0716067 HW2

Part 1

```
def bfs(start = 2270143902, end = 1079387396):
    with open('edges.csv', newline='') as csvfile:
        edges = csv.DictReader(csvfile)
        nodes = []
        edgess = []
        for edge in edges:
            edge['start'] = int(edge['start'])
            edge['end'] = int(edge['end'])
            edge['distance'] = float(edge['distance'])
            edge['speed limit'] = float(edge['speed limit'])
            nodes.append(edge['start'])
            nodes.append(edge['end'])
            edgess.append(edge)
        nodes = list(dict.fromkeys(nodes))
        nodes.sort()
        visited = {}
        previos = {}
        distance = {}
        for node in nodes:
            visited[node] = 0
            previos[node] = 0
            distance[node] = 0
        queue = []
        queue.append(start)
        visited[start] = 1
        num_visited = 0
        while queue:
            s = queue.pop(0)
            for edge in edgess:
                if edge['start'] == s:
                    if visited[edge['end']] == 0:
                        num_visited += 1
                        e = edge['end']
                        queue.append(e)
                        visited[e] = 1
                        previos[e] = edge['start']
                        distance[e] = edge['distance']
                        if e == end:
                            queue.clear()
                            break
        num_visited = sum(list(visited.values()))-1
        path = []
        dist = 0.0
        path.append(end)
        current_node = end
        while previos[current_node] != 0:
            path.append(previos[current_node])
            dist += distance[current_node]
            current_node = previos[current_node]
        path.reverse()
        return path, dist, num_visited
```

Store the edges into a list of dictionaries. Declare an array of "visited" to store if the node is visited, an array of "previous" to store the predecessor of the node and an array of "distance" to store the distance between current node and previous node. The most crucial part is to use a queue to store the nodes we are going to visit later. Start from the start node, push adjacent nodes into queue if it is not visited and pop the top one and repeat the process. After visiting the end node, rebuild the path from start to end using the previous array and get the distance adding the value from the distance array.

Part 2

```
def dfs(start = 2270143902, end = 1079387396):
    with open('edges.csv', newline='') as csvfile:
        edges = csv.DictReader(csvfile)
        nodes = []
        edgess = []
        for edge in edges:
            edge['start'] = int(edge['start'])
            edge['end'] = int(edge['end'])
            edge['distance'] = float(edge['distance'])
            edge['speed limit'] = float(edge['speed limit'])
            nodes.append(edge['start'])
            nodes.append(edge['end'])
            edgess.append(edge)
        nodes = list(dict.fromkeys(nodes))
        nodes.sort()
        visited = {}
        previos = {}
        distance = {}
        for node in nodes:
            visited[node] = 0
            previos[node] = 0
            distance[node] = 0
        stack = []
        stack.append(start)
        visited[start] = 1
        num_visited = 0
        while stack:
            s = stack.pop(-1)
            for edge in edgess:
                if edge['start'] == s:
                    if visited[edge['end']] == 0:
                        num_visited += 1
                        e = edge['end']
                        stack.append(e)
                        visited[e] = 1
                        previos[e] = edge['start']
                        distance[e] = edge['distance']
                        if e == end:
                            stack.clear()
                            break
        num_visited = sum(list(visited.values()))-1
        path = []
        dist = 0.0
        path.append(end)
        current_node = end
        while previos[current_node] != 0:
            path.append(previos[current_node])
            dist += distance[current_node]
            current_node = previos[current_node]
        path.reverse()
        return path, dist, num_visited
```

Store the edges into a list of dictionaries. Declare an array of "visited" to store if the node is visited, an array of "previous" to store the predecessor of the node and an array of "distance" to store the distance between current node and previous node. The most crucial part is to use a stack to store the nodes we are going to visit later. Start from the start node, push adjacent nodes into stack if it is not visited and pop the bottom one and repeat the process. After visiting the end node, rebuild the path from start to end using the previous array and get the distance adding the value from the distance array.

Part 3

```
def ucs(start = 2270143902, end = 1079387396):
    with open('edges.csv', newline='') as csvfile:
        edges = csv.DictReader(csvfile)
        nodes = []
        edgess = []
        for edge in edges:
            edge['start'] = int(edge['start'])
            edge['end'] = int(edge['end'])
            edge['distance'] = float(edge['distance'])
            edge['speed limit'] = float(edge['speed limit'])
            nodes.append(edge['start'])
            nodes.append(edge['end'])
            edgess.append(edge)
        nodes = list(dict.fromkeys(nodes))
        nodes.sort()
        visited = {}
        previos = {}
        distance = []
        for node in nodes:
            visited[node] = 0
            previos[node] = 0
        distance.append([start, 0.0, 0])
        visited[start] = 1
        num_visited = 0
        dist = 0.0
        while distance:
            distance.sort(key=lambda x: x[1])
            s = distance[0][0]
            d = distance[0][1]
            p = distance[0][2]
            if visited[s] == 0:
                visited[s] = 1
                previos[s] = p
            distance.pop(0)
            for edge in edgess:
                if edge['start'] == s:
                    if visited[edge['end']] == 0:
                        e = edge['end']
                        distance.append([edge['end'], d+edge['distance'], s])
                        dist = d+edge['distance']
                        if e == end:
                            previos[e] = s
                            distance.clear()
                            break
        num_visited += sum(list(visited.values()))-1
        path = []
        path.append(end)
        current_node = end
        while previos[current_node] != 0:
            path.append(previos[current_node])
            current_node = previos[current_node]
        path.reverse()
        return path, dist, num_visited
```

Store the edges into a list of dictionaries. Declare an array of "visited" to store if the node is visited, an array of "previous" to store the predecessor of the node and a list of "distance" to store current node, the path distance between current node and start node, and temporary predecessor. Start from the start node, push adjacent nodes into queue if it is not visited. Choose the next node with the least distance stored in the "distance" list and repeat the process. After visiting the end node, rebuild the path from start to end using the previous array and get the distance from the value we stored in "distance" list.

Part 4

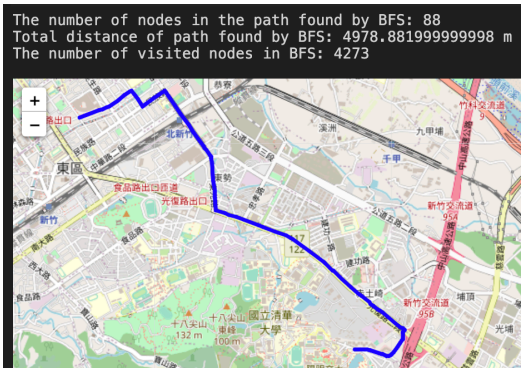
```python
def astar(start = 2270143902, end = 1079387396):
    with open('edges.csv', newline='') as csvfile:
        edges = csv.DictReader(csvfile)
        nodes = []
        edgess = []
        for edge in edges:
            edge['start'] = int(edge['start'])
            edge['end'] = int(edge['end'])
            edge['distance'] = float(edge['distance'])
            edge['speed limit'] = float(edge['speed limit'])
            nodes.append(edge['start'])
            nodes.append(edge['end'])
            edgess.append(edge)
        with open('heuristic.csv', newline='') as csvfile2:
            heuristics = csv.DictReader(csvfile2)
            heuristicss = {}
            for heuristic in heuristics:
                heuristicss[int(heuristic['node'])] = float(heuristic[str(end)])
            nodes = list(dict.fromkeys(nodes))
            nodes.sort()
            visited = {}
            previos = {}
            distance = []
            for node in nodes:
                visited[node] = 0
                previos[node] = 0
            distance.append([start, 0, heuristicss[start], 0])
            visited[start] = 1
            num_visited = 0
            dist = 0.0
            while distance:
                distance.sort(key=lambda x: x[1]+x[2])
                s = distance[0][0]
                d = distance[0][1]
                if visited[s] == 0:
                    visited[s] = 1
                    previos[s] = distance[0][3]
                distance.pop(0)
                for edge in edgess:
                    if edge['start'] == s:
                        if visited[edge['end']] == 0:
                            e = edge['end']
                            distance.append([edge['end'], d+edge['distance'], heuristicss[edge['end']], edge['start']])
                            dist = d+edge['distance']
                            if e == end:
                                previos[end] = edge['start']
                                distance.clear()
                                break
            num_visited = sum(list(visited.values()))-1
            path = []
            path.append(end)
            current_node = end
            while previos[current_node] != 0:
                path.append(previos[current_node])
                current_node = previos[current_node]
            path.reverse()
            return path, dist, num_visited
```

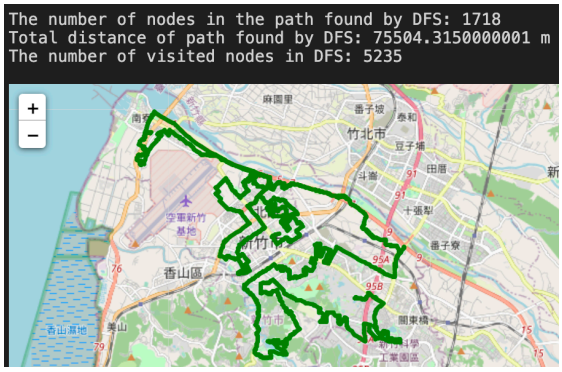
Store the edges into a list of dictionaries. Declare an array of "visited" to store if the node is visited, an array of "previous" to store the predecessor of the node and a list of "distance" to store current node, the path distance between current node and start node, the heuristic value, and temporary predecessor. Start from the start node, push adjacent nodes into queue if it is not visited. Choose the next node with the least value of path distance between start and current node plus the heuristic value and repeat the process. After visiting the end node, rebuild the path from start to end using the previous array and get the distance from the value we stored in "distance" list.

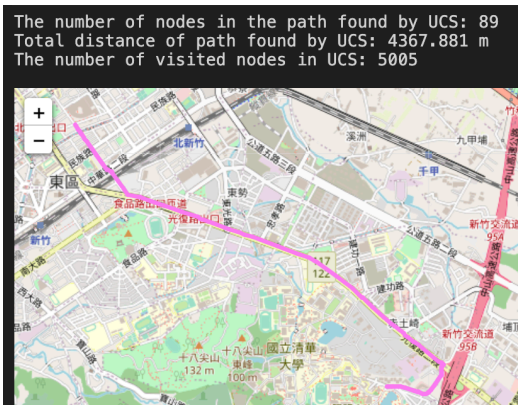
Part 5

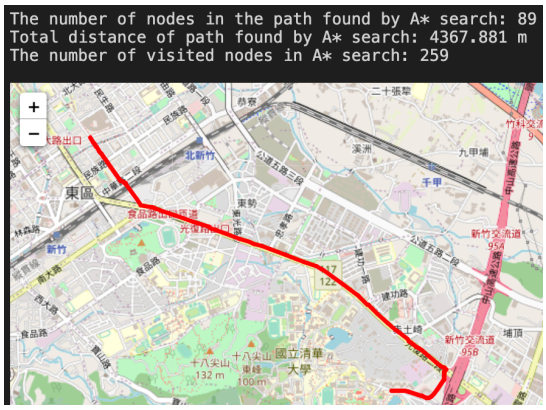
National Yang Ming Chiao Tung University -> Big City Shopping Mall

(ID: 2270143902) (ID: 1079387396)







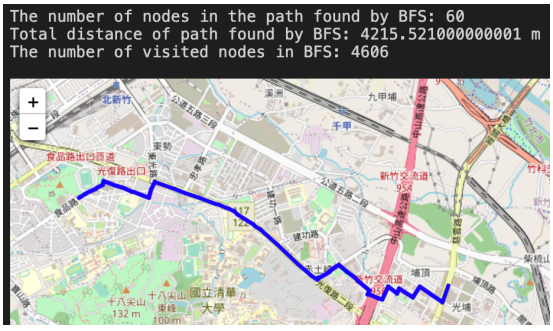


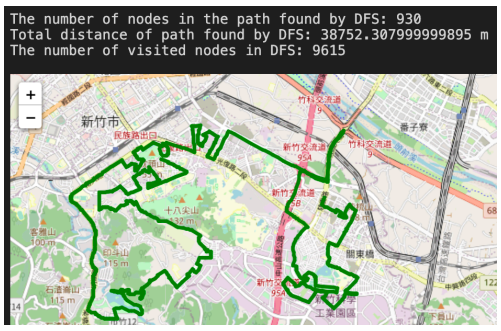
The number of nodes in the path: DFS >> UCS = A* > BFS

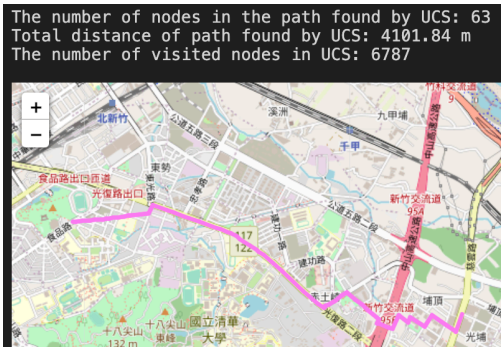
Total distance: DFS >> BFS > UCS = A*

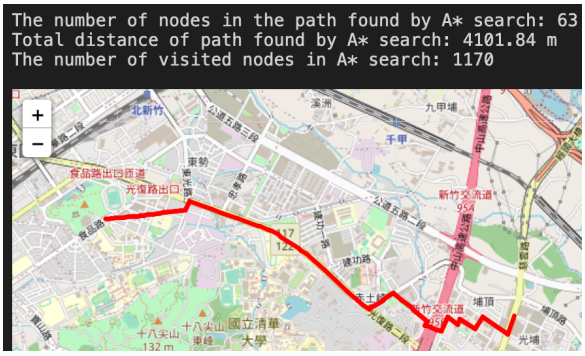
The number of visited nodes: DFS > UCS > BFS >> A*

Hsinchu Zoo (ID: 426882161) -> COSTCO Hsinchu Store (ID: 1737223506)









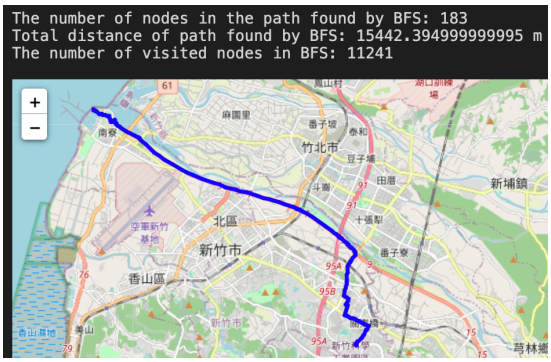
The number of nodes in the path: DFS >> UCS = A* > BFS

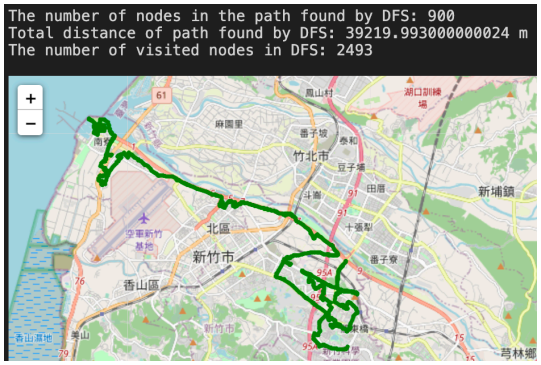
Total distance: DFS >> BFS > UCS = A*

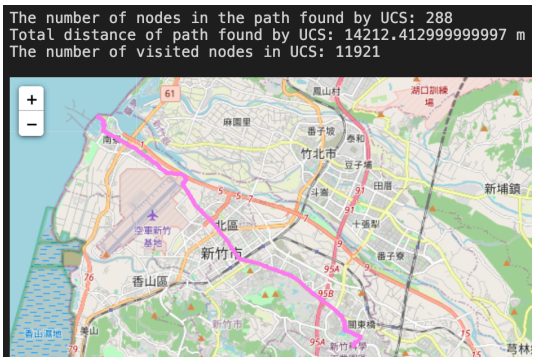
The number of visited nodes: DFS > UCS > BFS > A*

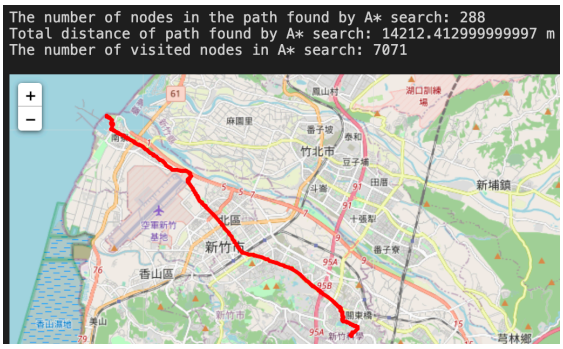
National Experimental High School -> Nanliao Fishing Port

(ID: 1718165260) (ID: 8513026827)









The number of nodes in the path: DFS >> UCS = A* > BFS

Total distance: DFS >> BFS > UCS = A*

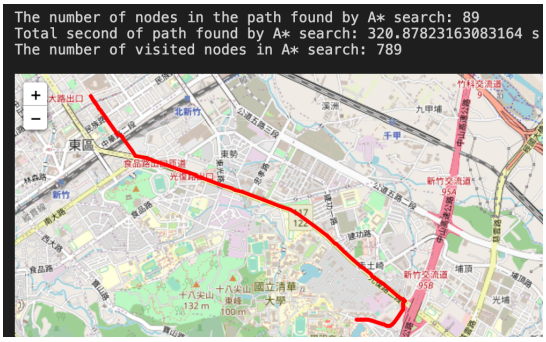
The number of visited nodes: UCS > BFS > A* > DFS

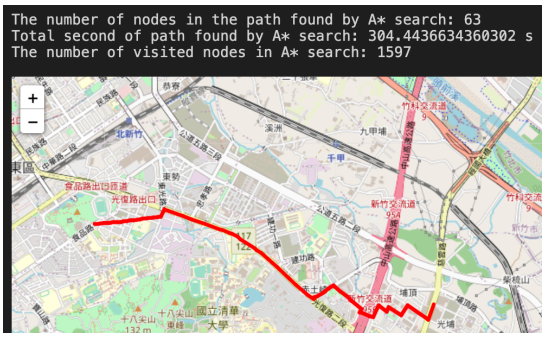
To sum up, UCS and A* has the best result, after that is BFS, and the last one is DFS. DFS uses the greatest number of nodes in the path, BFS uses the least number of nodes, and UCS and A* are in the middle. Most of the time, DFS visits the greatest number of nodes and A* visits the least number of nodes.

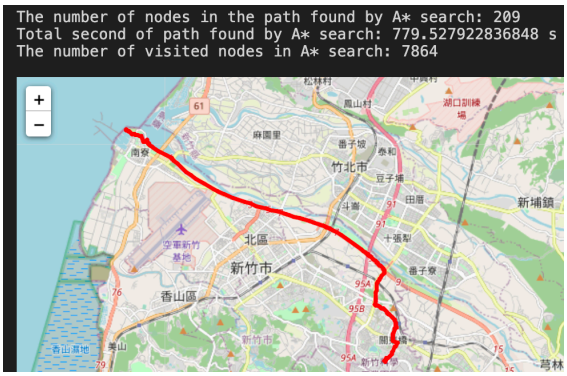
Part 6

```
def astar_time(start = 2270143902, end = 1079387396):
    with open('edges.csv', newline='') as csvfile:
        edges = csv.DictReader(csvfile)
        nodes = []
        edgess = []
        for edge in edges:
            edge['start'] = int(edge['start'])
            edge['end'] = int(edge['end'])
            edge['distance'] = float(edge['distance'])
            edge['speed limit'] = float(edge['speed limit'])
            edge['time'] = float(edge['distance'])/float(edge['speed limit'])*3.6
            nodes.append(edge['start'])
            nodes.append(edge['end'])
            edgess.append(edge)
        with open('heuristic.csv', newline='') as csvfile2:
            heuristics = csv.DictReader(csvfile2)
            heuristicss = {}
            for heuristic in heuristics:
                heuristicss[int(heuristic['node'])] = float(heuristic[str(end)])
            nodes = list(dict.fromkeys(nodes))
            nodes.sort()
            visited = {}
            previous = {}
            distance = []
            for node in nodes:
                visited[node] = 0
                previous[node] = 0
            distance.append([start, 0, heuristicss[start], 0])
            visited[start] = 1
            num_visited = 0
            time = 0.0
            while distance:
                distance.sort(key=lambda x: x[1]+x[2])
                s = distance[0][0]
                d = distance[0][1]
                if visited[s] == 0:
                    visited[s] = 1
                    previous[s] = distance[0][3]
                distance.pop(0)
                for edge in edgess:
                    if edge['start'] == s:
                        if visited[edge['end']] == 0:
                            e = edge['end']
                            distance.append([edge['end'], edge['time']+d, heuristicss[edge['end']]/60*3.6, edge['start']])
                            if e == end:
                                time = edge['time']+d
                                previous[end] = edge['start']
                                distance.clear()
                                break
            num_visited = sum(list(visited.values()))-1
            path = []
            path.append(end)
            current_node = end
            while previous[current_node] != 0:
                path.append(previous[current_node])
                current_node = previous[current_node]
            path.reverse()
            return path, time, num_visited
```

What's different from the original A* search is that I changed the edge cost from distance to travel time and divide the heuristic by 60, there are no speed limit greater than 60km/h, so I won't overestimate it.







Compare to the result from part 4, the first and second case has no difference. However, the third case has a completely different route to meet the minimum travel time.

Problems I met

At first, I didn't completely understand the idea of UCS and A* algorithm, after looking up lots of example, I have finally realized and gave a correct implementation.