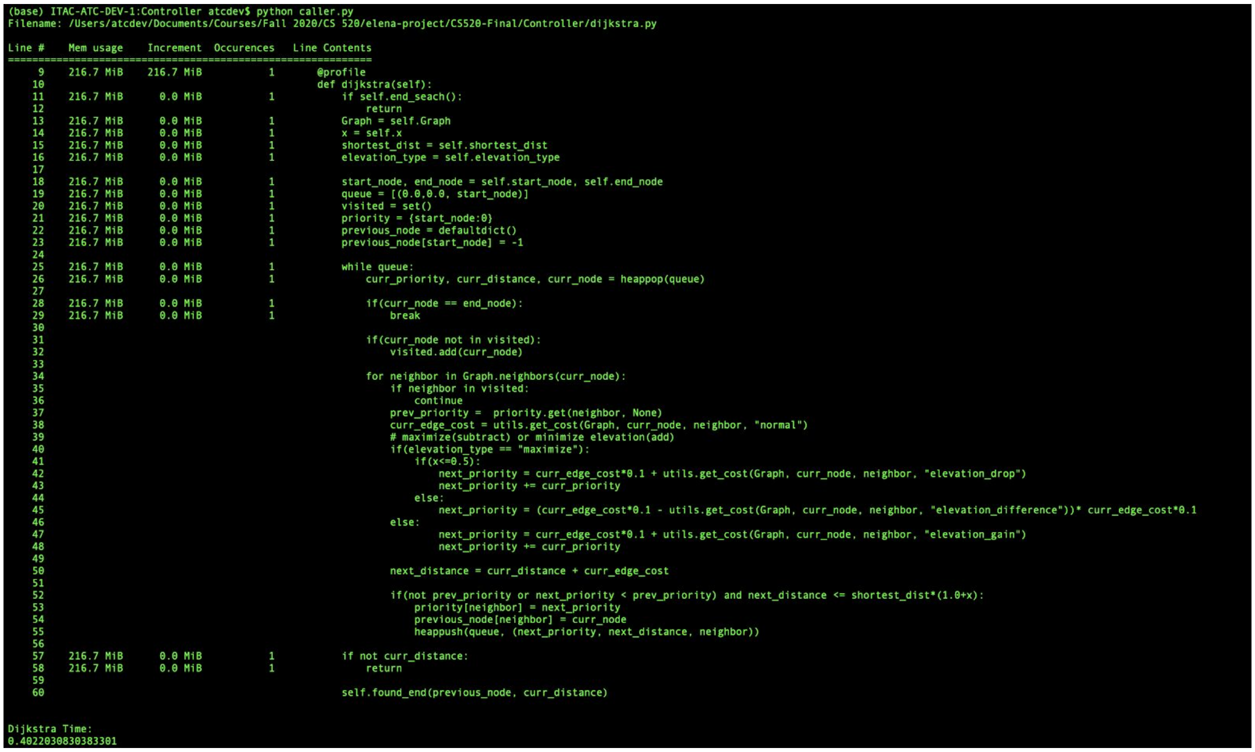
EleNA system

Group Members: Venkata (Dennis) Billagiri, Shreya Sawant, Dhruv Keyal, Anushree Jana, Aditya Vikram Agarwal

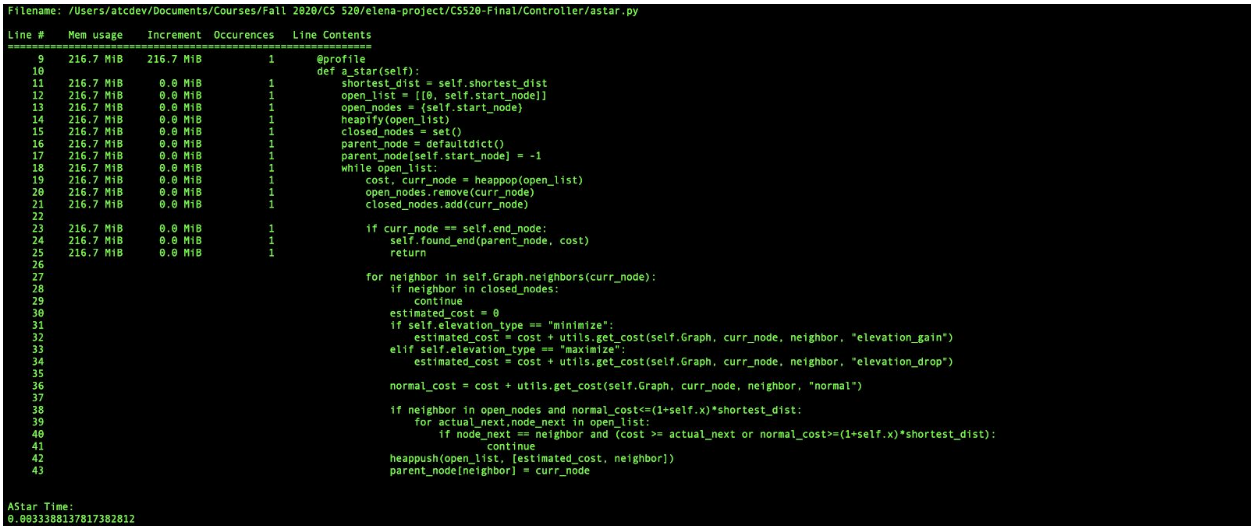
Test Cases and Performance



```
(base) ITAC-ATC-DEV-1:Controller atcdev$ python caller.py
Filename: /Users/atcdev/Documents/Courses/Fall 2020/CS 520/elena-project/CS520-Final/Controller/dijkstra.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9    216.7 MiB    216.7 MiB           1       @profile
    10                                             def dijkstra(self):
    11    216.7 MiB      0.0 MiB           1           if self.end_seach():
    12                                                     return
    13    216.7 MiB      0.0 MiB           1           Graph = self.Graph
    14    216.7 MiB      0.0 MiB           1           x = self.x
    15    216.7 MiB      0.0 MiB           1           shortest_dist = self.shortest_dist
    16    216.7 MiB      0.0 MiB           1           elevation_type = self.elevation_type
    17
    18    216.7 MiB      0.0 MiB           1           start_node, end_node = self.start_node, self.end_node
    19    216.7 MiB      0.0 MiB           1           queue = [(0.0,0.0, start_node)]
    20    216.7 MiB      0.0 MiB           1           visited = set()
    21    216.7 MiB      0.0 MiB           1           priority = {start_node:0}
    22    216.7 MiB      0.0 MiB           1           previous_node = defaultdict()
    23    216.7 MiB      0.0 MiB           1           previous_node[start_node] = -1
    24
    25    216.7 MiB      0.0 MiB           1           while queue:
    26    216.7 MiB      0.0 MiB           1               curr_priority, curr_distance, curr_node = heappop(queue)
    27
    28    216.7 MiB      0.0 MiB           1               if(curr_node == end_node):
    29    216.7 MiB      0.0 MiB           1                   break
    30
    31                                                     if(curr_node not in visited):
    32                                                         visited.add(curr_node)
    33
    34                                                     for neighbor in Graph.neighbors(curr_node):
    35                                                         if neighbor in visited:
    36                                                             continue
    37                                                         prev_priority =  priority.get(neighbor, None)
    38                                                         curr_edge_cost = utils.get_cost(Graph, curr_node, neighbor, "normal")
    39                                                         # maximize(subtract) or minimize elevation(add)
    40                                                         if(elevation_type == "maximize"):
    41                                                             if(x<=0.5):
    42                                                                 next_priority = curr_edge_cost*0.1 + utils.get_cost(Graph, curr_node, neighbor, "elevation_drop")
    43                                                                 next_priority += curr_priority
    44                                                             else:
    45                                                                 next_priority = (curr_edge_cost*0.1 - utils.get_cost(Graph, curr_node, neighbor, "elevation_difference"))* curr_edge_cost*0.1
    46                                                         else:
    47                                                                 next_priority = curr_edge_cost*0.1 + utils.get_cost(Graph, curr_node, neighbor, "elevation_gain")
    48                                                                 next_priority += curr_priority
    49
    50                                                         next_distance = curr_distance + curr_edge_cost
    51
    52                                                         if(not prev_priority or next_priority < prev_priority) and next_distance <= shortest_dist*(1.0+x):
    53                                                             priority[neighbor] = next_priority
    54                                                             previous_node[neighbor] = curr_node
    55                                                             heappush(queue, (next_priority, next_distance, neighbor))
    56
    57    216.7 MiB      0.0 MiB           1           if not curr_distance:
    58    216.7 MiB      0.0 MiB           1               return
    59
    60                                                 self.found_end(previous_node, curr_distance)


Dijkstra Time:
0.4022030830383301
```



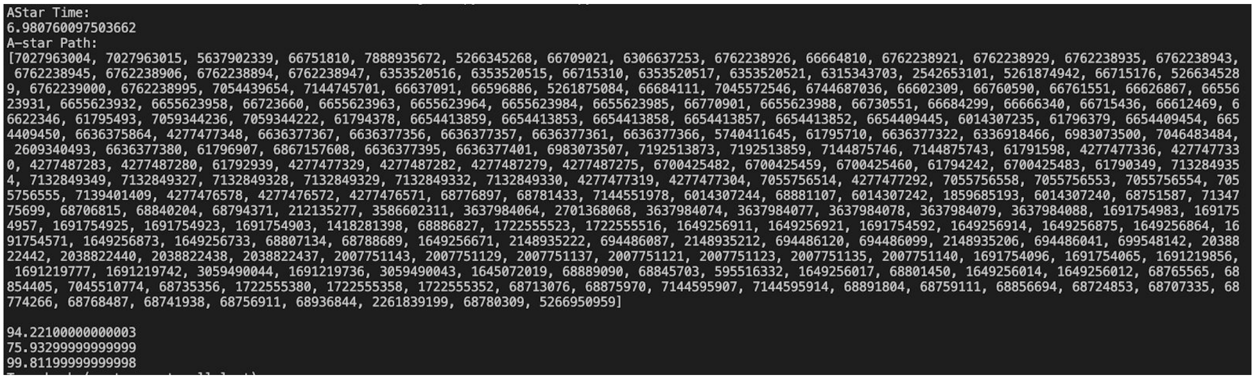
```
Filename: /Users/atcdev/Documents/Courses/Fall 2020/CS 520/elena-project/CS520-Final/Controller/astar.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9    216.7 MiB    216.7 MiB           1       @profile
    10                                             def a_star(self):
    11    216.7 MiB      0.0 MiB           1           shortest_dist = self.shortest_dist
    12    216.7 MiB      0.0 MiB           1           open_list = [[0, self.start_node]]
    13    216.7 MiB      0.0 MiB           1           open_nodes = {self.start_node}
    14    216.7 MiB      0.0 MiB           1           heapify(open_list)
    15    216.7 MiB      0.0 MiB           1           closed_nodes = set()
    16    216.7 MiB      0.0 MiB           1           parent_node = defaultdict()
    17    216.7 MiB      0.0 MiB           1           parent_node[self.start_node] = -1
    18    216.7 MiB      0.0 MiB           1           while open_list:
    19    216.7 MiB      0.0 MiB           1               cost, curr_node = heappop(open_list)
    20    216.7 MiB      0.0 MiB           1               open_nodes.remove(curr_node)
    21    216.7 MiB      0.0 MiB           1               closed_nodes.add(curr_node)
    22
    23    216.7 MiB      0.0 MiB           1               if curr_node == self.end_node:
    24    216.7 MiB      0.0 MiB           1                   self.found_end(parent_node, cost)
    25    216.7 MiB      0.0 MiB           1                   return
    26
    27                                                     for neighbor in self.Graph.neighbors(curr_node):
    28                                                         if neighbor in closed_nodes:
    29                                                             continue
    30                                                         estimated_cost = 0
    31                                                         if self.elevation_type == "minimize":
    32                                                             estimated_cost = cost + utils.get_cost(self.Graph, curr_node, neighbor, "elevation_gain")
    33                                                         elif self.elevation_type == "maximize":
    34                                                             estimated_cost = cost + utils.get_cost(self.Graph, curr_node, neighbor, "elevation_drop")
    35
    36                                                         normal_cost = cost + utils.get_cost(self.Graph, curr_node, neighbor, "normal")
    37
    38                                                         if neighbor in open_nodes and normal_cost<=(1+self.x)*shortest_dist:
    39                                                             for actual_next,node_next in open_list:
    40                                                                 if node_next == neighbor and (cost >= actual_next or normal_cost>=(1+self.x)*shortest_dist):
    41                                                                         continue
    42                                                         heappush(open_list, [estimated_cost, neighbor])
    43                                                         parent_node[neighbor] = curr_node


AStar Time:
0.0033388137817382812
```

We experimentally compared the results for the two search algorithms we used (Dijkstra and A*). By using this comparison technique, we measured their performance in terms of both space and time. In the case below, we see that both Dijkstra and A* take up the same amount of space but A* performs better in terms of time.

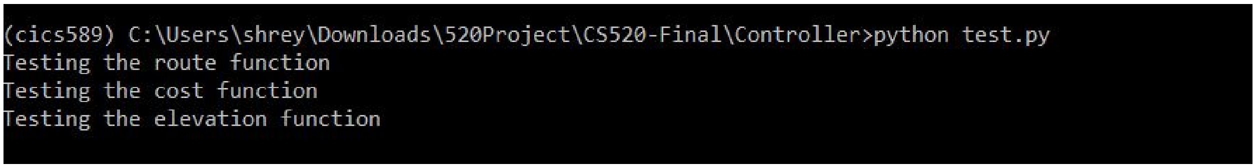
We also get the optimal path for A* below:



```
AStar Time:
6.980760097503662
A-star Path:
[7027963004, 7027963015, 5637902339, 66751810, 7888935672, 5266345268, 66709021, 6306637253, 6762238926, 66664810, 6762238921, 6762238929, 6762238935, 6762238943,
 6762238945, 6762238906, 6762238894, 6762238947, 6353520516, 6353520515, 66715310, 6353520517, 6353520521, 6315343703, 2542653101, 5261874942, 66715176, 526634528
9, 6762239000, 6762238995, 7054439654, 7144745701, 66637091, 66596886, 5261875084, 66684111, 7045572546, 6744687036, 66602309, 66760590, 66761551, 66626867, 66556
23931, 6655623932, 6655623958, 66723660, 6655623963, 6655623964, 6655623984, 6655623985, 66770901, 6655623988, 66730551, 66684299, 66666340, 66715436, 66612469, 6
6622346, 61795493, 7059344236, 7059344222, 61794378, 6654413859, 6654413853, 6654413858, 6654413857, 6654413852, 6654409445, 6014307235, 61796379, 6654409454, 665
4409450, 6636375864, 4277477348, 6636377367, 6636377356, 6636377357, 6636377361, 6636377366, 5740411645, 61795710, 6636377322, 6336918466, 6983073500, 7046483484,
 2609340493, 6636377380, 61796907, 6867157608, 6636377395, 6636377401, 6983073507, 7192513873, 7192513859, 7144875746, 7144875743, 61791598, 4277477336, 427747733
0, 4277487283, 4277487280, 61792939, 4277477329, 4277487282, 4277487279, 4277487275, 6700425482, 6700425459, 6700425460, 61794242, 6700425483, 61790349, 713284935
4, 7132849349, 7132849327, 7132849328, 7132849329, 7132849332, 7132849330, 4277477319, 4277477304, 7055756514, 4277477292, 7055756558, 7055756553, 7055756554, 705
5756555, 7139401409, 4277476578, 4277476572, 4277476571, 68776897, 68781433, 7144551978, 6014307244, 68881107, 6014307242, 1859685193, 6014307240, 68751587, 71347
75699, 68706815, 68840204, 68794371, 212135277, 3586602311, 3637984064, 2701368068, 3637984074, 3637984077, 3637984078, 3637984079, 3637984088, 1691754983, 169175
4957, 1691754925, 1691754923, 1691754903, 1418281398, 68886827, 1722555523, 1722555516, 1649256911, 1649256921, 1691754592, 1649256914, 1649256875, 1649256864, 16
91754571, 1649256873, 1649256733, 68807134, 68788689, 1649256671, 2148935222, 694486087, 2148935212, 694486120, 694486099, 2148935206, 694486041, 699548142, 20388
22442, 2038822440, 2038822438, 2038822437, 2007751143, 2007751129, 2007751137, 2007751121, 2007751123, 2007751135, 2007751140, 1691754096, 1691754065, 1691219856,
 1691219777, 1691219742, 3059490044, 1691219736, 3059490043, 1645072019, 68889090, 68845703, 595516332, 1649256017, 68801450, 1649256014, 1649256012, 68765565, 68
854405, 7045510774, 68735356, 1722555380, 1722555358, 1722555352, 68713076, 68875970, 7144595907, 7144595914, 68891804, 68759111, 68856694, 68724853, 68707335, 68
774266, 68768487, 68741938, 68756911, 68936844, 2261839199, 68780309, 5266950959]

94.22100000000003
75.93299999999999
99.81199999999998
```

Finally, this is our testing performance using test.py



```
(cics589) C:\Users\shrey\Downloads\520Project\CS520-Final\Controller>python test.py
Testing the route function
Testing the cost function
Testing the elevation function
```