Student Name: Keivan Jamali
Student ID: 99104468

Sharif University
of Technology

Intelligent Transportation Systems (20-504)
Assignment 3

---

1 Shortest Path

Question Number 1

This algorithm maintains two main values for each node: the cost of reaching that node from the start node (denoted as $g(n)$), and an estimate of the cost from that node to the target node (denoted as $h(n)$). The sum of these values, $f(n) = g(n) + h(n)$, represents the estimated total cost of the path through that node.
Steps through this method are:

1. Initialize the start node with g(start) = 0 and h(start), which represents the heuristic estimate from the start node to the target node.
2. Create two sets: the open set and the closed set. The open set contains nodes that have been discovered but not yet explored, while the closed set contains nodes that have been explored.
3. Add the start node to the open set.
4. Repeat the following steps until the open set is empty or the target node is reached:
   i. Select the node with the lowest $f(n)$ value from the open set. This node becomes the current node.
   ii. Move the current node from the open set to the closed set.
   iii. If the current node is the target node, the algorithm terminates, and the shortest path has been found.
   iv. For each neighbor of the current node:
   - Calculate the tentative g value by adding the weight of the edge from the current node to the neighbor.
   - If the tentative g value is lower than the neighbor's current g value or the neighbor is not in the open set:
     - Update the neighbor's g value to the tentative value.
     - Calculate the neighbor's h value, which represents the heuristic estimate from the neighbor to the target node.
     - If the neighbor is not in the open set, add it to the open set.
5. If the open set becomes empty before reaching the target node, there is no path from the start node to the target node, and the algorithm terminates.
6. To reconstruct the shortest path, start from the target node and follow the parent pointers (stored during the algorithm execution) until the start node is reached.

The efficiency and optimality of the this algorithm depend on the choice of heuristic function $h(n)$. The heuristic should be admissible, meaning it never overestimates the actual cost to reach the target node. If the heuristic is consistent (or monotonic), it also satisfies the triangle inequality property, which guarantees optimality in finding the shortest path.

In my code, however, the Euclidean distance between two nodes is used as the heuristic function. This heuristic provides a reasonable estimate of the distance between nodes in many scenarios.

Code Explanation - *ShortestPath_V0.py*

The `ShortestPath` class represents an implementation of the A* algorithm to find the shortest path between a start node and a target node in a graph.

i. Class Variables
   - `graph_`: A dictionary representing the graph.
   - `start`: The start node.
   - `target`: The target node.
   - `shortest`: An array of distances from the start node to each node in the graph.
   - `path`: A list representing the path taken to reach the target node.
   - `shortest_path`: A list representing the shortest path from the start node to the target node.

ii. Static Method: _heuristic
   This static method calculates the Euclidean distance between two nodes as the heuristic. It takes the current node and the target node as parameters and returns the Euclidean distance between them.

iii. Method: fit
   The `fit` method fits the given graph with the A* algorithm to find the shortest path from the start node to the target node. It takes the graph, start node, and target node as parameters and returns a tuple containing the shortest path from the start node to the target node and the path taken to reach the target node.

iv. Method: _astar_method
   This private method is the implementation of the A* algorithm. It performs the A* algorithm to find the shortest path between the start node and the target node in the graph. It returns the distances from the start node to each node in the graph and the shortest path as a tuple.

v. Method: _reconstruct_path
   This method reconstructs the path from the target node to the start node. It takes the parents dictionary as a parameter, which maps each node to its parent node. It returns a list representing the path from the target node to the start node.

vi. Method: graph_plotting
   This method generates a plot of the given networkx graph with a highlighted path. It takes a list of nodes representing the path to highlight as a parameter.

Question Number 2, 3 After running the code here are the results:
The shortest distance from node 0 to node 8 is 4.0 And this is the path: [0, 1, 5, 8]
We can see the plot in figure 1

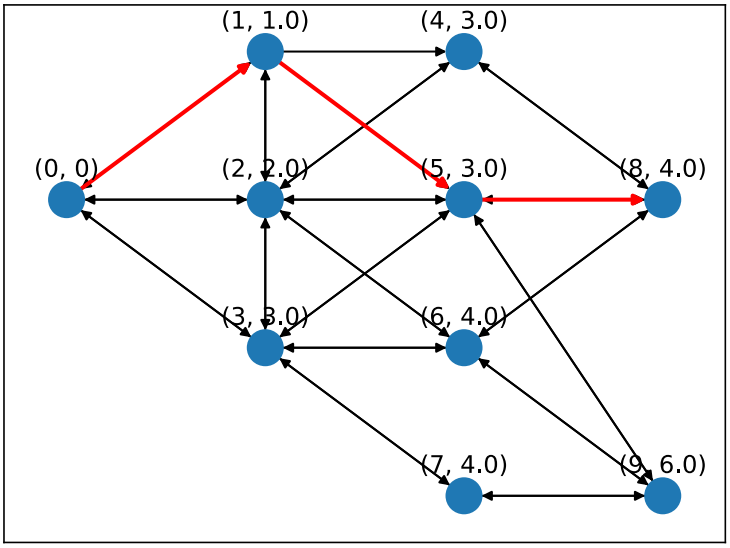


Figure 1: Search tree graph for the shortest path

Question Number 3
To modifing the code, I made two functions named: *_update_path* and fit_flow.

1. *_updated_path* Using the formula to update all the travel times in the shortest path.
2. fit_flow Represent a function that give flow and step to work on a loop and update the travel time too.

Question Number 4, 5 After running the code here are the results:
The shortest distance from node 0 to node 8 at the end is 8.25 And this is the path: [0, 1, 2, 4, 8]
We can see steps as some sub_plot in figure 2a, 2b, 2c, 2d and 2e

2 Shortest Path With Data

Question Number 1

To account this part I modified *_update_path* function to maintain the required.
Moreover, I made *DataLoader.py* file to prepare the data for analysis.

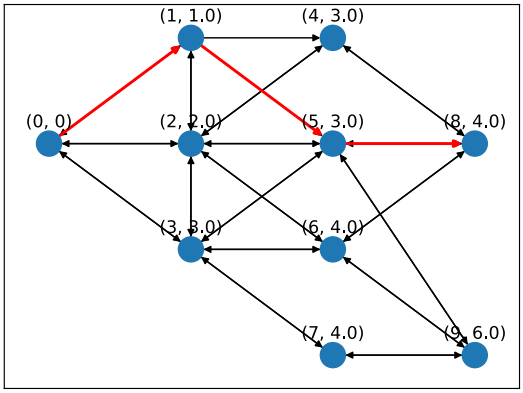
Question Number 2

according to my $id = 99104468$, source node is 3859 and sink node is 11.

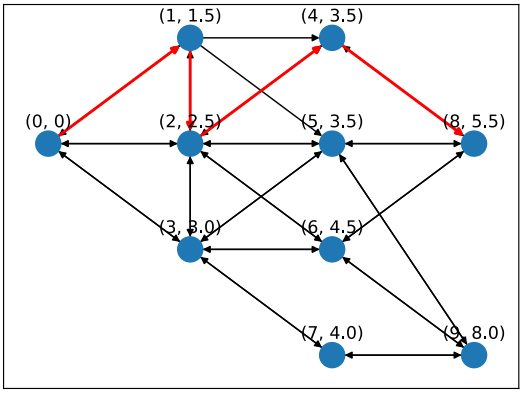
Question Number 3

a) I modified my code to address these problems as $ShortestPath\_V2$. here are all modifications:

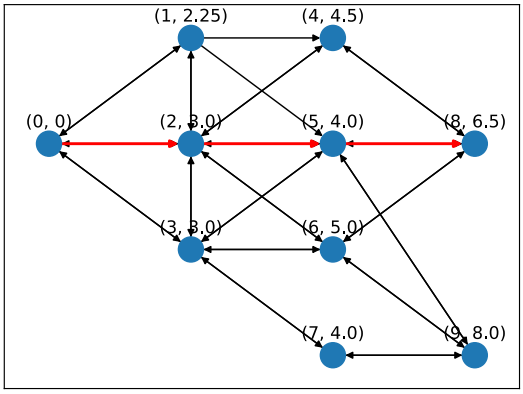
1. I add a list of nodes, which carry the index of each node. Because the name of node is not necessarily the index of the node and can vary. All index problems solved.
2. I also omit the plot-graph code due to the large scale of data.



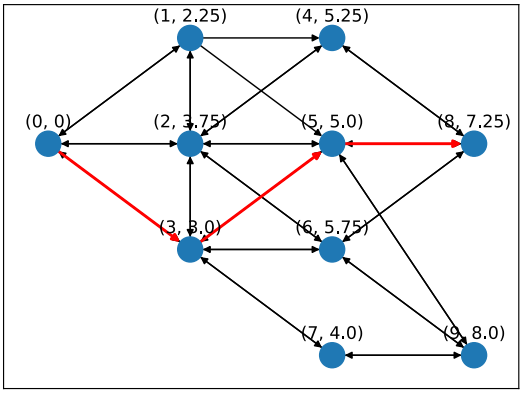
(a) step 0



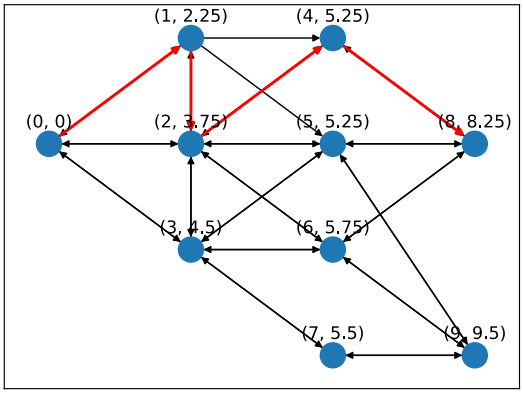
(b) step 1



(c) step2



(d) step 3



(e) step 4

3. Hopefully my code was perfect for both situations in part 1 and 2. Therefore, no significant changes was needed.

b) I made a new .py file, named *DataLoader.py* which can prepare the data for us as an ordered dictionary.

Question Number 4, 5, 6, 7

After running the code here are the results:

a) The total travel time for accommodating all 8000 vehicles to the shortest path in veh*hr is $TravelTime = 60348 veh * hr$.

b) Here are all orders in all 8 sections:

path1: 3859.0, 3864.0, 4140.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4090.0, 3887.0, 3603.0, 3602.0, 3564.0, 3562.0, 3563.0, 3611.0, 3615.0, 3617.0, 3613.0, 3549.0, 3543.0, 3540.0, 3541.0, 3597.0, 3551.0, 3496.0, 3497.0, 3664.0, 3593.0, 3587.0, 3582.0, 1859.0, 1854.0, 1851.0, 1749.0, 1720.0, 164.0, 180.0, 183.0, 1866.0, 1680.0, 1671.0, 1661.0, 1662.0, 1644.0, 99.0, 84.0, 1608.0, 70.0, 1593.0, 1585.0, 52.0, 51.0, 54.0, 55.0, 48.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path2: 3859.0, 3864.0, 4140.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4090.0, 3887.0, 3603.0, 3602.0, 3564.0, 3562.0, 3563.0, 3611.0, 3615.0, 3617.0, 3613.0, 3549.0, 3543.0, 3540.0, 3541.0, 3597.0, 3551.0, 3496.0, 3497.0, 3664.0, 3593.0, 3587.0, 3582.0, 1859.0, 1854.0, 1851.0, 1749.0, 1720.0, 164.0, 180.0, 183.0, 1866.0, 1680.0, 1671.0, 1661.0, 1662.0, 1644.0, 99.0, 84.0, 1608.0, 70.0, 1593.0, 1585.0, 52.0, 51.0, 54.0, 55.0, 48.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path3: 3859.0, 3861.0, 3990.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4090.0, 3887.0, 3603.0, 3602.0, 3564.0, 3562.0, 3563.0, 3611.0, 3615.0, 3617.0, 3613.0, 3549.0, 3543.0, 3540.0, 3541.0, 3597.0, 3551.0, 3496.0, 3497.0, 3664.0, 3593.0, 3587.0, 3582.0, 1859.0, 1854.0, 1851.0, 1749.0, 1720.0, 164.0, 180.0, 183.0, 1866.0, 1680.0, 1671.0, 1661.0, 1662.0, 1644.0, 99.0, 84.0, 1608.0, 70.0, 1593.0, 1585.0, 52.0, 51.0, 53.0, 176.0, 173.0, 174.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path4: 3859.0, 3864.0, 4140.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3856.0, 3858.0, 3995.0, 3979.0, 3828.0, 3827.0, 3829.0, 3806.0, 3790.0, 3781.0, 3780.0, 3766.0, 3767.0, 3959.0, 3701.0, 3698.0, 3694.0, 3695.0, 2235.0, 1883.0, 2221.0, 1845.0, 1710.0, 1712.0, 1713.0, 1841.0, 1828.0, 1691.0, 1667.0, 1669.0, 1663.0, 1657.0, 1658.0, 1869.0, 1868.0, 1670.0, 1665.0, 1666.0, 1819.0, 1820.0, 1621.0, 1617.0, 1594.0, 1581.0, 1585.0, 52.0, 51.0, 54.0, 55.0, 48.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path5: 3859.0, 3861.0, 3990.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4090.0, 3887.0, 3603.0, 3602.0, 3564.0, 3562.0, 3563.0, 3611.0, 3615.0, 3617.0, 3613.0, 3549.0, 3543.0, 3540.0, 3541.0, 3597.0, 3551.0, 3496.0, 3497.0, 3664.0, 3593.0, 3587.0, 3582.0, 1859.0, 1854.0, 1851.0, 1749.0, 1720.0, 164.0, 180.0, 183.0, 1866.0, 1680.0, 1662.0, 1644.0, 99.0, 84.0, 1608.0, 70.0, 1593.0, 1585.0, 52.0, 51.0, 53.0, 176.0, 173.0, 174.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path6: 3859, 3864, 4140, 11124, 11123, 4016, 3870, 3868, 3866, 4086, 4004, 4003, 3878, 3863, 4008, 4009, 4078, 3522, 3521, 3523, 3512, 3657, 3648, 3605, 3505, 3506, 3490, 3485, 3486, 3581, 3478, 3623, 3457, 3622, 3579, 3578, 3455, 1765, 1762, 1763, 1738, 1739, 1725, 1722, 1721, 296, 1720, 164, 180, 183, 1866, 1680, 1671, 1661, 1662, 1644, 99, 84, 1608, 70, 1593, 1585, 52, 51, 53, 176, 173, 174, 49, 175, 179, 18, 4, 19, 6, 11

path7: 3859.0, 3861.0, 3990.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4086.0, 4004.0, 4006.0, 4001.0, 3999.0, 4000.0, 4079.0, 3855.0, 3810.0, 3824.0, 4093.0, 4094.0, 3489.0,

3630.0, 3643.0, 3629.0, 3573.0, 3570.0, 1862.0, 1863.0, 1752.0, 1881.0, 1846.0, 1845.0, 1710.0, 1712.0, 1713.0, 1841.0, 1828.0, 1691.0, 1667.0, 1669.0, 1663.0, 1657.0, 1658.0, 1869.0, 1656.0, 1649.0, 1650.0, 1819.0, 1820.0, 1621.0, 1617.0, 1594.0, 1581.0, 1585.0, 52.0, 51.0, 54.0, 55.0, 48.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

path8: 3859.0, 3864.0, 4140.0, 11124.0, 11123.0, 4016.0, 3870.0, 3868.0, 3866.0, 4090.0, 3887.0, 3603.0, 3602.0, 3564.0, 3562.0, 3563.0, 3611.0, 3615.0, 3617.0, 3613.0, 3549.0, 3596.0, 3542.0, 3541.0, 3597.0, 3551.0, 3496.0, 3497.0, 3664.0, 3593.0, 3587.0, 3582.0, 1859.0, 1854.0, 1851.0, 1749.0, 1720.0, 164.0, 180.0, 183.0, 1866.0, 1680.0, 1662.0, 1644.0, 99.0, 84.0, 1608.0, 70.0, 1593.0, 1585.0, 52.0, 51.0, 53.0, 176.0, 173.0, 174.0, 49.0, 175.0, 179.0, 18.0, 4.0, 19.0, 6.0, 11.0

c) Here are number of nodes in each path:

in the 1th path the number of nodes is: 65

in the 2th path the number of nodes is: 65

in the 3th path the number of nodes is: 66

in the 4th path the number of nodes is: 66

in the 5th path the number of nodes is: 64

in the 6th path the number of nodes is: 76

in the 7th path the number of nodes is: 68

in the 8th path the number of nodes is: 64

d) Run time for this question is about 10 *hours* and 57 *min* and 33 *sec*