COMP 6721 Project 1 Report

Vida Abdollahi

40039052 vida.abdolahi@gmail.com

Table of Contents

1. Introduction: Algorithm

For this project, I will be focusing on the A* Algorithm. A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's Algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search. The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal. In the below diagram, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

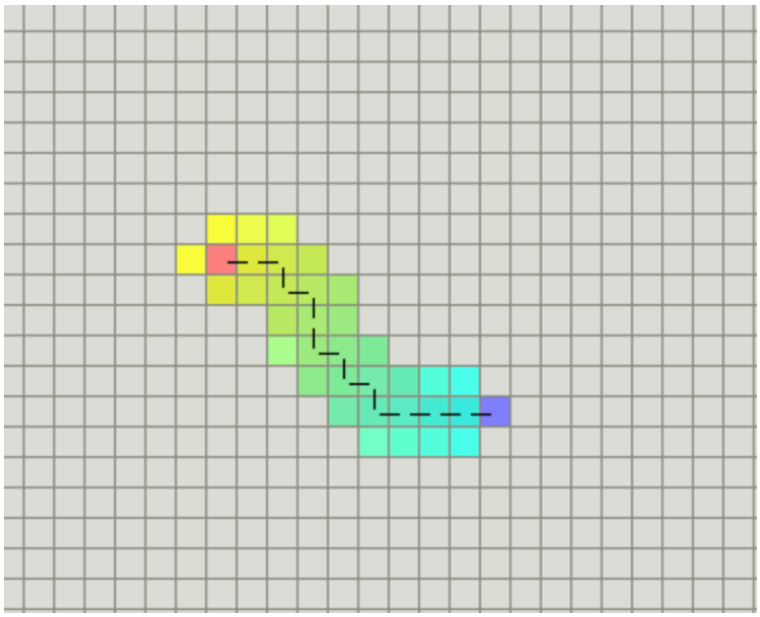

Fig. 1. A* algorithm

2. A* Heuristic

The heuristic function h(n) tells A* an estimate of the minimum cost from any vertex n to the goal. It's important to choose a good heuristic function. The heuristic can be used to control A*'s behavior.

Speed and Accuracy:

A*'s ability to vary its behavior based on the heuristic and cost functions can be very useful. The tradeoff between speed and accuracy can be exploited to make our search faster.

2.1 Heuristic for grid maps: Diagonal Distance

On a grid, there are well-known heuristic functions to use [1].

On a square grid that allows 4 directions of movement, we can use Manhattan distance.

On a square grid that allows 8 directions of movement, we can use Diagonal distance.

On a square grid that allows any direction of movement, we might or might not want Euclidean distance.

If A* is finding paths on the grid but we are allowing movement not on the grid, we may also want to consider other representations of the map.

Diagonal Distance

If our map allows diagonal movement we need a different heuristic. The Manhattan distance for (4 east, 4 north) will be 8×D. However, we could simply move (4 northeast) instead, so the heuristic should be 4×D2, where D2 is the cost of moving diagonally.

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Here we compute the number of steps we take if we can't take a diagonal, then subtract the steps we save by using the diagonal. There are min(dx, dy) diagonal steps, and each one costs D2 but saves us 2×D non-diagonal steps.

For this project, I set the diagonal cost to be 14 (which is the diameter of a square when the size is 1 unit multiply by 10) and also the non-diagonal cost to be 10 (a square of size 1 multiply by 10), so our heuristic functions is as following:

```
distX = abs(node_a.x - node_b.x)
distY = abs(node_a.y - node_b.y)
if distX > distY:
    return 14 *distY + 10 *(distX-distY)
```

```
else:
   return 14 *distX + 10 *(distY-distX)
```

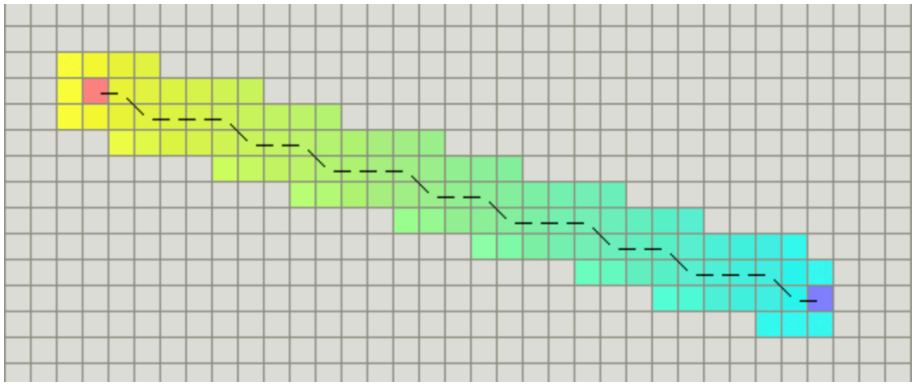

Fig. 2. Diagonal Heuristic Function

3. Implementation

The A* algorithm, stripped of all the code, is fairly simple. There are two sets, OPEN and CLOSED. The OPEN set contains those nodes that are candidates for examining. Initially, the OPEN set contains only one element: the starting position. The CLOSED set contains those nodes that have already been examined. Initially, the CLOSED set is empty. Graphically, the OPEN set is the "frontier" and the CLOSED set is the "interior" of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node n in OPEN (the node with the lowest f value) and examines it. If n is the goal, then we're done. Otherwise, node n is removed from OPEN and added to CLOSED. Then, its neighbors n′ are examined. A neighbor that is in CLOSED has already been seen, so we don't need to look at it. A neighbor that is in OPEN is scheduled to be looked at, so we don't need to look at it now. Otherwise, we add it to OPEN, with its parent set to n. The path cost to n′, g(n′), will be set to g(n) + movementcost(n, n′).

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
```

```
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): (2)
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

4. Performance: Threshold and Grid Size

There are few ways that can help us having a better performance in this sort of problems, like decreasing the size of the graph if possible. This will reduce the number of nodes that are processed, both those on the path and those that don't end up on the final path. We can also consider improving the accuracy of the heuristic. This will reduce the number of nodes that are not on the final path. The closer the heuristic to the actual path length (not the distance), the fewer nodes A* will explore [1].

The Threshold value defined for this project can vary between 50 to 90 percentage. This mean that the number of block nodes is dependent to the threshold. Obviously, having a higher threshold value will reduce the number of block nodes and increase the number of non-block nodes. In that case, it's easier for the algorithm to find its path through the map and takes less time. I evaluated my algorithm with all the 3 different threshold values from a same starting and ending point in the map, fixing the grid size to 0,002, the result is below:

Time to find the optimal path with threshold value 50: 0.0014947689999971203

Time to find the optimal path with threshold value 70: 0.0013959169999999688

Time to find the optimal path with threshold value 90: 0.0007535540000009888

As we can see, higher threshold leads to a simpler map and respectively the algorithm will find the solution in a lesser amount of time.

The second element needs to be consider for the performance is the size of the graph, or size of each grid in this problem. As mentioned, this will reduce the number of nodes that are processed, both those on the path and those that don't end up on the final path. I evaluated my algorithm on a same map with different size of grid cells (0.002 and 0.003), fixing the threshold value to 50, the result is shown below:

Time to find the optimal path with grids size of 0.002: 0.0014134180000002772

Time to find the optimal path with grids size of 0.003: 0.0007320350000004083

As we can see, increasing each grid size will reduce the number of nodes that are processed and this mean we need less time to find the solution.

Best result will be yield if we use the grid size of 0.003 with the threshold value of 90, in 0.0006172499999994585 seconds.

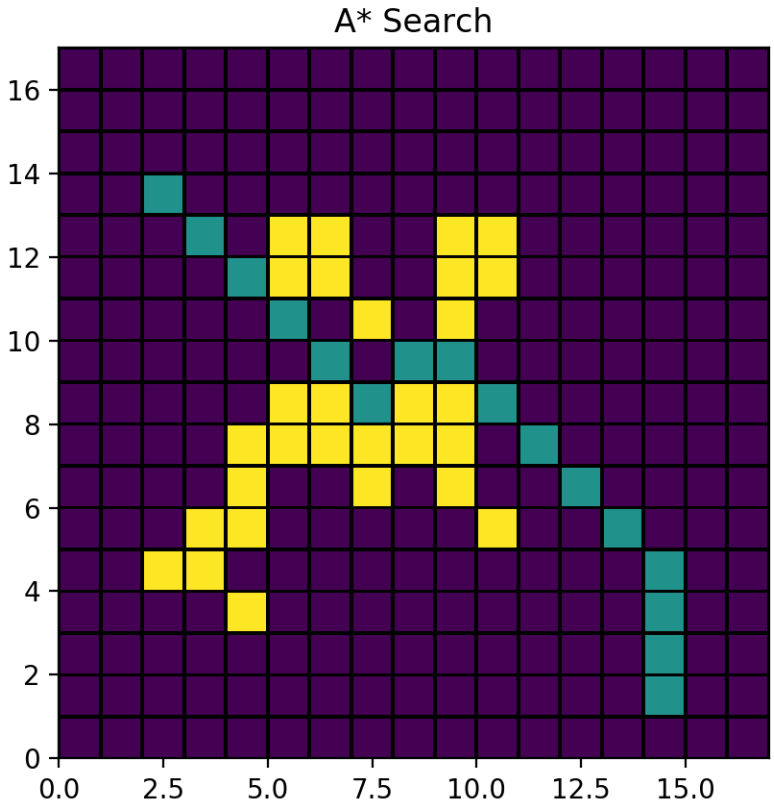

Fig. 3. A* Algorithm applied on grid size of 0.003 and threshold value 90

5. Difficulties

The heuristic function h(n) tells A* an estimate of the minimum cost from any vertex n to the goal. It's important to choose a good heuristic function. The heuristic can be used to control A*'s behavior and have effect on the speed and accuracy of the algorithm. Finding a heuristic function that can best suit this purpose and find a solution in less than 10 second was a challenge for me, I addressed this issue by implementing a heuristic function based on the diagonal movement as I explained in A* heuristic part.

Another challenging part that I encountered while working on this project was when I had to create a grid matrix based on the given map. To do it, I collected the coordination of all the points in the map that would be located on our grids of any size. Then using the center point of this cell, I generated a 2d numpy array to

store all the points that are located in the center of each cell. Using find_neighbours function, I was able to find the exact index of each points in the actual map and mapped them in the matrix.

6. Libraries

Geopandas: GeoPandas is an open source project to make working with geospatial data in python easier. GeoPandas extends the datatypes used by pandas to allow spatial operations on geometric types. Geometric operations are performed by shapely.

Shapely: Shapely is a Python package for set-theoretic analysis and manipulation of planar features using (via Python's ctypes module) functions from the well-known and widely deployed GEOS library

Numpy: NumPy is the fundamental package for scientific computing with Python.

Matplotlib: Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

References

1. https://www.redblobgames.com/pathfinding/a-star/introduction.html