

# Graph Search – A\* algorithm

---

This document will guide you through the practical work related to path planning algorithms for searching a graph efficiently. In particular, this practical exercise consists on programming the A\* algorithm to find the optimal path in a discrete environment.

---

## The A\* algorithm

To program an A\* algorithm to find the minimum path from the start position to the goal the following pseudocode ([source](#)) can be used.

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

% A* finds a path from start to goal.
% h is the heuristic function. h(n) estimates the cost to reach goal
from n.
function A_Star(start, goal, h)
    % The set of discovered nodes that may need to be (re-)expanded.
    % Initially, only the start node is known.
    % This is usually implemented as a min-heap or priority queue
    % rather than a hash-set.
    openSet := {start}

    % For node n, cameFrom[n] is the node immediately preceding it on
the
    % cheapest path from start to n currently known.
    cameFrom := an empty map

    % For node n, gScore[n] is the cost of the cheapest path from start
    % to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    % For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents
our
    % current best guess as to how short a path from start to finish can
    % be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := gScore[start] + h(start)

    while openSet is not empty
        % This operation can occur in O(1) time if openSet is a min-heap
        % or a priority queue
```

```

        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            % d(current,neighbour) is the weight of the edge from
current to
            % neighbour tentative_gScore is the distance from start to
the
            % neighbour through current
            tentative_gScore := gScore[current] + d(current, neighbour)
            if tentative_gScore < gScore[neighbour]
                % This path to neighbour is better than any previous
one.
                cameFrom[neighbor] := current
                gScore[neighbour] := tentative_gScore
                fScore[neighbour] := gScore[neighbour] + h(neighbour)
                if neighbor not in openSet
                    openSet.add(neighbour)

        % Open set is empty but goal was never reached
        return failure

```

## 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* locations 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)

## A\* algorithm implementation

Use the previous pseudocode to implement a discrete version of the A\*. The Euclidean distance can be used as heuristic function. For the *neighbours* of the current position, use any surrounding cell (i.e., connect-4 or connect-8) that is not occupied. E.g., Cell  $\$C\$$  at position  $[1, 1]$  has 8 neighbours if connect-8 is used. However, only  $[0, 1]$ ,  $[0, 2]$ ,  $[1, 2]$  and  $[2, 2]$  are valid neighbours, since the other cells are occupied:

```
$$  
\left[  
\begin{array}{ccc}  
1 & 0 & 0 \\  
1 & C & 0 \\  
1 & 1 & 0  
\end{array}  
\right]  
$$
```

Use the following interface for the A\* algorithm:

```
$ ./a_star_YOUR_NAME.py path_to_grid_map_image start_x start_y goal_x  
goal_y
```

Given the path to the grid map image as well as the *start* and *goal* location in pixels, the script must return the minimum *path* (i.e., list of  $(X, Y)$  positions in pixels) and the total *cost* (i.e., Euclidean path length). An image illustrating the resulting path on the original map must also be shown.

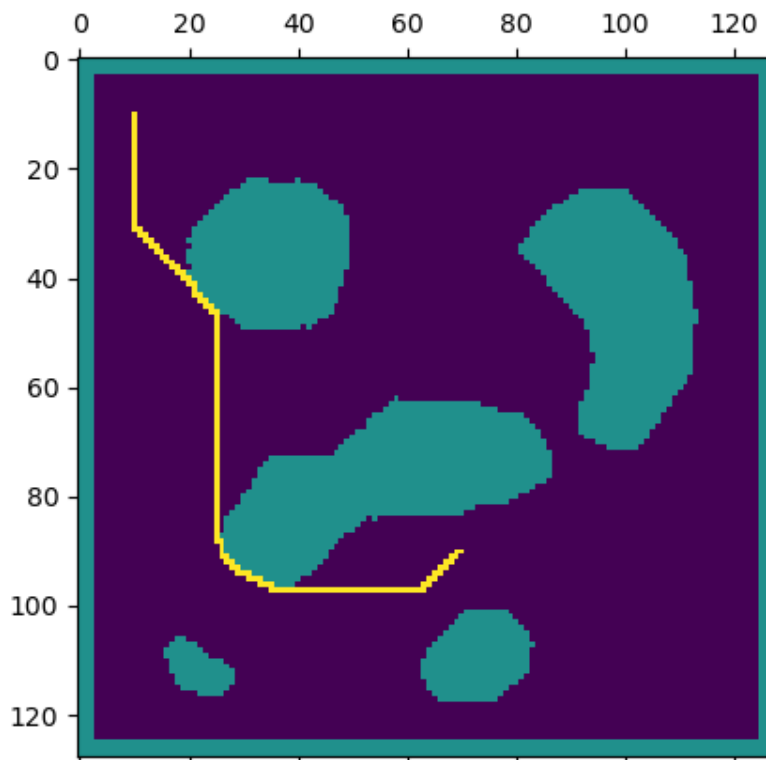
The *path* variable is an array that contains the ordered list of pixel positions that belong to the minimum path. The first pixel must always be the *start* position and the last one the *goal* (e.g., *path* =  $[(start\_x, start\_y), \dots, (goal\_x, goal\_y)]$ ). The *cost* is a scalar value that indicates the length of the optimal path.

When the A\* algorithm is called with the following input variables: *map0.png*, *start*=(10, 10) and *goal*=(90, 70), the following result should be obtained:

#### Using connectivity 4:

Path cost: 154.0

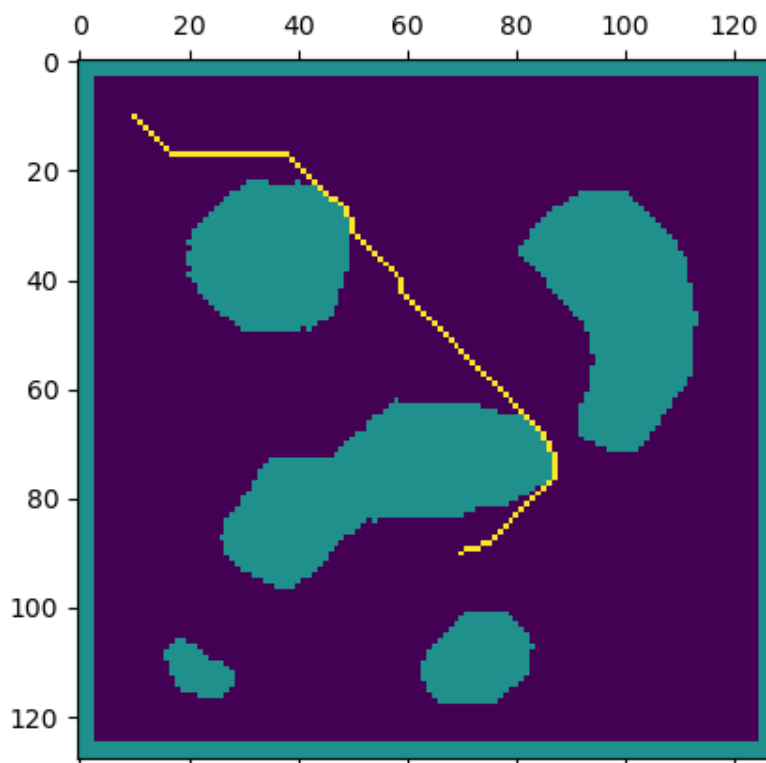
Path: [(10, 10), (11, 10), (12, 10), (13, 10), (14, 10), (15, 10), (16, 10), (17, 10), (18, 10), (19, 10), (20, 10), (21, 10), (22, 10), (23, 10), (24, 10), (25, 10), (26, 10), (27, 10), (28, 10), (29, 10), (30, 10), (31, 10), (31, 11), (32, 11), (32, 12), (33, 12), (33, 13), (34, 13), (34, 14), (35, 14), (35, 15), (36, 15), (36, 16), (37, 16), (37, 17), (38, 17), (38, 18), (39, 18), (39, 19), (40, 19), (40, 20), (41, 20), (41, 21), (42, 21), (43, 21), (43, 22), (44, 22), (44, 23), (45, 23), (45, 24), (46, 24), (46, 25), (47, 25), (48, 25), (49, 25), (50, 25), (51, 25), (52, 25), (53, 25), (54, 25), (55, 25), (56, 25), (57, 25), (58, 25), (59, 25), (60, 25), (61, 25), (62, 25), (63, 25), (64, 25), (65, 25), (66, 25), (67, 25), (68, 25), (69, 25), (70, 25), (71, 25), (72, 25), (73, 25), (74, 25), (75, 25), (76, 25), (77, 25), (78, 25), (79, 25), (80, 25), (81, 25), (82, 25), (83, 25), (84, 25), (85, 25), (86, 25), (87, 25), (88, 25), (88, 26), (89, 26), (90, 26), (91, 26), (91, 27), (92, 27), (92, 28), (93, 28), (93, 29), (94, 29), (94, 30), (94, 31), (95, 31), (95, 32), (95, 33), (96, 33), (96, 34), (96, 35), (97, 35), (97, 36), (97, 37), (97, 38), (97, 39), (97, 40), (97, 41), (97, 42), (97, 43), (97, 44), (97, 45), (97, 46), (97, 47), (97, 48), (97, 49), (97, 50), (97, 51), (97, 52), (97, 53), (97, 54), (97, 55), (97, 56), (97, 57), (97, 58), (97, 59), (97, 60), (97, 61), (97, 62), (97, 63), (96, 63), (96, 64), (95, 64), (95, 65), (94, 65), (94, 66), (93, 66), (93, 67), (92, 67), (92, 68), (91, 68), (91, 69), (90, 69), (90, 70)]



## Using connectivity 8:

Path cost: 133.58073580374347

Path: [(10, 10), (11, 11), (12, 12), (13, 13), (14, 14), (15, 15), (16, 16), (17, 17), (17, 18), (17, 19), (17, 20), (17, 21), (17, 22), (17, 23), (17, 24), (17, 25), (17, 26), (17, 27), (17, 28), (17, 29), (17, 30), (17, 31), (17, 32), (17, 33), (17, 34), (17, 35), (17, 36), (17, 37), (17, 38), (18, 39), (19, 40), (20, 41), (21, 42), (22, 43), (23, 44), (24, 45), (25, 46), (25, 47), (26, 48), (27, 49), (28, 49), (29, 50), (30, 50), (31, 50), (32, 51), (33, 52), (34, 53), (35, 54), (36, 55), (37, 56), (38, 57), (39, 58), (40, 59), (41, 59), (42, 59), (43, 60), (44, 61), (45, 62), (46, 63), (47, 64), (48, 65), (49, 66), (50, 67), (51, 68), (52, 69), (53, 70), (54, 71), (55, 72), (56, 73), (57, 74), (58, 75), (59, 76), (60, 77), (61, 78), (62, 79), (63, 80), (64, 81), (65, 82), (66, 83), (67, 84), (68, 85), (69, 85), (70, 86), (71, 86), (72, 87), (73, 87), (74, 87), (75, 87), (76, 87), (77, 86), (78, 85), (79, 84), (80, 83), (81, 82), (82, 81), (83, 80), (84, 79), (85, 78), (86, 77), (87, 76), (88, 75), (88, 74), (89, 73), (89, 72), (89, 71), (90, 70)]



PROF

## Submission

Submit a report in PDF and one Python script ([a\\_star\\_YOUR\\_NAME.py](#)). In the report, explain in detail the work done. For each proposed environment show the shortest path graphically. Explain also the

problems or difficulties faced while implementing the scripts. The algorithm may be evaluated with other environments than the ones provided here. **BE SURE** that your algorithm is able to correctly load any grid map environment!

---

## 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.