Sampling-based algorithms – Rapidly-exploring Random Tree (RRT) and Rapidly-exploring Random Tree Star (RRT*)

This document will guide you through the practical work related to path planning algorithms based on sampling-based algorithms. In particular, this practical exercise consists on programming the Rapidly-exploring random tree (RRT) algorithm, and its variant RRT Star (RRT*), to solve a 2D path planning problem. All the code has to be programmed in Python.

Grid map environment

We are going to use grayscale images to define our grid map environments. In Python, the `PIL` library can be used to load a map image and the `matplotlib` library can be used to show it. Transform the image to a 2D `numpy` array to simplify its manipulation. Note that when a grayscale image is used as an environment, 0 is black and it use to represent an obstacle while 1 (or 255) is white and it is normally used to represent the free space. Therefore, we need to binarize the loaded image to ensure that there are no intermediate values as well as to invert the values to have 0 as free space and 1 as obstacles, that is the definition of *free* and *occupied* space that we are going to use.

```
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image

# Load grid map
image = Image.open('map0.png').convert('L')
grid_map = np.array(image.getdata()).reshape(image.size[0], image.size[1])/255
# binarize the image
grid_map[grid_map > 0.5] = 1
grid_map[grid_map <= 0.5] = 0
# Invert colors to make 0 -> free and 1 -> occupied
grid_map = (grid_map * -1) + 1
# Show grid map
plt.matshow(grid_map)
plt.colorbar()
plt.show()
```

The following maps with the proposed *start* and *goal* loacations are provided:

grid map name	start	goal
map0.png	(10, 10)	(90, 70)
map1.png	(60, 60)	(90, 60)
map2.png	(8, 31)	(139, 38)
map3.png	(50, 90)	(375, 375)

The RRT algorithm

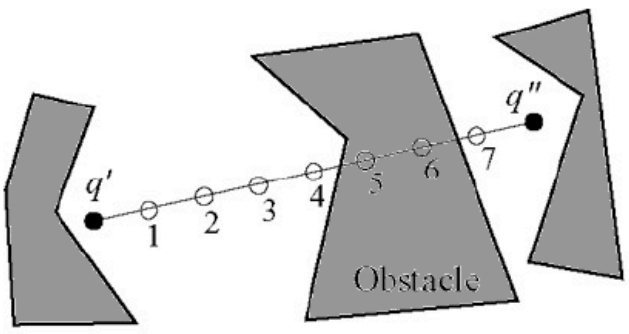
For a general configuration space `c`, the algorithm in pseudocode looks as follows:

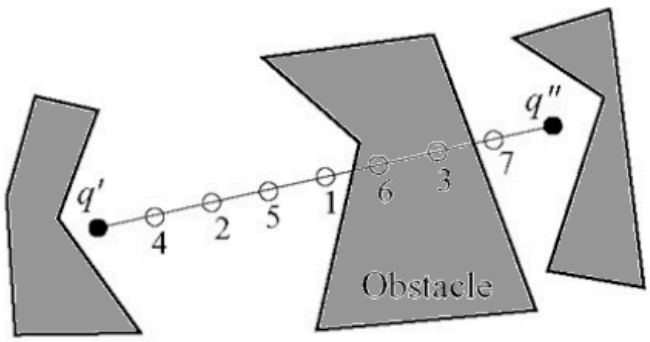
```
function [G, path] = RRT(C, K, Δq, p, qstart, qgoal)
    G.init(qstart)
    for k = 1 to K do
        qrand ← RAND_CONF(C, p, qgoal)
        qnear ← NEAREST_VERTEX(qrand, G)
        qnew ← NEW_CONF(qnear, qrand, Δq)
        if IS_SEGMENT_FREE(qnear, qnew, C)
            G.add_vertex(qnew)
            G.add_edge(qnear, qnew)
            if DISTANCE(qnew, qgoal)<min_dist
                G.add_edge(qnew, qgoal)
                return G
    return "No solution found"
```

where '←' denotes assignment and `return` terminates the algorithm and outputs the following value.

In the algorithm above:

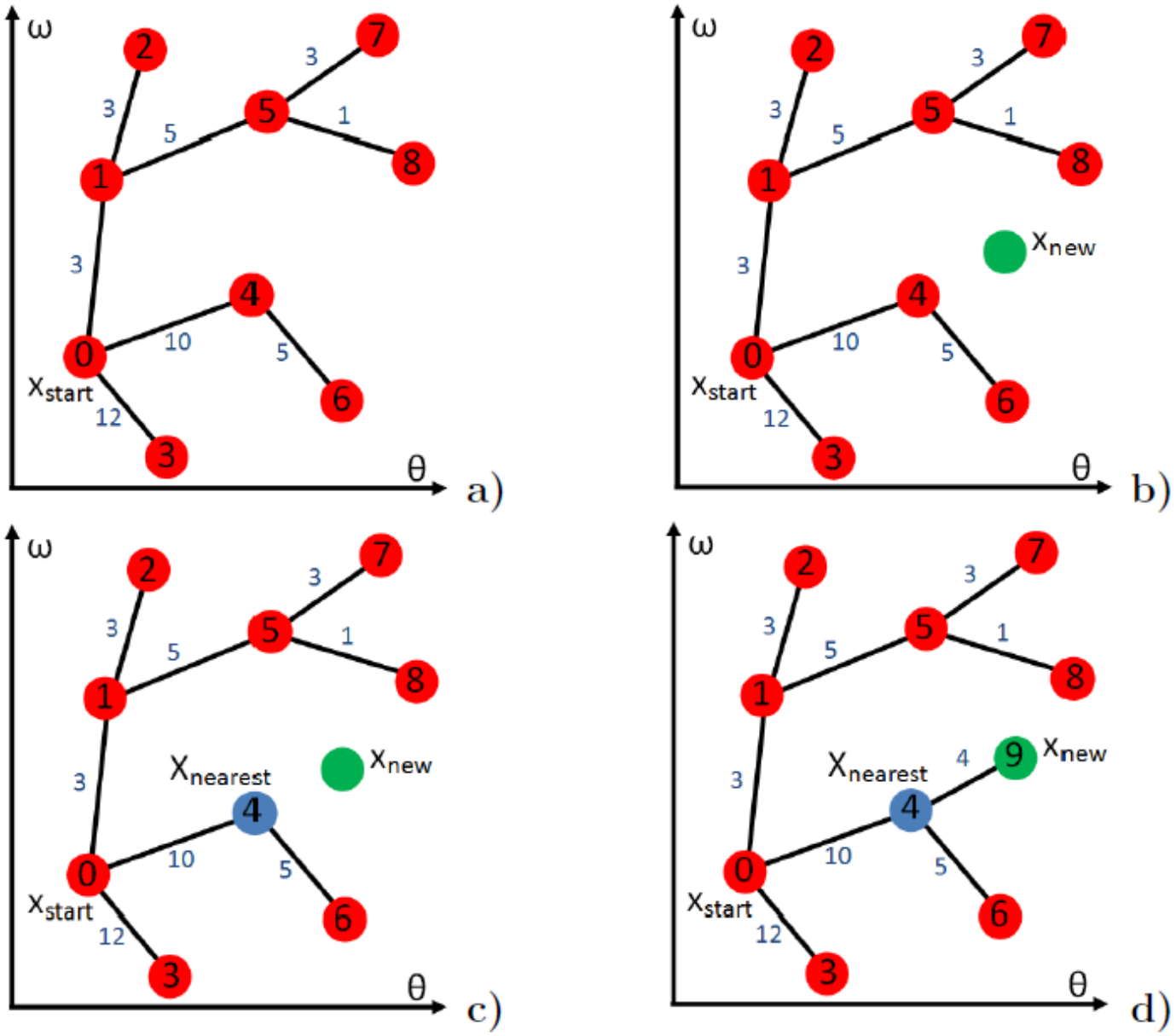
- `RAND_CONF` generates a random configuration `qrand` in `C` with probability $(1-$ `p` $)$ or generates `qrand = qgoal` with probability `p` .
- `NEAREST_VERTEX` is a function that runs through all vertices `v` in graph `G` , calculates the distance between `qrand` and `v` using some measurement function (in our case the Euclidean distance) and returns the nearest vertex.
- `NEW_CONF` selects a new configuration `qnew` by moving an incremental distance `Δq` from `qnear` in the direction of `qrand` without overshooting `qrand` .
- `IS_SEGMENT_FREE` checks if the segment defined by the vertices `qnear` and `qnew` crosses only free space in `C` (returns `True`) or if it crosses any obstacle (returns `False`). To check if a segment belongs to the free space, use the incremental (figure left) or bisection (figure right) strategies.





You can use as many points as the distance between both vertices. Despite vertices are real numbers, to check if `v` is *free* or *occupied* in `C` turn these numbers into integers.

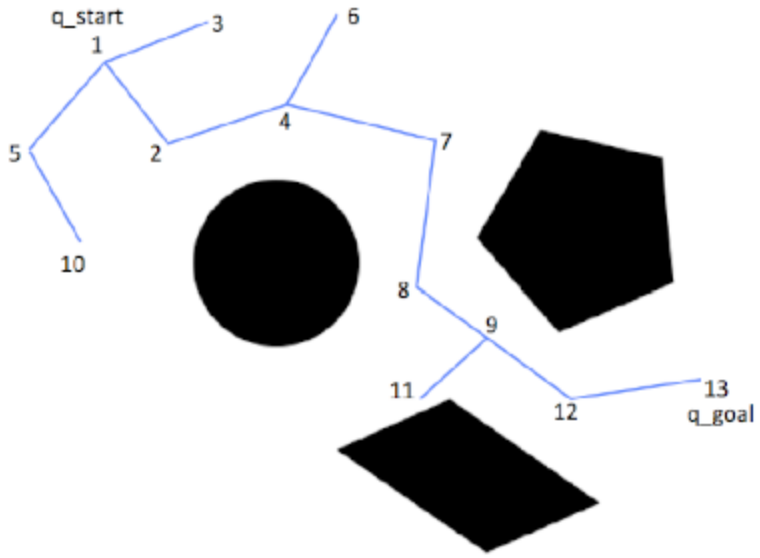
An example of how the RRT algorithm expands a new vertex is sown here (source)



When a new vertex is added to the graph `G`, close to `qgoal` given a minimun distance, then an edge can be formed between this vertex and `qgoal`, and a solution has been found.

To get the path from `start` to `goal` you can use:

- `FILL_PATH` function moves form the `qgoal` (last vertex in the graph `G`) to the `qstart` (initial vertex in `G`) and computes the `path`. This function returns both the graph `G` (if you prefer as a list of 2D vertices and a list of edges containing the indexes to these vertices) and the `path` as a list of indexes to vertices. See the following example:



vertices

index	vertex_x	vertex_y
1	x_1	y_1
2	x_2	y_2
3	x_3	y_3
4	x_4	y_4
5	x_5	y_5
6	x_6	y_6
7	x_7	y_7
8	x_8	y_8
9	x_9	y_9
10	x_{10}	y_{10}
11	x_{11}	y_{11}
12	x_{12}	y_{12}
13	x_{13}	y_{13}

edges

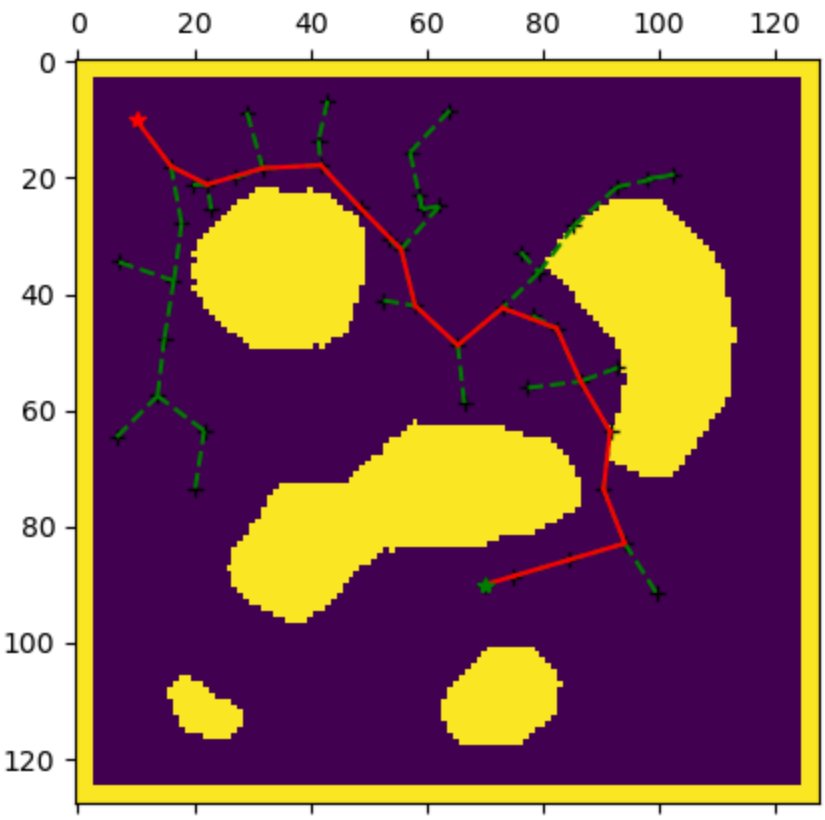
edge_init	edge_end
2	1
3	1
4	2
5	1
6	4
7	4
8	7
9	8
10	5
11	9
12	9
13	12

path

path = [1, 2, 4, 7, 8, 9, 12, 13]

Using `C = 'map0.png'`, `K` =10000, `Δq` =10, `p` =0.2, `qstart` =(10, 10) and `qgoal` =(90, 70) a possible solution is:

Path found in 96 iterations
Distance: 162.09352297574452
PATH to follow:
(10.0, 10.0)
(18.0, 16.0)
(21.17, 22.21)
(18.36, 31.81)
(17.85, 41.8)
(25.19, 48.59)
(32.31, 55.6)
(42.01, 58.03)
(48.81, 65.36)
(42.4, 73.03)
(45.92, 82.39)
(55.03, 86.5)
(63.66, 91.56)
(73.59, 90.44)
(82.9, 94.1)
(85.73, 84.5)
(88.55, 74.91)
(90.0, 70.0)



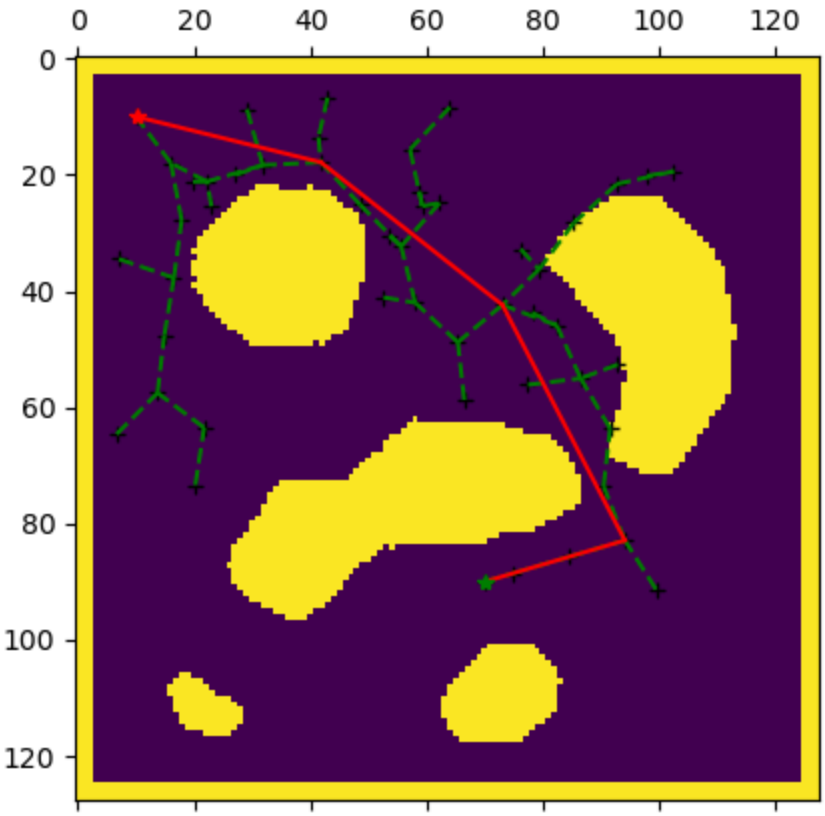
Smoothing

Once you finish the basic RRT algorithm, you must program a smoothing algorithm for obtaining a shorter and less abrupt path. We will use a greedy approach:

> Try to connect from `path[0]` (i.e., `qstart`) to `path[n]` being `n` the last element in the path (i.e. `qgoal`). If the path is invalid (i.e., it crosses an obstacle) try from a closer position (i.e., `path[1]` to `path[n]`, then `path[2]` to `path[n]`, etc.) until a valid path is found. Once `path[i]` to `path[n]` is connected using a valid path, repeat the whole procedure doing `n = i` and `i = 0`. This process is repeated until `n = 0`.

If we smooth the previous obtained path the result is:

```
Smooth distance: 143.24867642790463
Smooth PATH to follow:
(10.0, 10.0)
(17.85, 41.8)
(42.4, 73.03)
(82.9, 94.1)
(90.0, 70.0)
```



As you can see the improvement is significant!

Exercise

Implement both the `RRT` and the `smooting` algorithms and create a script to call them. This script **MUST USE** the following interface:

```
$ ./rrt_YOUR_NAME.py path_to_grid_map_image K Δq p qstart_x qstart_y qgoal_x qgoal_y
```

The script must also plot the generated graph (`vertices` and `edges`) as well as the resulting paths (`path` and `smooth_path`) graphically (i.e., plot the vertices, edges and path over the given grid map). Use 2 different figures if you prefer. Compute also the total path length.

Rapidly-exploring Random Tree Star (RRT*)

A common variation of the RRT is the Rapidly-exploring Random Tree Star (RRT*). RRT* is an incremental sampling-based motion planning approach that is **asymptotically optimal**. It means that when the number of nodes approaches infinity, the RRT* algorithm offers the shortest possible path to the goal.

RRT* algorithm

The RRT* algorithm is similar to the RRT one. However, when the `q_new` configuration is generated, 2 additional optimizations are applied:

- **Cost**: For all the nodes *near* to `q_new` , connect `q_new` not with the *nearest* one but with the one with less cost from `qstart` to `q_new` .
- **Rewire**: For all the nodes *near* to `q_new` , check if it is possible to reach any of this nodes from `q_new` with less cost than with the previous path.

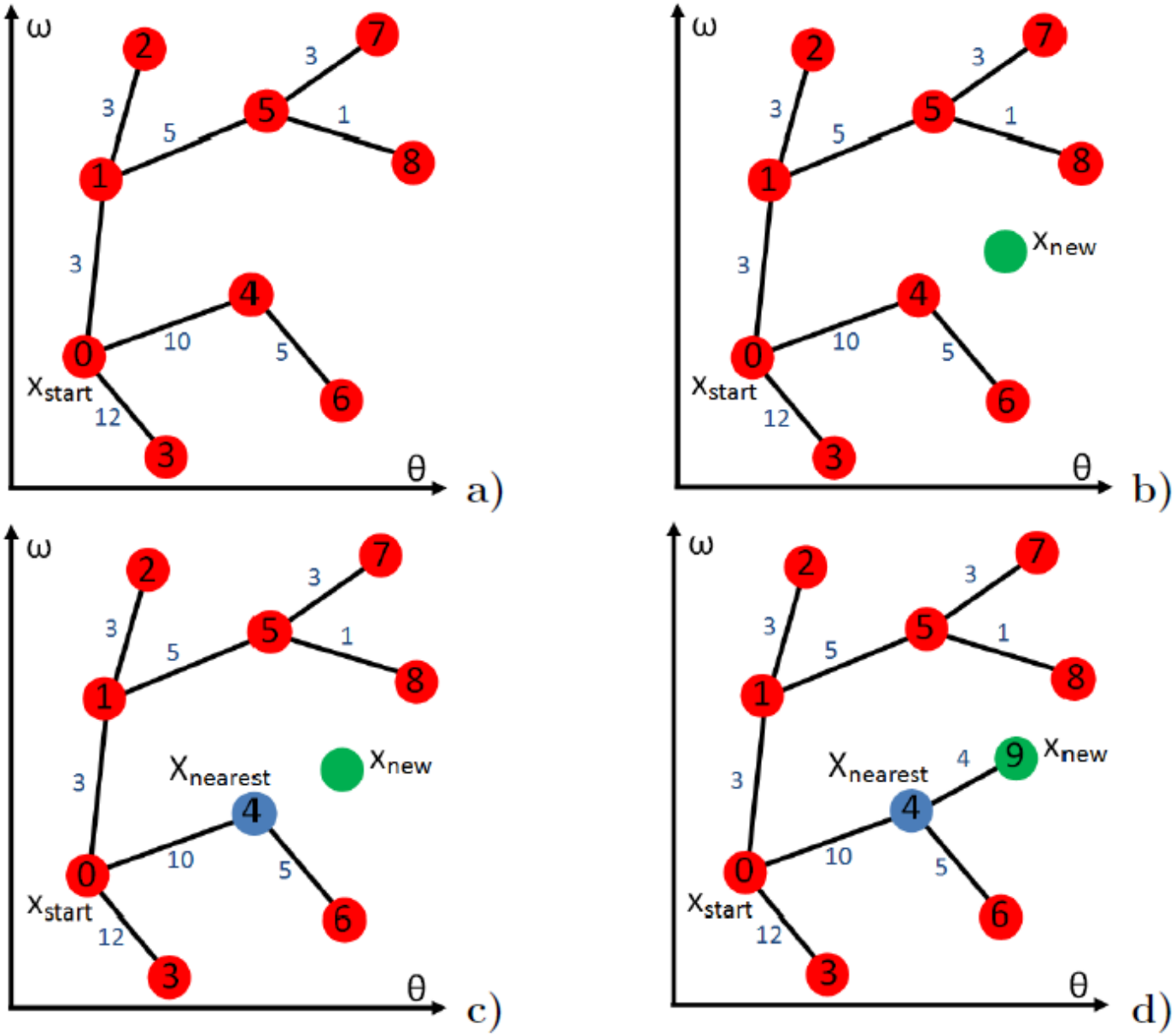
An example of the RRT* pseudocode for the graph extend function is shown next (source). Be aware that it uses `x_rand` , `x_near` , `x_new` , ... instead of `q_rand` , `q_near` , `q_new` , ...

Algorithm 4: $\texttt{Extend}_{RRT^*}(G, x)$

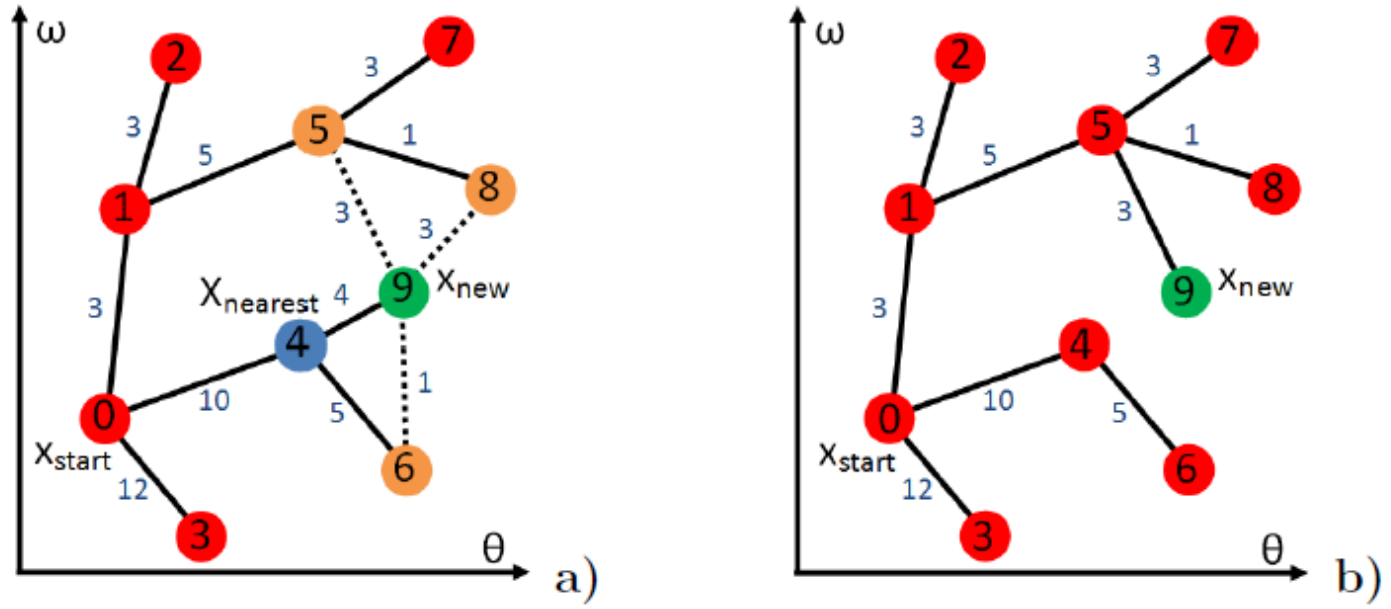
```
1  V' ← V; E' ← E;
2  x_nearest ← Nearest(G, x);
3  x_new ← Steer(x_nearest, x);
4  if ObstacleFree(x_nearest, x_new) then
5      V' ← V' ∪ {x_new};
6      x_min ← x_nearest;
7      X_near ← Near(G, x_new, |V|);
8      for all x_near ∈ X_near do
9          if ObstacleFree(x_near, x_new) then
10             c' ← Cost(x_near) + c(Line(x_near, x_new));
11             if c' < Cost(x_new) then
12                 x_min ← x_near;
13     E' ← E' ∪ {(x_min, x_new)};
14     for all x_near ∈ X_near \ {x_min} do
15         if ObstacleFree(x_new, x_near) and
           Cost(x_near) > Cost(x_new) + c(Line(x_new, x_near))
           then
16             x_parent ← Parent(x_near);
17             E' ← E' \ {(x_parent, x_near)};
               E' ← E' ∪ {(x_new, x_near)};
18 return G' = (V', E')
```

Also, a graphical example can be visualized in the following figures (source)

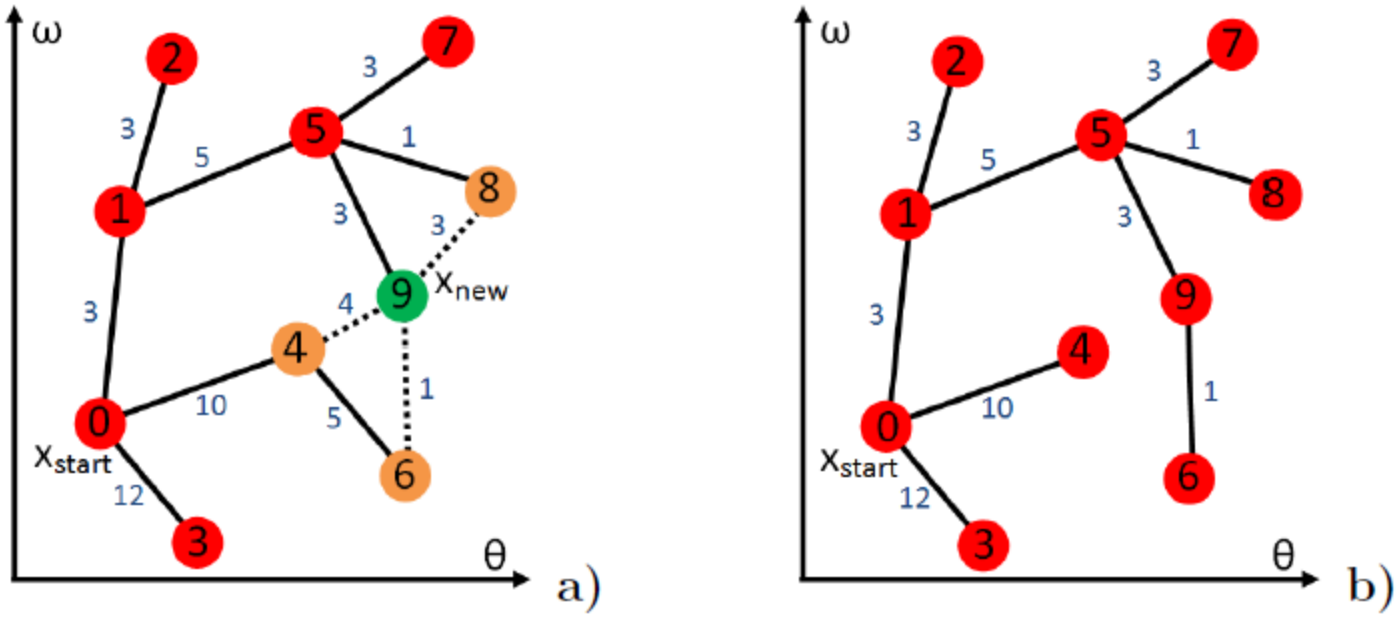
RRT new node expansion



RRT* cost optimization



RRT* rewire



Exercise

Implement the `RRT*` algorithm. You **MUST FOLLOW** the proposed interface:

```
$ ./rrt_star_YOUR_NAME.py path_to_grid_map_image K Δq p max_distance start_x start_y goa
```

where:

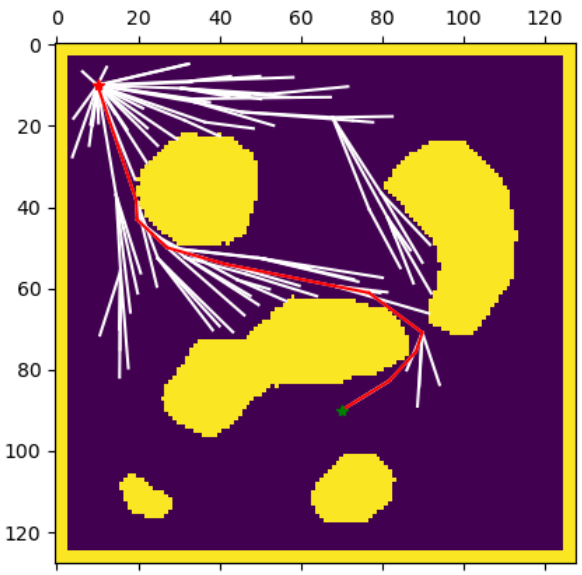
- `path_to_grid_map_image` is the bitmap image containing the environment.
- `K` is the number of iterations to perform.
- `Δq` the maximum distance at which `q_new` is created with respect `q_nearest` pointing `q_rand`.
- `p` is the probability of `q_rand` to be `goal`.
- `max_distance` is the maximum distance to `q_new` to consider a vertex *near* to `q_new`.
- `start` is a 2D array `[start_x, start_y]` containing the initial position.
- `goal` is a 2D array `[goal_x, goal_y]` containing the final position.

The script must plot the generated graph (`vertices` and `edges`) as well as the resulting `path` graphically (i.e., plot the vertices, edges and path over the given grid map). Compute also the total path length.

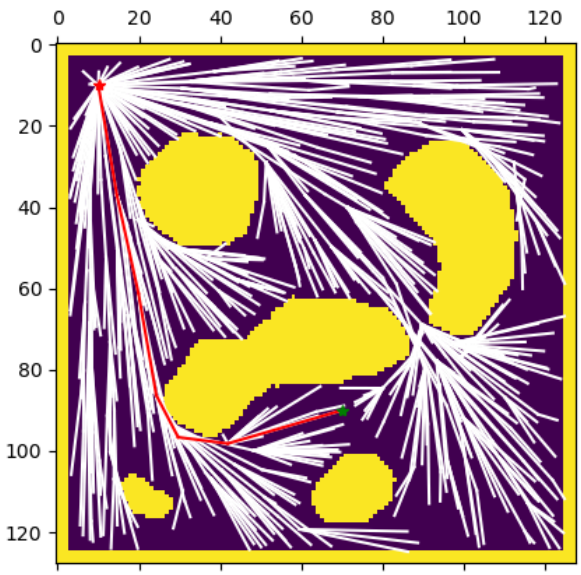
To verify that RRT* is asymptotically optimal, you can plot the `graph` and the `path` as soon as a solution is found and the `graph` and the `path` after `K` iterations. You should notice that the final `path` is better (shorter and smoother) than the original one.

See the following execution:

```
$ ./rrt_star.py ./map0.png 1000 5 0.2 30 10 10 90 70
Goal reached in 293 iterations. Path distance: 140.3928103797893
Path distance after 1000 iteration: 130.91107714174987
```



After 293 iteration



After 1000 iteration

Submission

Submit a report in PDF and one or two Python files (one for the RRT algorithm and another for the RRT*). In the report, explain in detail, and using figures, the work done. Explain also the problems you found. You might want to test your algorithms using other environments than the ones provided. Remember to add your name to the Python files: `rrt_YOUR_NAME.py` and `rrt_star_YOUR_NAME.py` .

We encourage you to use classes to encapsulate the RRT and RRT* algorithms as well as, for instance, for the graph representation.

WARNING:

We encourage you to help or ask your classmates for help, but the direct copy of a lab will result in a failure (with a grade of 0) for all the students involved.

It is possible to use functions or parts of code found on the internet only if they are limited to a few lines and correctly cited (a comment with a link to where the code was taken from must be included).

Deliberately copying entire or almost entire works will not only result in the failure of the laboratory but may lead to a failure of the entire course or even to disciplinary action such as temporal or permanent expulsion of the university. Rules of the evaluation and grading process for UdG students.

Narcís Palomeras Last review September 2024.