VIETNAM NATIONAL UNIVERSITY, HCMC -
UNIVERSITY OF SCIENCE

FACULTY OF INFORMATICS TECHNOLOGY
ADVANCED PROGRAM IN COMPUTER SCIENCE

DATA STRUCTURES - CS163

Technical Report

Task 2

Students:
Le Duc Tung Duong - 23125081

Instructors:
Thanh Ho Tuan, M.Sc.
Dung Nguyen Le Hoang, M.Sc.,
Tien Dinh Ba Ph.D.

2024

Summary

§1 Introduction

In the previous course *CS162*, I built a graph based on a given set of data of a bus map and implemented *Dijkstra's algorithm* to find the shortest path between two nodes in a positive weighted directed graph. In this task of *CS163*, I will continue to use this graph to implement some improvements to find the shortest path more efficiently.

§2 Improvements In Algorithm

In the report of task one, I showed that *Dijkstra's algorithm* is not efficient in finding the shortest path between two nodes in a graph with millions of nodes. Thus, in this task, I implement two algorithms which can run faster than *Dijkstra's*: *A* algorithm* and *Contraction Hierarchies*, which are built based on *Dijkstra's.*

§2.1 A* Algorithm

§2.1.1 Introduction

*A** is an algorithm developed in 1968 with the same purpose as *Dijkstra's*, pathfinding. However, it is a combination of a formal path-finding algorithm *Dijkstra's algorithm* and a heuristic approach *Greedy Best-First-Search*. Thus, it determines the step by favoring vertices close to both the source (*Dijkstra's algorithm*) and the target node (*Greedy Best-First-Search*). Although *A** is built on top of a heuristic approach, which usually does not guarantee the result, it can find the optimal shortest path.

§2.1.2 Implementation

In *A**, the only difference is $f(n)$, the cost of each vertex n and the sum of two values:

- $g(n)$: the exact lowest cost from the source node to n
- $h(n)$: the estimated cost to the target node and also the heuristic function. Here, I use Euclidean distance for heuristic value.

Below is the **pseudocode** of *A* algorithm*:

```
Initialize open_list as a priority queue
Initialize close_list as an empty set
function HEURISTIC(current_node, target_node)
    return Euclidean distance between current_node and target_node
end function
while open_list is not empty do
    n ← node in open_list with the smallest f(n) value
    if n is the target node then
        return path to n
    end if
    Add n to close_list
    for each neighbor m of n do
        if m is in close_list then
            continue
```

end if
tentative_g $\leftarrow g(n) + \text{cost}(n, m)$
if m is not in *open_list* or tentative_g $< g(m)$ **then**
Update $g(m)$ to tentative_g
$h(m) \leftarrow \text{heuristic}(m, \text{target\_node})$
$f(m) \leftarrow g(m) + h(m)$
if m is not in *open_list* **then**
Add m to *open_list*
end if
end if
end for
end while
return failure if *open_list* is empty and target node is not found

Below the **source code** of *A* algorithm* I implemented:

```python
def a_star(self, source_node, target_node): # Load stop info before use
    close_list = set()
    open_list = []
    g = {}
    f = {}
    h = {}
    parents = {}

    def heuristic(cur_id, target_id):
        h[cur_id] = self.get_distance(self.stop_info[str(cur_id)][0],
    ↪  self.stop_info[str(cur_id)][1],
                                  self.stop_info[str(target_id)][0],
                              ↪  self.stop_info[str(target_id)][1])

    g[source_node] = 0
    heuristic(source_node, target_node)
    f[source_node] = g[source_node] + h[source_node]
    parents[source_node] = source_node
    heapq.heappush(open_list, (f[source_node], source_node))
    while open_list:
        _, n = heapq.heappop(open_list)

        if n in close_list:
            continue

        if n == target_node:
            path = self.get_path(n, parents)
            return f[target_node], path

        close_list.add(n)

        for m in self.G.neighbors(n):
            g_new = g[n] + self.times_all[n][m]
```

```
            if m not in h:
                heuristic(m, target_node)
            f_new = g_new + h[m]

            if m not in f or f_new < f[m]:
                g[m] = g_new
                f[m] = f_new
                parents[m] = n
                heapq.heappush(open_list, (f_new, m))
    return 0, []
```

§2.1.3 Conclusion

Space complexity: $O(V)$

Time complexity: $O((E+V)\log V$ in the worst case

Although the time complexity of *A* algorithm* in the worst case is just the same as *Dijkstra's*, *A** in this case runs slower. This is because it has to consider two values, $g(n)$ and $h(x)$, with every node. Thus, it takes longer.

The efficiency of *A* algorithm* largely depends on the heuristic function. In a small road network, *Euclidean distance* is good enough to get the shortest path without checking all nodes, except for one case. When there are no shortest path between two nodes, it has to check all nodes, thus, even worse than *Dijkstra's*.

In summary, in road networks, *A** is more efficient than *Dijkstra's* in terms of time, especially in a large graph. With the given graph, there are only nearly 5000 nodes, so *Dijkstra's* and *A** do not show much difference. Sometimes, the former is even faster.

e.g. Given the problem of finding shortest path from stop 6860 to 1568. We compare *A** and *Dijkstra's:*

Algorithm	Time (s)	Nodes checked
A*	0.0184	2,799
Dijkstra	0.0156	4,397

Although number of nodes *A** checks is fewer than *Dijkstra's*, it takes a little bit longer to do the query. As I explained, this graph is not really large and *A** has to check each node longer, thus executing time is slightly longer.

§2.2 Contraction Hierarchies

§2.2.1 Introduction

With a graph of millions of nodes, as road networks do, people try to upgrade *Dijkstra's algorithm* to quickly query the shortest path between two nodes. In a road network, these extension algorithms are called *route-planning algorithms*. Among these algorithms, *Contraction Hierarchies* offers a speedup in query with modest preprocessing time.

§2.2.2 Implementation

Given a directed graph $G = (V, E)$, where each edge e in E has a weight representing the time it takes to travel on the road, we need to find the shortest path between source node and target node in V.

In a map, each node has its own importance. Nodes on highways, with many paths going through, usually have higher importance or higher chance to be passed in the shortest path while others, with fewer paths, have lower importance. This is the *hierarchical information* of a graph. We use this information to find the shortest path by getting to higher nodes from the source, then when getting close to the target node, we get to lower nodes until reaching the target.

Thus, *Contraction Hierarchies* has two processes:

- **Preprocess:** Categorize nodes and edges based on the importance.
- **Query:** Run a bidirectional search that only considers increasingly important edges.

§2.2.3 Preprocessing

Contraction Hierarchies uses a process of node contraction based on orders of importance to create a hierarchy. After each contraction, some shortcut edges may be added to reduce the number of nodes checked during the query process. When contracting a node v, if the shortest path between two neighbors u and w of v includes v, then we add a shortcut between u and w with the weight being the cost of the shortest path.

Consider contracting node v, we have two lists: U containing v's predecessors and W containing v's neighbors (or children). With each node u in U, we calculate $P(w) = \text{weight}(u, v) + \text{weight}(v, w)$ with each w in W.

Then, we run the local *Dijkstra* of u in a graph having no v until the cost reaches $P_{\max}$. If the path we found is longer than the one with v, $\text{dist}(w) > P(w)$, then a shortcut edge is added between u and w with the weight $P(w)$.

Below is the process of adding shortcuts when contracting v:

```
for u in list(self.G.predecessors(v)):
    if u in self.order_of:
        continue
    P = {}
    for w in list(self.G.neighbors(v)):
        if w in self.order_of:
            continue
        P[w] = self.times_all[u][v] + self.times_all[v][w]
    if not P:
        continue
    P_max = max(P.values())

    D = self.local_dijkstra_without_v(u, v, P_max)
    for w in list(self.G.neighbors(v)):
        if w in self.order_of:
```

```
            continue

        if D[w] > P[w]:
            if self.G.has_edge(u, w):
                self.G.get_edge_data(u, w)[0]['shortcut_node'] = v
            else:
                self.G.add_edge(u, w, shortcut_node=v)
            self.times_all[u][w] = P[w]
```

Below is the local *Dijkstra's algorithm* used in above code with the source node u in a graph having no v:

```
def local_dijkstra_without_v(self, u, v, P_max):
    vertices = list(self.G.nodes)
    visited = set()
    pq = [(0, u)]
    D = {v: float('inf') for v in vertices}
    visited.add(v)
    D[u] = 0
    while pq:
        cost, n = heapq.heappop(pq)
        if n in visited:
            continue
        if cost > P_max:
            break
        visited.add(n)

        for neighbor in list(self.G.neighbors(n)):
            if neighbor in self.order_of:
                continue
            old_cost = D[neighbor]
            new_cost = D[n] + self.times_all[n][neighbor]
            if new_cost < old_cost:
                D[neighbor] = new_cost
                heapq.heappush(pq, (new_cost, neighbor))
    return D
```

We do this with each node in order of importance. This order is just used to create shortcuts then we use these shortcuts to search. Thus, *the fewer shortcuts, the better the search is*. I implemented two ways of determining the order: *degree* and *edge difference*.

- **Degree:** With this approach, to get the degree of v, we just need to calculate the number of edges containing v by adding the number of predecessors with the number of neighbors. Then, sort all the nodes with ascending degree and get the order.

  Here, I use *Min Heap* to store the degree and the node along with it. Then, I gradually take each element in the heap to store in two predefined lists, which store orders of all the nodes.

```
    def get_node_order(self):
        node_pq = []
        for v in list(self.G.nodes()):
            val = len(list(self.G.neighbors(v))) +
        ↪  len(list(self.G.predecessors(v)))
            heapq.heappush(node_pq, (val, v))
        i = 1
        while node_pq:
            _, v = heapq.heappop(node_pq)
            self.node_order[i] = v
            self.order_of[v] = i
            i += 1
```

- **Edge Difference:** The edge difference of node v is the difference between the number of shortcuts added when contracting only v in the graph and its degree (number of edges containing v in the beginning).

  To get the number of shortcuts added when contracting v, we run the local *Dijkstra* in a graph with no v, as introduced above. Then, we sort all the nodes based on the edge difference and get the initial order.

```
    def edge_difference(self, v):
        dif = - len(list(self.G.neighbors(v))) -
    ↪  len(list(self.G.predecessors(v)))
        for u in list(self.G.predecessors(v)):
            if u in self.order_of:
                continue
            P = {}
            for w in list(self.G.neighbors(v)):
                if w in self.order_of:
                    continue
                P[w] = self.times_all[u][v] + self.times_all[v][w]
            if not P:
                continue
            P_max = max(P.values())

            D = self.local_dijkstra_without_v(u, v, P_max)

            for w in list(self.G.neighbors(v)):
                if w in self.order_of:
                    continue
                if D[w] > P[w]:
                    dif += 1
        return dif
```

However, during the process of preprocessing, when a node is contracted, the edge

difference of others may be affected. Thus, we need to precompute the edge difference until we get the minimum edge difference of the current graph.

```
while node_pq:
    # Calculate edge difference again to update the pq and get the next node
    _, v = heapq.heappop(node_pq)

    new_dif = self.edge_difference(v)
    if node_pq and new_dif > node_pq[0][0]:
        heapq.heappush(node_pq, (new_dif, v))
        continue

    # Continue with the process of adding shortcuts when contracting v
```

§2.2.4 Query

There are many approaches for bidirectional search but here I implemented *bidirectional Dijkstra*.

- From the source node, I go forward to higher-ordered nodes of the current node v, which are also the neighbors (or children) of v.
- From the target node, I go backward to higher-ordered nodes of the current node v which are also the predecessors of v.

```
def bidirectional_dijkstra(self, source_node, target_node):
    vertices = list(self.G.nodes())
    visited_start = set()
    visited_end = set()
    parents1 = {}
    parents2 = {}
    dist1 = {v: float('inf') for v in vertices}
    dist2 = {v: float('inf') for v in vertices}

    parents1[source_node] = source_node
    parents2[target_node] = target_node
    dist1[source_node] = 0
    dist2[target_node] = 0
    pq_start = [(0, source_node)]
    pq_end = [(0, target_node)]
    while pq_start or pq_end:
        if pq_start:
            _, current_vertex = heapq.heappop(pq_start)
            if current_vertex in visited_start:
                continue
            visited_start.add(current_vertex)

```

```
            for neighbor in self.G.neighbors(current_vertex):
                if self.order_of[neighbor] <= self.order_of[current_vertex]:
                    continue

                new_cost = dist1[current_vertex] +
            ↪  self.times_all[current_vertex][neighbor]
                if new_cost < dist1[neighbor]:
                    parents1[neighbor] = current_vertex
                    dist1[neighbor] = new_cost
                    heapq.heappush(pq_start, (new_cost, neighbor))
        if pq_end:
            _, current_vertex = heapq.heappop(pq_end)
            if current_vertex in visited_end:
                continue
            visited_end.add(current_vertex)

            for neighbor in self.G.predecessors(current_vertex):
                if self.order_of[neighbor] <= self.order_of[current_vertex]:
                    continue

                new_cost = dist2[current_vertex] +
            ↪  self.times_all[neighbor][current_vertex]
                if new_cost < dist2[neighbor]:
                    parents2[neighbor] = current_vertex
                    dist2[neighbor] = new_cost
                    heapq.heappush(pq_end, (new_cost, neighbor))
```

After that, we store the all the nodes that exist in both searches in L. From this, we find the one that satisfy:

$$\text{dist}(s,t) = \min\{\text{dist}(s,v) + \text{dist}(v,t)\} \quad \text{over all } v \in L$$

```
    L = [v for v in self.G.nodes if dist1[v] != float('inf') and dist2[v] !=
↪  float('inf')]
    if not L:
        return 0, []

    shortest_time = math.inf
    common_node = 0
    for v in L:
        if shortest_time > dist1[v] + dist2[v]:
            shortest_time = dist1[v] + dist2[v]
            common_node = v
```

After that, we trace back the path by combining it with a recursive function to get the shortcut node stored in an edge. Thus, we get a path from the source node to the highest-ordered node that can be reached and a path from this node to the target node.

```
    def generate_shortcut(start_node, end_node):
        shortcut_node = self.G.get_edge_data(start_node,
    ↪   end_node)[0]['shortcut_node']
        if shortcut_node != 0:
            return generate_shortcut(start_node, shortcut_node) + [shortcut_node] +
        ↪   generate_shortcut(shortcut_node, end_node)
        else:
            return []

    shortest_path = []
    path1 = []
    cur_node = common_node

    while parents1[cur_node] != cur_node:
        tmp_node = parents1[cur_node]
        path = []
        if self.G.get_edge_data(tmp_node, cur_node)[0]['shortcut_node'] != 0:
            path = generate_shortcut(tmp_node, cur_node)
        path1 = path + path1
        path1 = [tmp_node] + path1
        cur_node = tmp_node

    cur_node = common_node
    path2 = []
    while parents2[cur_node] != cur_node:
        path2.append(cur_node)
        tmp_node = parents2[cur_node]
        path = []
        if self.G.get_edge_data(cur_node, tmp_node)[0]['shortcut_node'] != 0:
            path = generate_shortcut(cur_node, tmp_node)
        path2 += path
        cur_node = tmp_node
    path2.append(cur_node)

    shortest_path = path1 + path2
```

§2.2.5 Conclusion

The complexity of *Contraction Hierarchies* is very complicated to precisely calculate. So, I will take example to show how efficient it is.

e.g. Continue with two nodes in 2.1.3., from 6860 to 1568 , I will do some comparison:

Algorithm	Preprocess Time (s)	Query Time (s)	Nodes checked
A*	0.0	0.0184	2,799
Dijkstra	0.0	0.0156	4,397
Contraction Hierarchies	17.4383	0.0032	226

By taking a modest process time, the number of nodes needed to be checked is significantly reduced to 226 nodes (nearly **20 times** smaller than *Dijkstra's* and over **14 times** smaller than $A*$). Thus, query time is also reduced to **3.2 milliseconds** (nearly 6 times faster than A* and nearly 5 times with Dijkstra's).

§3 Further Improvement - Path Caching

§3.1 Introduction

Suppose we have district U and V. From a source node u in U to a target node v in V, there can be a fixed route which all shortest paths from u to v go through. Our target is to store this route and use it when doing query.

§3.2 Implementation

By calculating all shortest paths from U to V, we can take the LCS (Longest Common String) of all of these paths. I implemented two approaches to find LCS:

- **Approach 1:** Manually check each node in *path2* with *path1*

```
    def update_fixed_point(self, path1, path2, time1, time2):
        new_path = []
        cnt = {v: 1 for v in path1}

        for u in path2:
            if cnt.get(u, 0) == 1:
                i = path1.index(u)
                j = path2.index(u)

                while i < len(path1) and j < len(path2) and path1[i] ==
            ↪  path2[j]:
                    new_path.append(path1[i])
                    i += 1
                    j += 1
                break

        if not new_path:
            if time2 < time1:
                return time2, path2
            else:
                return time1, path1

        new_time = 0
        for u, v in zip(new_path, new_path[1:]):
            new_time += self.times_all[u][v]
        return new_time, new_path
```

- **Approach 2:** *Counter()* function in *collections* library

```
    def update_intersection(self, path1, path2, time1, time2):
        result = collections.Counter(path1) & collections.Counter(path2)
        new_path = list(result.elements())

        new_time = 0
        for u, v in zip(new_path, new_path[1:]):
            new_time += self.times_all[u][v]

        return new_time, new_path
```

Because we have to calculate shortest paths of all pairs of nodes, I use *Dijkstra's algorithm* to do this work. From a node u, it can visit all other nodes in modest time with $O((E + V) \log V)$. Besides, I store a boolean variable *have_fixed_point* to indicate whether there is a fixed route between two districts or not.

```
def generate_caching_json(self):
    with open("input_data/stop_coordinate.json", "r", encoding='utf-8') as f:
        stop_zone = json.load(f)
    with open("input_data/path_caching_demo.json", "r", encoding='utf-8') as f:
        fixed_point = json.load(f)

    vertices = list(self.G.nodes)
    for source_node in vertices:
        shortest_time, parents =  self.dijkstra(source_node)
        source_zone = stop_zone[str(source_node)][2]

        for target_node in vertices:
            path = self.get_path(target_node, parents)
            target_zone = stop_zone[str(target_node)][2]

            if fixed_point[source_zone][target_zone]['have_fixed_point'] == False:
                continue
            elif fixed_point[source_zone][target_zone]['have_fixed_point'] == True:
                fixed_point[source_zone][target_zone]['time'],
            ↪  fixed_point[source_zone][target_zone]['path'] =
            ↪  self.update_intersection(fixed_point[source_zone][target_zone]['path']
            ↪  , path, fixed_point[source_zone][target_zone]['time'],
            ↪  shortest_time[target_node])

                if fixed_point[source_zone][target_zone]['time'] == 0:
                    fixed_point[source_zone][target_zone]['have_fixed_point'] = False
            else:
                fixed_point[source_zone][target_zone]['have_fixed_point'] = False
                for v in path:
                    if stop_zone[str(v)][2] != source_zone and stop_zone[str(v)][2]
                    ↪  != target_zone:
```

```
                        fixed_point[source_zone][target_zone]['have_fixed_point'] =
                    ↪   True
                        fixed_point[source_zone][target_zone]['time'] =
                    ↪   shortest_time[target_node]
                        fixed_point[source_zone][target_zone]['path'] = path
                        break

    with open("input_data/path_caching_demo.json", "w", encoding='utf-8') as f:
        json_object = json.dumps(fixed_point, indent=4, ensure_ascii=False)
        f.write(json_object)
```

§3.3 Conclusion

Although path caching is a feature to optimize searching process, it is not suitable with all other algorithms, especially it needs to find shortest paths of two segments.

e.g. Continue with example in 2.1.3. and 2.2.5., from 6860 to 1568, I add path caching to each algorithm to see their behaviour:

Algorithm	Preprocess Time (s)	Query Time (s)	Nodes checked
A*	0.0	0.0184	2,799
A* (Path Caching)	0.0	0.0073	2,706
Dijkstra	0.0	0.0156	4,397
Dijkstra (Path Caching)	0.0	0.0192	8,794
CH	17.4383	0.0032	226
CH (Path Caching)	17.0597	0.0054	331

**CH: Contraction Hierarchies*

With algorithms which need to check all other nodes like *Dijkstra* or all higher-ordered nodes like *Contraction Hierarchies*, we need to do this process twice for two segments. Thus, number of nodes checked is bigger (331 nodes $>$ 226 nodes - *Contraction Hierarchies* or 8794 nodes $>$ 4397 nodes - *Dijkstra*), resulting slower query time.

Meanwhile, *A** go straight to the node we need to find. Combined with path caching, it can reduce number of nodes and speed up query process **over 2 times**.

In summary, only algorithms going straight to the target node like A* uses path caching efficiently.

§4 Bonus

§4.1 More comparison

To have an overview about each algorithm with and without caching, I query 15*4397 or 65955 pairs and calculate executing time of each.

The result is the same as before. In conclusion, path caching is suitable with *A** than *Contraction Hierarchies*. However, the latter is faster in all cases because it has preprocessed.

Algorithm	Query Time (s)	Query Time (Path Caching) (s)
A*	0.0114	0.0088
Contraction Hierarchies	0.0017	0.0018

§4.2 Generate .json file for stop information

To quickly get the coordinate and district (or zone) of a stop, I write a function to do the queries with all stops.

```
def get_stop_info(self):
    stop_query = StopQuery()
    stop_query.read_stop()
    vertices = list(self.G.nodes())
    coordinate = {}
    cnt = 1
    for u in vertices:
        print(f".........Loading {cnt}/4397 stop coordinates.........")
        cnt+=1
        nodes = stop_query.searchByABC(StopId = u)
        coordinate[u] = [nodes[0].get_lng(), nodes[0].get_lat(), nodes[0].get_zone()]
    return coordinate
```

Then, I write information of all stops into a file.

```
def write_stop_info_json(self):
    stop_info = self.read_stop_info()
    with open("input_data/stop_coordinate.json", "w", encoding='utf-8') as f:
        json_object = json.dumps(stop_info, indent=4, ensure_ascii=False)
        f.write(json_object)
    print("ALL STOP COORDINATES HAVE BEEN WRITTEN INTO FILE!")
```

Now, coordinate and district of all nodes are stored in the file *stop_coordinate.json*. Each time we need stop information, we just need to call the function below to load all information into a dict.

```
def load_stop_info_json(self):
    with open("input_data/stop_coordinate.json", "r", encoding='utf-8') as f:
        self.stop_info = json.load(f)
```

§4.3 Query along with updating cache

Suppose having a huge graph with hundreds of millions of nodes and millions of edges, the preprocessing process of finding fixed routes may take too many resources. In this case, I think we can do the query and update the cache at the same time.

Idea:

- **Step 1:** We need to find the shortest path from source node u to target node v. Suppose between 2 districts that 2 nodes belong to has no fixed route, we do apply

any search algorithm to find the path from u to v. Otherwise, there is a fixed route from a to b.

- **Step 2:** We find the shortest paths from u to a and from b to v. Adding three parts, together we have the path from u to v. However, this is not guaranteed the optimal solution, so we do a search from u to v alone. If it is shorter, we update the cache.

Here, the search I use is A*, but it can be changed into any others.

```
def get_shortest_path_astar_pathcaching_updating(self, source_node, target_node):
    self.load_stop_info_json()
    with open("input_data/path_caching.json", "r", encoding='utf-8') as f:
        fixed_point = json.load(f)
    time = 0
    path = []
    source_zone = self.stop_info[str(source_node)][2]
    target_zone = self.stop_info[str(target_node)][2]
    if fixed_point[source_zone][target_zone]['have_fixed_point'] == False:
        self.a_star(source_node, target_node)
    elif fixed_point[source_zone][target_zone]['have_fixed_point'] == True:
        new_time, new_path = self.a_star(source_node, target_node)
        if new_time is not None:
            tmp_time = fixed_point[source_zone][target_zone]['time']
            tmp_path = fixed_point[source_zone][target_zone]['path']
            time1, path1 = self.a_star(source_node, tmp_path[0])
            time2, path2 = self.a_star(tmp_path[-1], target_node)
            cur_time = time1 + tmp_time + time2
            cur_path = path1 + tmp_path + path2

            if new_time >= cur_time:
                time = cur_time
                path = cur_path
            else:
                fixed_point[source_zone][target_zone]['time'],
            ↪ fixed_point[source_zone][target_zone]['path'] =
            ↪ self.update_intersection(tmp_path, new_path, tmp_time, new_time)
                if fixed_point[source_zone][target_zone]['time'] == 0:
                    fixed_point[source_zone][target_zone]['have_fixed_point'] = False
                time = new_time
                path = new_path
    else:
        new_time, new_path = self.a_star(source_node, target_node)
        fixed_point[source_zone][target_zone]['have_fixed_point'] = False
        if source_zone != "Ngoại thành" and target_zone != "Ngoại thành":
            for v in new_path:
                if self.stop_info[str(v)] != source_zone and self.stop_info[str(v)]
            ↪ != target_zone:
                    fixed_point[source_zone][target_zone]['have_fixed_point'] = True
                    fixed_point[source_zone][target_zone]['time'] = new_time
```

```
                    fixed_point[source_zone][target_zone]['path'] = new_path
                    break
        time = new_time
        path = new_path

    print(time, path)
    with open("input_data/path_caching.json", "w", encoding='utf-8') as f:
        json_object = json.dumps(fixed_point, indent=4, ensure_ascii=False)
        f.write(json_object)
    if not path:
        print(f'No p found from {source_node} to {target_node}!')
        return 0, path
    return time, path
```

Although it can be slower than A* alone several milliseconds, it is so small that users may not care about. When the data is enough, we can do the query with path caching immediately as I introduced in section 3.

§4.4 Output shortest path into .geojson file

To view the path more easily, I implement a function to view all stops and path on the map.

```
def output_shortest_path_json(self, paths):
    # geojson file to view the path on the map
    stop_query = StopQuery()
    stop_query.read_stop()
    geojson_data = {
        "type": "FeatureCollection",
        "features": [
            {
                "type": "Feature",
                "geometry": {
                    "type": "LineString",
                    "coordinates": []
                },
            }
        ]
    }

    for (a, b) in zip(paths, paths[1:]):
        coordinates = self.G.get_edge_data(a, b)[0]['coordinates']
        for coordinate in coordinates:
            geojson_data['features'][0]['geometry']['coordinates'].append(coordinate)

    stop_objects = [stop_query.searchByABC(StopId=stopid) for stopid in paths]
    for stop_coordinate, stop_id in zip(stop_objects, paths):
        feature = {
```

```
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": [stop_coordinate[0].Lng, stop_coordinate[0].Lat]
            },
            "properties": {
                "StopId": stop_id,
            }
        }
        geojson_data['features'].append(feature)

    json_object = json.dumps(geojson_data, indent = 4)
    with open("shortest_path_coordinates.geojson", "w") as outfile:
        outfile.write(json_object)

    # json file for all the edges gone through and start, end stop of each
    stop_query = StopQuery()
    stop_query.read_stop()
    edges_data = []
    first_edge = False
    for (a, b) in zip(paths, paths[1:]):
        if first_edge is False:
            edge = self.G.get_edge_data(a, b)[0]
            first_edge = True
            prev_route_id = edge['route_id']
            prev_route_var_id = edge['route_var_id']
        else:
            edges = self.G.get_edge_data(a, b)
            find = False
            for key, item in edges.items():
                if item['route_id'] == prev_route_id and item['route_var_id'] ==
                ↪  prev_route_var_id:
                    edge = item
                    find = True
                    break
            if find is False:
                edge = edges[0]
                prev_route_id = edge['route_id']
                prev_route_var_id = edge['route_var_id']
        start_stop = stop_query.searchByABC(StopId=a, RouteId=str(edge['route_id']),
    ↪  RouteVarId=str(edge['route_var_id']))
        end_stop = stop_query.searchByABC(StopId=b, RouteId=str(edge['route_id']),
    ↪  RouteVarId=str(edge['route_var_id']))
        edge_data = {
            'From': a,
            'To': b,
            'RouteId': edge['route_id'],
            'RouteVarId': edge['route_var_id'],
```

```
            'Distance': edge['distance'],
            'Time': edge['time'],
            'StartStop': start_stop[0].get_stop(),
            'EndStop': end_stop[0].get_stop()
        }
        edges_data.append(edge_data)
    with open("shortest_path_stops.json", "w", newline='', encoding='utf-8') as f:
        for edge in edges_data:
            json_object = json.dumps(edge, ensure_ascii=False)
            f.write(json_object + '\n')
```

§5 Reference

John Lazarsfeld - Tufts Comp 150 Alg Project - Contraction Hierarchies

Introduction to A* From Amit's Thoughts on Pathfinding

Geeks for Geeks - A* Search Algorithm

This is the link to my Github Repository.