Path planning using A*
Correction

David FILLIAT - ENSTA Paris

December 13, 2021

1 Question 1

Question 1 : With parameter `heuristic_weight = 0.0` what is the algorithm that is executed ? What are the advantages and disadvantages with respect to the version with `heuristic_weight = 1.0` ?

With parameter `heuristic_weight = 0.0`, the A star algorithm is simply the Dijkstra algorithm, as the next updated node is the one that is closer from the starting point. The main disadvantage is that the algorithm is slower because more nodes are explored to reach the goal (more or less depending on the environment). There is no real advantages of Dijkstra compared to A*, except if you use the result as a policy because the computation has been done for more nodes, thus increasing the chance of the robot staying in an already planned area. But in practice it is probably more relevant to resort to fast replanning algorithms such as D*.

2 Question 2

Question 2 : With parameter `heuristic_weight = 5.0` what is the result of the algorithm ? What are the advantages and disadvantages with respect to the version with `heuristic_weight = 1.0` ? What is the theoretical reason for this behavior ?

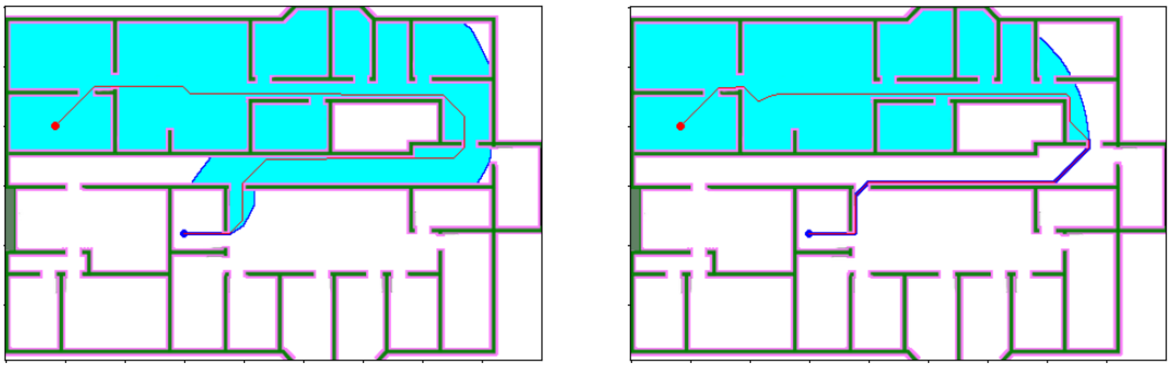

Figure 1: Shortest path with `heuristic_weight = 1.0` (left) and with with `heuristic_weight = 5.0` (right)

Figure 1 shows the resulting path with `heuristic_weight = 5.0`. It is longer than the path with `heuristic_weight = 1.0` (680.97 vs 676.00) and we can see that fewer nodes have been explored, which translate in a shorter computation time (1.28 s vs 1.55). The reason of the lack of optimality is that the heuristic with weight = 5 is not *admissible*, i.e. it is not alway smaller than the shortest path to the goal[1]. However, using a higher weights more strongly orients the computation towards the goal and often reduces computation times, which in some environment and some applications where optimality is not crucial can be interesting.

[1]See `https://en.wikipedia.org/wiki/Admissible_heuristic`

3 Question 3

Question 3 : Compare the performance of the algorithm with `heuristic_weight = 0.0` and with `heuristic_weight = 1.0` in the different environments provided with the code. What do you see? In which cases does the A* algorithm bring the most gain?

In all environments, the paths are optimal in the two configurations (as it is theoretically guaranteed). The following table shows the computation times for the three environments, and the gain of using A* versus using Dijkstra:

Env.	*weight* $= 0$	*weight* $= 1$	*Gain*
$office$	1.81	1.60	11.6%
$labyrinthe$	1.11	1.06	4.5%
$freespace$	2.84	0.16	94.3%

We can see that the gain strongly depends on the environment. In mostly free space, where the heuristics correspond almost perfectly to the real path cost, the gain can be huge. On the contrary, in the labyrinth where the path is complex and very far from the straight line, the heuristics guides the computations in many dead-ends, and thus offer a very limited gain. The office environment represents an intermediate complexity, quite common in practice, where A* offers a good speedup.

4 Question 4

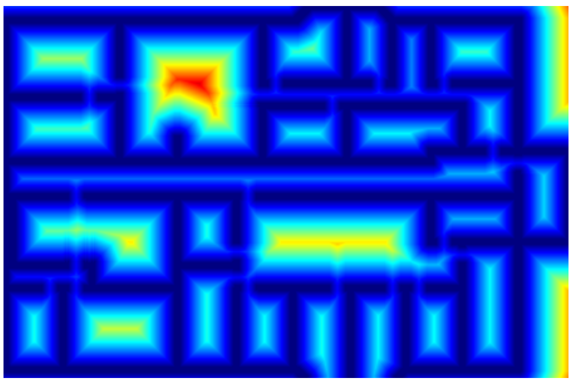

Figure 2: Result of the distance transform, from blue(0) to red (max value).

Question 4 : Modify the weight of the nodes so that the paths do not pass too close to obstacles (Fig. 3, right). You can use the opencv `cv2.distanceTransform(map, cv2.DIST_L2, 3)` function which makes it possible to compute, for each free position of the map, the distance to the nearest obstacle. Illustrate the trajectories obtained on the environment proposed by default in the script.

You should start by computing the distance transform in the class constructor by adding the following line at the end of the `__init__` function (Figure 2 show the resulting distance map). It is important to put this here, and not in the planning function, because its a computation that depends on the map, and not on the goal, thus it should be computed when the map is loaded and don't need to be recomputed for each new plan:

```python
self.node_cost = cv2.distanceTransform(self.map, cv2.DIST_L2, 3)
cv2.imwrite('cost.png', cv2.applyColorMap(np.uint8(self.node_cost*5), cv2.COLORMAP_JET))
```

Then you use this distance map when computing the cost of the new node by replacing:

```python
#new_cost = cost_so_far[current] + self.distance(current, next)
new_cost = cost_so_far[current] + 5.0/self.node_cost[current] + self.distance(current,
    next)
```

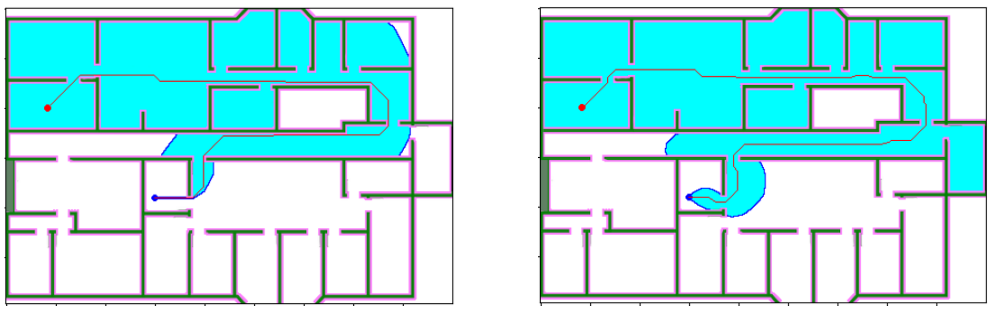

Figure 3: Shortest path without node cost (left) and with a penalty for proximity of obstacles (right)

Figure 3 illustrates the behavior of this modification, finding longer paths, but remaining farther from the obstacles. Note that this also entails a longer computation time as more nodes are processed because the heuristics more strongly underestimates the path cost.