == self.start):
                tmpstr += "<"
                flag = True
            if(i == self.end):
                tmpstr += ">"
                flag = True

            if(flag):
```

```
                tmpstr += '| ' + str(q[i]) + '|\n'
            else:
                tmpstr += ' | ' + str(q[i]) + '|\n'

        return tmpstr

    def __call__(self):
        return self.queue

    def initialize_queue(self,init_queue = []):
        self.queue = copy(init_queue)

    def sort(self,key=str.lower):
        self.queue = sorted(self.queue,key=key)

    def push(self,data):
        self.queue.append(data)
        self.end += 1

    def pop(self):
        p = self.queue.pop(self.start)
        self.end = len(self.queue)-1
        return p

class Node():
    def __init__(self,name):
        self.name = name
        self.children = []
        self.weight = []

    def __repr__(self):
        return self.name

    def add_children(self,node,w=None):
        if w == None:
            w = [1]*len(node)
        self.children.extend(node)
        self.weight.extend(w)

class Tree():
    def __init__(self,name):
        self.name = name
        self.root = 0
        self.end = 0
        self.g = {}
        self.g_visual = Graph('G')

    def __call__(self):
        for name,node in self.g.items():
            if(self.root == name):
                self.g_visual.node(name,name,color='red')
```

```
            elif(self.end == name):
                self.g_visual.node(name,name,color='blue')
            else:
                self.g_visual.node(name,name)
            for i in range(len(node.children)):
                c = node.children[i]
                w = node.weight[i]
                #print('%s -> %s'%(name,c.name))
                if w == 0:
                    self.g_visual.edge(name,c.name)
                else:
                    self.g_visual.edge(name,c.name,label=str(w))
        return self.g_visual

    def add_node(self, node, start = False, end = False):
        self.g[node.name] = node
        if(start):
            self.root = node.name
        elif(end):
            self.end = node.name

    def set_as_root(self,node):
        # These are exclusive conditions
        self.root = True
        self.end = False

    def set_as_end(self,node):
        # These are exclusive conditions
        self.root = False
        self.end = True




a = Node('a')
b = Node('b')
c = Node('c')
d = Node('d')
e = Node('e')
f = Node('f')

a.add_children([c],[1])
b.add_children([c,e],[1,1])
c.add_children([b,e,d],[1,3,1])
e.add_children([b,c],[1,3])
d.add_children([c],[1])

tree = Tree('tree')

tree.add_node(a,start=True)
tree.add_node(b)
tree.add_node(c)
```

```
tree.add_node(d)
tree.add_node(e,end=True)
tree.add_node(f)


tree()
```

Breadth First Search Algorithm

The breadth first search goes through the nodes in an unweighted graph and keeps a queue of the visited and unvisited nodes. For that, BFS checks the child nodes at every iteration and adds them to the queue, if a particular child node was already in the queue, it just skips to the next. At the end of the iteration, the parent node is marked as visited and it goes to the next entry in the queue. In the graph above, if we consider the node 'a' as the start node, then the queue would look like this after the first iteration.

1st iteration

```
c
a <- visited
```

2nd iteration

```
d
e
b
c <- visited
a <- visited
```

3rd iteration

```
d
e
b <- visited
c <- visited
a <- visited
```

```
class BFS():
    def __init__(self,tree):
        self.q = Queue()
        self.visited = {name:False for name,node in tree.g.items()}
        self.via = {name:0 for name,node in tree.g.items()}
        self.dist = {name:0 for name,node in tree.g.items()}

    def solve(self,sn):
        self.q.push(sn)
        self.visited[sn.name] = True
        while len(self.q) > 0:
            node = self.q.pop()
            for i in range(len(node.children)):
                c = node.children[i]
                w = node.weight[i]
                if self.visited[c.name] == False:
                    self.q.push(c)
                    self.visited[c.name] = True
                    self.via[c.name] = node.name
                    self.dist[c.name] = self.dist[node.name] + w
            #print(node.name,self.q.queue)
            #print(self.dist)
        return self.via

    def reconstruct_path(self,sn=0,en=0):
        path = []
        node = en.name
        path.append(node)
        dist = self.dist[en.name]
        while True:
            node = self.via[node]
```

```
            if node == 0:
                break
            else:
                path.append(node)
        path.reverse()
        if path[0] != sn.name:
            path = []
        return path,dist
```

```
bfs = BFS(tree)
bfs.solve(tree.g[tree.root])
path,dist = bfs.reconstruct_path(tree.g[tree.root],tree.g[tree.end])
print(path)
print(dist)
```

```
['a', 'c', 'e']
4
```

Dijkstra's Algorithm

Dijkstra's finds the shortest path between two points given a weighted graph. Unlike BFS, Dijkstra's algorithm takes into consideration how difficult is to get from one node to another. Once we pass the graph and the start point to the algorithm it will build a data structure, where we can find the shortest path from the given start point to any other node in the graph (if they are connected). If we only require to find the path to one end point, then the algorithm can be shortened to break once that end point is found.

The key difference of this algorithms lies on the `dist` and `via` lists, which give the ability to propritize the search of the shortest path. This also has the disadvantage that, if no other information is given, the algorithm will start looking for the shortest edges, regardless of where the target/end node is.

```
a = Node('a')
b = Node('b')
c = Node('c')
d = Node('d')
e = Node('e')
f = Node('f')

a.add_children([c],[1])
b.add_children([c,e],[1,1])
c.add_children([b,e,d],[1,3,1])
e.add_children([b,c],[1,3])
```

```
d.add_children([c],[1])

tree = Tree('tree1')
tree.add_node(a,start=True)
tree.add_node(b)
tree.add_node(c)
tree.add_node(d)
tree.add_node(e,end=True)
tree.add_node(f)


tree()
```

```
class Dijkstra():
    def __init__(self,in_tree):
        self.q = Queue()
        self.dist = {name:np.Inf for name,node in in_tree.g.items()}
        self.via = {name:0 for name,node in in_tree.g.items()}
        self.visited = {name:False for name,node in in_tree.g.items()}
        for __,node in in_tree.g.items():
            self.q.push(node)

    def __get_dist_to_node(self,node):
        return self.dist[node.name]

    def solve(self, sn, en):
        self.dist[sn.name] = 0
        while len(self.q) > 0:
```

```
            self.q.sort(key=self.__get_dist_to_node)
            u = self.q.pop()
            #print(u.name,self.q.queue)
            if u.name == en.name:
                break
            for i in range(len(u.children)):
                c = u.children[i]
                w = u.weight[i]
                new_dist = self.dist[u.name] + w
                if new_dist < self.dist[c.name]:
                    self.dist[c.name] = new_dist
                    self.via[c.name] = u.name


    def reconstruct_path(self,sn,en):
        start_key = sn.name
        end_key = en.name
        dist = self.dist[end_key]
        u = end_key
        path = [u]
        while u != start_key:
            u = self.via[u]
            path.append(u)
        path.reverse()
        return path,dist
```

```
dj = Dijkstra(tree)
dj.solve(tree.g[tree.root],tree.g[tree.end])
dj.reconstruct_path(tree.g[tree.root],tree.g[tree.end])
```

```
    (['a', 'c', 'b', 'e'], 3)
```

A* Algorithm

A* can be seen as an extension of Dijkstra's algorithm. While Dijkstra's fails to do a "smart" search, A* introduces an heuristic function h to provide more information to the search process, which is aimed to improve the speed of Dijkstra's. In other words, the search queue in A* is prioritized based on a function $f(n) = g(n) + h(n)$, where $g(n)$ is the `dist` vector containing the shortest distance up to the node n, while $h(n)$ is an heuristic that provides an idea of how "good" is to move to the node n while searching for the path. Thus the score function f weights not just how close is the node n fr om the current node, but how "good" is to move towards that point, which at the end helps to proritize the paths that make more sense.

```
class AStar():
```

```
    def __init__(self,in_tree):
        self.q = Queue()
        self.dist = {name:np.Inf for name,node in in_tree.g.items()}
        self.via = {name:0 for name,node in in_tree.g.items()}
        self.visited = {name:False for name,node in in_tree.g.items()}

        self.in_tree = in_tree
        self.h = {name:0 for name,node in in_tree.g.items()}
        self.f = {name:np.Inf for name,node in in_tree.g.items()}

        for name,node in in_tree.g.items():
            start = tuple(map(int, name.split(',')))
            end = tuple(map(int, self.in_tree.end.split(',')))
            self.h[name] = np.sqrt((end[0]-start[0])**2 + (end[1]-start[1])**2)

        for __,node in in_tree.g.items():
            self.q.push(node)

    def __get_f_score(self,node):
        self.f[node.name] = self.dist[node.name] + self.h[node.name]
        return self.f[node.name]

    def solve(self, sn, en):
        self.dist[sn.name] = 0
        while len(self.q) > 0:
            self.q.sort(key=self.__get_f_score)
            u = self.q.pop()
            #print(u.name,self.q.queue)
            if u.name == en.name:
                break
            for i in range(len(u.children)):
                c = u.children[i]
                w = u.weight[i]
                new_dist = self.dist[u.name] + w
                if new_dist < self.dist[c.name]:
                    self.dist[c.name] = new_dist
                    self.via[c.name] = u.name

    def reconstruct_path(self,sn,en):
        start_key = sn.name
        end_key = en.name
        dist = self.dist[end_key]
        u = end_key
        path = [u]
        while u != start_key:
            u = self.via[u]
            path.append(u)
        path.reverse()
        return path,dist
```

Create A Graph From A Map

```
from google.colab import files
uploaded = files.upload()

class MapProcessor():
    def __init__(self,name):
        self.map = Map(name)
        self.inf_map_img_array = np.zeros(self.map.image_array.shape)
        self.map_graph = Tree(name)

    def __modify_map_pixel(self,map_array,i,j,value,absolute):
        if( (i >= 0) and
            (i < map_array.shape[0]) and
            (j >= 0) and
            (j < map_array.shape[1]) ):
            if absolute:
                map_array[i][j] = value
            else:
                map_array[i][j] += value

    def __inflate_obstacle(self,kernel,map_array,i,j,absolute):
        dx = int(kernel.shape[0]//2)
        dy = int(kernel.shape[1]//2)
        if (dx == 0) and (dy == 0):
            self.__modify_map_pixel(map_array,i,j,kernel[0][0],absolute)
        else:
            for k in range(i-dx,i+dx):
                for l in range(j-dy,j+dy):
                    self.__modify_map_pixel(map_array,k,l,kernel[k-i+dx][l-j+dy],abso

    def inflate_map(self,kernel,absolute=True):
        # Perform an operation like dilation, such that the small wall found during t
        # are increased in size, thus forcing a safer path.
        self.inf_map_img_array = np.zeros(self.map.image_array.shape)
        for i in range(self.map.image_array.shape[0]):
            for j in range(self.map.image_array.shape[1]):
                if self.map.image_array[i][j] == 0:
                    self.__inflate_obstacle(kernel,self.inf_map_img_array,i,j,absolut
        r = np.max(self.inf_map_img_array)-np.min(self.inf_map_img_array)
        if r == 0:
            r = 1
        self.inf_map_img_array = (self.inf_map_img_array - np.min(self.inf_map_img_ar

    def get_graph_from_map(self):
        # Create the nodes that will be part of the graph, considering only valid nod
        for i in range(self.map.image_array.shape[0]):
            for j in range(self.map.image_array.shape[1]):
                if self.inf_map_img_array[i][j] == 0:
                    node = Node('%d,%d'%(i,j))
                    self.map_graph.add_node(node)
```

```
        # Connect the nodes through edges
        for i in range(self.map.image_array.shape[0]):
            for j in range(self.map.image_array.shape[1]):
                if self.inf_map_img_array[i][j] == 0:
                    if (i > 0):
                        if self.inf_map_img_array[i-1][j] == 0:
                            # add an edge up
                            child_up = self.map_graph.g['%d,%d'%(i-1,j)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_up],[
                    if (i < (self.map.image_array.shape[0] - 1)):
                        if self.inf_map_img_array[i+1][j] == 0:
                            # add an edge down
                            child_dw = self.map_graph.g['%d,%d'%(i+1,j)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_dw],[
                    if (j > 0):
                        if self.inf_map_img_array[i][j-1] == 0:
                            # add an edge to the left
                            child_lf = self.map_graph.g['%d,%d'%(i,j-1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_lf],[
                    if (j < (self.map.image_array.shape[1] - 1)):
                        if self.inf_map_img_array[i][j+1] == 0:
                            # add an edge to the right
                            child_rg = self.map_graph.g['%d,%d'%(i,j+1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_rg],[
                    if ((i > 0) and (j > 0)):
                        if self.inf_map_img_array[i-1][j-1] == 0:
                            # add an edge up-left
                            child_up_lf = self.map_graph.g['%d,%d'%(i-1,j-1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_up_lf
                    if ((i > 0) and (j < (self.map.image_array.shape[1] - 1))):
                        if self.inf_map_img_array[i-1][j+1] == 0:
                            # add an edge up-right
                            child_up_rg = self.map_graph.g['%d,%d'%(i-1,j+1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_up_rg
                    if ((i < (self.map.image_array.shape[0] - 1)) and (j > 0)):
                        if self.inf_map_img_array[i+1][j-1] == 0:
                            # add an edge down-left
                            child_dw_lf = self.map_graph.g['%d,%d'%(i+1,j-1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_dw_lf
                    if ((i < (self.map.image_array.shape[0] - 1)) and (j < (self.map.
                        if self.inf_map_img_array[i+1][j+1] == 0:
                            # add an edge down-right
                            child_dw_rg = self.map_graph.g['%d,%d'%(i+1,j+1)]
                            self.map_graph.g['%d,%d'%(i,j)].add_children([child_dw_rg

    def gaussian_kernel(self, size, sigma=1):
        size = int(size) // 2
        x, y = np.mgrid[-size:size+1, -size:size+1]
        normal = 1 / (2.0 * np.pi * sigma**2)
        g =  np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
        r = np.max(g)-np.min(g)
        sm = (g - np.min(g))*1/r
```

```
        return sm

    def rect_kernel(self, size, value):
        m = np.ones(shape=(size,size))
        return m

    def draw_path(self,path):
        path_tuple_list = []
        path_array = copy(self.inf_map_img_array)
        for idx in path:
            tup = tuple(map(int, idx.split(',')))
            path_tuple_list.append(tup)
            path_array[tup] = 0.5
        return path_array
```

```
mp = MapProcessor('my_map')
mp
```

```
<__main__.MapProcessor at 0x7f20f40e8890>
```

```
kr = mp.rect_kernel(5,1)
#kr = mp.rect_kernel(1,1)
mp.inflate_map(kr,True)

mp.get_graph_from_map()

fig, ax = plt.subplots(dpi=100)
plt.imshow(mp.inf_map_img_array)
plt.colorbar()
plt.show()
```

```
start_pt = "95,100"
end_pt = "85,110"


mp.map_graph.root = start_pt
mp.map_graph.end = end_pt

bfs_maze = BFS(mp.map_graph)

start = time.time()
bfs_maze.solve(mp.map_graph.g[mp.map_graph.root])
end = time.time()
print('Elapsed Time: %.3f'%(end - start))

path_bfs,dist_bfs = bfs_maze.reconstruct_path(mp.map_graph.g[mp.map_graph.root],mp.ma
```

```
Elapsed Time: 0.010
```

```
path_arr_bfs = mp.draw_path(path_bfs)
```

```
path_bfs
```

```
['95,100',
 '94,100',
 '93,100',
 '92,100',
 '91,100',
 '90,100',
 '89,100',
 '88,100',
 '87,101',
 '86,102',
 '85,103',
 '85,104',
 '85,105',
 '85,106',
```

```
      '85,107',
      '85,108',
      '85,109',
      '85,110']
```

```
mp.map_graph.root = start_pt
mp.map_graph.end = end_pt
dj_maze = Dijkstra(mp.map_graph)

start = time.time()
dj_maze.solve(mp.map_graph.g[mp.map_graph.root],mp.map_graph.g[mp.map_graph.end])
end = time.time()
print('Elapsed Time: %.3f'%(end - start))

path_djk,dist_djk = dj_maze.reconstruct_path(mp.map_graph.g[mp.map_graph.root],mp.map
```

```
     Elapsed Time: 9.393
```

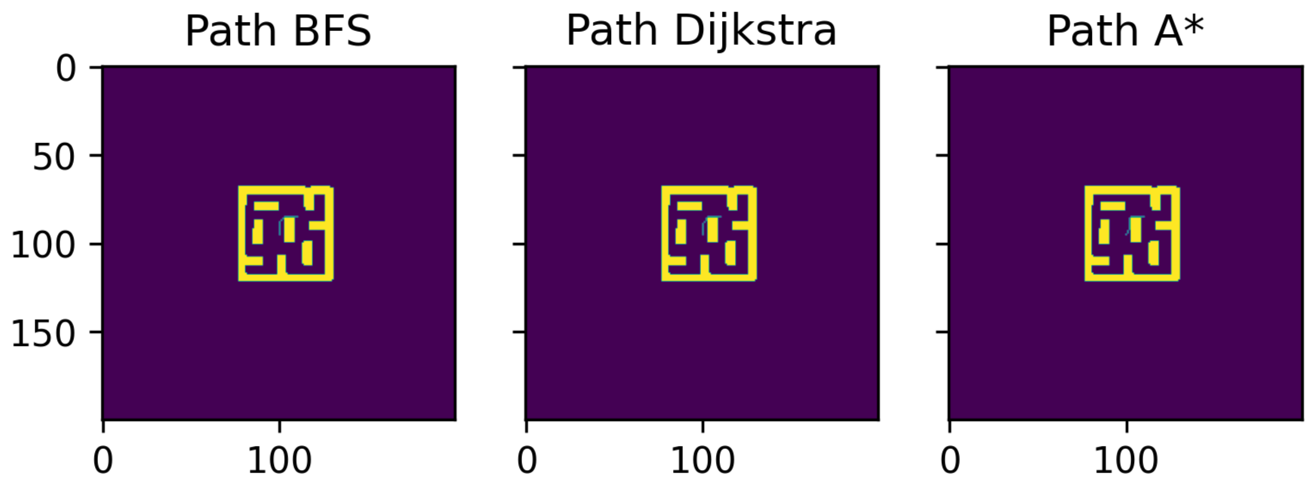
```
path_arr_djk = mp.draw_path(path_djk)
```

```
mp.map_graph.root = start_pt
mp.map_graph.end = end_pt

as_maze = AStar(mp.map_graph)

start = time.time()
as_maze.solve(mp.map_graph.g[mp.map_graph.root],mp.map_graph.g[mp.map_graph.end])
end = time.time()
print('Elapsed Time: %.3f'%(end - start))

path_as,dist_as = as_maze.reconstruct_path(mp.map_graph.g[mp.map_graph.root],mp.map_g
```

```
     Elapsed Time: 2.516
```

```
path_arr_as = mp.draw_path(path_as)
```

```
fig, ax = plt.subplots(nrows = 1, ncols = 3, dpi=300, sharex=True, sharey=True)
ax[0].imshow(path_arr_bfs)
ax[0].set_title('Path BFS')
ax[1].imshow(path_arr_djk)
ax[1].set_title('Path Dijkstra')
ax[2].imshow(path_arr_as)
ax[2].set_title('Path A*')

plt.show()
```




```
print("BFS dist =      ", dist_bfs)
print("Dijkstra dist = ", dist_djk)
print("A star dist =   ", dist_as)
```

```
BFS dist =       18.242640687119287
Dijkstra dist =  18.242640687119284
A star dist =    18.242640687119284
```

✓ 0s completed at 11:42 PM