Homework 2: Route Finding Report

Part I. Implementation (6%):

- **Screenshots and explanations of my codes:**

  Part 1 BFS:

```python
def bfs(start, end):
    # Begin your code (Part 1)
    file = open(edgeFile)
    csvreader = csv.reader(file)

    path = []                   # Path found by BFS.
    dist = 0.0                  # The distance of the path.
    num_visited = -1            # The number of visited nodes in searching.

    queue = []                  # queue[]     : used in BFS process.
    adj = {}                    # adj[x][]    : stores adjacent nodes for node x.
    dis = {}                    # dis[x, y]   : distance between node x and y.
    visited = {}                # visited[x]  : records whether a node x is visited.
    have_adj = {}               # have_adj[x] : records whether a node x has adjacent nodes.
    prev = {}                   # prev[x]     : records previous passing node x of each node.

    title = 1
    prev_id = ''
    for row in csvreader:       # Load data from edges.csv with csvreader and a for-loop.
        if title:
            title = 0
            continue
        if row[0] != prev_id:
            prev_id = row[0]
            adj[row[0]] = []
            have_adj[row[0]] = True
            visited[row[0]] = False
        visited[row[1]] = False
        dis[row[0], row[1]] = float(row[2])
        adj[row[0]].append(row[1])

    str_start = str(start)
    str_end = str(end)
    queue.append(str_start)                 # Initially, add start node into queue.
    visited[str_start] = True
    while len(queue) > 0:                   # Use a while-loop and a queue to implement BFS.
        p = queue.pop(0)                    # Pop the first element from queue and assign it to p per time.
        num_visited += 1
        if p == str_end:                    # Finish searching if p is the end node.
            break
        if have_adj.get(p, 0) == 0:         # Determine whether p has adjacent node.
            continue
        for x in adj[p]:
            if not visited[x]:              # If x has not been visited, add it into queue
                queue.append(x)             # and set its previous node to p.
                visited[x] = True
                prev[x] = p

    path.append(end)                        # From end to start node, add prvious node into path,
    p = str_end                             # and sum up their distance.
    while p != str_start:
        path.append(int(prev[p]))
        dist += dis[prev[p], p]
        p = prev[p]
    path.reverse()                          # Reverse the list to let start node at the beginning of path.

```

```
    file.close()
    return path, dist, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

Part 2 DFS (stack):

```
def dfs(start, end):
    # Begin your code (Part 2)
    file = open(edgeFile)
    csvreader = csv.reader(file)

    path = []                   # Path found by DFS.
    dist = 0.0                  # The distance of the path.
    num_visited = 0             # The number of visited nodes in searching.

    stack = []                  # stack[]     : used in DFS process.
    adj = {}                    # adj[x][]    : stores adjacent nodes for node x.
    dis = {}                    # dis[x, y]   : distance between node x and y.
    visited = {}                # visited[x]  : records whether a node x is visited.
    have_adj = {}               # have_adj[x] : records whether a node x has adjacent nodes.
    prev = {}                   # prev[x]     : records previous passing node x of each node.

    title = 1
    prev_id = ''
    for row in csvreader:       # Load data from edges.csv with csvreader and a for-loop.
        if title:
            title = 0
            continue
        if row[0] != prev_id:
            prev_id = row[0]
            adj[row[0]] = []
            have_adj[row[0]] = True
            visited[row[0]] = False
        visited[row[1]] = False
        dis[row[0], row[1]] = float(row[2])
        adj[row[0]].append(row[1])

    str_start = str(start)
    str_end = str(end)
    stack.append(str_start)                # Initially, add start node into stack.
    visited[str_start] = True
    while len(stack) > 0:                  # Use a while-loop and a stack to implement DFS.
        p = stack[-1]                      # Assign top elememt of stack to p.
        flg = 0                            # flg is to record whether p has unvisited adjacent node.
        if p == str_end:                   # Finish searching if p is the end node.
            break
        if have_adj.get(p, 0) == 0:        # Pop the top element from stack if p does not has adjacent node.
            stack.pop()
            continue
        for x in adj[p]:
            if not visited[x]:             # If x has not been visited, add it into stack
                stack.append(x)            # and set its previous node to p.
                num_visited += 1
                visited[x] = True
                prev[x] = p
                flg = 1
                break
        if flg:
            continue
        else:
            stack.pop()                    # Pop the top element of stack if all adjacent nodes of p are visited.

    path.append(end)                       # From end to start node, add prvious node into path,
    p = str_end                            # and sum up their distance.
    while p != str_start:
        path.append(int(prev[p]))
        dist += dis[prev[p], p]
        p = prev[p]
    path.reverse()                         # Reverse the list to let start node at the beginning of path.

    file.close()
    return path, dist, num_visited
```

```
    raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

Part 3 UCS:

```
def ucs(start, end):
    # Begin your code (Part 3)
    file = open(edgeFile)
    csvreader = csv.reader(file)

    path = []                   # Path found by UCS.
    dist = 0.0                  # The distance of the path.
    num_visited = 0             # The number of visited nodes in searching.

    set = {}                    # set[key]    : used to implement priority queue in UCS.
    d = {}                      # d[x]        : current minimum distance to start node for node x.
    adj = {}                    # adj[x][]    : stores adjacent nodes for node x.
    dis = {}                    # dis[x, y]   : distance between node x and y.
    visited = {}                # visited[x]  : records whether a node x is visited.
    have_adj = {}               # have_adj[x] : records whether a node x has adjacent nodes.
    prev = {}                   # prev[x]     : records previous passing node x of each node.

    title = 1
    prev_id = ''
    for row in csvreader:       # Load data from edges.csv with csvreader and a for-loop.
        if title:
            title = 0
            continue
        if row[0] != prev_id:
            prev_id = row[0]
            adj[row[0]] = []
            have_adj[row[0]] = True
            visited[row[0]] = False
        visited[row[1]] = False
        dis[row[0], row[1]] = float(row[2])
        adj[row[0]].append(row[1])

    str_start = str(start)
    str_end = str(end)
    visited[str_start] = True
    d[str_start] = 0                        # Initially, set d[start] to 0, and update set[start].
    set[str_start] = d[str_start]
    while len(set) > 0:                     # Use a while-loop and a priority queue to implement UCS.
        p = min(set, key=set.get)           # Get the node which has minimum set[node] from set and assign it to p.
        if not visited[p]:                  # If p is unvisited, then mark it as visited.
            visited[p] = True
            num_visited += 1
        if p == str_end:                    # Finish searching if p is the end node.
            break
        if have_adj.get(p, 0) == 0:         # Pop p from the priority queue if it does not has adjacent node.
            set.pop(p)
            continue
        for x in adj[p]:
            if d.get(x, 0) == 0 or d[p] + dis[p, x] < d[x]: # If d[x] does not have value or d[p] + dis[p, x] < d[x],
                d[x] = dis[p, x] + d[p]                     # then update d[x] with new minimum distance,
                prev[x] = p                                 # set x's previous node to p.
                set[x] = d[x]                               # and, add x into priority queue.
        set.pop(p)                          # Pop p from the priority queue

    dist = d[str_end]                       # Assign d[end] to dist.
    path.append(end)                        # From end to start node, add prvious node into path.
    p = str_end
    while p != str_start:
        path.append(int(prev[p]))
        p = prev[p]
    path.reverse()                          # Reverse the list to let start node at the beginning of path.

    file.close()
    return path, dist, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

Part 4 A*:

```python
def astar(start, end):
    # Begin your code (Part 4)
    file1 = open(edgeFile)
    file2 = open(heuristicFile)
    csvreader1 = csv.reader(file1)
    csvreader2 = csv.reader(file2)

    path = []                       # Path found by A*.
    dist = 0.0                      # The distance of the path.
    num_visited = 0                 # The number of visited nodes in searching.

    set = {}                        # set[key]          : used to implement priority queue in A*.
    d = {}                          # d[x]              : current minimum distance to start node for node x.
    adj = {}                        # adj[x][]          : stores adjacent nodes for node x.
    dis = {}                        # dis[x, y]         : distance between node x and y.
    dis_To_Goal = {}                # dis_To_Goal[x, y] : Euclidean distance from node x to y.
    visited = {}                    # visited[x]        : records whether a node x is visited.
    have_adj = {}                   # have_adj[x]       : records whether a node x has adjacent nodes.
    prev = {}                       # prev[x]           : records previous passing node x of each node.

    title = 1
    prev_id = ''
    for row in csvreader1:          # Load data from edges.csv with csvreader1 and a for-loop.
        if title:
            title = 0
            continue
        if row[0] != prev_id:
            prev_id = row[0]
            adj[row[0]] = []
            have_adj[row[0]] = True
            visited[row[0]] = False
        visited[row[1]] = False
        dis[row[0], row[1]] = float(row[2])
        adj[row[0]].append(row[1])

    title = 1
    for row in csvreader2:          # Load data from heuristic.csv with csvreader2 and a for-loop.
        if title:
            goal1 = row[1]
            goal2 = row[2]
            goal3 = row[3]
            title = 0
            continue
        dis_To_Goal[row[0]] = {}
        dis_To_Goal[row[0]][goal1] = float(row[1])
        dis_To_Goal[row[0]][goal2] = float(row[2])
        dis_To_Goal[row[0]][goal3] = float(row[3])

    str_start = str(start)
    str_end = str(end)
    visited[str_start] = True
    d[str_start] = 0                        # Initially, set d[start] to 0, and update set[start].
    set[str_start] = d[str_start]
    while len(set) > 0:                     # Use a while-loop and a priority queue to implement A*.
        p = min(set, key=set.get)           # Get the node which has minimum set[node] from set and assign it to p.
        if not visited[p]:                  # If p is unvisited, then mark it as visited.
            visited[p] = True
            num_visited += 1
        if p == str_end:                    # Finish searching if p is the end node.
            break
        if have_adj.get(p, 0) == 0:         # Pop p from the priority queue if it does not has adjacent node.
            set.pop(p)
            continue
        for x in adj[p]:
            if d.get(x, 0) == 0 or d[p] + dis[p, x] < d[x]: # If d[x] does not have value or d[p] + dis[p, x] < d[x]
                d[x] = dis[p, x] + d[p]                     # then update d[x] with new minimum distance,
                prev[x] = p                                 # set x's previous node to p,
                set[x] = d[x] + dis_To_Goal[x][str_end]     # and update set[x] with d[x] + dis_To_Goal[x][str_end].
        set.pop(p)                          # Pop p from the priority queue.

    dist = d[str_end]                       # Assign d[end] to dist.
    path.append(end)                        # From end to start node, add prvious node into path.
```

```
    p = str_end
    while p != str_start:
        path.append(int(prev[p]))
        p = prev[p]
    path.reverse()                          # Reverse the list to let start node at the beginning of path.

    file1.close()
    file2.close()
    return path, dist, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part 6 (Bonus) A* (time):

```
def astar_time(start, end):
    # Begin your code (Part 6)
    file1 = open(edgeFile)
    csvreader1 = csv.reader(file1)
    file2 = open(heuristicFile)
    csvreader2 = csv.reader(file2)

    path = []                       # Path found by A*(time).
    time = 0.0                      # The time of the path.
    num_visited = 0                 # The number of visited nodes in searching.

    set = {}                        # set[key]          : used to implement priority queue in A*.
    pass_time = {}                  # pass_time[x]      : current minimum time from start node to node x.
    adj = {}                        # adj[x][]          : stores adjacent nodes for node x.
    dis = {}                        # dis[x, y]         : distance between node x and y.
    speed = {}                      # speed[x, y]       : speed limit between node x and y.
    dis_To_Goal = {}                # dis_To_Goal[x, y] : Euclidean distance from node x to y.
    visited = {}                    # visited[x]        : records whether a node x is visited.
    have_adj = {}                   # have_adj[x]       : records whether a node x has adjacent nodes.
    prev = {}                       # prev[x]           : records previous passing node x of each node.
    max_speed = 0.0                 # Records max speed limit from heuristic.csv.

    title = 1
    prev_id = ''
    for row in csvreader1:          # Load data from edges.csv with csvreader1 and a for-loop.
        if title:
            title = 0
            continue
        if row[0] != prev_id:
            prev_id = row[0]
            adj[row[0]] = []
            have_adj[row[0]] = True
            visited[row[0]] = False
        visited[row[1]] = False
        dis[row[0], row[1]] = float(row[2])
        speed[row[0], row[1]] = float(row[3]) * 5/18
        if speed[row[0], row[1]] > max_speed:
            max_speed = speed[row[0], row[1]]
        adj[row[0]].append(row[1])

    title = 1
    for row in csvreader2:          # Load data from heuristic.csv with csvreader2 and a for-loop.
        if title:
            goal1 = row[1]
            goal2 = row[2]
            goal3 = row[3]
            title = 0
            continue
        dis_To_Goal[row[0]] = {}
        dis_To_Goal[row[0], goal1] = float(row[1])
        dis_To_Goal[row[0], goal2] = float(row[2])
        dis_To_Goal[row[0], goal3] = float(row[3])

    str_start = str(start)
    str_end = str(end)
    visited[str_start] = True
    pass_time[str_start] = 0                # Initially, set pass_time[start] to 0, and update set[start].
    set[str_start] = pass_time[str_start]
    while len(set) > 0:                     # Use a while-loop and a priority queue to implement A*(time).
        p = min(set, key=set.get)           # Get the node which has minimum set[node] from set and assign it to p.
        if not visited[p]:                  # If p is unvisited, then mark it as visited.
            visited[p] = True
            num_visited += 1
```

```
        if p == str_end:                    # Finish searching if p is the end node.
            break
        if have_adj.get(p, 0) == 0:         # Pop p from the priority queue if it does not has adjacent node.
            set.pop(p)
            continue
        for x in adj[p]:
            '''
            I change the heuristic function with 'min time to the end' (distance to end / max speed limit) in this part.
            If pass_time[x] does not have value or pass_time[p] + (dis[p, x] / speed[p, x]) < pass_time[x],
            then update pass_time[x] with new minimum time, set x's previous node to p,
            and update set[x] with pass_time[x] + (dis_To_Goal[x, str_end] / max_speed).
            '''
            if pass_time.get(x, 0) == 0 or pass_time[p] + (dis[p, x] / speed[p, x]) < pass_time[x]:
                pass_time[x] = (dis[p, x] / speed[p, x]) + pass_time[p]
                prev[x] = p
                set[x] =  pass_time[x] + (dis_To_Goal[x, str_end] / max_speed)
        set.pop(p)                          # Pop p from the priority queue.

    time = pass_time[str_end]               # Assign pass_time[end] to time.
    path.append(end)                        # From end to start node, add prvious node into path.
    p = str_end
    while p != str_start:
        path.append(int(prev[p]))
        p = prev[p]
    path.reverse()                          # Reverse the list to let start node at the beginning of path.

    file1.close()
    file2.close()
    return path, time, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 6)
```

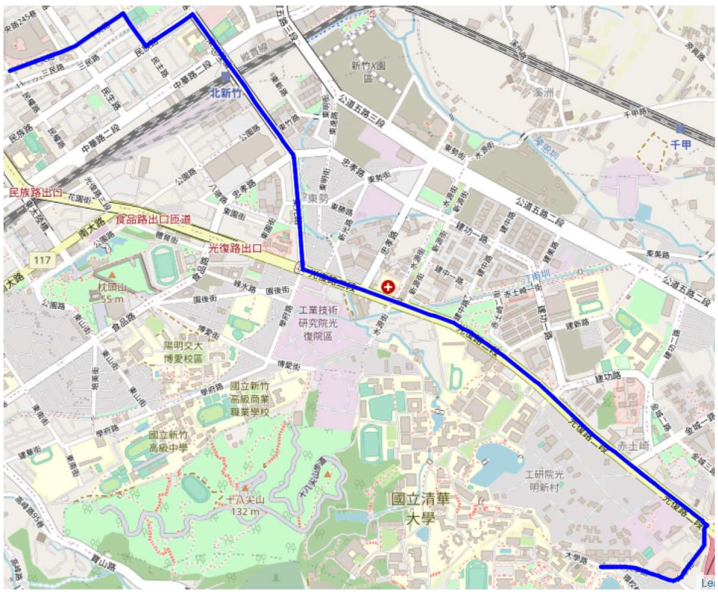
Part II. Results & Analysis (12%):

- **Screenshots of the results:**

  Test 1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

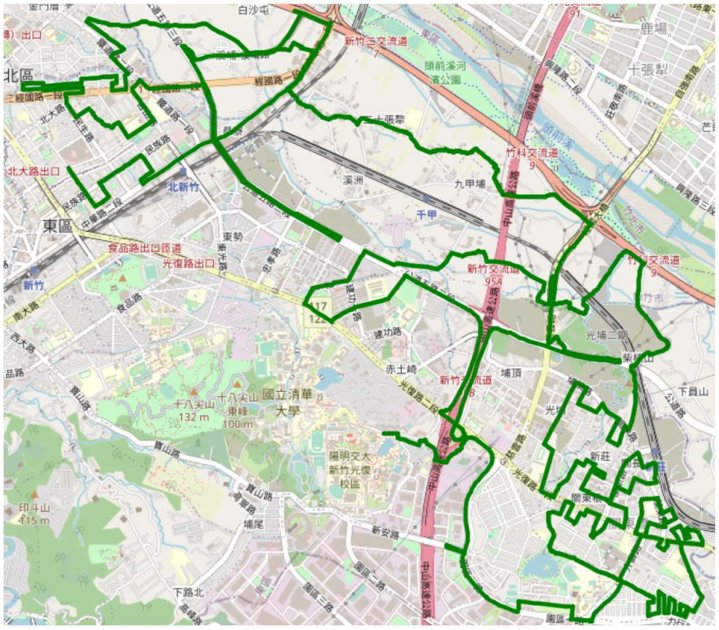
  BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4273
```



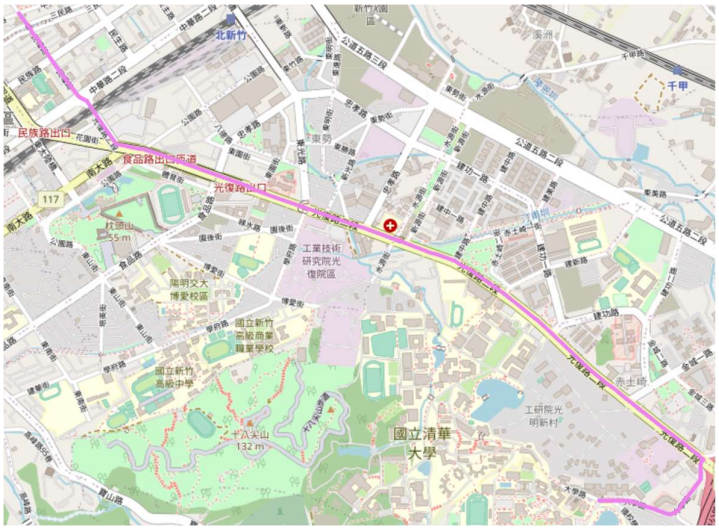
  DFS (stack):

```
The number of nodes in the path found by DFS: 1311
Total distance of path found by DFS: 48954.32099999998 m
The number of visited nodes in DFS: 3518
```



UCS:

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5085
```



A*:

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 260
```

Part 6 A*(time):

```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 1933
```

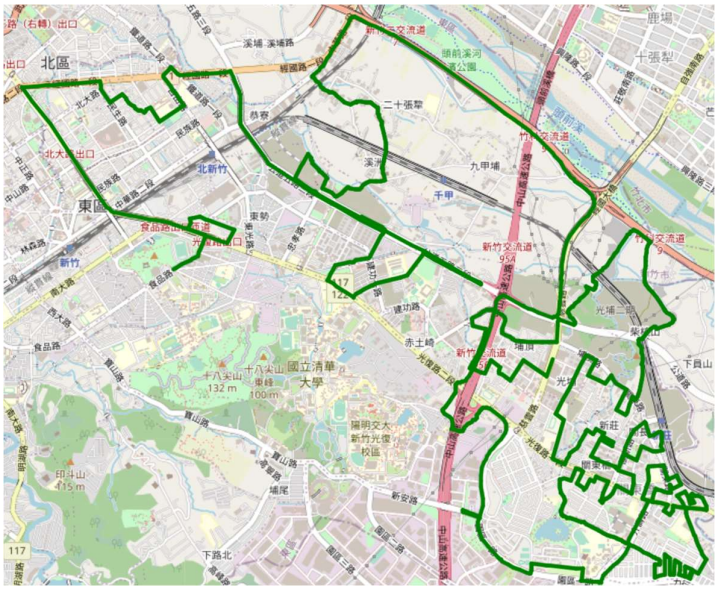
Test 2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

BFS:

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4606
```

DFS (stack):

```
The number of nodes in the path found by DFS: 1016
Total distance of path found by DFS: 43504.76899999997 m
The number of visited nodes in DFS: 10622
```



UCS:

```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7212
```

A*:

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1171
```

Part 6 A*(time):

```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603014 s
The number of visited nodes in A* search: 2869
```

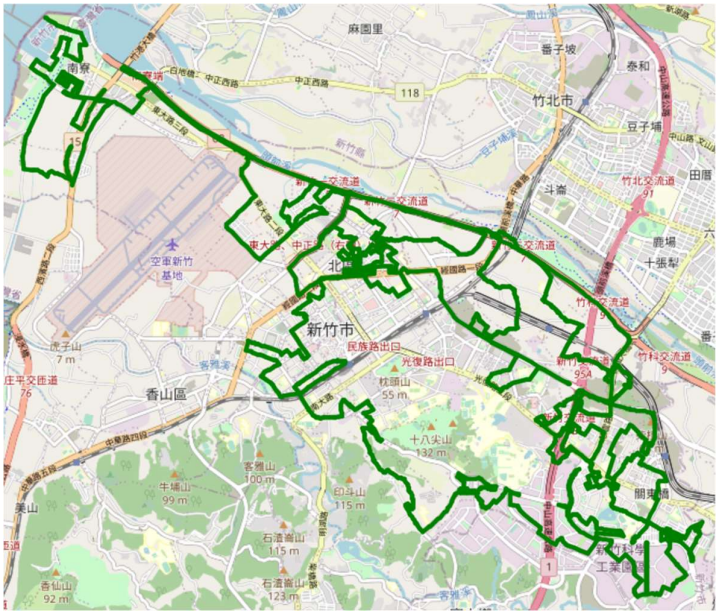
Test 3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)

BFS:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11241
```

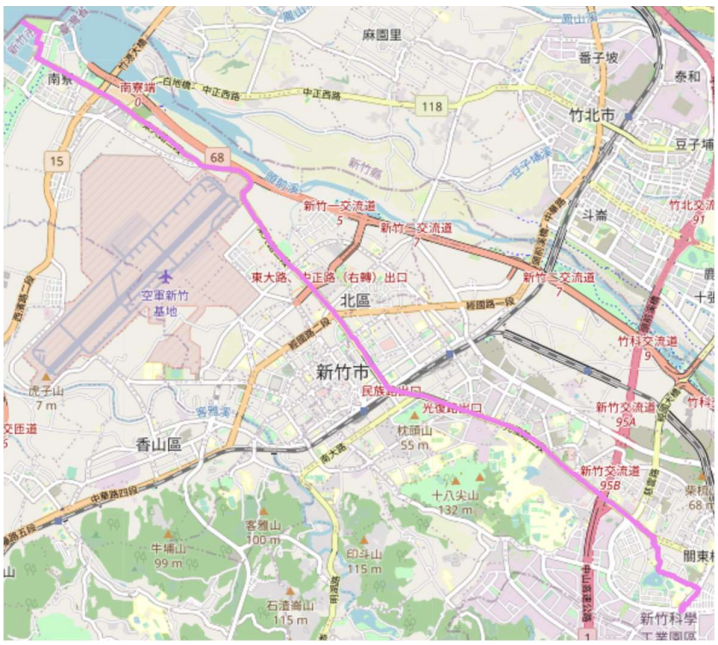
DFS (stack):

```
The number of nodes in the path found by DFS: 2635
Total distance of path found by DFS: 120440.44300000017 m
The number of visited nodes in DFS: 7517
```



UCS:

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11925
```



A*:

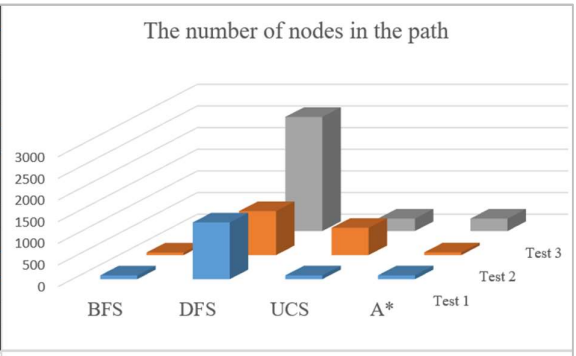
```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7072
```

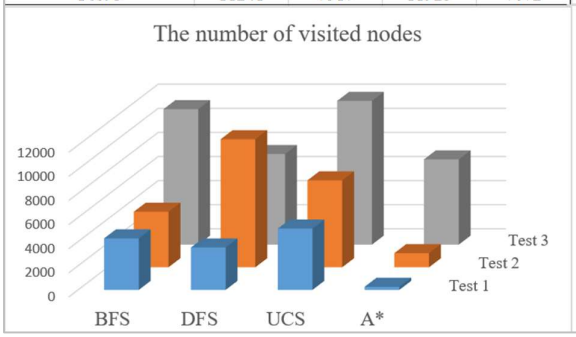
Part 6 A*(time):

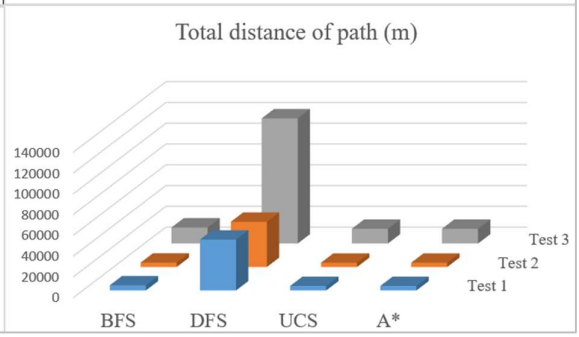
```
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.527922836848 s
The number of visited nodes in A* search: 8457
```

- **My analysis and observation:**

# of nodes in the path	BFS	DFS	UCS	A*
Test 1	88	1311	89	89
Test 2	60	1016	630	63
Test 3	183	2635	288	288
Total distance of path (m)	**BFS**	**DFS**	**UCS**	**A***
Test 1	4978.8819	48954.321	4367.881	4367.881
Test 2	4215.521	43504.769	4101.84	4101.84
Test 3	15442.395	120440.44	14212.413	14212.413
# of visited nodes	**BFS**	**DFS**	**UCS**	**A***
Test 1	4273	3518	5085	260
Test 2	4606	10622	7212	1171
Test 3	11241	7517	11925	7072



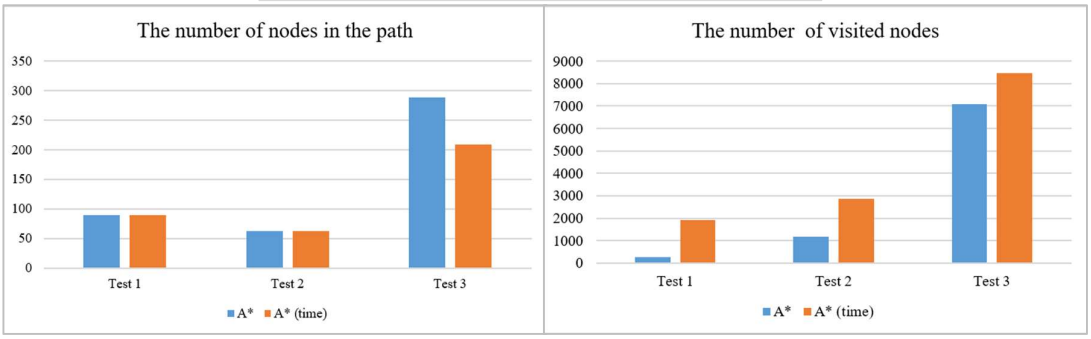




a. From the charts in the upper right and the lower right corner, we can observe that the DFS algorithm found the greatest number of nodes in the path and longest distance in each test. That means that if we use DFS as searching algorithm of our navigating system, it very likely finds a path whose number of nodes and total distance will more and longer than them of optimal solution generally.

b. From the chart in the lower left corner, we can observe that the A* algorithm visited minimum number of nodes when searching in each test. Besides, same as UCS, A* is also a minimum distance searching algorithm, but it only needs less times of visiting nodes. In sum, A* algorithm is most suitable for navigating system in this homework.

# of nodes in the path	A*	A* (time)
Test 1	89	89
Test 2	63	63
Test 3	288	209
# of visited nodes	**A***	**A* (time)**
Test 1	260	1933
Test 2	1171	2869
Test 3	7072	8457



The charts above is used to compare the differences between A* and A*(speed limit considered) algorithms in 3 tests.

After adding speed limit and pass time of a road into our heuristic function, we can observe that they show different results based on same algorithm and map information. And we know that what heuristic function we choose will deeply affect the performance of this algorithm. Take right chart above for instance, A*(time) version needs to visit more nodes than the original version. So, what attributes should we need to design heuristic function will be an important issue for us in the future.

Part III. Answer the questions (12%):

1. **Please describe a problem you encountered and how you solved it.**

   My problem: In HW2, the problem I met is designing a heuristic function in bonus part 6. In beginning, I had no idea what attributes should I use to design an admissible heuristic function.

   How did I solve the problem: After trying and modify my code repeatedly, I finally choose minimum pass time as my heuristic function whose formula is “distance to destination” / “maximum speed limit (100 km/h)”.

2. **Besides speed limit and distance, could you please come up with another attribute that is essential for route finding in the real world? Please explain the rationale.**

   I come up with two potential attributes, “traffic volume” (also known as traffic flow) and “the number of traffic light”, which would may be essential for route finding in the real world.

   “Traffic Volume” means the number of vehicles occupying a unit length of road at a given instant of time. The traffic condition would get worse or even traffic jam when the traffic volume is high although with high speed limit of that road. Based on that, traffic volume may be an important factor in route finding in the real world, and it will affect whether car drivers can reach their destinations quickly and efficiently.

   “The number of traffic light” is also a factor we would consider when choosing a driving route, I think no one would want to spend time waiting for traffic lights. In real world, we choose the route whose number of traffic light is as less as possible to get to destination more quickly.

3. **As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for mapping and localization components?**

   Before doing localization, a navigation system should do mapping first to get a map to know “how the world looks like”. Then it figures out “where it is” through previous map. The following is my idea about possible solution to mapping and localization:

   Mapping: Use “lidar device” to gain distances of everything in the lens by targeting every object seen by lens with a laser and measuring the time for the reflected light to return to the receiver. Thus, we can use these distance data to build the 3-D map. And this map can help us to do subsequent localization.

   Localization: Use the map above and map it to the 3-D coordinate system such as Cartesian coordinate system. And from that coordinate system, we can get our coordinate information of our location.

4. **The estimated time of arrival (ETA) is one of the features of Uber Eats. To provide accurate estimates for users, Uber Eats needs to dynamically update based on other attributes. Please define a dynamic heuristic function for ETA. Please explain the rationale of your design.**

My dynamic heuristic function for ETA:

$h(n) = \sum_{i<j}^{S} The\ driving\ time\ between\ location\ i\ to\ location\ j,$

$S\ is\ the\ set\ of\ estimated\ places\ to\ arrive\ for\ a\ delivery\ man\ such\ as\ stores\ or\ clients'\ residence.$

Explanation:

As we know, when a delivery man receives a delivery order, he would ride to several places to pick the meals up or take the meals to clients, that is, he will have many stops in his delivery route. In my idea, I see finishing the order as the goal of heuristic function, and I use the summation of each time interval of two stops in set S. Thus, by h(n), we can know the expected finishing time of an order for every road n, and use it to implement searching algorithm. But the problem of my function design is that it needs too much calculation and has high time complexity.