Homework 2: Route Finding

110550168

Part I. Implementation (6%):

- **Please screenshot your code snippets of Part 1 ~ Part 4, and explain your implementation.**

  Part I. BFS:

```python
    # Begin your code (Part 1)
    node = [] # adjacent list
    split = [] # to temperarily store the adjacent list
    count = 0 # check if it is the first line
    idx = [] # to store the idx (the first line)
    nodeidx = [] # to store the node index
    with open(edgeFile,'r') as file: # open file
        csvlist = csv.reader(file) # read as .csv flie
        for line in csvlist: # go through all line
            if count == 0: # if is first line, store in idx
                idx = line
                count+=1
                continue
            if split == []: # store in temp adjacent list
                split.append(line) # add line to temp adj list
                nodeidx.append(int(line[0])) # add the line index to nodeidx list
            elif split[0][0] != line[0]: # if the temp adj list has different node with current node
                node.append(split) # add temp adj list to adj list
                nodeidx.append(int(line[0])) # add current node to nodeidx list
                split = [] # reset temp adj list
                split.append(line) # add line to temp adj list
            elif line is not None: # else, add line to temp adj list
                split.append(line)
        node.append(split) # add the last temp adj list to adj list

    qu = queue.Queue() # create a queue
    anslist = [] # create a list to store parent, dist, visited
    for i in range(len(nodeidx)): # set all value to 0
        anslist.append([0,0,0])
    flag = 0 # check if is at the end
    num_visited = 0 # calculate the visited point
```

```python
    x = nodeidx.index(start) # get the start node's index
    anslist[x][0] = -1 # parent
    anslist[x][1] = 0 # distance
    anslist[x][2] = 1 # visited
    qu.put(start) # put start in queue
    while ~qu.empty(): # loop if queue is not empty
        cur = qu.get() # get the first element of queue
        if(cur != end): # if current node is not end node
            x = nodeidx.index(cur) # get current node's index
            for j in range(len(node[x])): # go through all nodes adjacent to current node
                if int(node[x][j][1]) != end: # if the adjacent node is not end node
                    try: # try to find its index
                        y = nodeidx.index(int(node[x][j][1]))
                    except ValueError:
                        # cannot find, then it means that the node has no adjacent node, skip it
                        continue
                    if anslist[y][2] == 0: # if the adjacent node is not visited
                        anslist[y][0] = x # set the adj node's parent to be current node
                        anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up the distance
                        anslist[y][2] = 1 # set the node visited
                        num_visited += 1 # the number of visited node + 1
                        qu.put(nodeidx[y]) # push back the adjacent node
```

```python
            else:
                y = nodeidx.index(end) # get end node's index
                num_visited += 1 # the number of visited node + 1
                anslist[y][0] = x # set current node to be end node's parent
                anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up the distance
```

```python
                anslist[y][2] = 1 # set the node visited
                flag = 1 # set flag to 1 when end node is finded and break
                break
        if flag == 1: # break if find end node
            break
if(flag): # if find end node
    path = [] # create path list
    y = nodeidx.index(end) # find end node's index
    dist = anslist[y][1] # total distance walked
    cur = y # set end node's index as current index
    path.append(end) # add end to path list
    while anslist[cur][0] != -1: # loop if not find the node that its parent is -1
        cur = int(anslist[cur][0]) # current index change to current node's parent's index
        path.append(nodeidx[cur]) # add current node to path index
    path.reverse() # reverse the path and get the right path
    return path,dist,num_visited # return value
else: return [], 0, num_visited # if did't find end, return the right value
# End your code (Part 1)
```

I import queue at the top, which is in the standard python library. The key point of the code is I used a queue to do bfs, and anslist helped me to store the parent node, distance that has been through to this node, and visited or not. At last, I go from the end to the start and store the path’s node in the path list, and then reverse it to get the right path.

Part 2. DFS:

```python
# Begin your code (Part 2)
node = [] # adjacent list
split = [] # to temperarily store the adjacent list
count = 0 # check if it is the first line
idx = [] # to store the idx (the first line)
nodeidx = [] # to store the node index
with open(edgeFile,'r') as file: # open file
    csvlist = csv.reader(file) # read as .csv flie
    for line in csvlist: # go through all line
        if count == 0: # if is first line, store in idx
            idx = line
            count+=1
            continue
        if split == []: # store in temp adjacent list
            split.append(line) # add line to temp adj list
            nodeidx.append(int(line[0])) # add the line index to nodeidx list
        elif split[0][0] != line[0]: # if the temp adj list has different node with current node
            node.append(split) # add temp adj list to adj list
            nodeidx.append(int(line[0])) # add current node to nodeidx list
            split = [] # reset temp adj list
            split.append(line) # add line to temp adj list
        elif line is not None: # else, add line to temp adj list
            split.append(line)
    node.append(split) # add the last temp adj list to adj list

st = [] # create a stack
anslist = [] # create a list to store parent, dist, visited
for i in range(len(nodeidx)): # set all value to 0
    anslist.append([0,0,0])
flag = 0 # check if is at the end
num_visited = 0 # calculate the visited point
x = nodeidx.index(start) # get the start node's index
anslist[x][0] = -1 # parent
anslist[x][1] = 0 # distance
anslist[x][2] = 1 # visit
st.append(start) # add start node to stack
while ~len(st): # loop if stack is not empty
    cur = st.pop() # get the top element of the stack
    if(cur != end): # if current node is not end node
        x = nodeidx.index(cur) # get current node index
        for j in range(len(node[x])): # go through all nodes adjacent to current node
            if int(node[x][j][1]) != end: # if the adj node is not end
                try: # try to find its index
                    y = nodeidx.index(int(node[x][j][1]))
                except ValueError:
                    # cannot find, then it means that the node has no adjacent node, skip it
                    continue
                if anslist[y][2] == 0: # if the adjacent node is not visited
                    anslist[y][0] = x # set the adj node's parent to be current node
                    anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up the distance
                    anslist[y][2] = 1 # set the node visited
                    num_visited += 1 # the number of visited node + 1
                    st.append(nodeidx[y]) # add the adj node to stack
            else:
                y = nodeidx.index(end) # get the end node index
                num_visited += 1 # the number of visited node + 1
                anslist[y][0] = x # set current node to be end node's parent
                anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up the distance
                anslist[y][2] = 1 # set end node visited
                flag = 1 # set flag to 1 when end node is finded and break
                break
        if flag == 1: # break if find end node
            break
```

```python
    if(flag): # if find end node
        path = [] # create path list
        y = nodeidx.index(end) # find end node's index
        dist = anslist[y][1] # total distance walked
        cur = y # set end node's index as current index
        path.append(end) # add end to path list
        while anslist[cur][0] != -1: # loop if not find the node that its parent is -1
            cur = int(anslist[cur][0]) # current index change to current node's parent's index
            path.append(nodeidx[cur]) # add current node to path index
        path.reverse() # reverse the path and get the right path
        return path,dist,num_visited # return value
    else: return [], 0, num_visited # if did't find end, return the right value
    # End your code (Part 2)
```

I use stack to do DFS, and since in python, I can use a list to implement it. And others are the same as BFS, use anslist to store the parent node, distance that has been through to this node, and visited or not, then go backward to find the path's node, store in the path list, and reverse to get the right path.

Part 3. UCS:

```python
# Begin your code (Part 3)
node = [] # adjacent list
split = [] # to temperarily store the adjacent list
count = 0 # check if it is the first line
idx = [] # to store the idx (the first line)
nodeidx = [] # to store the node index
with open(edgeFile,'r') as file: # open file
    csvlist = csv.reader(file) # read as .csv flie
    for line in csvlist: # go through all line
        if count == 0: # if is first line, store in idx
            idx = line
            count+=1
            continue
        if split == []: # store in temp adjacent list
            split.append(line) # add line to temp adj list
            nodeidx.append(int(line[0])) # add the line index to nodeidx list
        elif split[0][0] != line[0]: # if the temp adj list has different node with current node
            node.append(split) # add temp adj list to adj list
            nodeidx.append(int(line[0])) # add current node to nodeidx list
            split = [] # reset temp adj list
            split.append(line) # add line to temp adj list
        elif line is not None: # else, add line to temp adj list
            split.append(line)
    node.append(split) # add the last temp adj list to adj list

pq = queue.PriorityQueue() # create a priority queue
anslist = [] # create a list to store parent, dist, visited
for i in range(len(nodeidx)): # set all value to 0
    anslist.append([0,0,0])
flag = 0 # check if is at the end
num_visited = 0 # calculate the visited point
x = nodeidx.index(start) # get the start node index

pq.put((0,start,-1)) # push start to pq (distance, current, parent)
while ~pq.empty(): # loop if pq is not empty
    cur = pq.get() # get the first element
    x = nodeidx.index(cur[1]) # get current node's index
    if(anslist[x][2] == 0): # if current node is not visited
        num_visited += 1 # visited node + 1
        anslist[x][0] = cur[2] # set parent
        anslist[x][1] = cur[0] # set distance
        anslist[x][2] = 1 # set visited
        for j in range(len(node[x])): # go through all adjacent node
            if int(node[x][j][1]) != end: # if adj node is not end
                try: # try to find its index
                    y = nodeidx.index(int(node[x][j][1]))
                except ValueError:
                    # cannot find, then it means that the node has no adjacent node, skip it
                    continue
                if anslist[y][2] == 0: # if the adj node is not visited
                    pq.put((anslist[x][1] + float(node[x][j][2]), nodeidx[y], x)) # put it in pq
            else: # if the adj node is end
                y = nodeidx.index(end) # get end node's index
                num_visited += 1 # visited node + 1
                anslist[y][0] = x # set the end node's parent to be current node
                anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up distance
                anslist[y][2] = 1 # set end node visited
                flag = 1 # set flag to 1 when end node is finded and break
                break
        if flag == 1: # if flag is 1, find end node, break
            break
```

```python
if(flag): # if find end node
    path = [] # create path list
    y = nodeidx.index(end) # find end node's index
    dist = anslist[y][1] # total distance walked
    cur = y # set end node's index as current index
    path.append(end) # add end to path list
    while anslist[cur][0] != -1: # loop if not find the node that its parent is -1
        cur = int(anslist[cur][0]) # current index change to current node's parent's index
        path.append(nodeidx[cur]) # add current node to path index
    path.reverse() # reverse the path and get the right path
    return path,dist,num_visited # return value
else: return [], 0, num_visited # if did't find end, return the right value
# End your code (Part 3)
```

I import queue at the top, and use the priority queue to do UCS. I push the element in the priority queue if the node is adjacent to the current node (get from the top of the priority queue, so it is chosen to be visited, and nodes adjacent to the current node are the nodes that are unexpanded and should be added to priority queue) into the priority queue and I use tuples with three elements, the total distance that will be passed, the adjacent node, the parent node.

Other parts are same as BFS, DFS, use anslist to store the parent node, distance that has been through to this node, and visited or not, then go backward to find the path’s node, store in the path list, and reverse to get the right path.

Part 4. A* Search:

```python
    # Begin your code (Part 4)
    # read edge list
    node = [] # adjacent list
    split = [] # to temperarily store the adjacent list
    count = 0 # check if it is the first line
    nodeidx = [] # to store the node index
    with open(edgeFile,'r') as file: # open file
        csvlist = csv.reader(file) # read as .csv flie
        for line in csvlist: # go through all line
            if count == 0: # if is first line, count + 1
                count+=1
                continue
            if split == []: # store in temp adjacent list
                split.append(line) # add line to temp adj list
                nodeidx.append(int(line[0])) # add the line index to nodeidx list
            elif split[0][0] != line[0]: # if the temp adj list has different node with current node
                node.append(split) # add temp adj list to adj list
                nodeidx.append(int(line[0])) # add current node to nodeidx list
                split = [] # reset temp adj list
                split.append(line) # add line to temp adj list
            elif line is not None: # else, add line to temp adj list
                split.append(line)
        node.append(split) # add the last temp adj list to adj list

    # read heuristic list
    heuristic = [] # distance to the end list
    idx = [] # first line of heuristic
    heur_idx = [] # to store the index of each heuristic
    count = 0 # check if it is the first line
    with open(heuristicFile,'r') as file2: # open file
        csvlist = csv.reader(file2) # read as .csv flie
        for line in csvlist: # go through all line
            if count == 0: # if is first line, store in idx
                idx = line
                count+=1
                continue
            else: # if not first line
                heuristic.append(line) # add line to heuristic list
                heur_idx.append(int(line[0])) # store the line's index, search by node
    # A* algorithm
    endnode = idx.index(str(end)) # find which end node is
    pq = queue.PriorityQueue() # create a priority queue
    anslist = [] # create a list to store parent, dist, visited
    for i in range(len(nodeidx)): # set all value to 0
        anslist.append([0,0,0])
    flag = 0 # check if is at the end
    num_visited = 0 # calculate the visited point
    x = nodeidx.index(start) # get the start node's adj list's index
    h = heur_idx.index(start) # get the start node's heuristic's index

    pq.put((0 + float(heuristic[h][endnode]), start, -1, 0)) # push start to pq, (g(x) + h(x), current, parent, distance)
    while ~pq.empty(): # loop if pq is empty
        weight, cur, par, dis = pq.get() # get the current element
        x = nodeidx.index(cur) # get current node's index
        if(anslist[x][2] == 0): # if current node is not visited
            anslist[x][0] = par # set current node's parent
            anslist[x][1] = dis # set current node's distance
            anslist[x][2] = 1 # current node visited
            num_visited += 1 # visited node + 1
```

```python
            for j in range(len(node[x])): # go through all adj node
                if int(node[x][j][1]) != end: # if adj node is not visited
                    try: # try to find its index
                        y = nodeidx.index(int(node[x][j][1]))
                    except ValueError:
                        # cannot find, then it means that the node has no adjacent node, skip it
                        continue
                    h = heur_idx.index(int(node[x][j][1])) # get adj node's heuristic
                    if anslist[y][2] == 0: # if adj node is not visited
                        tmp_weight = anslist[x][1] + float(node[x][j][2]) + float(heuristic[h][endnode]) # calculate g(x) + h(x)
                        tmp_dist = float(node[x][j][2]) + anslist[x][1] # calculate distance
                        pq.put((tmp_weight, nodeidx[y], x, tmp_dist)) # push it to pq
                else: # if adj node is end
                    y = nodeidx.index(end) # get end node's index
                    num_visited += 1 # visited number + 1
                    anslist[y][0] = x # set end node's parent
                    anslist[y][1] = anslist[x][1] + float(node[x][j][2]) # sum up distance
                    anslist[y][2] = 1 # set end node visited
                    flag = 1 # set flag to 1 when end node is finded and break
                    break
            if flag == 1: # if flag is 1, find end node, break
                break
    if(flag): # if find end node
        path = [] # create path list
        y = nodeidx.index(end) # find end node's index
        dist = anslist[y][1] # total distance walked
        cur = y # set end node's index as current index
        path.append(end) # add end to path list
        while anslist[cur][0] != -1: # loop if not find the node that its parent is -1
            cur = int(anslist[cur][0]) # current index change to current node's parent's index
            path.append(nodeidx[cur]) # add current node to path index
        path.reverse() # reverse the path and get the right path
        return path,dist,num_visited # return value
    else: return [], 0, num_visited # if did't find end, return the right value
    # End your code (Part 4)
```

Same as UCS, but I push the node that is adjacent to the current node (get from the top of the priority queue) into the priority queue. Then I use a tuple with four elements, the total distance that will be passed plus the distance between the adjacent node and the end node, the adjacent node, the parent node and the total distance that will be passed.

Other parts are same as BFS, DFS and UCS, use anslist to store the parent node, distance that has been through to this node, and visited or not, then go backward to find the path's node, store in the path list, and reverse to get the right path.

Part 6. A* Search with Different Heuristic:

```python
    # Begin your code (Part 6)
    # read edge list
    node = [] # adjacent list
    split = [] # to temperarily store the adjacent list
    count = 0 # check if it is the first line
    nodeidx = [] # to store the node index
    with open(edgeFile,'r') as file1: # open file
        csvlist = csv.reader(file1) # read as .csv flie
        for line in csvlist: # go through all line
            if count == 0: # if is first line, count + 1
                count+=1
                continue
            else: # if is not first line, calculate the time to pass the edge
                m_to_s = float(line[2]) / (float(line[3]) * 10 / 36) # change km/h to m/s, store the time to pass the edge
                line.append(m_to_s) # add the time at the last of the line
            if split == []: # store in temp adjacent list
                split.append(line) # add line to temp adj list
                nodeidx.append(int(line[0])) # add the line index to nodeidx list
            elif split[0][0] != line[0]: # if the temp adj list has different node with current node
                node.append(split) # add temp adj list to adj list
                nodeidx.append(int(line[0])) # add current node to nodeidx list
                split = [] # reset temp adj list
                split.append(line) # add line to temp adj list
            elif line is not None: # else, add line to temp adj list
                split.append(line)
        node.append(split) # add the last temp adj list to adj list

    # read heuristic list
    heuristic = [] # distance to the end list
    idx = [] # first line of heuristic
    heur_idx = [] # to store the index of each heuristic
    count = 0 # check if it is the first line
    with open(heuristicFile,'r') as file2: # open file
        csvlist = csv.reader(file2) # read as .csv flie
        for line in csvlist: # go through all line
            if count == 0: # if is first line, store in idx
                idx = line
                count+=1
                continue
            else: # if not first line
                heuristic.append(line) # add line to heuristic list
                heur_idx.append(int(line[0])) # store the line's index, search by node
    # A* algorithm
    endnode = idx.index(str(end)) # find which end node is
    pq = queue.PriorityQueue() # create a priority queue
    anslist = [] # create a list to store parent, dist, visited
    for i in range(len(nodeidx)): # set all value to 0
        anslist.append([0,0,0])
    flag = 0 # check if is at the end
    num_visited = 0 # calculate the visited point
    x = nodeidx.index(start) # get the start node's adj list's index
    h = heur_idx.index(start) # get the start node's heuristic's index

    pq.put((0 + float(heuristic[h][endnode])/1, start, -1, 0)) # push start to pq, (g(x) + h(x), current, parent, distance)
    while ~pq.empty(): # loop if pq is empty
        weight, cur, par, sec = pq.get() # get the current element
        x = nodeidx.index(cur) # get current node's index
        if(anslist[x][2] == 0): # if current node is not visited
            anslist[x][0] = par # set current node's parent
            anslist[x][1] = sec # set current node's time
            anslist[x][2] = 1 # current node visited
            num_visited += 1 # visited node + 1
```

```python
            for j in range(len(node[x])): # go through all adj node
                if int(node[x][j][1]) != end: # if adj node is not visited
                    try: # try to find its index
                        y = nodeidx.index(int(node[x][j][1]))
                    except ValueError:
                        # cannot find, then it means that the node has no adjacent node, skip it
                        continue
                    h = heur_idx.index(int(node[x][j][1])) # get adj node's heuristic
                    if anslist[y][2] == 0: # if adj node is not visited
                        next_time = float(node[x][j][3]) + float(heuristic[h][endnode]) / (float(node[x][j][3]) * 10 / 36)
                        # next time = pass edge time + next node to end node's distance / speed
                        pq.put((anslist[x][1] + next_time, nodeidx[y], x, float(node[x][j][4]) + anslist[x][1])) # push data
                else: # if adj node is end
                    y = nodeidx.index(end) # get end node's index
                    num_visited += 1 # visited number + 1
                    anslist[y][0] = x # set end node's parent
                    anslist[y][1] = anslist[x][1] + float(node[x][j][4]) # sum up time
                    anslist[y][2] = 1 # set end node visited
                    flag = 1 # set flag to 1 when end node is finded and break
                    break
            if flag == 1: # if flag is 1, find end node, break
                break
    if(flag): # if find end node
        path = [] # create path list
        y = nodeidx.index(end) # find end node's index
        time = anslist[y][1] # the total time to reach the end node
        cur = y # set end node's index as current index
        path.append(end) # add end to path list
        while anslist[cur][0] != -1: # loop if not find the node that its parent is -1
            cur = int(anslist[cur][0]) # current index change to current node's parent's index
            path.append(nodeidx[cur]) # add current node to path index
        path.reverse() # reverse the path and get the right path
        return path,time,num_visited # return value
    else:
        return [], 0, num_visited # if did't find end, return the right value
    # End your code (Part 6)
```

I import queue at the top. Then I push the node that is adjacent to the current node (get from the top of the priority queue) into the priority queue.

The difference is that I use the time instead of the distance as the admissible heuristic. I consider the time that gets to the current point, and also the time from the adjacent node to reach the end node. I assume the speed limit to be the same as the road between the current node and the adjacent node. Then I use a tuple with four elements, the remaining time to reach the end node plus the total time to reach the adjacent node, the adjacent node, the parent node and the total distance that will be passed.

Other parts are same as BFS, DFS and UCS, use anslist to store the parent node, distance that has been through to this node, and visited or not, then go backward to find the path's node, store in the path list, and reverse to get the right path.

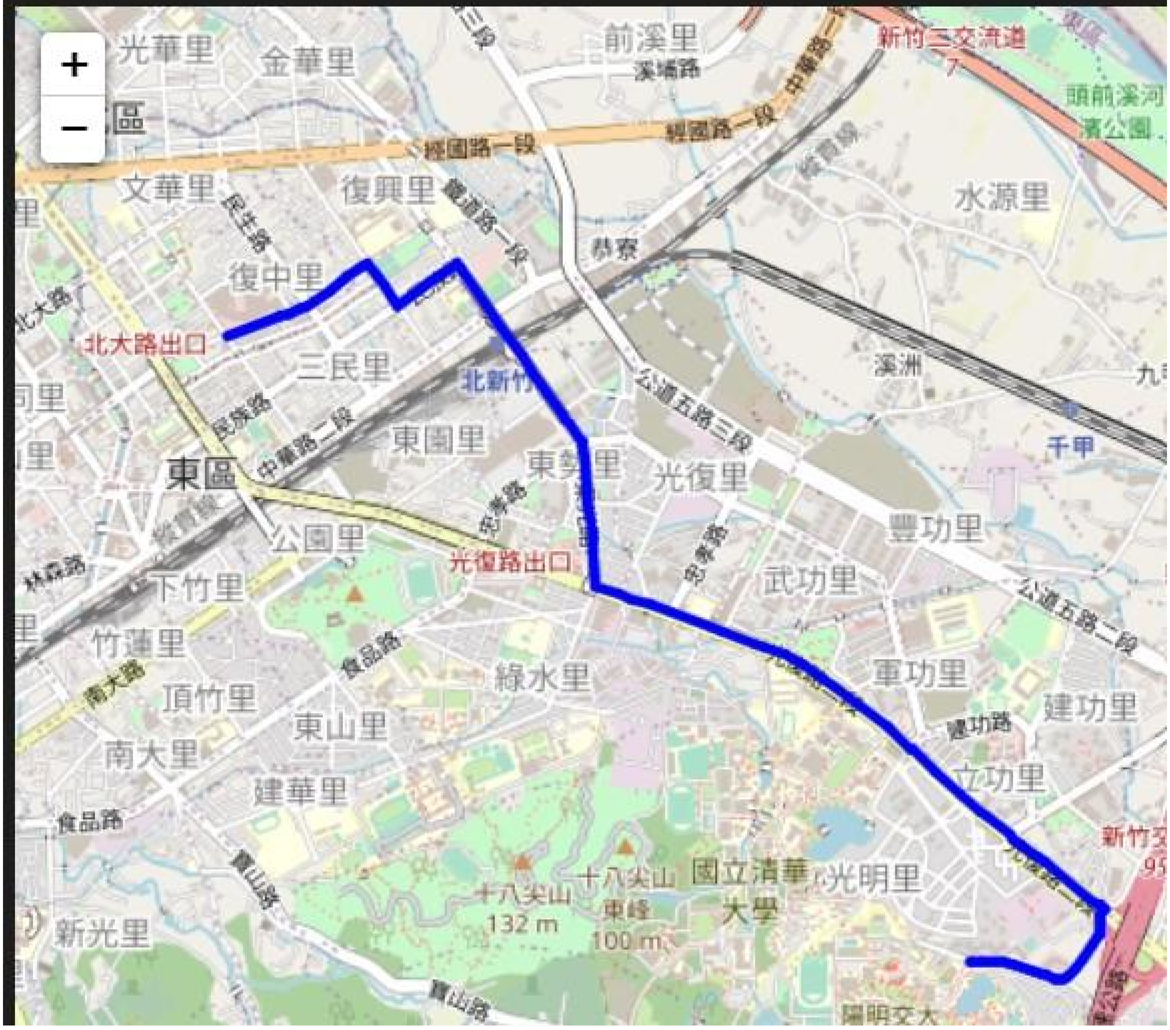
Part II. Results & Analysis (12%):

- **Please screenshot the results.**

  Test1 : from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

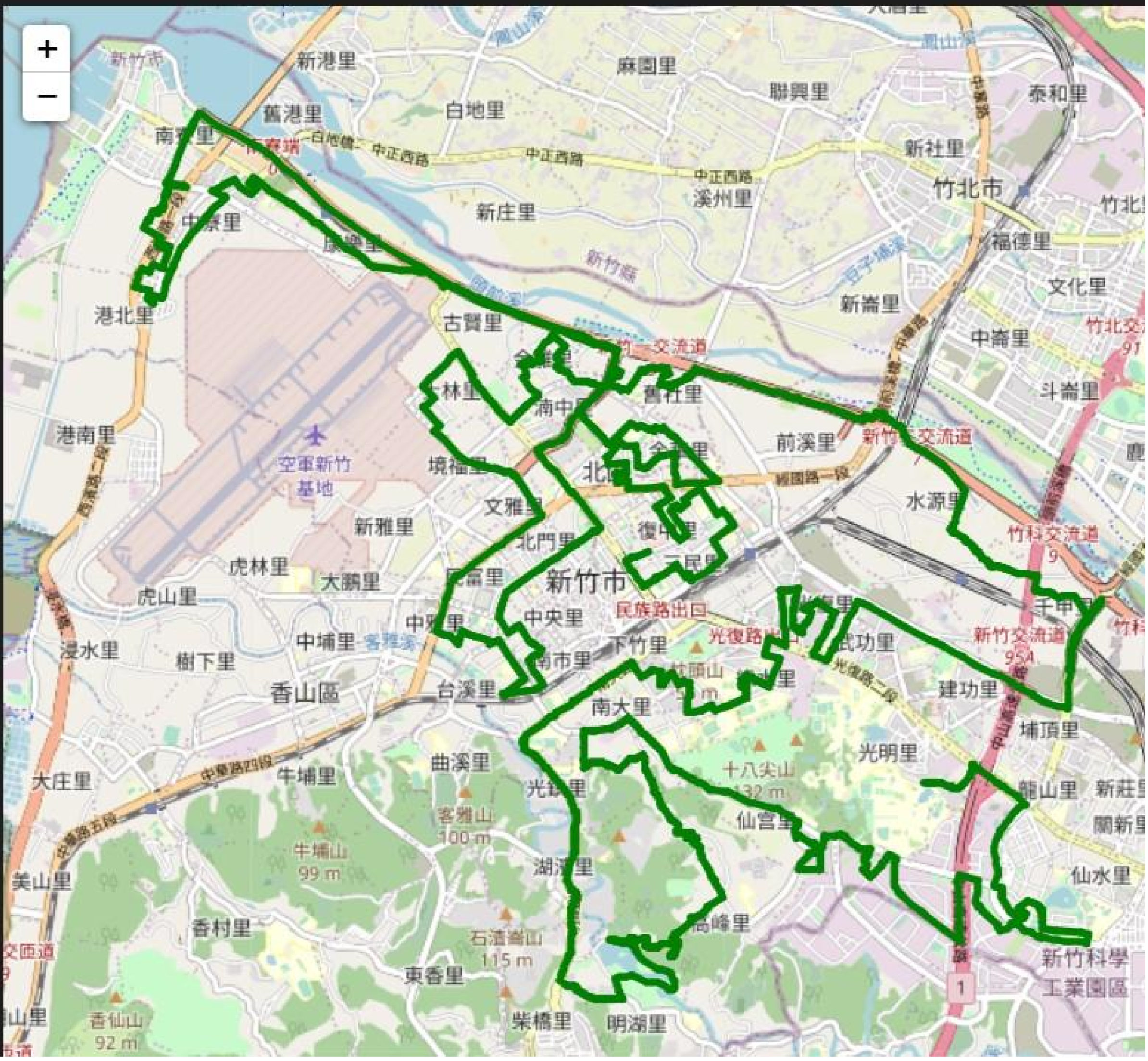
BFS :

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 4266
```



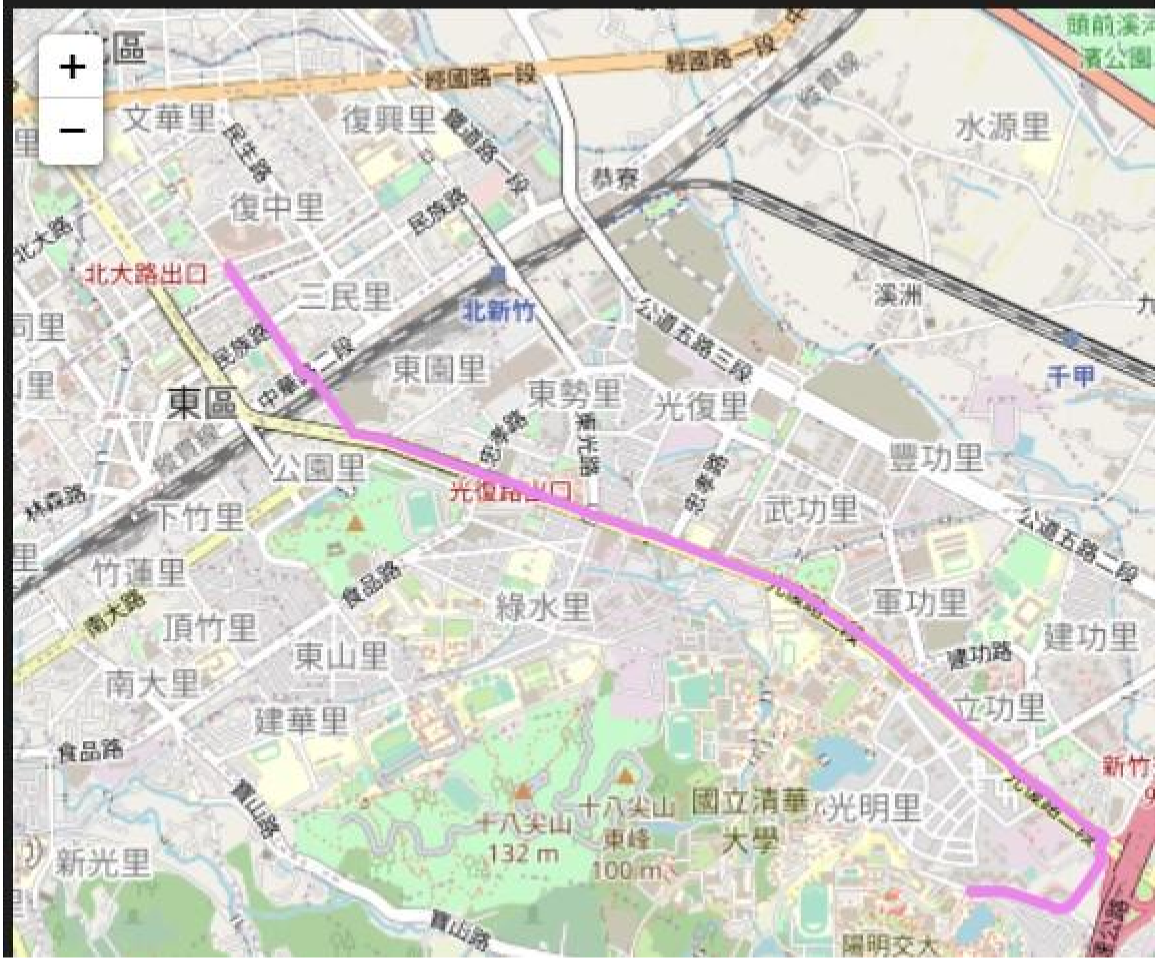
DFS (stack) :

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.31499999983 m
The number of visited nodes in DFS: 5227
```



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: 4998
```



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: 261
```

A* with different heuristic:

```
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: 226
```

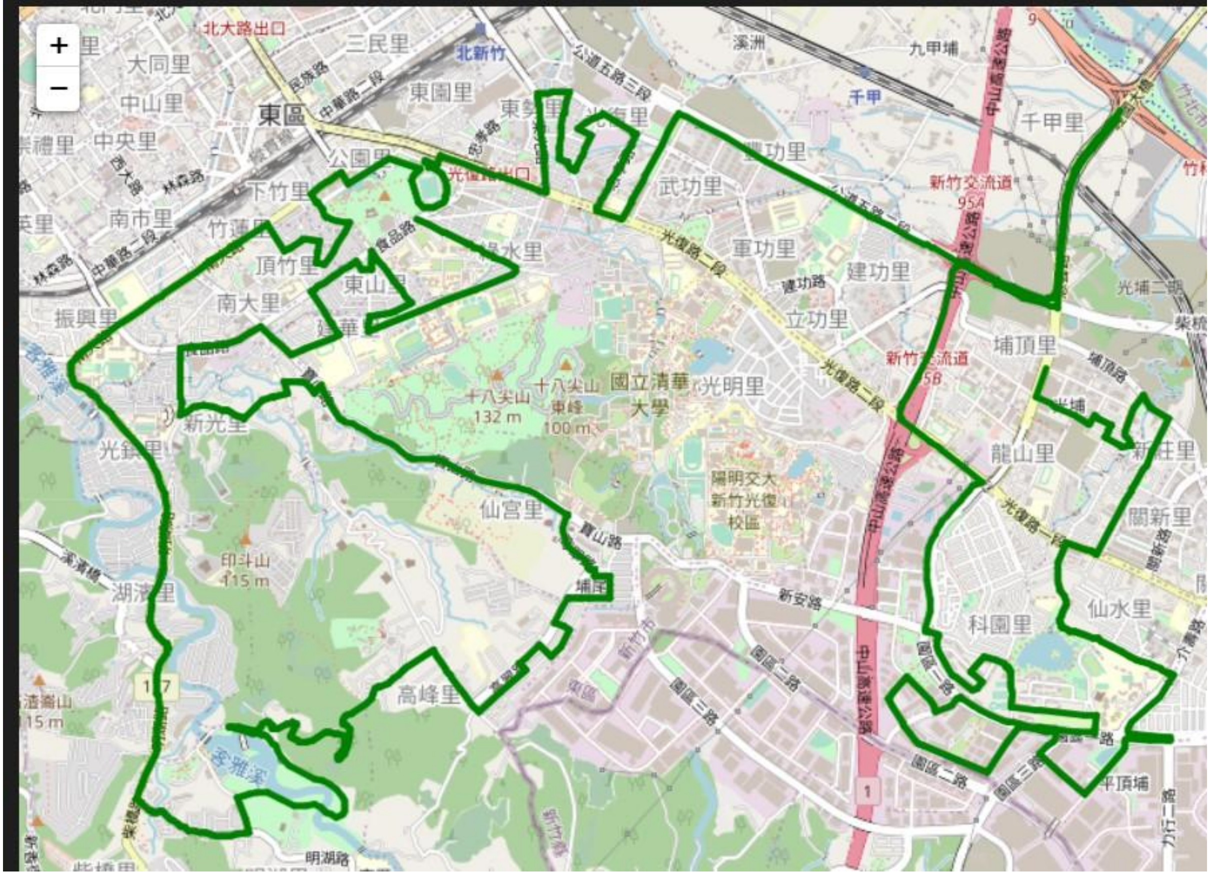
Test 2：from Hsinchu Zoo (ID: 426882161)
to COSTCO Hsinchu Store (ID: 1737223506)

BFS :

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 4603
```

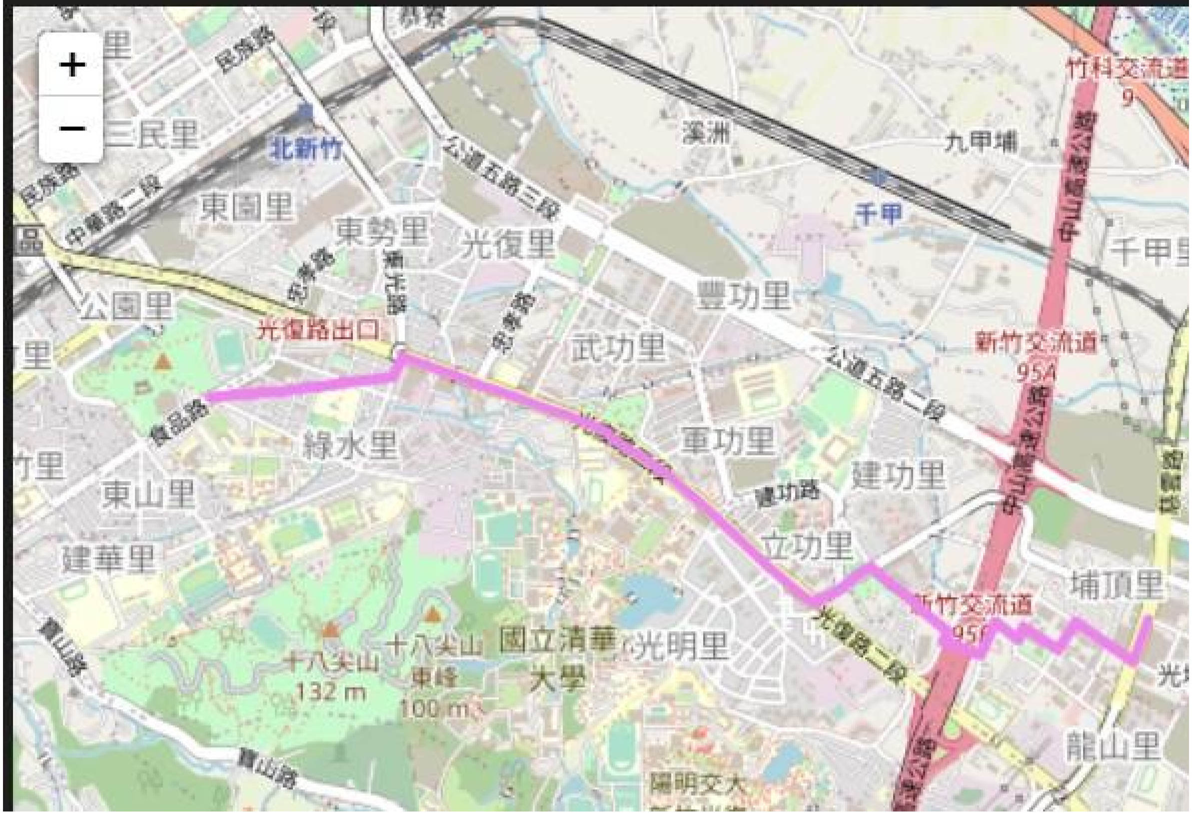
DFS (stack) :

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.30799999996 m
The number of visited nodes in DFS: 9599
```



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: 6783
```



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
```

A* with different heuristic:

```
The number of nodes in the path found by A* search: 67
Total second of path found by A* search: 321.1662525152129 s
The number of visited nodes in A* search: 1233
```

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

BFS :

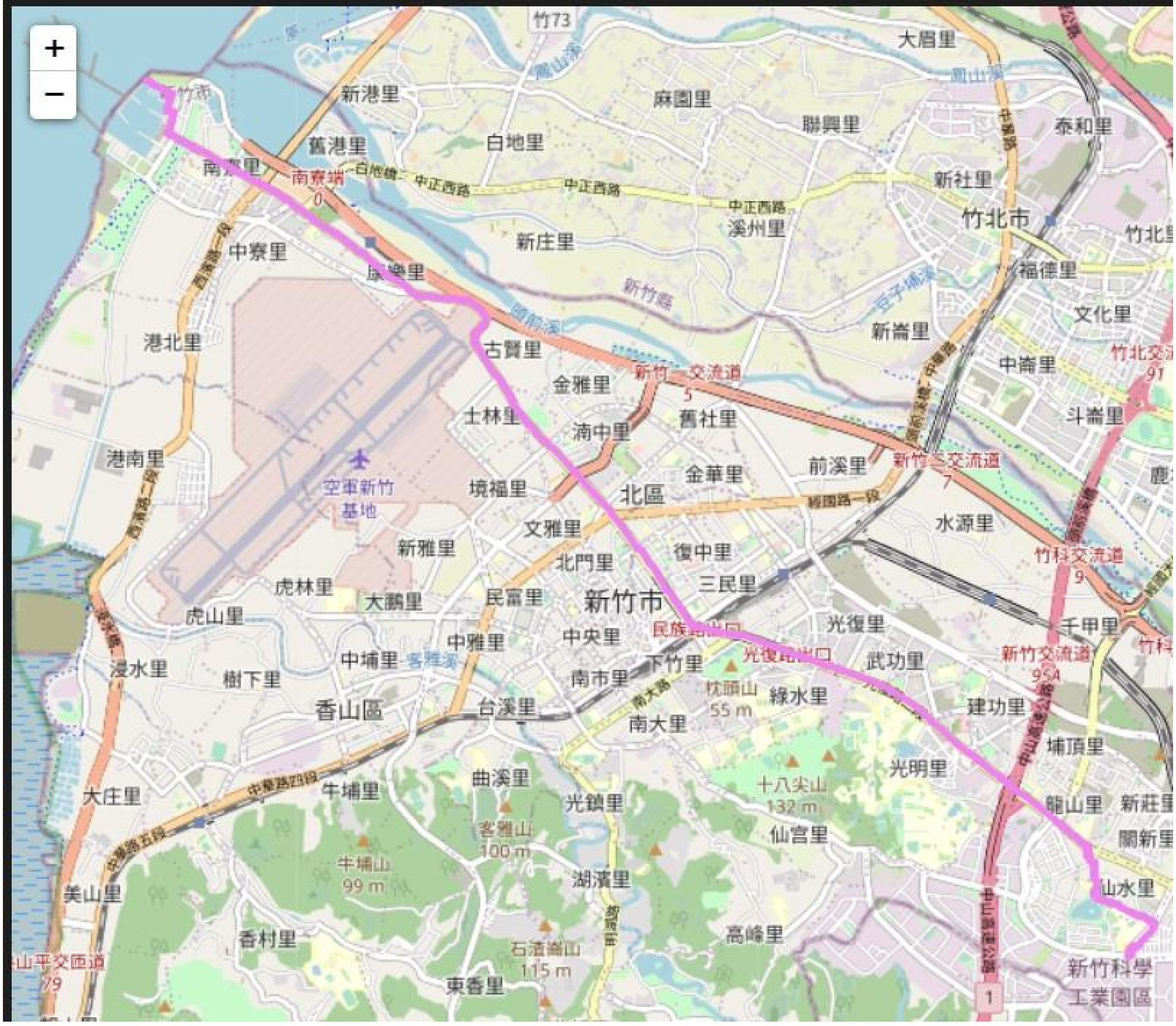
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 11226
```

DFS (stack) :

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.99299999996 m
The number of visited nodes in DFS: 2488
```

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: 11906
```

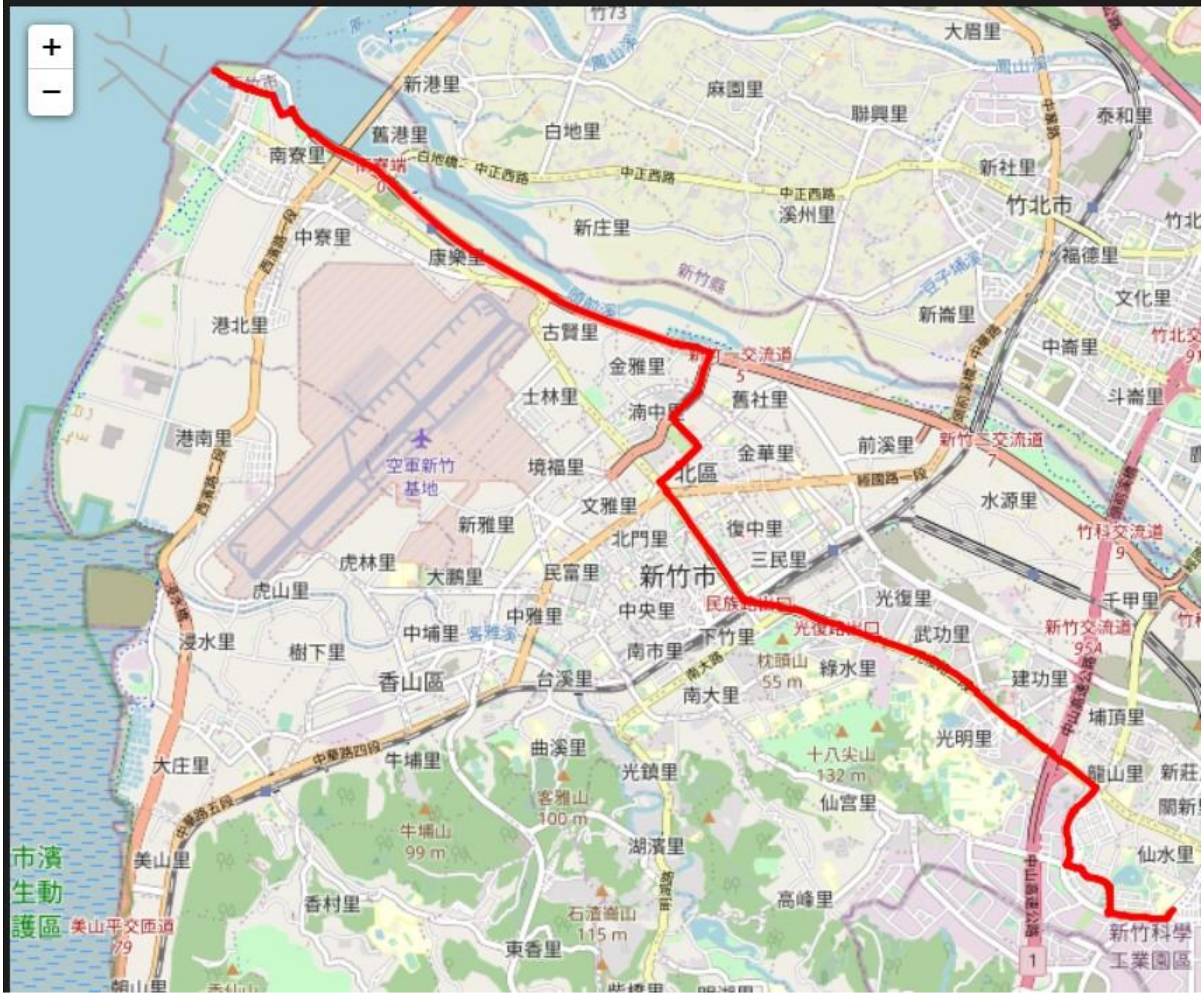


A* :

```
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: 7067
```

A* with different heuristic:

```
The number of nodes in the path found by A* search: 236
Total second of path found by A* search: 924.4499983740104 s
The number of visited nodes in A* search: 6640
```



Compare table between algorithm:

1. from National Yang Ming Chiao Tung University to Big City Shopping Mall
2. from Hsinchu Zoo to COSTCO Hsinchu Store
3. from National Experimental High School At Hsinchu Science Park to Nanliao Fishing Port

	1. node in path	1. total distance	1. visited node	2. node in path	2. total distance	2. visited node	3. node in path	3. total distance	3. visited node
BFS	88	4978.882	4266	60	4215.521	4603	183	15442.395	11226
DFS	1718	75504.314	5227	930	38752.307	9599	900	39219.992	2488
UCS	89	4367.881	4998	63	4104.84	6783	288	14212.412	11906
A*	89	4367.881	261	63	4104.84	1171	288	14212.412	7067

Discussion:
1. BFS, UCS, A* are useful when dealing with routing questions. Since they can find a shorter path
2. BFS needs to visit lots of nodes and cannot get the shortest path, but due to its characteristic, the nodes in path are less than other algorithms.
3. UCS can get the shortest path but needs to visit lots of nodes.
4. A* is the best way to figure out the path.
5. DFS is unstable to get the path to the end node, it may have the possibility to visit few nodes to get to the end, but also may visit lots of nodes. And its total distance may be so high because its algorithm goes deeper until the road is to the end.
6. The number of the visited nodes is smaller than the given test result. I think that I break out the loop as soon as I reach the end node.

Discussion A* with different heuristic:
1. In my heuristic function, the path will choose the fastest path to reach the end, so that in the third case, the path goes to the highway instead of the surface street.
2. But my design can not find the fastest path if the highway is too far from the current node, it is not possible to reach there.

Part III. Question Answering (12%):

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

   (1) I encounter that the answer of UCS is different from the test result, but BFS and DFS are the same. At last, I found out that UCS needed to use unvisited nodes, but the old way I used marked unvisited nodes visited before putting it into priority queue. Why BFS and DFS are the same is because their algorithms have order (FIFO, FILO), but the priority queue in UCS needs to pick the smallest one, so you cannot mark visited first, but need to mark it when getting out of the priority queue.

   (2) I encountered the problem that the .csv files read strings, but my operation needed floats. I use type() to find out the type and use float() or str() to change its type.

   (3) I used a numpy array before, and I figured out that a numpy array is different from list ([ ]), and also, numpy array is not in the python standard library.

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.

   A : red light, the red light will slow down the speed to reach the end because we need to stop and wait, and when we need to accelerate after.

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

   mapping : I think that we need to get all interactions and all of the distance between the interactions, then we can create a map by using an adjacency list. To visualize the map on the device, we can scale down the road distance, but don't need to be so accurate, since it is for visualization.

   localization : We need the help of GPS to get the current location, by the latitude and longitude. And also, we need the latitude and longitude of the interaction points on the map, so that we can get a close point on the map.

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 ETA based on their mechanism. Please define a **dynamic heuristic equation** for ETA and explain the rationale of your design. Hint: You can consider meal prep time, delivery priority, multiple orders, etc.

   (1) assume that prep time is 5 - 15 for each order

   h(t) = prep time + traffic time from store to customer (if single order)

   h(t) = prep time * (1 + 0.5N) +

   traffic time from store to customer + (1 + 0.4(P - 1))

   (if multiple order and order number is N, the priority is P)

   traffic time = distance / speed limit - 15(if busy time) (car)

   traffic time = distance / speed limit - 5 - 5(if busy time) (motor)

   I assume that the preparation time will different between orders, so I give a range. Also, orders will overlap, so I use the average, 0.5, to become the parameter. And the reason to use times N is that I sum up all the preparation time of the order.

   For the traffic time, I consider the isosceles right triangle, using the traffic time from the store to current customer times (1 + 1)/2^(½), about 1.4, and the first order's customer don't need to time the parameter, so I subtract 1 from P.

   And the traffic time needs multiple considerations. If it is a car, then it can drive at the same speed as the speed limit, motors can't, so I subtract 5 from it. During busy times, cars will be stuck in traffic jams, so subtract 15 from the speed limit, but motors will not be that sensitive to traffic jams, so subtract less than cars.

I think 1 or 2 orders commonly happen, so I consider 1 or 2 cases at most, for order more than 3, I just give a rough time.