

AssignmentOne

SOFE 3720/ CSCI 4610: Introduction to Artificial
Intelligence/ Artificial Intelligence

Winter 2019

Dr. Sukhwant Kaur Sagar

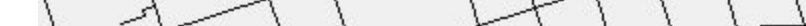
Student number	Banner ID	Student Name
100555278		Christian Ivanov
100737224		Federico Hauque
100720471		You Cun Tan

Introduction

This is an assignment for the SOFE 3720/ CSCI 4610: Introduction to Artificial Intelligence course. The concept behind the problem statement is to use the A* algorithm in order to solve a path finding problem. We were tasked to use the given code and choose a map section to work on. From then on we used given data to make a program that can solve for the shortest path given a starting and ending point.

Correctness of A* search

The program would take a starting and ending point on a given node. After which, by clicking the "Plan!" button, the A* algorithm would run, find and map the shortest path. The following screenshots are some example on how the program runs but we do encourage personal testing.



In the example shown in the map on the right, there are two dots representing the start (green) and goal (red) nodes. As we can see, taking the path to the west of the start



node will calculate lower heuristics than the other paths. And since that path leads to a loop, if



the A* search happens not to be correctly implemented, it would get caught in that loop (which is the problem that Best First Search would have in a situation like this).

Fortunately, the implemented A* search does not fail in this task and finds the best possible path, highlighting it with the orange lines that can be seen in the picture on the left.

After passing this test, we were able to try more complex paths in the map, checking if the provided solution is actually well enough to find either long and short paths. The result of another path we wanted to try to calculate can be seen in the following picture.



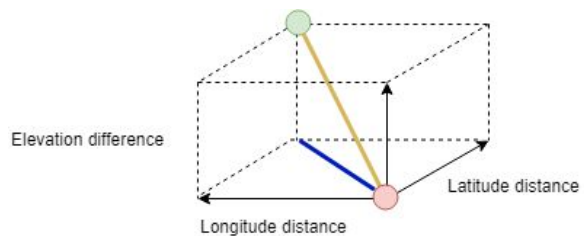
Heuristic

We used a modified version of the `node_dist` function in order to solve the value. We took the original position distance found and we included the elevation. The exact equations and elevation statements can be found in the following:

```
def node_dist(n1, n2):
    ''' Distance between nodes n1 and n2, considering the elevation difference (in meters) '''
    dx = (n2.pos[0] - n1.pos[0]) * MPERLON
    dy = (n2.pos[1] - n1.pos[1]) * MPERLAT
    plain_distance = math.sqrt(dx * dx + dy * dy)
    elev_diff = n1.elev - n2.elev
    return math.sqrt(elev_diff * elev_diff + plain_distance * plain_distance) # in meters
```

To illustrate the chosen heuristic (and cost function), we can see that the former heuristic (provided with the code) as a blue line, stating the flat distance between the start (red circle) and goal (green circle) node.

The new heuristic considers the elevation difference and calculates a more precise distance between two nodes, and since it is still underestimating the actual path cost, it can be considered an admissible heuristic.



Data handling

The data used in our project includes a map of the location and an elevation map. The map is located in southern Oshawa and we did not pick it for any specific reason. The following picture shows the Oshawa area chosen just as it can be seen in OpenStreetMap website.



The map import was done by using the given code and modifying it for the given chosen location. The coordinates used are between -78.8623 to -76.832 longitude and between 43.8983 to 43.8881 latitude.

Secondly we imported the elevation map. This was adapted from a suggested code from the internet (first reference). The code would just map the elevation to the correct node and the code is as follows:

```
def build_elevs():    #Code adapted from a suggested code in the internet - Reference: https://bit.ly/2ElrIYs
    height = EPIX
    width = EPIX
    fi = open(r"n43_w079_1arc_v2.bil", "rb")
    contents = fi.read()
    fi.close()
    s = "<%dH" % (int(width * height),)
    z = struct.unpack(s, contents)
    heights = np.zeros((height, width))
    for r in range(0, height):
        for c in range(0, width):
            elevation = z[((width) * r) + c]
            heights[r][c] = float(elevation)
    print(len(heights))
    return heights
```

Further data handling problems were introduced once we got to the mapping. The major one was having no coastal regions to map. This was quite simple to fix by just removing the code that would handle coasts. The second one was that we have stored the heights in a matrix instead of a vector, and the given code manipulated all heights as a vector. However, the heights vector indexes were always calculated from two values, row and column, which were the indexes we needed to obtain the same height in our matrix. With the rest of the functions being defined, we were able to solve the rest. Furthermore, testing did show that placing the starting and ending node on the same spot would result in a fatal error but that was handled with an exit message.

Defense of path cost function

In this solution the algorithm would constantly choose the least cost path. The A* solution is quite different from Dijkstra and BFS because it looks at multiple locations at once. The Heuristic function would find the best possible option from all the unvisited neighboring nodes. The best possible option would be the one with the least cost. From then, if the least cost is constantly chosen, it is a matter of time before the starting and ending nodes are connected and a solution is found. Furthermore, it is important to mention that this works when looking at all parts of the map even straight paths as they would still give the same incentive to follow them. Further information on the cost function used can be found in the following paragraph and screenshot.

Efficiency of solution

The complexity of the solution is highly dependant on the heuristics. The less accurate the heuristic is, the more nodes will be unnecessarily explored. Usually, A* search is known for having an exponential time (and space) complexity in the worst case. However, a good heuristic leads to the exploration of best paths more rapidly, and since we have enhanced the given heuristic considering the elevation difference between two nodes, this better accuracy surely has impacted in the overall complexity. The following code implements the A* algorithm relying on the previously explained heuristics and cost function.

```

def plan(self, s, g):
    """
    Standard A* search
    """
    if(s!=g):
        parents = {}
        costs = {}
        q = PriorityQueue()
        q.put((self.heur(s, g), s))
        parents[s] = None
        costs[s] = 0
        while not q.empty():
            cf, cnode = q.get()
            if cnode == g:
                print("Path found, time will be", costs[g] * 60 / 5000, " minutes.") # 5 km/hr known as the preferred walking distance
                return self.make_path(parents, g)
            for edge in cnode.ways:
                newcost = costs[cnode] + edge.cost
                if edge.dest not in parents or newcost < costs[edge.dest]:
                    parents[edge.dest] = (cnode, edge.way)
                    costs[edge.dest] = newcost
                    q.put((self.heur(edge.dest, g) + newcost, edge.dest))
        else:
            print('Start and goal nodes are the same! - Closing the window')
            sys.exit(0)

```