HW2 Route Finding

0616098 黃秉茂

prerequisite

Import modules and read csv

```python
# import module
import csv
from collections import deque
import heapq

# read csv
with open('edges.csv', newline='') as f:
    reader = csv.reader(f)
    edges = list(reader)

with open('heuristic.csv', newline='') as f:
    reader = csv.reader(f)
    heuristic = list(reader)
```

csv for read csv

deque for queue and stack

heapq for priority queue

read csv and store to lists of list.

Part 1

```python
# Part 1
# version 1.0
def bfs(start, end):
    """
    Parameters:
        start: Integer. The starting node ID.
        end: Integer. The end node ID.
    Returns:
        path: List of integer. The path you found, stored as a list of node IDs. The first is starting node ID.
              The last is end node ID.
        dist: : Float. The distance of the path you found. (Unit:meter)
        num_visited: Integer. The number of nodes were visited when you search.
    """
    queue = deque()
    queue.append(start)
    id_path = {start: [start]}
    id_distance = {start: 0.0}
    num_visited_nodes = 0
    is_reach_end = False
    while True:
        current_node_idx = queue.popleft()
        next_nodes = []
        for row in edges:
            if row[0] == str(current_node_idx):
                next_nodes.append(row)
        next_nodes_idx = [int(row[1]) for row in next_nodes]
        next_nodes_distance = [float(row[2]) for row in next_nodes]
        for row_idx in range(len(next_nodes)):
            next_node_idx = next_nodes_idx[row_idx]
            if id_distance.get(next_node_idx) == None:
                if next_node_idx == end:
                    is_reach_end = True
                num_visited_nodes += 1
                queue.append(next_node_idx)
                id_distance[next_node_idx] = id_distance[current_node_idx] + next_nodes_distance[row_idx]
                id_path[next_node_idx] = list(id_path[current_node_idx])
                id_path[next_node_idx].append(next_node_idx)
                if is_reach_end:
                    break
        if is_reach_end or len(queue) == 0:
            break
    if is_reach_end:
        path = id_path[end]
        dist = id_distance[end]
    else:
        path = -1
        dist = -1
        print(f'The path from {start} to {end} with BFS doesn\'t exist!')
    num_visited = num_visited_nodes
    return path, dist, num_visited
```

Complete bfs with queue (implemented by deque)
queue is only with the starting point first.
id_path is a dictionary which record the path to the node with id.
id_path[a] can get the path from the starting point to node a.
id_distance is a dictionary which records the distance from starting point to the node with index.
id_distance [a] can get the distance from the starting point to node a.
is_reach_end marks the end point is reach or not and the time to break from loop.
num_visited_nodes is a value which record the number of the visited value. It begins with zero and plus one when the new node is visited.
The current node is popped from queue, which is the node we are visiting, and then find the following nodes by the current node.
Getting following nodes by search in lists of list with starting point.
Getting following nodes index and distance from lists of following nodes.
After adding the distance from the starting point to the current node and the distance between the current nodes and the following nodes, record it and push the value to id_distance.
Append following node to the path from the starting point to the current node, and record it and push the value to id_path.
And then push following nodes into the queue.
id_path and id_distance are recorded a kind of recursive with the current node which attach to the following nodes.
Jump out of the loop if the end point is reached or the queue is empty.

Part 2

```
Part 2
version 1.0
f dfs(start, end):
  """
  Parameters:
      start: Integer. The starting node ID.
      end: Integer. The end node ID.
  Returns:
      path: List of integer. The path you found, stored as a list of node IDs. The first is starting node ID.
            The last is end node ID.
      dist: : Float. The distance of the path you found. (Unit:meter)
      num_visited: Integer. The number of nodes were visited when you search.
  """
  end_points = [row[1] for row in edges]
  n_idx = len(set(end_points))
  distance = [[0] * n_idx for _ in range(n_idx)]
  visited_idx = []
  visited_idx.append(start)
  precede = []
  precede.append(0)
  stack = deque()
  stack.append(start)
  is_reach_end = False
  num_visited_nodes = 0
  is_should_pop = True
  while True:
      current_node_idx = stack[-1]
      next_nodes = []
      for row in edges:
          if row[0] == str(current_node_idx):
              next_nodes.append(row)
      next_nodes_idx = [int(row[1]) for row in next_nodes]
      next_nodes_distance = [float(row[2]) for row in next_nodes]
      is_should_pop = True
      for row_idx in range(len(next_nodes) - 1, -1, -1):
          next_node_idx = next_nodes_idx[row_idx]
          if next_node_idx not in visited_idx:
              if next_node_idx == end:
                  is_reach_end = True
              visited_idx.append(next_node_idx)
              precede.append(visited_idx.index(current_node_idx))
              num_visited_nodes += 1
              distance[visited_idx.index(current_node_idx)][visited_idx.index(next_node_idx)] = next_nodes_distance[row_idx]
              distance[visited_idx.index(next_node_idx)][visited_idx.index(current_node_idx)] = next_nodes_distance[row_idx]
              stack.append(next_node_idx)
              is_should_pop = False
              break
      if is_should_pop:
          current_node_idx = stack.pop()
      if is_reach_end or len(stack) == 0:
          break

  if is_reach_end:
      path = [len(visited_idx) - 1]
      dist = 0
      while True:
          now_idx = path[-1]
          precede_idx = precede[now_idx]
          dist += distance[precede_idx][now_idx]
          path.append(precede_idx)
          if precede_idx == 0:
              break
      path.reverse()
      for i in range(len(path)):
          path[i] = visited_idx[path[i]]
  else:
      path = -1
      dist = -1
      print(f'The path from {start} to {end} with DFS doesn\'t exist!')
  num_visited = num_visited_nodes
  return path, dist, num_visited
```

Complete dfs with stack (implemented by deque)
stack is only with the starting point first.
visit_idx is a list which recordthe nodes with index we have visited.
distance is a numpy matrix which record the distance between two nodes.
distance[i, j] can get the distance between the node I and the node j.
precede is a list which records the preceding nodes index in visit_idx.
precede[i] can get the index of preceding node i in visit_idx.
is_reach_end marks the end point is reach or not and the time to break from loop.
is_should_pop marks the stack should pop or not. It should pop something when the all following nodes are visited.
num_visited_nodes is a value which record the number of the visited value. It begins with zero and plus one when the new node is visited.
The current node is the top of the stack, which is the node we are visiting, and then find the following nodes by the current node.
Getting following nodes by search in lists of list with starting point.
Getting following nodes index and distance from lists of following nodes.
The former node it is in the lists, the latter we push into the stack.
After visiting a node, add the node to the list visit_idx and add its preceding nodes index in visit_idx to the list precede.
Record the distance between the current nodes and the following nodes to the matrix distance.
Push the following node we deal with now into the stack, and then break the loop.
The other following nodes will process after we find that this following node can't bring us to the end point.
Stack should pop something only when the all following nodes are visited.
Jump out of the loop if the end point is reached or the stack is empty.
With the list precede which records the preceding nodes index in visit_idx, we can trace the path and calculating the distance from the starting point to the end point.
From the end point, we use the list precede to get to the preceding point, and record the path and the distance between the current point and the preceding point.
Recursively record the path and calculating the distance until we get to the starting point by keep tracing the preceding nodes.

Part 3

```python
# Part 3
def ucs(start, end):
    """
    Parameters:
        start: Integer. The starting node ID.
        end: Integer. The end node ID.
    Returns:
        path: List of integer. The path you found, stored as a list of node IDs. The first is starting node ID.
              The last is end node ID.
        dist: : Float. The distance of the path you found. (Unit:meter)
        num_visited: Integer. The number of nodes were visited when you search.
    """
    end_points = [row[1] for row in edges]
    n_idx = len(set(end_points))
    distance = [[0] * n_idx for _ in range(n_idx)]
    visited_idx = []
    visited_idx.append(start)
    precede = []
    precede.append(0)
    is_reach_end = False
    num_visited_nodes = 0
    heap = [(0, start)]
    accumulated_distance = [0]
    while True:
        current_node_idx = heapq.heappop(heap)[-1]
        next_nodes = []
        for row in edges:
            if row[0] == str(current_node_idx):
                next_nodes.append(row)
        next_nodes_idx = [int(row[1]) for row in next_nodes]
        next_nodes_distance = [float(row[2]) for row in next_nodes]
        next_nodes_accumulated_distance = []
        if current_node_idx in visited_idx:
            for row_idx in range(len(next_nodes_idx)):
                next_node_accumulated_distance = next_nodes_distance[row_idx] + accumulated_distance[visited_idx.index(curre
                next_nodes_accumulated_distance.append(next_node_accumulated_distance)
```

```python
        node_tuples = list(zip(next_nodes_accumulated_distance, next_nodes_idx))
        for node_tuple in node_tuples:
            if node_tuple[-1] not in visited_idx:
                heapq.heappush(heap, node_tuple)
        for next_node_tuple in heap:
            next_node_accumulated_distance, next_node_idx = next_node_tuple
            if next_node_idx not in visited_idx:
                visited_idx.append(next_node_idx)
                num_visited_nodes += 1
                precede.append(visited_idx.index(current_node_idx))
                accumulated_distance.append(next_node_accumulated_distance)
                distance[visited_idx.index(current_node_idx)][visited_idx.index(next_node_idx)] = next_node_accumulated_dist
                distance[visited_idx.index(next_node_idx)][visited_idx.index(current_node_idx)] = next_node_accumulated_dist
            if next_node_idx == end:
                is_reach_end = True
        if is_reach_end or len(heap) == 0:
            break
    if is_reach_end:
        path = [len(visited_idx) - 1]
        dist = 0
        while True:
            now_idx = path[-1]
            precede_idx = precede[now_idx]
            dist += distance[precede_idx][now_idx]
            path.append(precede_idx)
            if precede_idx == 0:
                break
        path.reverse()
        for i in range(len(path)):
            path[i] = visited_idx[path[i]]
    else:
        path = -1
        dist = -1
        print(f'The path from {start} to {end} with UCS doesn\'t exist!')
    num_visited = num_visited_nodes
    return path, dist, num_visited
```

Complete ucs with priority queue (inplemented by heap)
heap is only with the starting point first.
visit_idx is a list which recordthe nodes with index we have visited.
distance is a numpy matrix which record the distance between two nodes.
distance[i, j] can get the distance between the node I and the node j.
precede is a list which records the preceding nodes index in visit_idx.
precede[i] can get the index of preceding node i in visit_idx.
accumulated_distance is a list which records the distance from the starting point to the node with index.
is_reach_end marks the end point is reach or not and the time to break from loop.
num_visited_nodes is a value which record the number of the visited value. It begins with zero and plus one when the new node is visited.
The current node is popped from heap(priority queue), which is the node we are visiting and the distance from sthe tarting point to the node with index is the least in the heap. And then find the following nodes by the current node.
Getting following nodes by search in lists of list with starting point.
Getting following nodes index and distance from lists of following nodes.
heap(priority queue) is sorted by the accumulated distance (the distance from the starting point to the node with index), so the distance is the addition of the distance from the starting point to the current node and the distance between the current node and the following node.
Push the tuple of the accumulated distance and the index of the following nodes into the heap(priority queue).
After visiting a node, add the node to the list visit_idx and add its preceding nodes index in visit_idx to the list precede.
The distance between the current nodes and the following nodes is get from the tuple in the heap(priority queue). The distance received by the tuple is accumulated, so we have to minus it with the distance accumulated to the preceding nodes.
Record the distance between the current nodes and the following nodes to the matrix distance.
Jump out of the loop if the end point is reached or the stack is empty.
With the list precede which records the preceding nodes index in visit_idx, we can trace the path and calculating the distance from the starting point to the end point.
From the end point, we use the list precede to get to the preceding point, and record the path and the distance between the current point and the preceding point.
Recursively record the path and calculating the distance until we get to the starting point by keep tracing the preceding nodes.

Part 4

```python
# Part 4
def astar(start, end):
    """
    Parameters:
        start: Integer. The starting node ID.
        end: Integer. The end node ID.
    Returns:
        path: List of integer. The path you found, stored as a list of node IDs. The first is starting node ID.
              The last is end node ID.
        dist: : Float. The distance of the path you found. (Unit:meter)
        num_visited: Integer. The number of nodes were visited when you search.
    """
    end_points = [row[1] for row in edges]
    n_idx = len(set(end_points))
    distance = [[0] * n_idx for _ in range(n_idx)]
    visited_idx = []
    visited_idx.append(start)
    precede = []
    precede.append(0)
    is_reach_end = False
    num_visited_nodes = 0
    heap = [(0, start, 99999)]
    accumulated_distance = [0]
    while True:
        current_node_idx = heapq.heappop(heap)[1]
        next_nodes = []
        for row in edges:
            if row[0] == str(current_node_idx):
                next_nodes.append(row)
        next_nodes_idx = [int(row[1]) for row in next_nodes]
        next_nodes_distance = [float(row[2]) for row in next_nodes]
        heuristic_distance = []
        nodes_total_distance = []
        heuristic_distance_list = []
        if current_node_idx in visited_idx:
            for row_idx in range(len(next_nodes_idx)):
                next_nodes_accumulated_distance = next_nodes_distance[row_idx] + accumulated_distance[visited_idx.index(curr
                heuristic_distance = 0
                for row in heuristic:
                    if row[0] == str(next_nodes_idx[row_idx]):
                        end_idx = -1
                        for end_id in heuristic[0]:
                            if end_id == str(end):
                                break
                            end_idx += 1
                        heuristic_distance = float(row[1 + end_idx])
                heuristic_distance_list.append(heuristic_distance)
                nodes_total_distance.append(next_nodes_accumulated_distance + heuristic_distance)
        node_tuples = list(zip(nodes_total_distance, next_nodes_idx, heuristic_distance_list))
        for node_tuple in node_tuples:
            if node_tuple[1] not in visited_idx:
                heapq.heappush(heap, node_tuple)
        for next_node_tuple in heap:
            node_total_distance, next_node_idx, heuristic_distance = next_node_tuple
            next_node_accumulated_distance = node_total_distance - heuristic_distance
            if next_node_idx not in visited_idx:
                visited_idx.append(next_node_idx)
                num_visited_nodes += 1
                precede.append(visited_idx.index(current_node_idx))
                accumulated_distance.append(next_node_accumulated_distance)
                distance[visited_idx.index(current_node_idx)][visited_idx.index(next_node_idx)] = next_node_accumulated_dist
                distance[visited_idx.index(next_node_idx)][visited_idx.index(current_node_idx)] = next_node_accumulated_dist
            if next_node_idx == end:
                is_reach_end = True
        if is_reach_end or len(heap) == 0:
            break
    if is_reach_end:
        path = [len(visited_idx) - 1]
        dist = 0
        while True:
            now_idx = path[-1]
            precede_idx = precede[now_idx]
            dist += distance[precede_idx][now_idx]
            path.append(precede_idx)
            if precede_idx == 0:
                break
        path.reverse()
        for i in range(len(path)):
            path[i] = visited_idx[path[i]]
    else:
        path = -1
        dist = -1
        print(f'The path from {start} to {end} with ASTAR doesn\'t exist!')
    num_visited = num_visited_nodes
    return path, dist, num_visited
```

Complete A* with priority queue (inplemented by heap)
It's implement is similar to UCS + heuristic distance.
heap(priority queue) is only with the starting point first.
visit_idx is a list which recordthe nodes with index we have visited.
distance is a numpy matrix which record the distance between two nodes.
distance[i, j] can get the distance between the node I and the node j.
precede is a list which records the preceding nodes index in visit_idx.
precede[i] can get the index of preceding node i in visit_idx.
accumulated_distance is a list which records the distance from the starting point to the node with index.
is_reach_end marks the end point is reach or not and the time to break from loop.
num_visited_nodes is a value which record the number of the visited value. It begins with zero and plus one when the new node is visited.
The current node is popped from heap(priority queue), which is the node we are visiting and the distance from sthe tarting point to the node with index is the least in the heap. And then find the following nodes by the current node.
heap(priority queue) is sorted by the total distance (the distance from the starting point to the node with index and the heuristic distance), so the distance is the addition of the distance from the starting point to the current node, the distance between the current node and the following node and the heuristic distance.
Push the tuple of the total distance, the index of the following nodes, and the heuristic distance into the heap(priority queue).
Getting following nodes by search in lists of list with starting point.
Getting following nodes index and distance from lists of following nodes.
After visiting a node, add the node to the list visit_idx and add its preceding nodes index in visit_idx to the list precede.
The distance between the current nodes and the following nodes is get from the tuple in the heap(priority queue). The distance received by the tuple is total distance, so we have to minus it with the distance accumulated to the preceding nodes and the heuristic distance.
Record the distance between current nodes and following nodes to matrix distance.
Jump out of the loop if the end point is reached or the stack is empty.
With the list precede which records the preceding nodes index in visit_idx, we can trace the path and calculating the distance from the starting point to the end point.
From the end point, we use the list precede to get to the preceding point, and record the path and the distance between the current point and the preceding point.
Recursively record the path and calculating the distance until we get to the starting point by keep tracing the preceding nodes.

Part 5

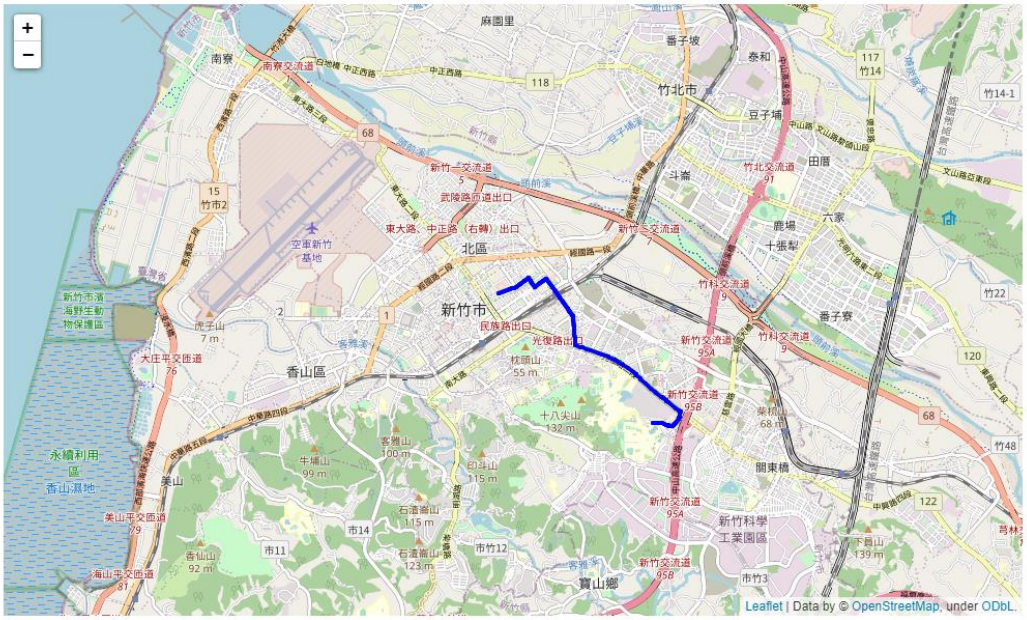
Screenshot

starting node: 2270143902 end node: 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: 4273
```

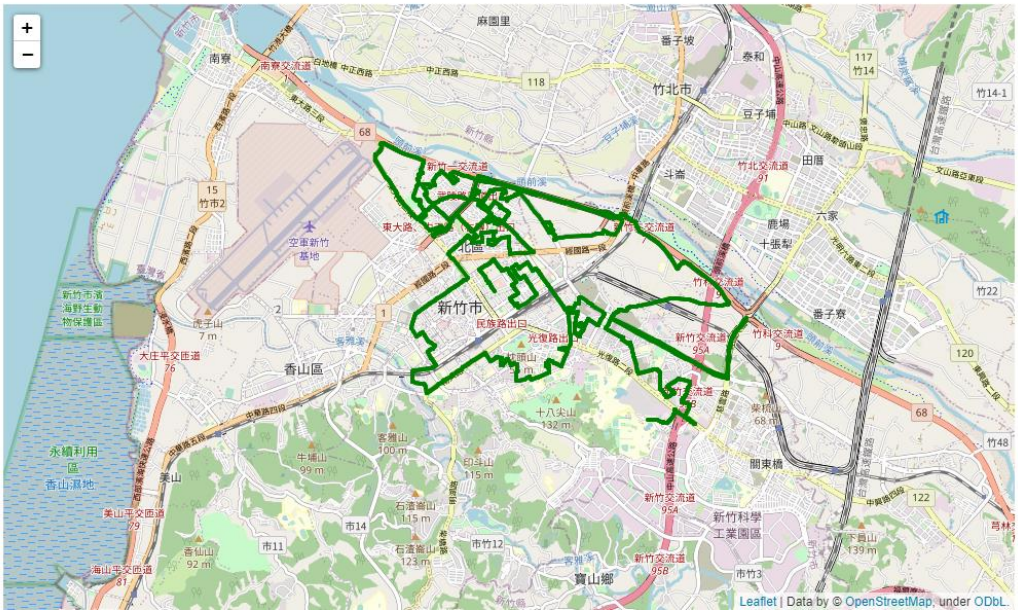
Out[9]:



DFS

```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987 m
The number of visited nodes in DFS: 4210
```

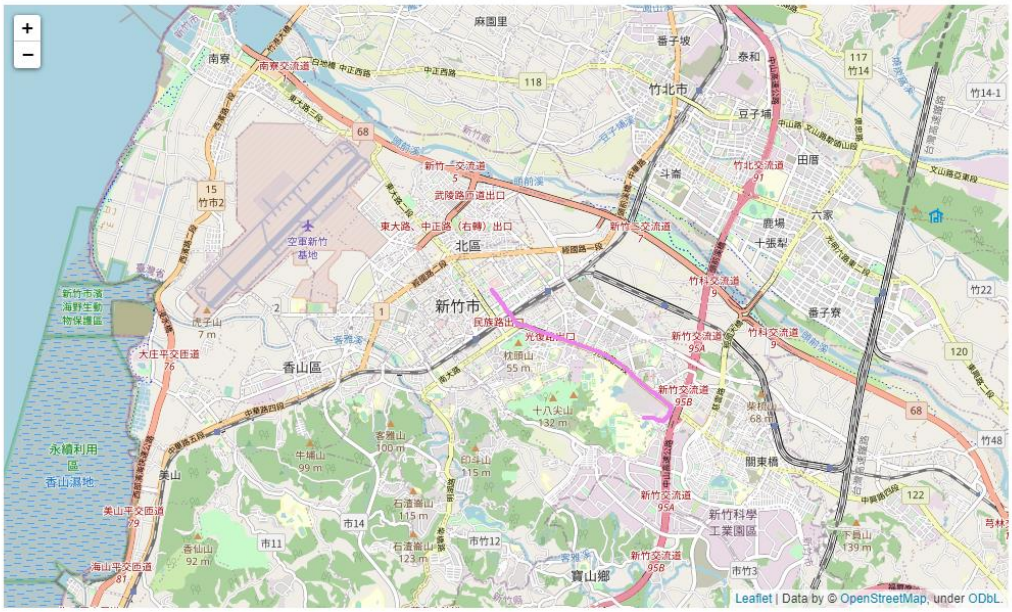
Out[10]:



UCS

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.880999999999 m
The number of visited nodes in UCS: 5034
```

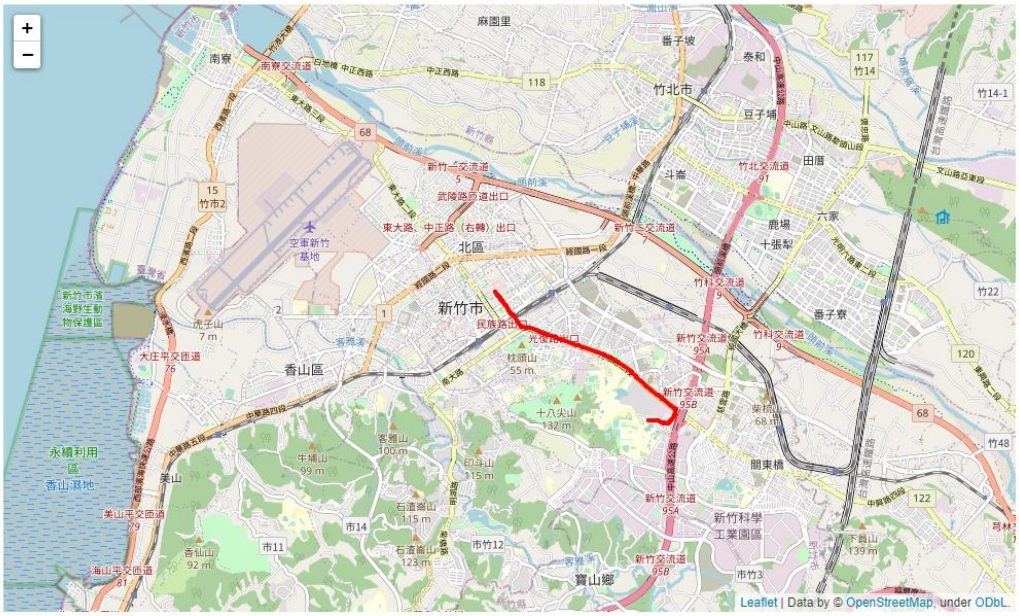
Out[11]:



A*

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881000000001 m
The number of visited nodes in A* search: 305
```

Out[12]:

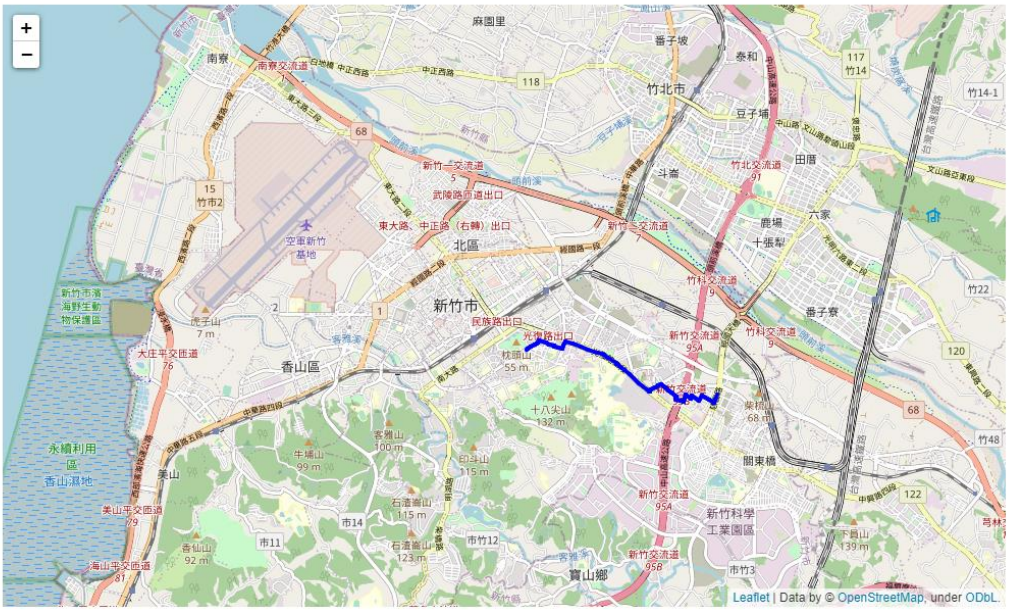


starting node: 426882161 end node: 1737223506

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: 4606
```

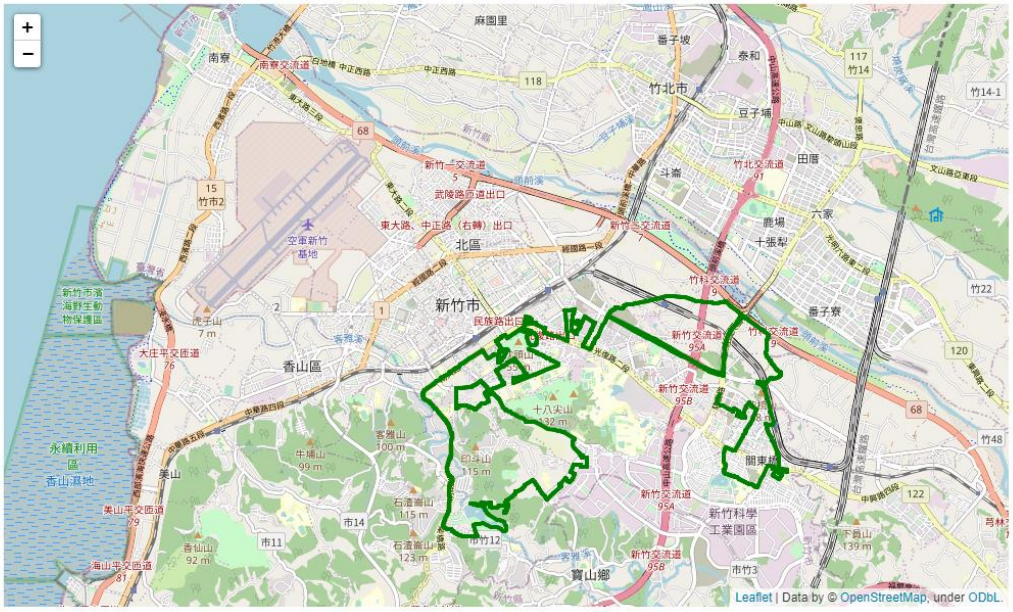
Out[16]:



DFS

```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.657999999916 m
The number of visited nodes in DFS: 8030
```

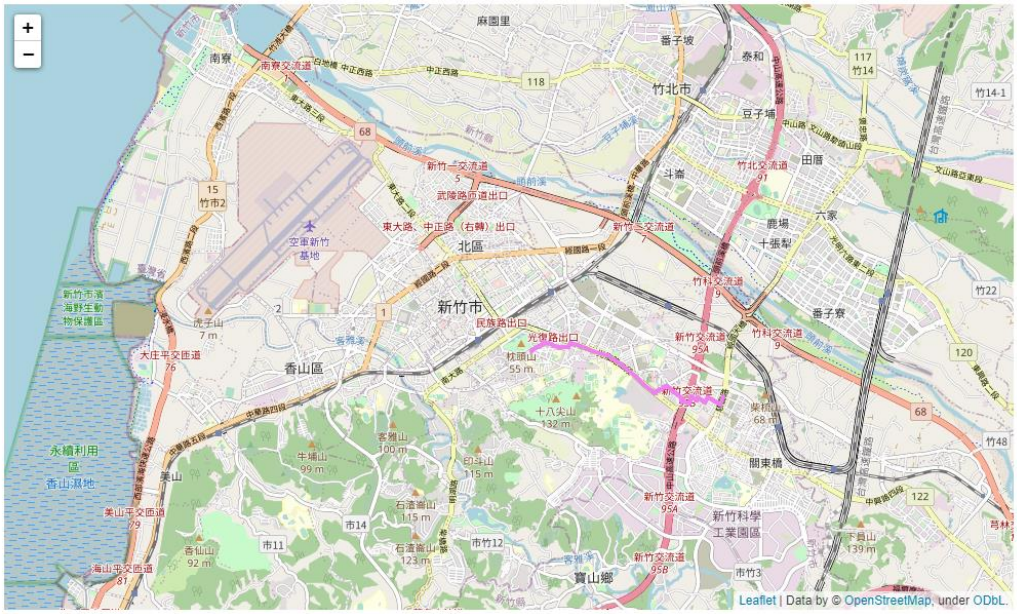
Out[17]:



UCS

```
The number of nodes in the path found by UCS: 60
Total distance of path found by UCS: 4162.608999999999 m
The number of visited nodes in UCS: 6946
```

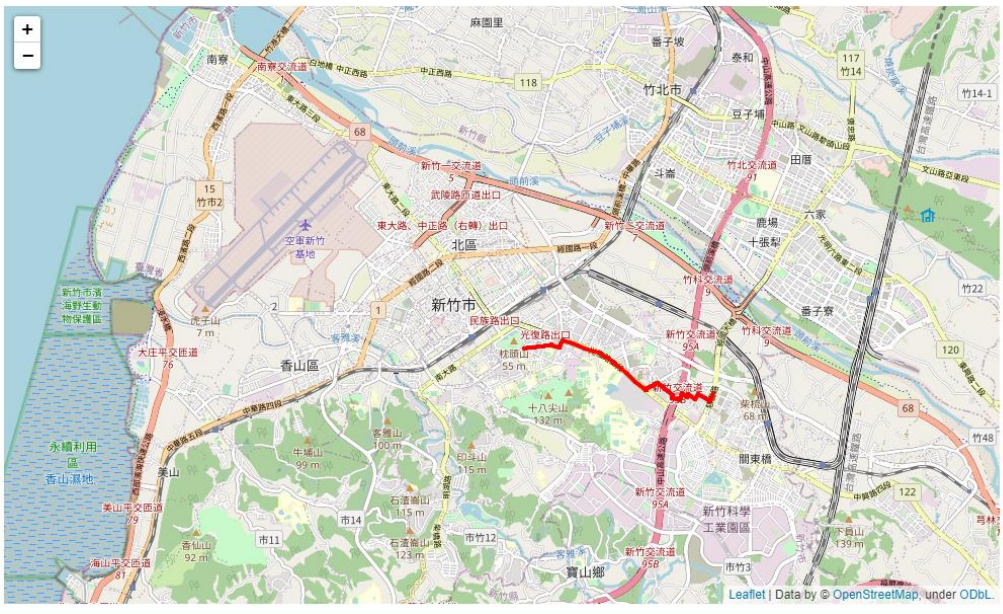
Out[18]:



A*

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4154.752 m
The number of visited nodes in A* search: 1197
```

Out[19]:

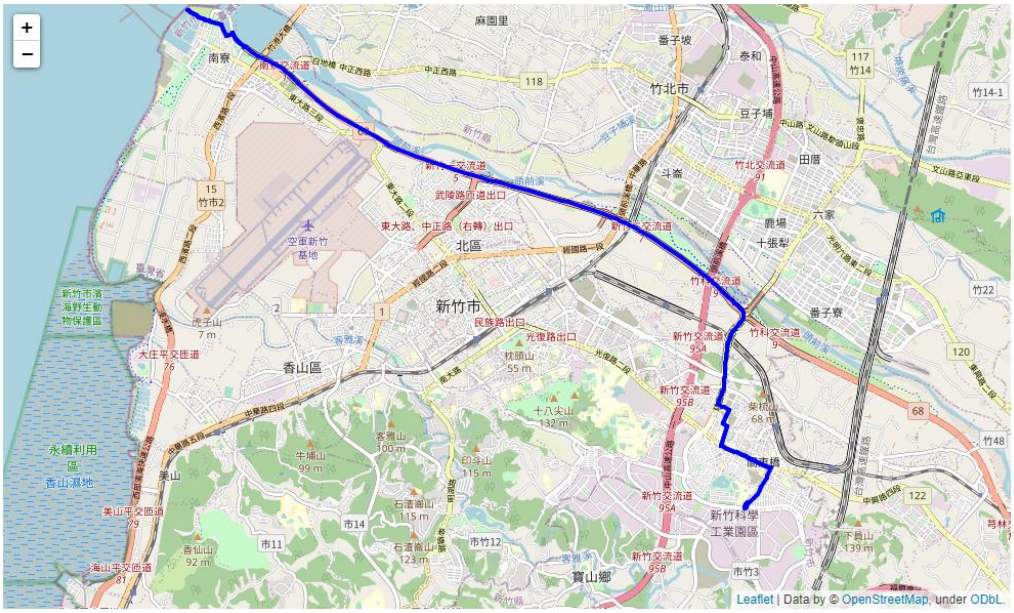


starting node: 1718165260 end node: 8513026827

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: 11241
```

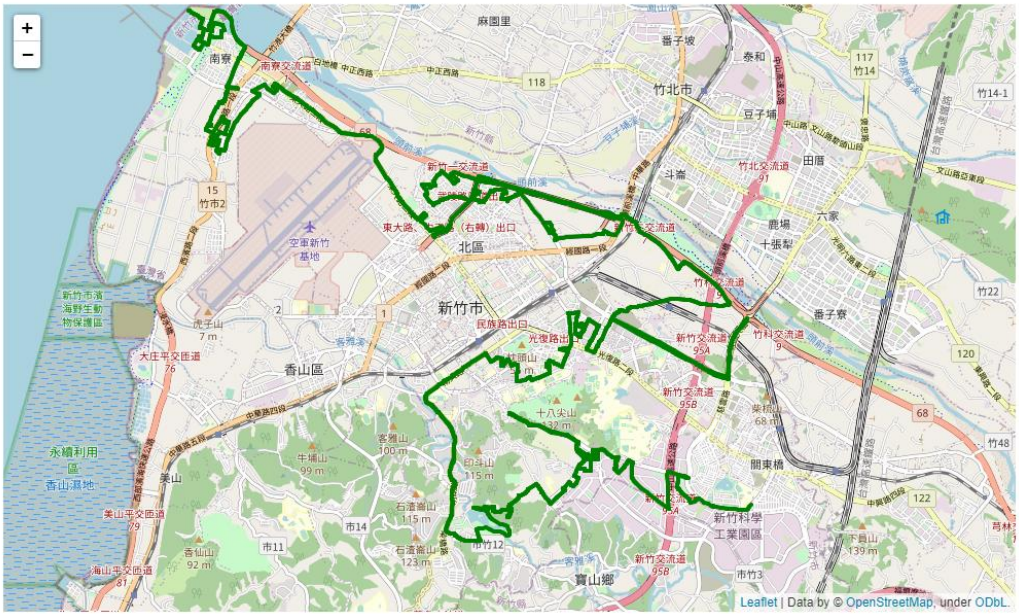
Out[23]:



DFS

```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999999 m
The number of visited nodes in DFS: 3291
```

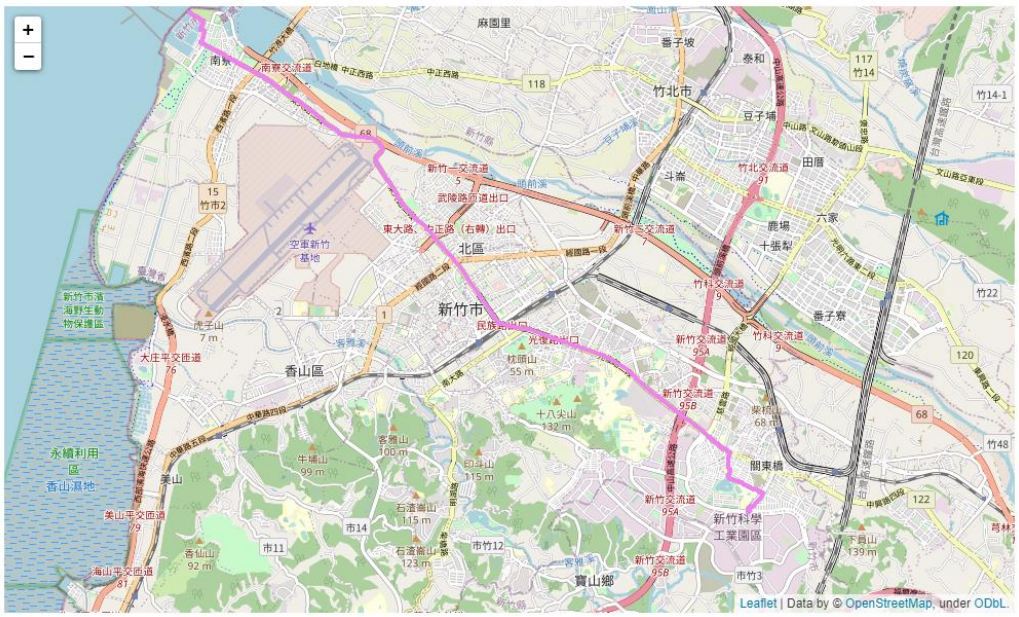
Out[24]:



UCS

```
The number of nodes in the path found by UCS: 275
Total distance of path found by UCS: 14241.51 m
The number of visited nodes in UCS: 11929
```

Out[25]:



A*

```
The number of nodes in the path found by A* search: 277
Total distance of path found by A* search: 14218.25500000001 m
The number of visited nodes in A* search: 7089
```

Out[12]:

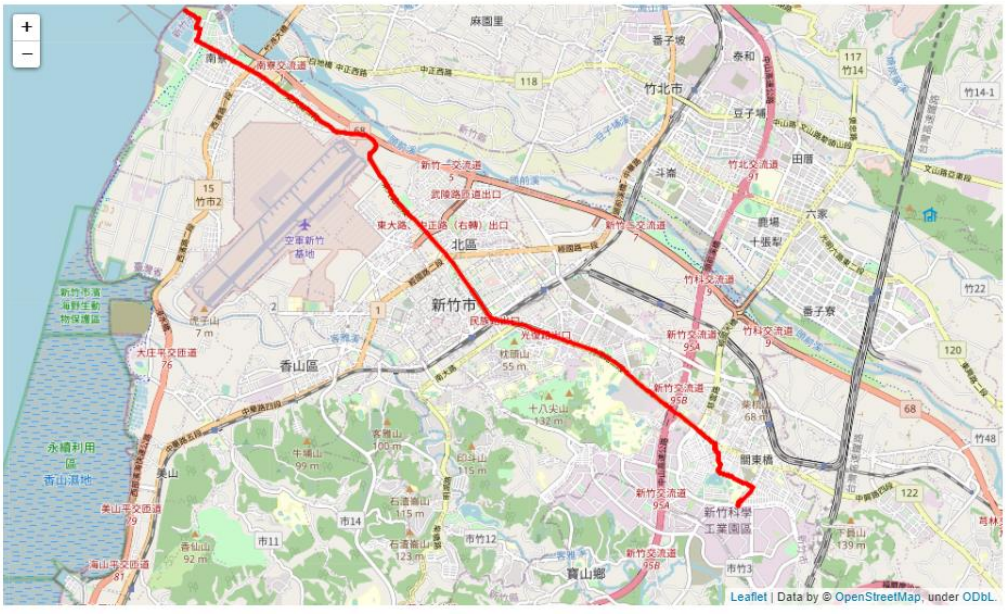


table comparison

starting node: 2270143902 end node: 1079387396

	# nodes in path	distance (m)	# visited
BFS	88	4978.882	4273
DFS	1232	57208.987	4210
UCS	89	4367.881	5034
A*	89	4367.881	305

starting node: 426882161 end node: 1737223506

	# nodes in path	distance (m)	# visited
BFS	60	4215.521	4606
DFS	998	41094.658	8030
UCS	60	4162.609	6946
A*	63	4154.752	1197

starting node: 1718165260 end node: 8513026827

	# nodes in path	distance (m)	# visited
BFS	183	15442.395	11241
DFS	1521	64821.604	3291
UCS	275	14241.51	11929
A*	277	14218.255	7089

The most # nodes in path is always in DFS.
The least distance is always in A*.
Compare with UCS, A* is often more efficient and better performance. In other words, A* beat UCS.
In some conditions, DFS can be the most efficient. However, DFS always lead to the worst performance not only # nodes in path but also distance
The performance in BFS, UCS, and A* doesn’t differ a lot.

Part 6

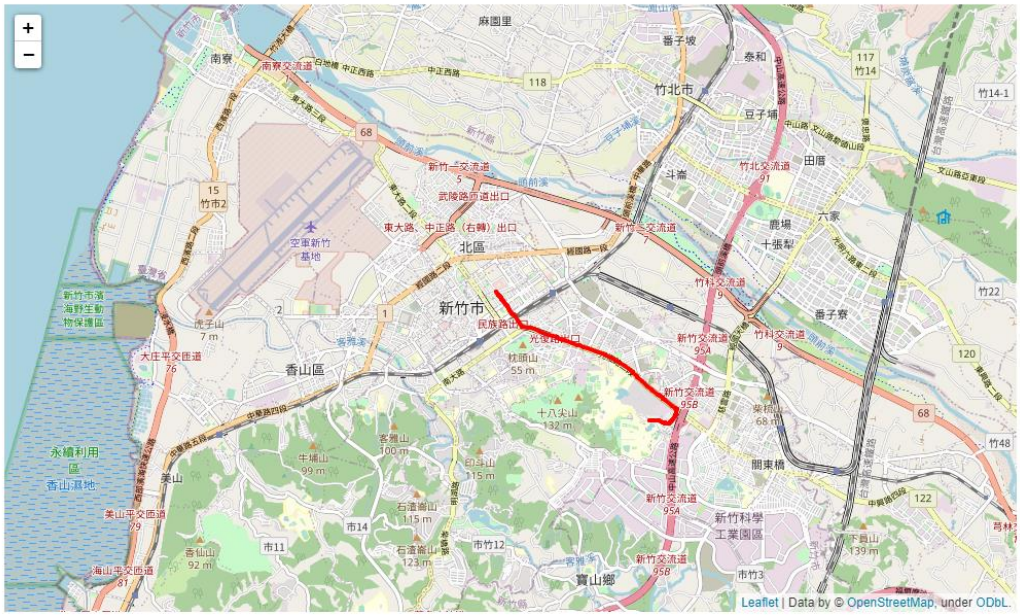
```python
# Part 6 (Bonus)
# admissible heuristic function is heuristic_distance / speed_limit
def astar_time(start, end):
    """
    Parameters:
        start: Integer. The starting node ID.
        end: Integer. The end node ID.
    Returns:
        path: List of integer. The path you found, stored as a list of node IDs. The first is starting node ID.
              The last is end node ID.
        dist: : Float. The distance of the path you found. (Unit:meter)
        num_visited: Integer. The number of nodes were visited when you search.
    """
    end_points = [row[1] for row in edges]
    n_idx = len(set(end_points))
    time = [[0] * n_idx for _ in range(n_idx)]
    visited_idx = []
    visited_idx.append(start)
    precede = []
    precede.append(0)
    is_reach_end = False
    num_visited_nodes = 0
    # dist, index, heuristic, speed limit
    heap = [(0, start, 999999, 1)]
    accumulated_time = [0]
    while True:
        current_node_idx = heapq.heappop(heap)[1]
        next_nodes = []
        for row in edges:
            if row[0] == str(current_node_idx):
                next_nodes.append(row)
        next_nodes_idx = [int(row[1]) for row in next_nodes]
        next_nodes_distance = [float(row[2]) for row in next_nodes]
        next_nodes_speed_limit = [float(row[-1]) for row in next_nodes]
        next_nodes_time = []
        nodes_total_accumulated_time = []
        heuristic_distance_list = []
        if current_node_idx in visited_idx:
            for row_idx in range(len(next_nodes_idx)):
                next_node_speed_limit = next_nodes_speed_limit[row_idx] * 1000 / 3600
                next_node_time = next_nodes_distance[row_idx] / next_node_speed_limit
                next_nodes_time.append(next_node_time)
                next_nodes_accumulated_time = next_node_time + accumulated_time[visited_idx.index(current_node_idx)]
                heuristic_distance = 0
                for row in heuristic:
                    if row[0] == str(next_nodes_idx[row_idx]):
                        end_idx = -1
                        for end_id in heuristic[0]:
                            if end_id == str(end):
                                break
                            end_idx += 1
                        heuristic_distance = float(row[1 + end_idx])
                heuristic_distance_list.append(heuristic_distance)
                heuristic_time = heuristic_distance / next_node_speed_limit
                nodes_total_accumulated_time.append(next_nodes_accumulated_time + heuristic_time)
        node_tuples = list(zip(nodes_total_accumulated_time, next_nodes_idx, heuristic_distance_list, next_nodes_speed_limit
        for node_tuple in node_tuples:
            if node_tuple[1] not in visited_idx:
                heapq.heappush(heap, node_tuple)
        for next_node_tuple in heap:
            next_node_total_accumulated_time, next_node_idx, heuristic_distance, next_node_speed_limit = next_node_tuple
            next_node_speed_limit = next_node_speed_limit * 1000 / 3600
            heuristic_time = heuristic_distance / next_node_speed_limit
            next_node_accumulated_time = next_node_total_accumulated_time - heuristic_time
            if next_node_idx not in visited_idx:
                visited_idx.append(next_node_idx)
                num_visited_nodes += 1
                precede.append(visited_idx.index(current_node_idx))
                accumulated_time.append(next_node_accumulated_time)
                time[visited_idx.index(current_node_idx)][visited_idx.index(next_node_idx)] = next_node_accumulated_time - a
                time[visited_idx.index(next_node_idx)][visited_idx.index(current_node_idx)] = next_node_accumulated_time - a
            if next_node_idx == end:
                is_reach_end = True
        if is_reach_end or len(heap) == 0:
            break
    if is_reach_end:
        path = [len(visited_idx) - 1]
        dist = 0
        while True:
            now_idx = path[-1]
            precede_idx = precede[now_idx]
            dist += time[precede_idx][now_idx]
            path.append(precede_idx)
            if precede_idx == 0:
                break
        path.reverse()
        for i in range(len(path)):
            path[i] = visited_idx[path[i]]
    else:
        path = -1
        dist = -1
        print(f'The path from {start} to {end} with ASTAR doesn\'t exist!')
    num_visited = num_visited_nodes
    return path, dist, num_visited
```

Complete A* time with priority queue (inplemented by heap)
It's implement is similar to A* and change distance to time.
admissible heuristic function is heuristic distance divide the speed limit.
heap(priority queue) is only with the starting point first.
visit_idx is a list which recordthe nodes with index we have visited.
time is a numpy matrix which record the time you take between two nodes.
time[i, j] can get the time you take between the node I and the node j.
precede is a list which records the preceding nodes index in visit_idx.
precede[i] can get the index of preceding node i in visit_idx.
accumulated_time is a list which records the time from the starting point to the node with index.
is_reach_end marks the end point is reach or not and the time to break from loop.
num_visited_nodes is a value which record the number of the visited value. It begins with zero and plus one when the new node is visited.
The current node is popped from heap(priority queue), which is the node we are visiting and the time from the starting point to the node with index is the least in the heap. And then find the following nodes by the current node.
heap(priority queue) is sorted by the total time (the time from the starting point to the node with index and the heuristic time), so the time is the addition of the time from the starting point to the current node, the time between the current node and the following node and the heuristic time.
All the time is the distance divide the speed limit.
Push the tuple of the total time, the index of the following nodes, the heuristic distance, and the speed limit into the heap(priority queue).
Getting following nodes by search in lists of list with starting point.
After visiting a node, add the node to the list visit_idx and add its preceding nodes index in visit_idx to the list precede.
The time between the current nodes and the following nodes is get from the tuple in the heap(priority queue). The time received by the tuple is total time, so we have to minus it with the time accumulated to the preceding nodes and the heuristic time.
Record the time between current nodes and the following nodes to the matrix time.
Jump out of the loop if the end point is reached or the stack is empty.
With the list precede which records the preceding nodes index in visit_idx, we can trace the path and calculating the time from the starting point to the end point.
From the end point, we use the list precede to get to the preceding point, and record the path and the time between the current point and the preceding point.
Recursively record the path and calculating the time until we get to the starting point by keep tracing the preceding nodes.

starting node: 2270143902 end node: 1079387396

```
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: 266
```

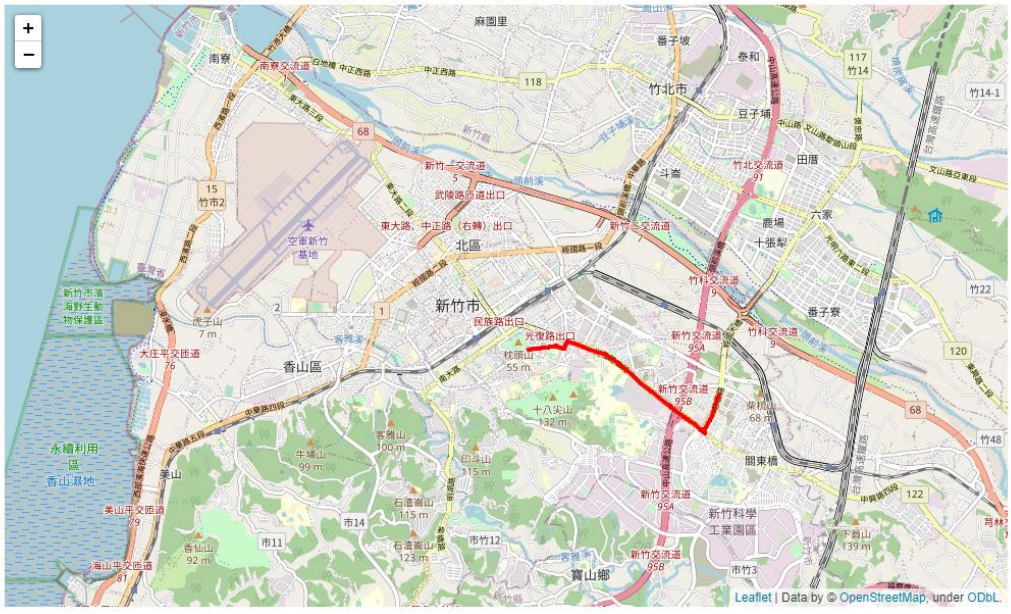
Out[14]:



starting node: 426882161 end node: 1737223506

```
The number of nodes in the path found by A* search: 74
Total second of path found by A* search: 317.5605111724135 s
The number of visited nodes in A* search: 1248
```

Out[21]:



starting node: 1718165260 end node: 8513026827

```
The number of nodes in the path found by A* search: 233
Total second of path found by A* search: 915.3859068561261 s
The number of visited nodes in A* search: 5996
```

Out[14]:

starting node: 2270143902 end node: 1079387396

	# nodes in path	distance (m)	# visited
BFS	88	4978.882	4273
DFS	1232	57208.987	4210
UCS	89	4367.881	5034
A*	89	4367.881	305
A* time	89	320.8782 (s)	266

starting node: 426882161 end node: 1737223506

	# nodes in path	distance (m)	# visited
BFS	60	4215.521	4606
DFS	998	41094.658	8030
UCS	60	4162.609	6946
A*	63	4154.752	1197
A* time	74	317.5605 (s)	1248

starting node: 1718165260 end node: 8513026827

	# nodes in path	distance (m)	# visited
BFS	183	15442.395	11241
DFS	1521	64821.604	3291
UCS	275	14241.51	11929
A*	277	14218.255	7089
A* time	233	915.3859 (s)	5996

The path in A* time is similar to that in A*.
The efficiency of A* time is often beat that of A*, so I think A* time is a kind of efficient algorithm.

Describe problems you meet and how you solve them.

- How to find the following nodes?
  - With the assist of the lists of list and find it with searching the column with specific matching values like the starting points.
- Why does dfs not work when I just change the queue to the stack from bfs?
  - Although changing the queue to the stack from bfs, some nodes should push into the stack later. You have to push they after we recognize the node we encounter former can't lead us to the end point instead of pushing they into the stack when we met at first.
- How to create 2D array to record the distance between 2 nodes?
  - I first use [[0] * n] * n, but it will make some mistakes when assign some values to the element in the 2D list. Therefore, I change to use [[0] * n for _ in range(n)]and it works.
- How to record the path?
  - Using a list to record the preceding node of the currunt visiting node.
- How to complete UCS?
  - BFS and priority queue, and priority queue can implemented by heapq. Moreover, you have to compare the accumulated distance.
- How to calculate the accumulated distance?
  - With the help of the numpy 2D array which record the distance between two nodes. And record to the list.