**UNIVERSITY OF SCIENCE, HCM-VNU**
**FACULTY OF INFORMATION TECHNOLOGY**

# BIG GRAPH PROCESSING - CS163

**Report**

Name: Đinh Hồ Gia Huy
Student ID: 231253036
Class: 23TT1

## About this report

 We were asked to make improvements to the algorithm aimed at reducing the runtime for finding the shortest path. including storing previously computed paths.

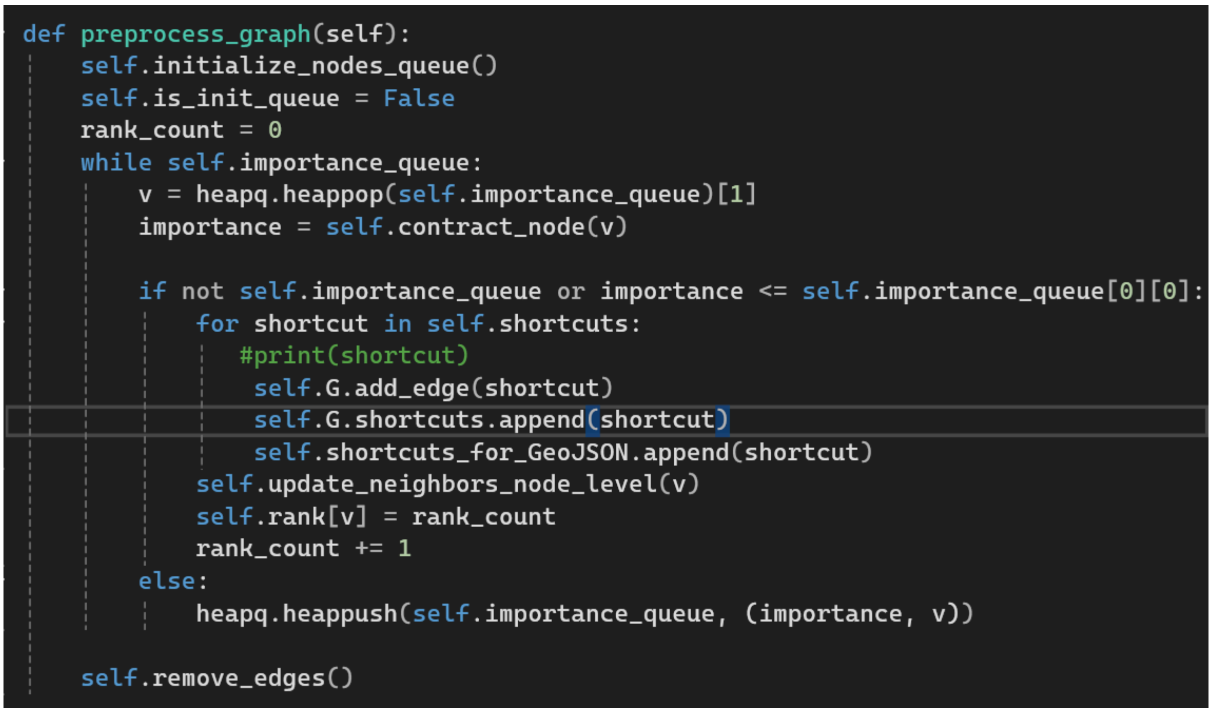
*“For instance, if a route from District 1 to Hoc Mon always includes certain fixed bus routes, these fixed connections can be precomputed and reused in future calculations. “*

I noticed that this perfectly fits the description of Transit Node Routing framework: we choose certain transit nodes based on some particular criteria, then precompute the paths between them, store the paths, and each time we make a query, we try to connect the source and target node to the best transit nodes we could find. Since the paths between transit nodes are already known, this will greatly improve our query time.

## Contraction Hierarchies

***Implementations are found in ContractionHierarchies.py***

Contraction Hierarchies heuristically order the nodes by some measure of importance and contract them one by one in this order. Contracting means that a node is removed (temporarily) and as few shortcut edges as possible are inserted to preserve shortest path distances.



```
def preprocess_graph(self):
    self.initialize_nodes_queue()
    self.is_init_queue = False
    rank_count = 0
    while self.importance_queue:
        v = heapq.heappop(self.importance_queue)[1]
        importance = self.contract_node(v)

        if not self.importance_queue or importance <= self.importance_queue[0][0]:
            for shortcut in self.shortcuts:
                #print(shortcut)
                 self.G.add_edge(shortcut)
                 self.G.shortcuts.append(shortcut)
                 self.shortcuts_for_GeoJSON.append(shortcut)
            self.update_neighbors_node_level(v)
            self.rank[v] = rank_count
            rank_count += 1
        else:
            heapq.heappush(self.importance_queue, (importance, v))

    self.remove_edges()
```

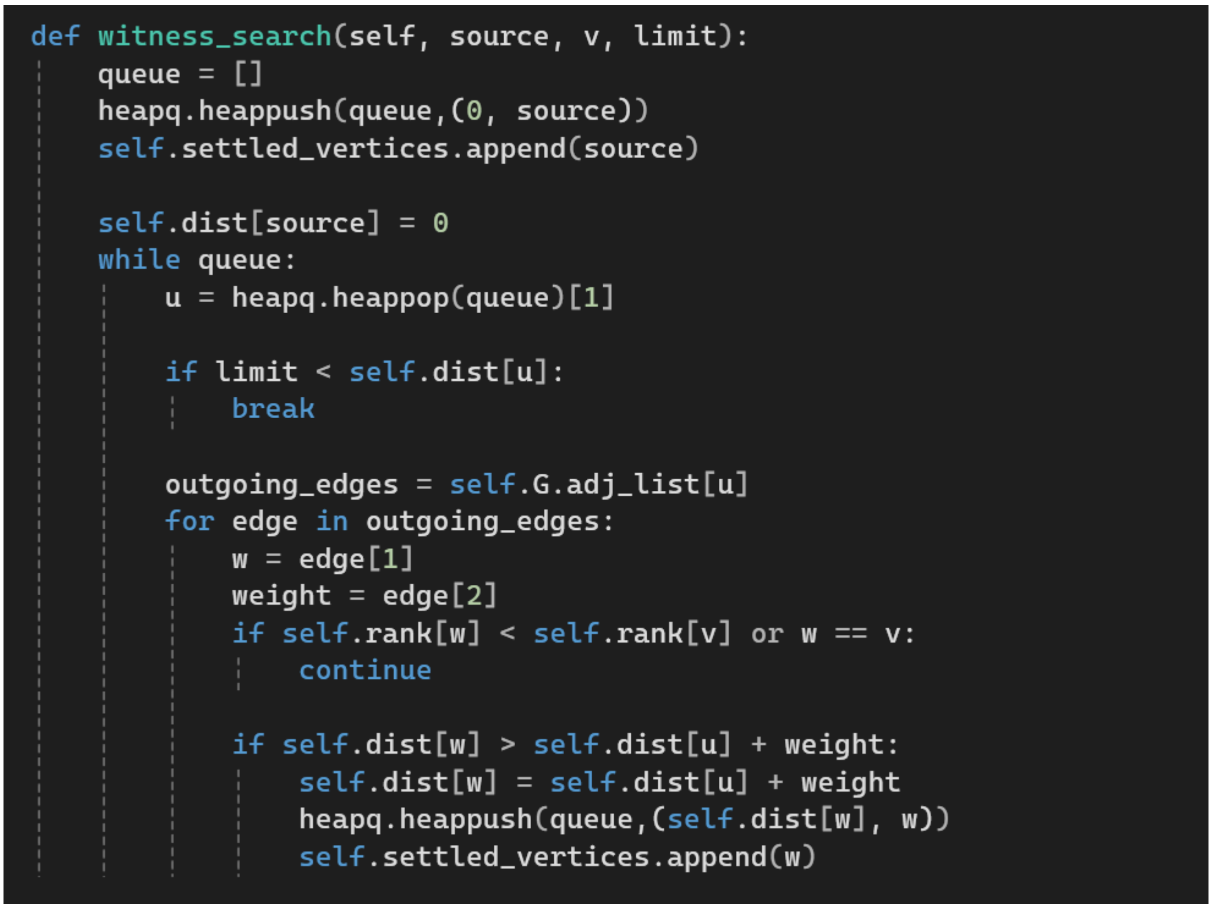
This function perfectly summarizes the core idea of Contraction Hierarchies.

- We push all nodes of the graph into a priority queue, which in Python is basically a Min-Heap.
- The queue's order is the importance of a node, i.e. how disconnected it would leave the graph were it to be removed.
- From that queue, we take out the top node, one by one, and contract it, i.e. remove the node from the graph, and make necessary shortcuts between neighboring nodes that would preserve the shortest paths.

- After all nodes are contracted, the graph is significantly simplified, making shortest-path queries much faster, as they can skip over contracted nodes by using the shortcuts.
- After the graph has been processed and nodes contracted, remove edges that are no longer necessary, i.e. those where the rank of the starting node is higher than the rank of the ending node.

I will now go into details of the main parts .

**Witness search:**



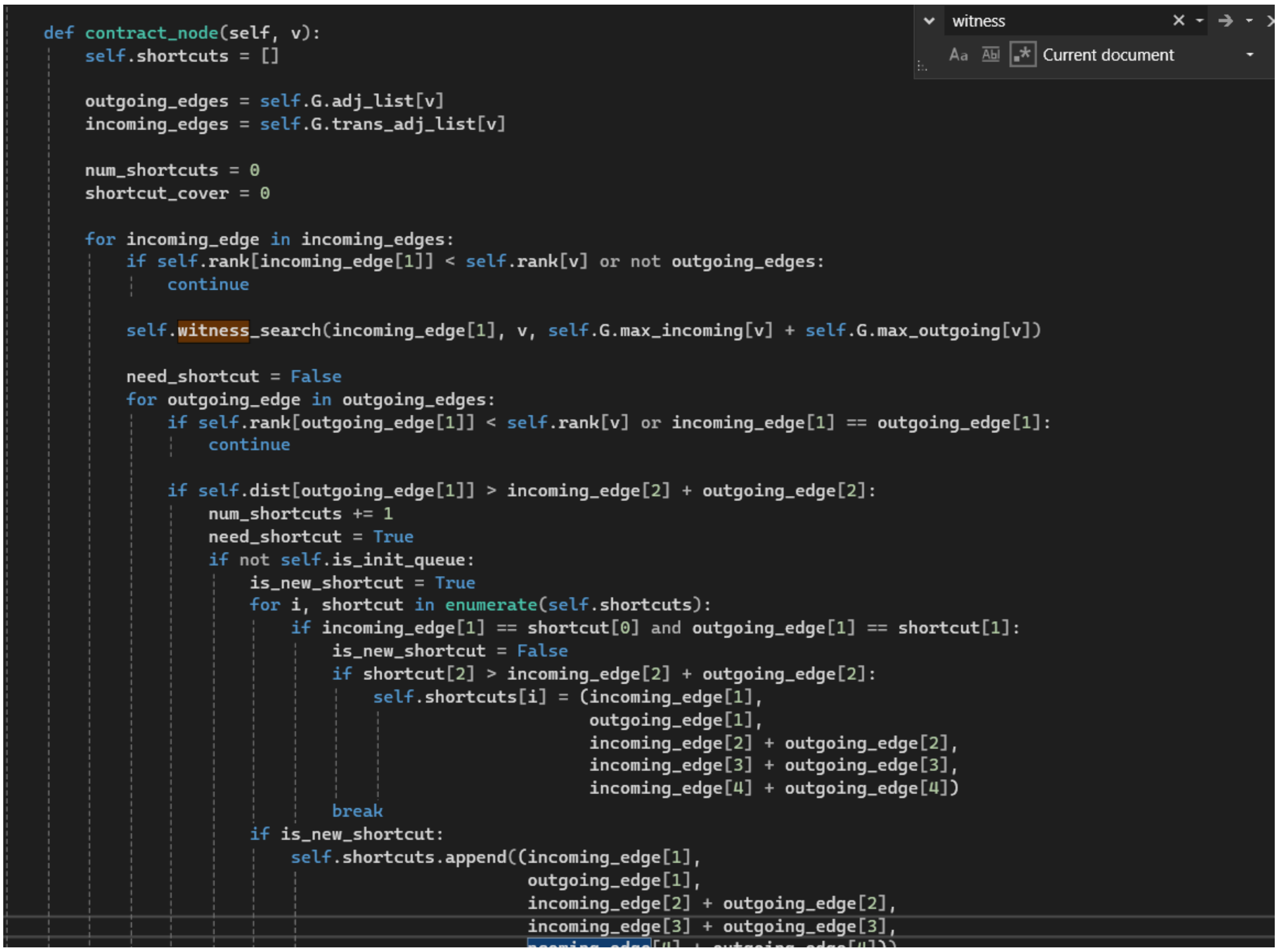
```python
def witness_search(self, source, v, limit):
    queue = []
    heapq.heappush(queue,(0, source))
    self.settled_vertices.append(source)

    self.dist[source] = 0
    while queue:
        u = heapq.heappop(queue)[1]

        if limit < self.dist[u]:
            break

        outgoing_edges = self.G.adj_list[u]
        for edge in outgoing_edges:
            w = edge[1]
            weight = edge[2]
            if self.rank[w] < self.rank[v] or w == v:
                continue

            if self.dist[w] > self.dist[u] + weight:
                self.dist[w] = self.dist[u] + weight
                heapq.heappush(queue,(self.dist[w], w))
                self.settled_vertices.append(w)
```

We perform a Dijkstra-like search from a node source to determine if there is a shorter path to a node v through existing edges than what would be provided by a potential shortcut. The search is bounded by a limit to prevent unnecessary exploration.

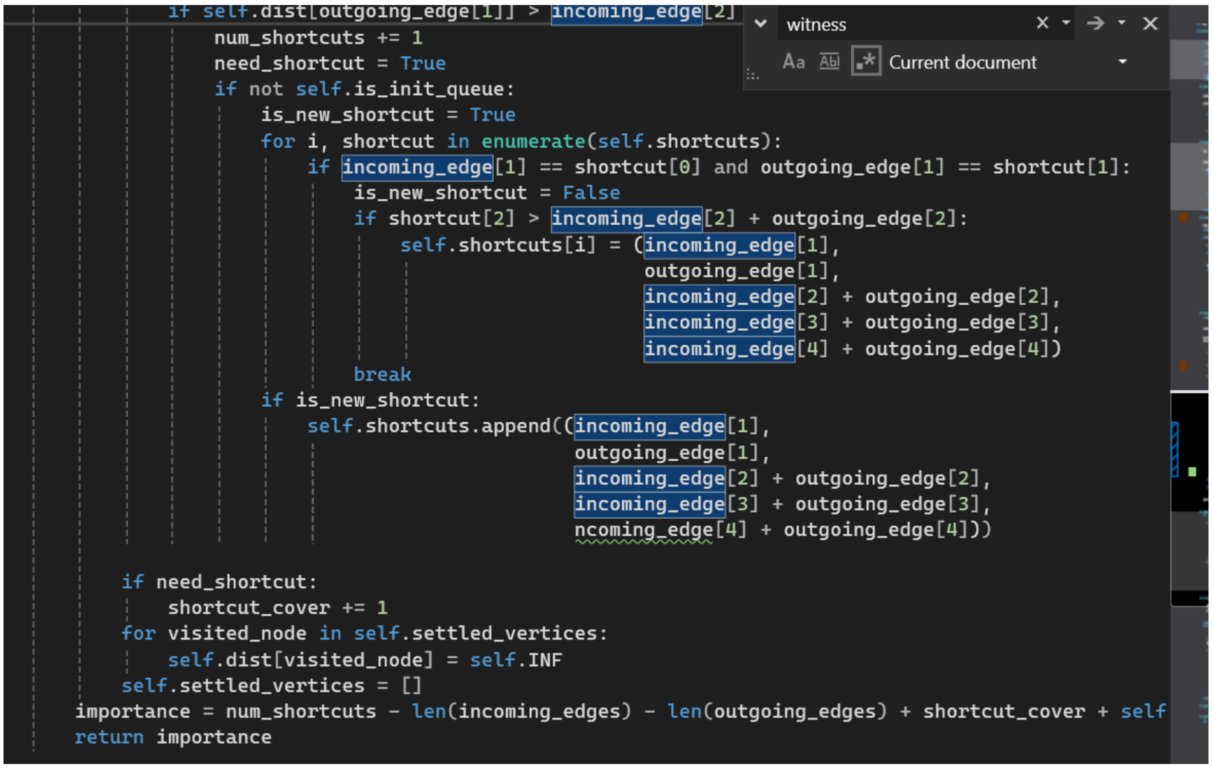
The limit for a witness search of a node we chose is the weight of the heaviest edge going into that node, plus the weight of the heaviest edge coming out of it.

**There are papers on the Internet suggesting that we should also use a limit called *hop*, that is basically the number of local searches we'll be performing. Past that number and we shall exit the function. However I find this optimization unsuitable for our graph since, once again, our graph is sparse, so limiting the number of local searches actually hurts our performance.

## Contracting a node



```
def contract_node(self, v):
    self.shortcuts = []

    outgoing_edges = self.G.adj_list[v]
    incoming_edges = self.G.trans_adj_list[v]

    num_shortcuts = 0
    shortcut_cover = 0

    for incoming_edge in incoming_edges:
        if self.rank[incoming_edge[1]] < self.rank[v] or not outgoing_edges:
            continue

        self.witness_search(incoming_edge[1], v, self.G.max_incoming[v] + self.G.max_outgoing[v])

        need_shortcut = False
        for outgoing_edge in outgoing_edges:
            if self.rank[outgoing_edge[1]] < self.rank[v] or incoming_edge[1] == outgoing_edge[1]:
                continue

            if self.dist[outgoing_edge[1]] > incoming_edge[2] + outgoing_edge[2]:
                num_shortcuts += 1
                need_shortcut = True
                if not self.is_init_queue:
                    is_new_shortcut = True
                    for i, shortcut in enumerate(self.shortcuts):
                        if incoming_edge[1] == shortcut[0] and outgoing_edge[1] == shortcut[1]:
                            is_new_shortcut = False
                            if shortcut[2] > incoming_edge[2] + outgoing_edge[2]:
                                self.shortcuts[i] = (incoming_edge[1],
                                                     outgoing_edge[1],
                                                     incoming_edge[2] + outgoing_edge[2],
                                                     incoming_edge[3] + outgoing_edge[3],
                                                     incoming_edge[4] + outgoing_edge[4])
                            break
                    if is_new_shortcut:
                        self.shortcuts.append((incoming_edge[1],
                                               outgoing_edge[1],
                                               incoming_edge[2] + outgoing_edge[2],
                                               incoming_edge[3] + outgoing_edge[3],
                                               incoming_edge[4] + outgoing_edge[4]))
```

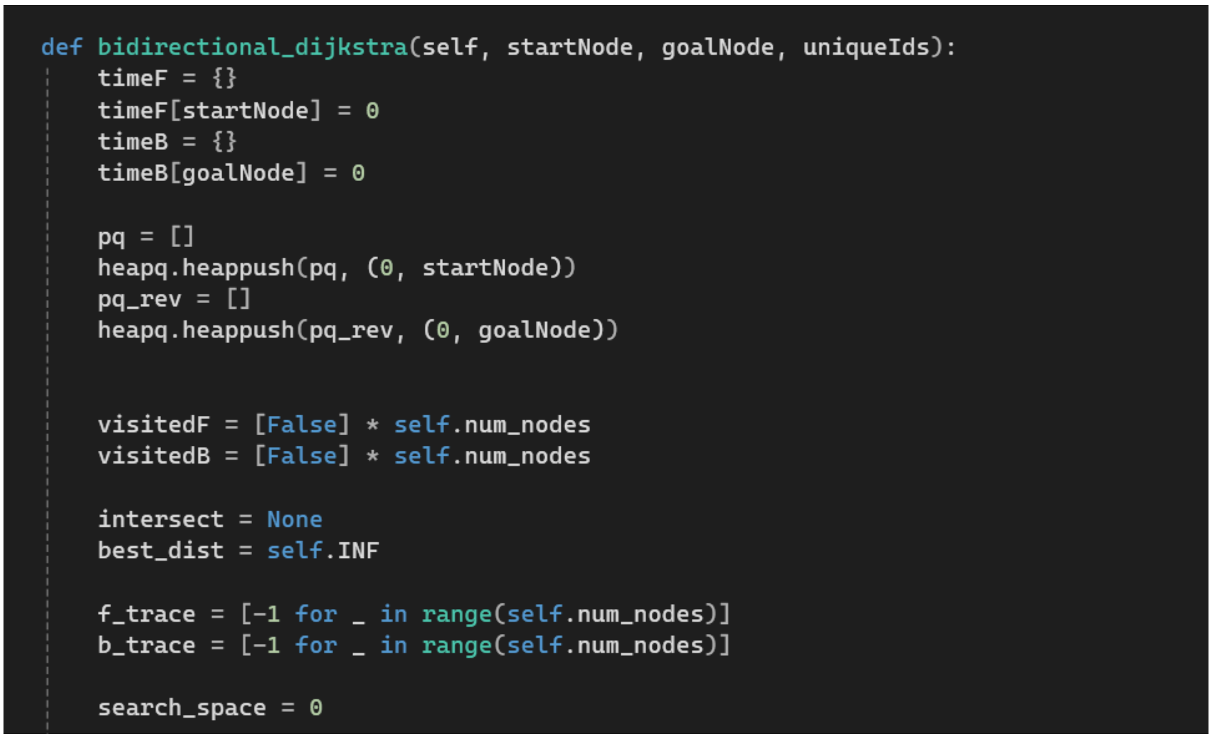


```
            if self.dist[outgoing_edge[1]] > incoming_edge[2]
                num_shortcuts += 1
                need_shortcut = True
                if not self.is_init_queue:
                    is_new_shortcut = True
                    for i, shortcut in enumerate(self.shortcuts):
                        if incoming_edge[1] == shortcut[0] and outgoing_edge[1] == shortcut[1]:
                            is_new_shortcut = False
                            if shortcut[2] > incoming_edge[2] + outgoing_edge[2]:
                                self.shortcuts[i] = (incoming_edge[1],
                                                     outgoing_edge[1],
                                                     incoming_edge[2] + outgoing_edge[2],
                                                     incoming_edge[3] + outgoing_edge[3],
                                                     incoming_edge[4] + outgoing_edge[4])
                            break
                    if is_new_shortcut:
                        self.shortcuts.append((incoming_edge[1],
                                               outgoing_edge[1],
                                               incoming_edge[2] + outgoing_edge[2],
                                               incoming_edge[3] + outgoing_edge[3],
                                               ncoming_edge[4] + outgoing_edge[4]))

        if need_shortcut:
            shortcut_cover += 1
        for visited_node in self.settled_vertices:
            self.dist[visited_node] = self.INF
        self.settled_vertices = []
    importance = num_shortcuts - len(incoming_edges) - len(outgoing_edges) + shortcut_cover + self
    return importance
```

This is the main part of preprocessing: contracts a node v, adding shortcuts if necessary. The function checks incoming and outgoing edges of v to see if a direct connection (shortcut) is needed between nodes that would otherwise be connected through v. It also computes the importance of the node

**The number of shortcuts I added after preprocessing CH is around 6700 edges.**

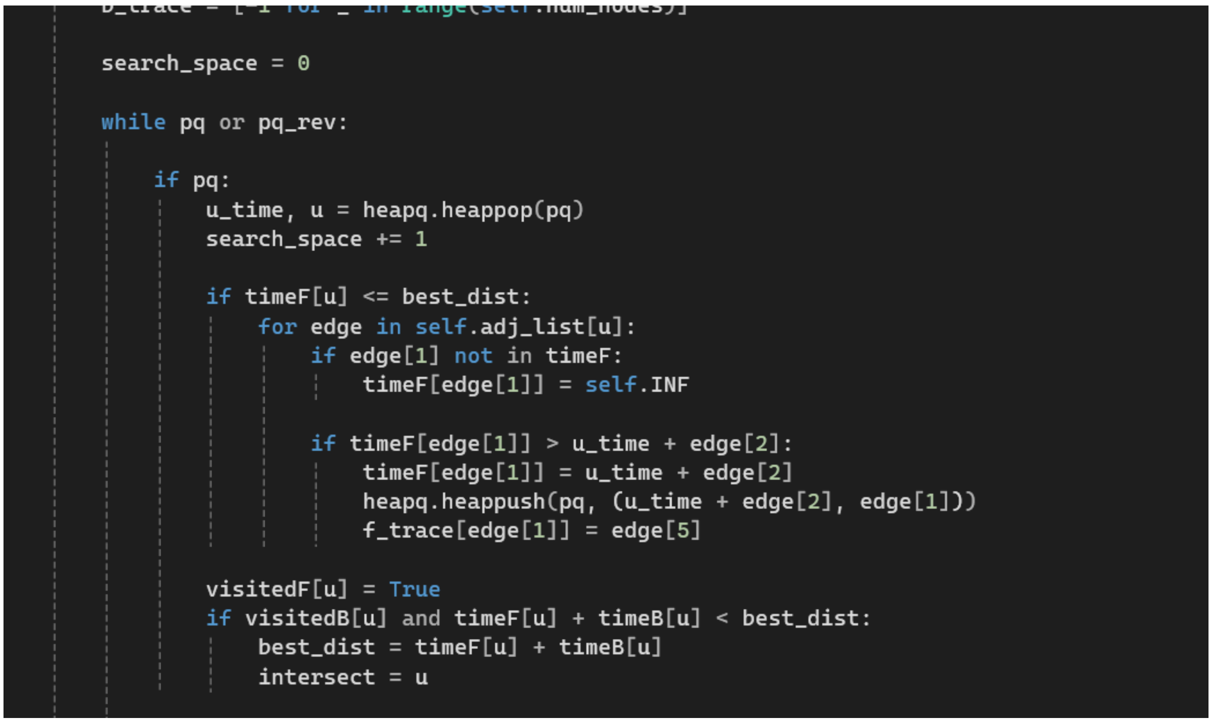
After preprocessing, we can only perform **Bidirectional Dijkstra's** on the graph for a query, with greatly reduced query time.



```python
def bidirectional_dijkstra(self, startNode, goalNode, uniqueIds):
    timeF = {}
    timeF[startNode] = 0
    timeB = {}
    timeB[goalNode] = 0

    pq = []
    heapq.heappush(pq, (0, startNode))
    pq_rev = []
    heapq.heappush(pq_rev, (0, goalNode))


    visitedF = [False] * self.num_nodes
    visitedB = [False] * self.num_nodes

    intersect = None
    best_dist = self.INF

    f_trace = [-1 for _ in range(self.num_nodes)]
    b_trace = [-1 for _ in range(self.num_nodes)]

    search_space = 0
```

Above is the setup, we have the distance dictionaries for forward and backward search (denoted by 'f' and 'b') as well as the corresponding priority queues and traces for return paths.

The intersect node is the node that will be settled first by both the forward search from source and backward search from target.



```
        b_trace = [-1 for _ in range(self.num_nodes)]

        search_space = 0

        while pq or pq_rev:

            if pq:
                u_time, u = heapq.heappop(pq)
                search_space += 1

                if timeF[u] <= best_dist:
                    for edge in self.adj_list[u]:
                        if edge[1] not in timeF:
                            timeF[edge[1]] = self.INF

                        if timeF[edge[1]] > u_time + edge[2]:
                            timeF[edge[1]] = u_time + edge[2]
                            heapq.heappush(pq, (u_time + edge[2], edge[1]))
                            f_trace[edge[1]] = edge[5]

                visitedF[u] = True
                if visitedB[u] and timeF[u] + timeB[u] < best_dist:
                    best_dist = timeF[u] + timeB[u]
                    intersect = u
```

This is the settling for the forward search. The same goes for the backward search.

We pop a node out of the queue. If its estimated distance is smaller than the best distance we could find, we relax it.
We update the best distance when a node is visited by the backward search and its total estimated distance to the source and target is better than the current best distance.

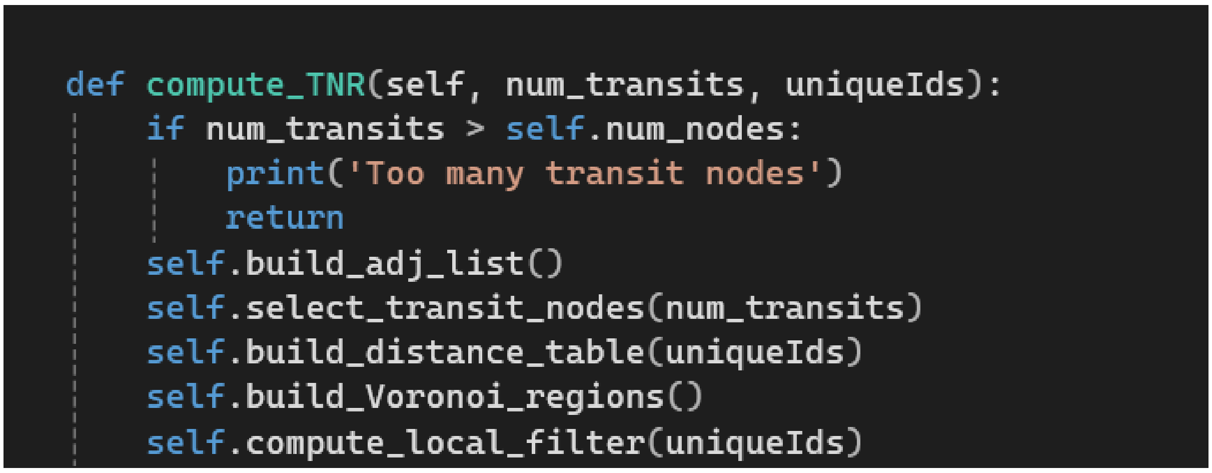
**I noticed that when I tried to use an optimization called ***stalling on demand*** , it actually slows down the routing, so I discarded it. I suppose the technique works better for dense graphs,  (since our graph with close to 5000 nodes is comparatively small) so I'll take note of it and use it later on (if ever).

## Transit Node Routing

***Implementations are found in TransitNodeRouting.py***

The idea of TNR is fairly straightforward as I've mentioned at the start of this report:

"It has to identify a set of transit nodes. It has to find an access node for all nodes. And is has to deal with the fact that some queries between nearby nodes cannot answered via the transit nodes set"
(**Transit Node Routing Reconsidered (arxiv.org)**)



```
def compute_TNR(self, num_transits, uniqueIds):
    if num_transits > self.num_nodes:
        print('Too many transit nodes')
        return
    self.build_adj_list()
    self.select_transit_nodes(num_transits)
    self.build_distance_table(uniqueIds)
    self.build_Voronoi_regions()
    self.compute_local_filter(uniqueIds)
```

This is the main function that orchestrates the TNR preprocessing:

- Selects transit nodes.

  (I find the optimized number of transit nodes for our graph is **250**.)

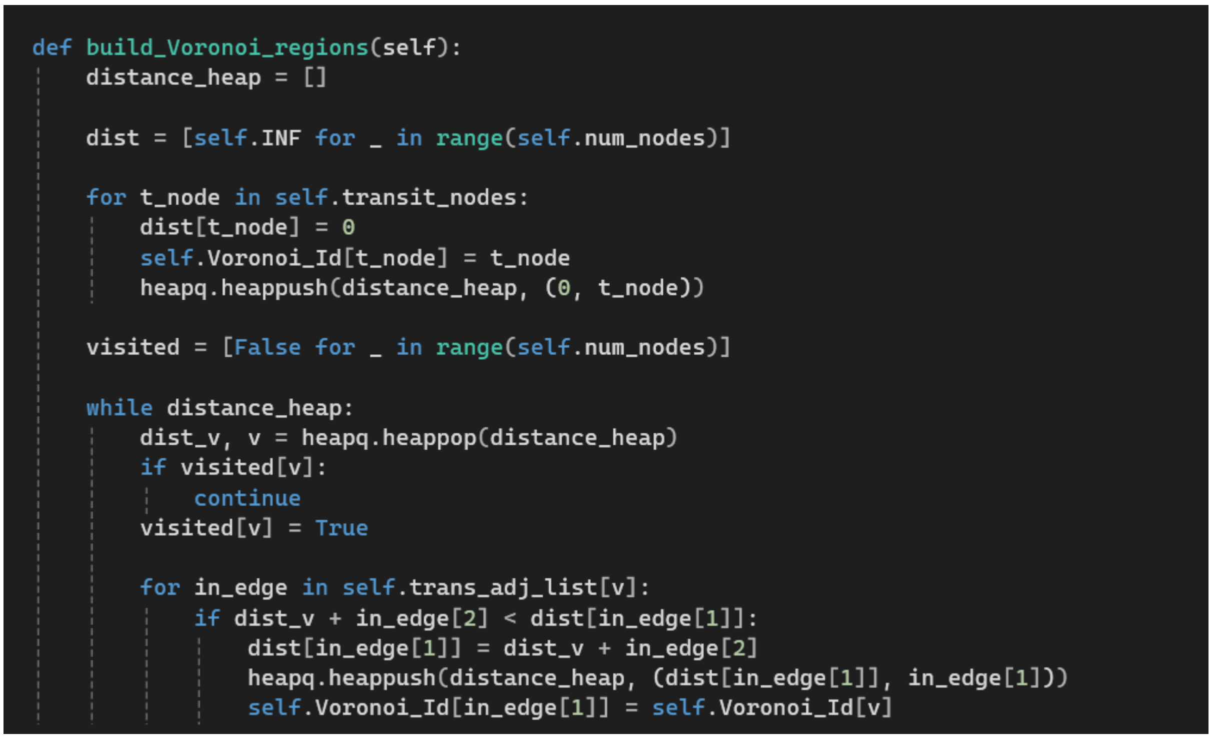
- Builds the distance table between transit nodes.
- Builds Voronoi regions around transit nodes.
- Computes local filters for each node to ensure efficient queries.

Let’s first get some terminologies straight.

A **Voronoi region** is a partitioning of the graph where each node is assigned to the nearest transit node. In other words, nodes in the same Voronoi region will take the same transit node to some other node.

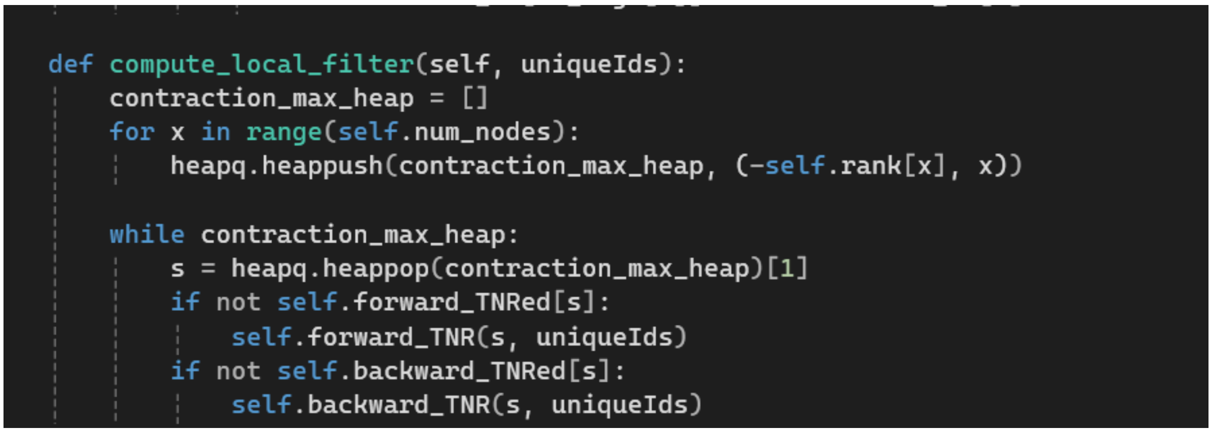
A **local filter** is a way of deciding whether or not we should use a transit node for a node for a query, or if it’s better if we just perform local search on that.

The notable functions that I want to highlight are below:



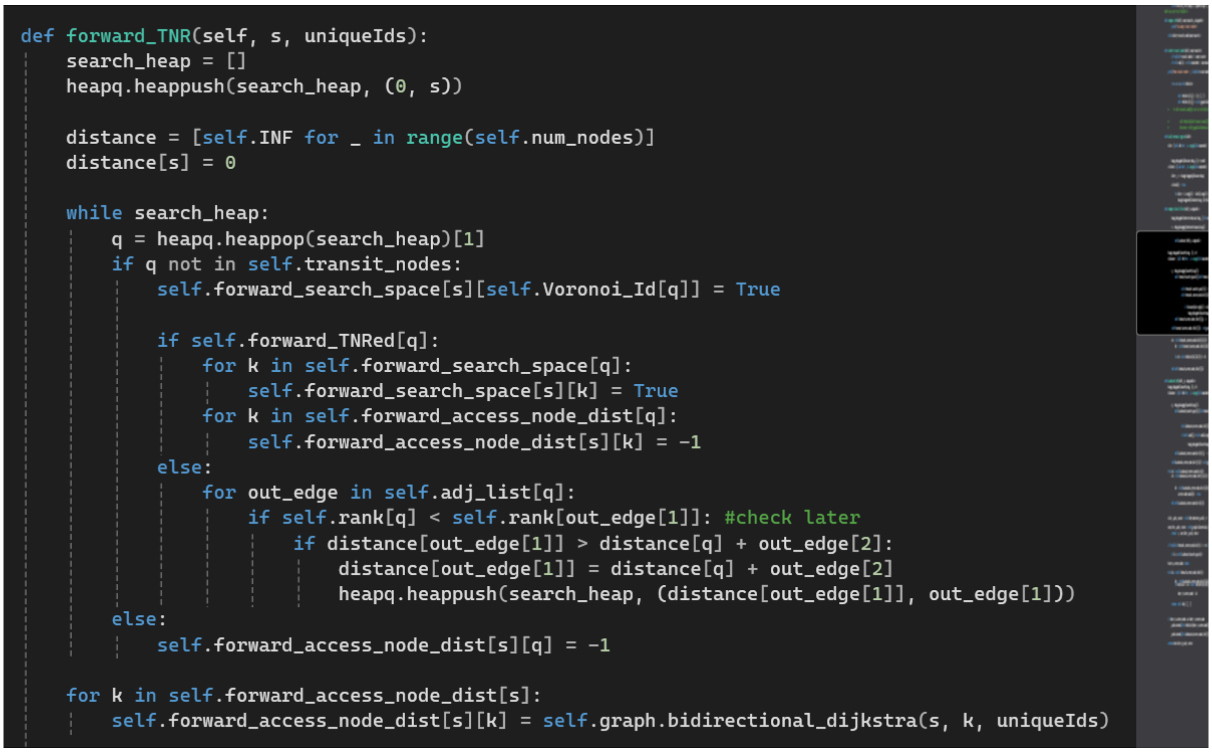
```python
def build_Voronoi_regions(self):
    distance_heap = []

    dist = [self.INF for _ in range(self.num_nodes)]

    for t_node in self.transit_nodes:
        dist[t_node] = 0
        self.Voronoi_Id[t_node] = t_node
        heapq.heappush(distance_heap, (0, t_node))

    visited = [False for _ in range(self.num_nodes)]

    while distance_heap:
        dist_v, v = heapq.heappop(distance_heap)
        if visited[v]:
            continue
        visited[v] = True

        for in_edge in self.trans_adj_list[v]:
            if dist_v + in_edge[2] < dist[in_edge[1]]:
                dist[in_edge[1]] = dist_v + in_edge[2]
                heapq.heappush(distance_heap, (dist[in_edge[1]], in_edge[1]))
                self.Voronoi_Id[in_edge[1]] = self.Voronoi_Id[v]
```

This function assigns every node in the graph to its nearest transit node, effectively creating Voronoi regions. Each node is associated with a "Voronoi ID," which is the nearest transit node.



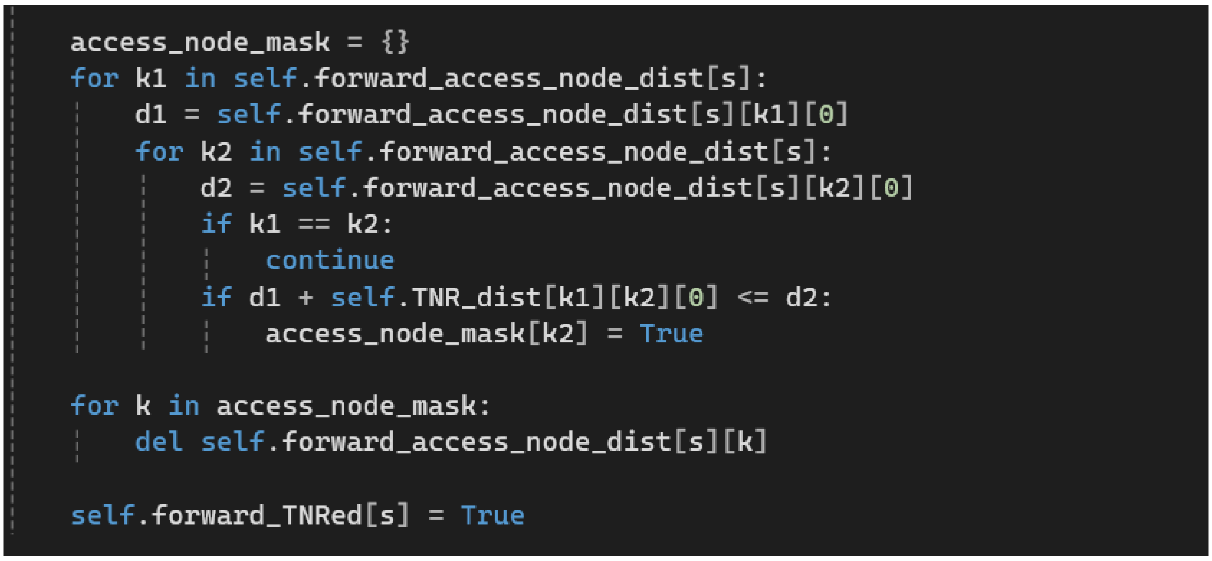
```python
def compute_local_filter(self, uniqueIds):
    contraction_max_heap = []
    for x in range(self.num_nodes):
        heapq.heappush(contraction_max_heap, (-self.rank[x], x))

    while contraction_max_heap:
        s = heapq.heappop(contraction_max_heap)[1]
        if not self.forward_TNRed[s]:
            self.forward_TNR(s, uniqueIds)
        if not self.backward_TNRed[s]:
            self.backward_TNR(s, uniqueIds)
```

When we compute local filter, we first push all nodes into a max heap with their respective rank (which is the importance of a node after CH preprocessing), but since Python priority queues are Min-Heap by nature, we must add the minus sign for the rank of the nodes



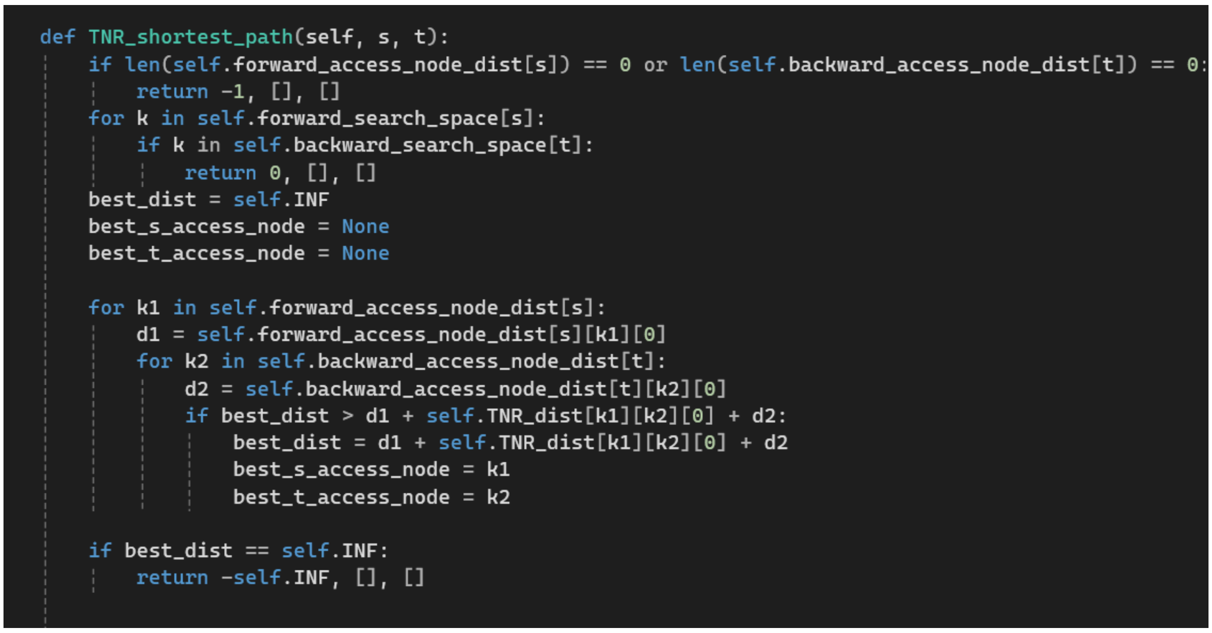
```python
def forward_TNR(self, s, uniqueIds):
    search_heap = []
    heapq.heappush(search_heap, (0, s))

    distance = [self.INF for _ in range(self.num_nodes)]
    distance[s] = 0

    while search_heap:
        q = heapq.heappop(search_heap)[1]
        if q not in self.transit_nodes:
            self.forward_search_space[s][self.Voronoi_Id[q]] = True

            if self.forward_TNRed[q]:
                for k in self.forward_search_space[q]:
                    self.forward_search_space[s][k] = True
                for k in self.forward_access_node_dist[q]:
                    self.forward_access_node_dist[s][k] = -1
            else:
                for out_edge in self.adj_list[q]:
                    if self.rank[q] < self.rank[out_edge[1]]: #check later
                        if distance[out_edge[1]] > distance[q] + out_edge[2]:
                            distance[out_edge[1]] = distance[q] + out_edge[2]
                            heapq.heappush(search_heap, (distance[out_edge[1]], out_edge[1]))
        else:
            self.forward_access_node_dist[s][q] = -1

    for k in self.forward_access_node_dist[s]:
        self.forward_access_node_dist[s][k] = self.graph.bidirectional_dijkstra(s, k, uniqueIds)
```

Searches from the node s to all reachable nodes until it encounters a transit node. It records the shortest path distances to transit nodes and updates the forward search space and access node distances. The same goes for the backward TNR function.



```python
access_node_mask = {}
for k1 in self.forward_access_node_dist[s]:
    d1 = self.forward_access_node_dist[s][k1][0]
    for k2 in self.forward_access_node_dist[s]:
        d2 = self.forward_access_node_dist[s][k2][0]
        if k1 == k2:
            continue
        if d1 + self.TNR_dist[k1][k2][0] <= d2:
            access_node_mask[k2] = True

for k in access_node_mask:
    del self.forward_access_node_dist[s][k]

self.forward_TNRed[s] = True
```

This function filters out redundant access nodes to minimize the number of transit nodes considered during a query.

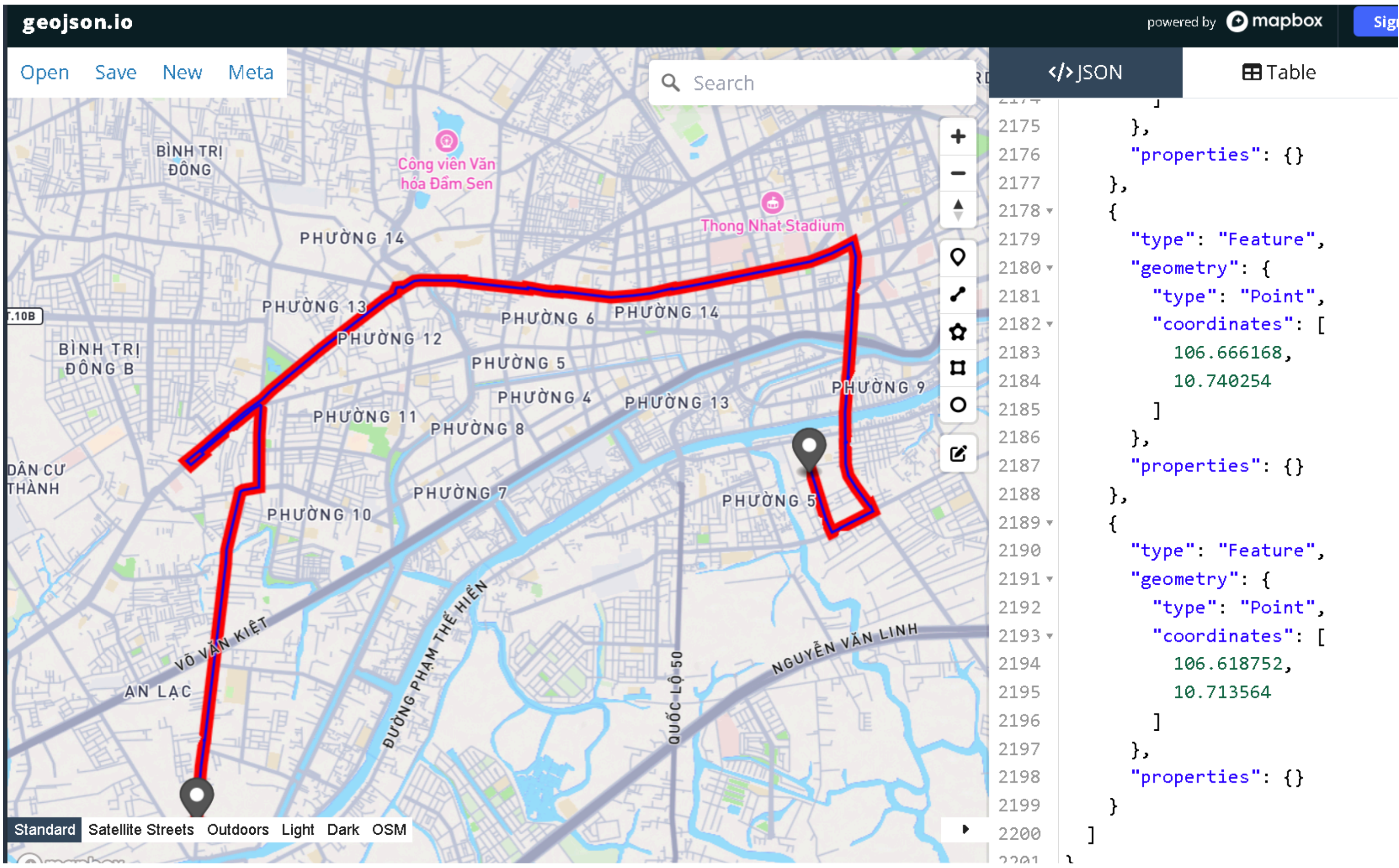


```
def TNR_shortest_path(self, s, t):
    if len(self.forward_access_node_dist[s]) == 0 or len(self.backward_access_node_dist[t]) == 0:
        return -1, [], []
    for k in self.forward_search_space[s]:
        if k in self.backward_search_space[t]:
            return 0, [], []
    best_dist = self.INF
    best_s_access_node = None
    best_t_access_node = None

    for k1 in self.forward_access_node_dist[s]:
        d1 = self.forward_access_node_dist[s][k1][0]
        for k2 in self.backward_access_node_dist[t]:
            d2 = self.backward_access_node_dist[t][k2][0]
            if best_dist > d1 + self.TNR_dist[k1][k2][0] + d2:
                best_dist = d1 + self.TNR_dist[k1][k2][0] + d2
                best_s_access_node = k1
                best_t_access_node = k2

    if best_dist == self.INF:
        return -self.INF, [], []
```

The checks if there is a common transit node in the forward and backward search spaces.
It then adds the precomputed distances from the forward search, the distance between transit nodes, and the distances from the backward search to compute the shortest path.

I also implemented the return path for the shortes TNR path.

## GeoJSON illustration



```
      },
      "properties": {}
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          106.666168,
          10.740254
        ]
      },
      "properties": {}
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          106.618752,
          10.713564
        ]
      },
      "properties": {}
    }
  ]
```

*Visualization of Bidirectional Dijkstra's on TNR and CH*

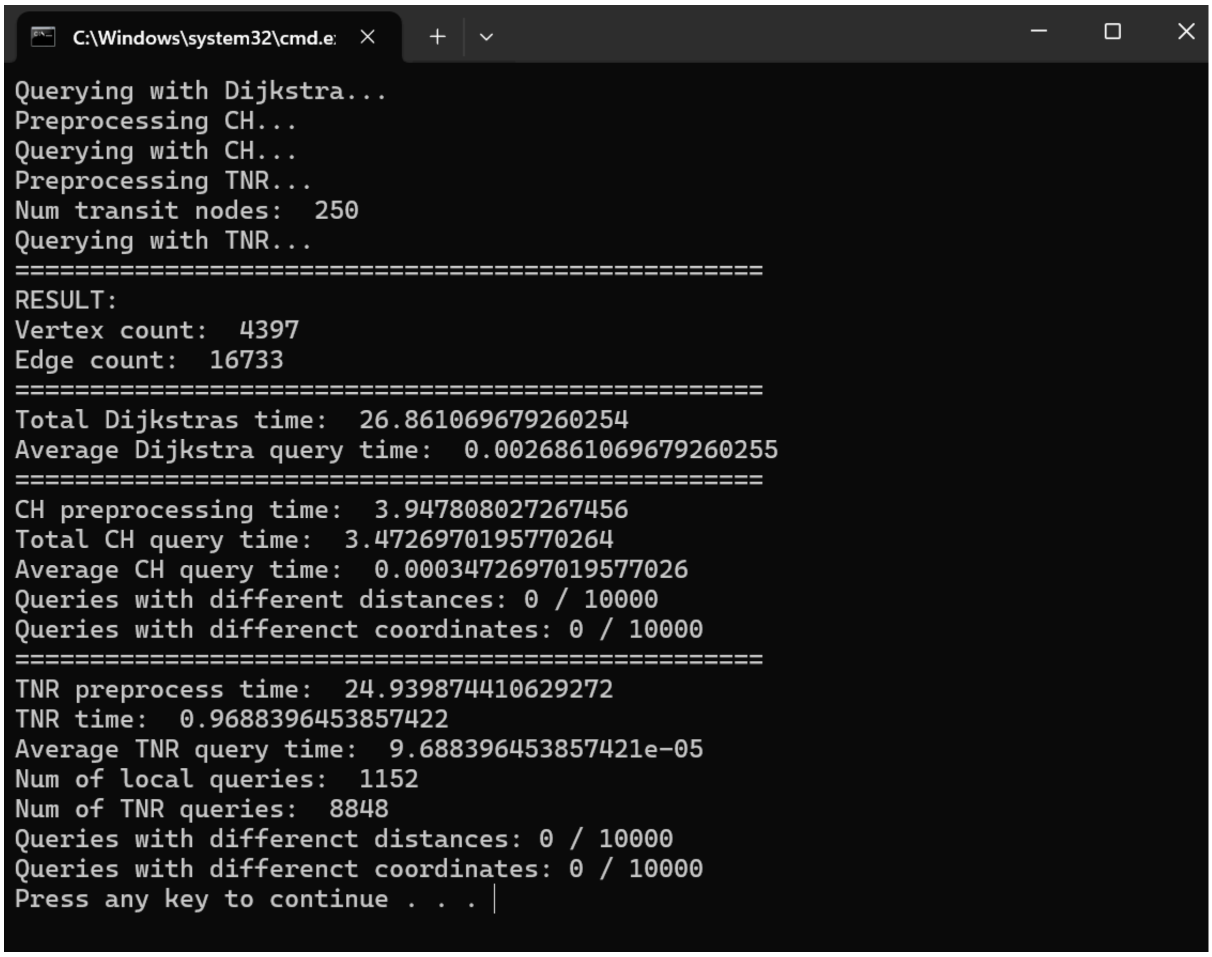
I just want to add this GeoJSON map to verify the correctness of my methods and to illustrate the path found by the algorithm. The thick red line is found by normal Dijkstra's and the thin black line is found by TNR query. As can be observed, both methods found the same best path.

The generated geojson data can be found in the *coordinates.geojson* file in the *data* folder.

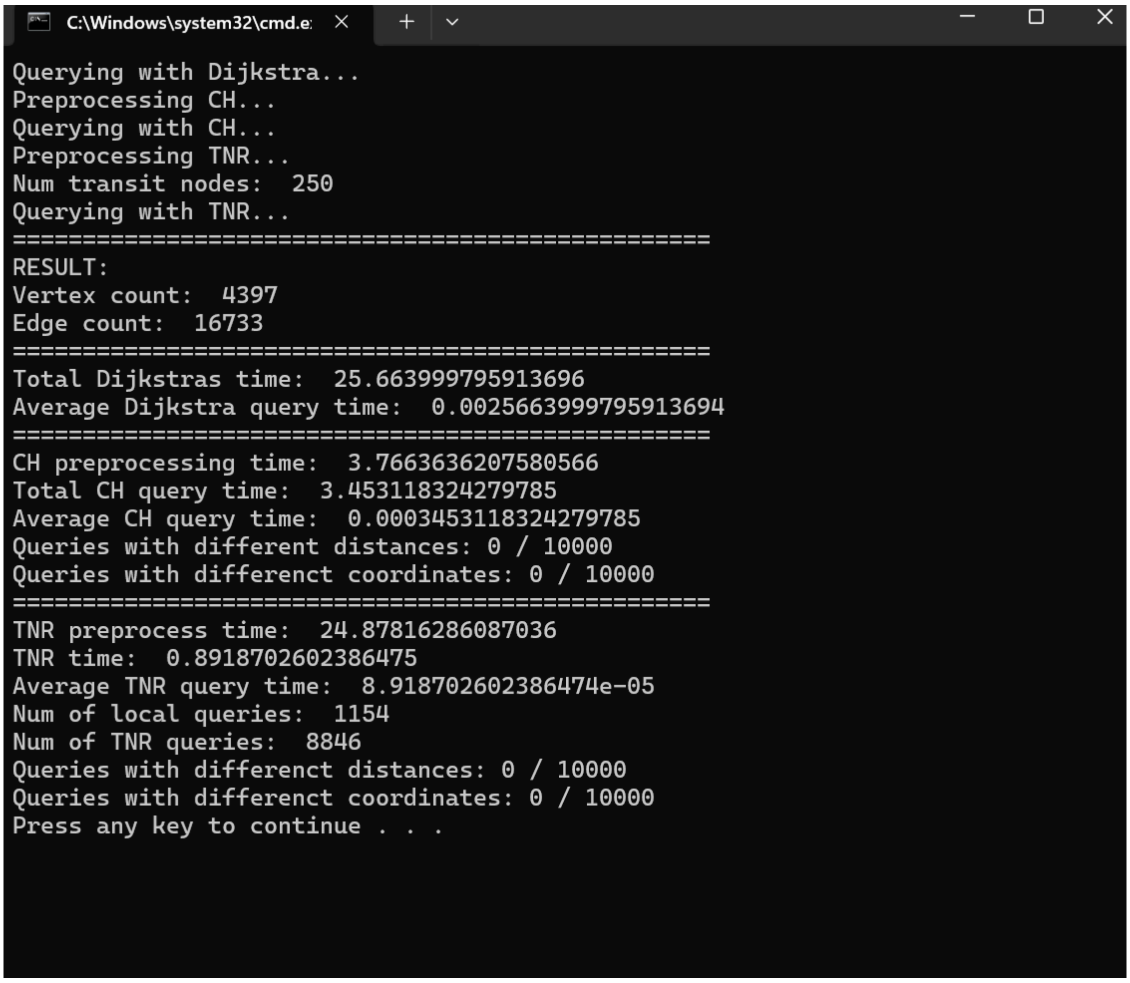
## Real-time performance

To check the real-time performance of my implementation, I ran 5 tests . For each test, I generated **10,000 random distinct pairs of nodes**. Then I ran a query with each pair using the 3 methods that I have built: Dijkstra's algorithm, Contraction Hierarchies, and Transit Node Routing.

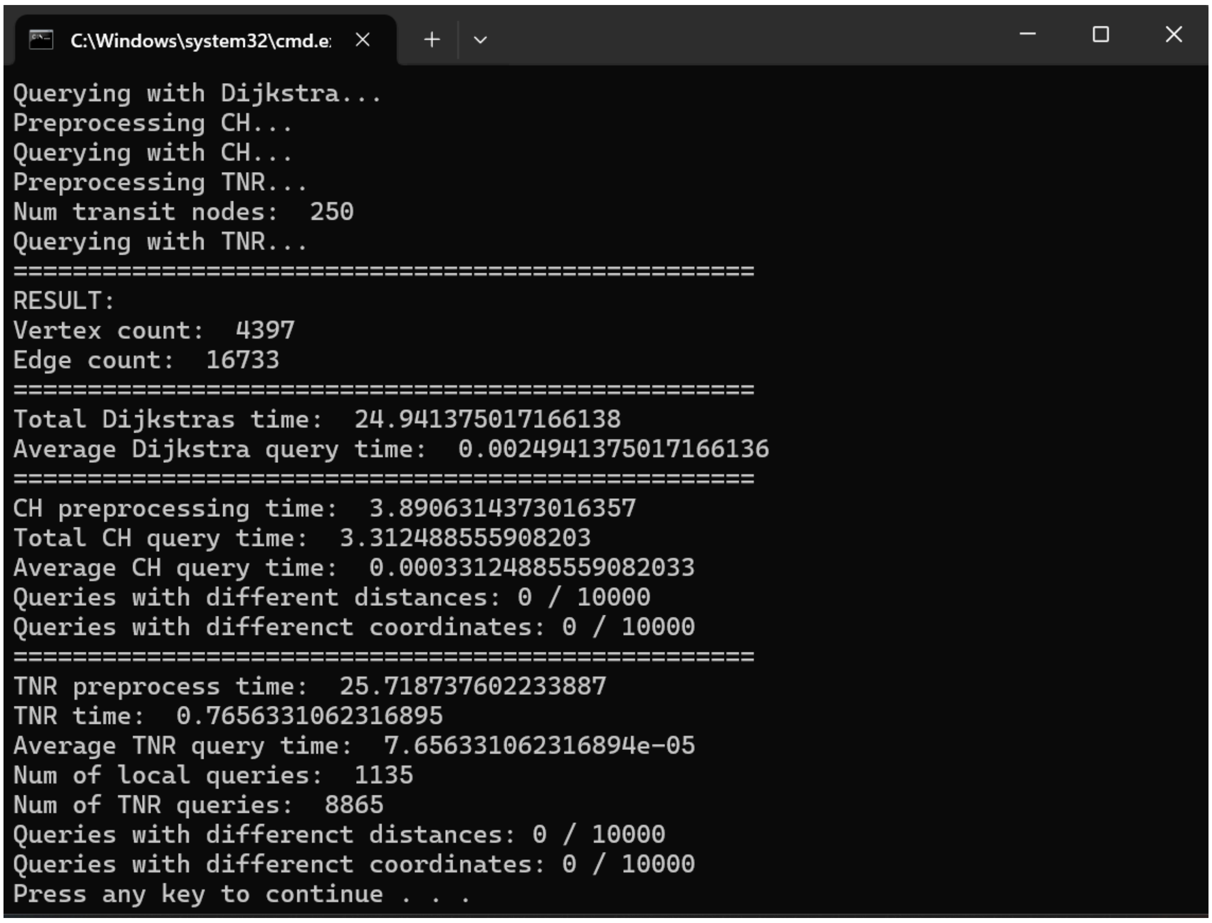
In the next few pages are 5 screenshots of the terminal that I took for the tests:



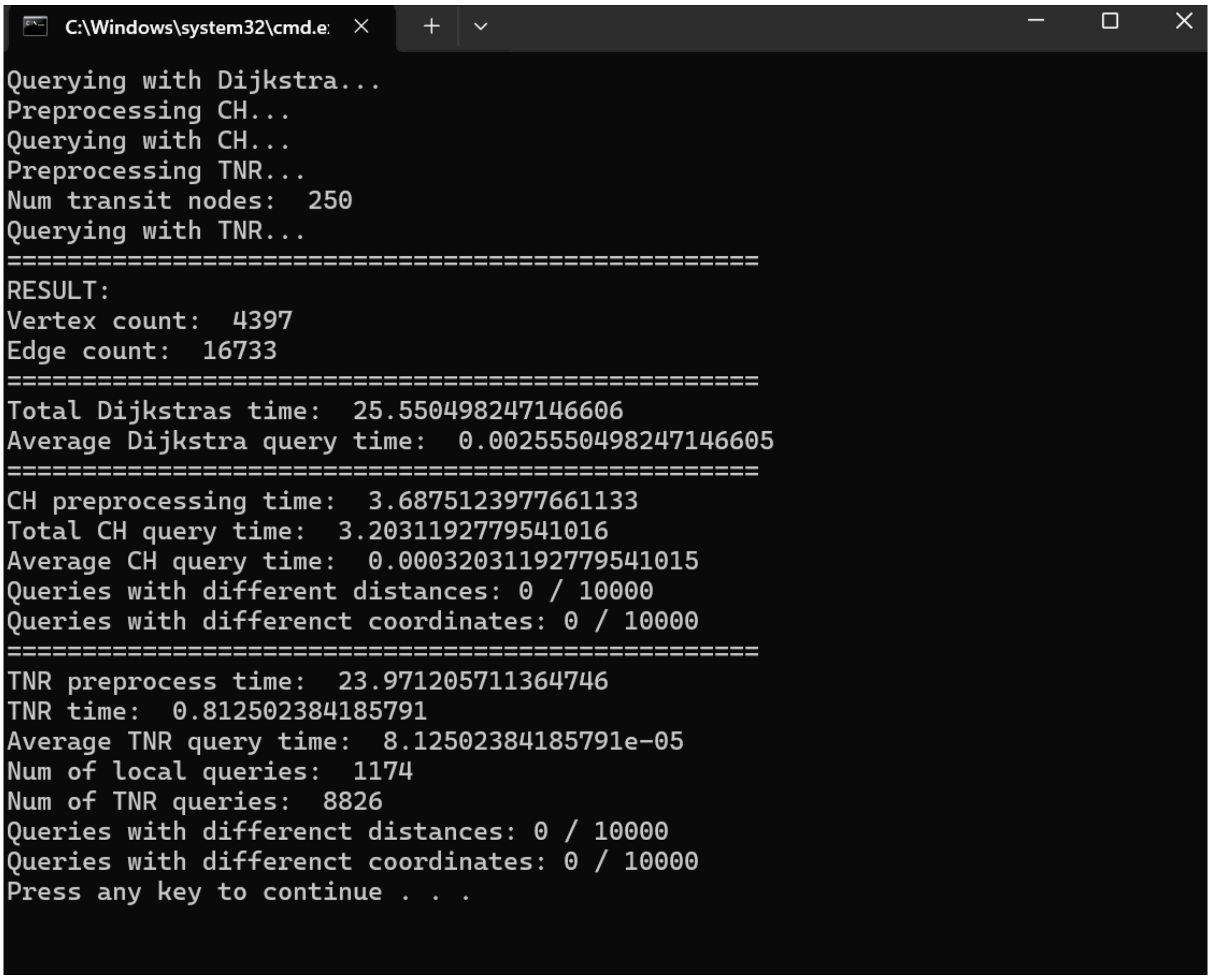
```
Querying with Dijkstra...
Preprocessing CH...
Querying with CH...
Preprocessing TNR...
Num transit nodes:  250
Querying with TNR...
==================================================
RESULT:
Vertex count:  4397
Edge count:  16733
==================================================
Total Dijkstras time:  26.861069679260254
Average Dijkstra query time:  0.0026861069679260255
==================================================
CH preprocessing time:  3.947808027267456
Total CH query time:  3.4726970195770264
Average CH query time:  0.00034726970195770264
Queries with different distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
==================================================
TNR preprocess time:  24.939874410629272
TNR time:  0.9688396453857422
Average TNR query time:  9.688396453857421e-05
Num of local queries:  1152
Num of TNR queries:  8848
Queries with differenct distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
Press any key to continue . . .
```



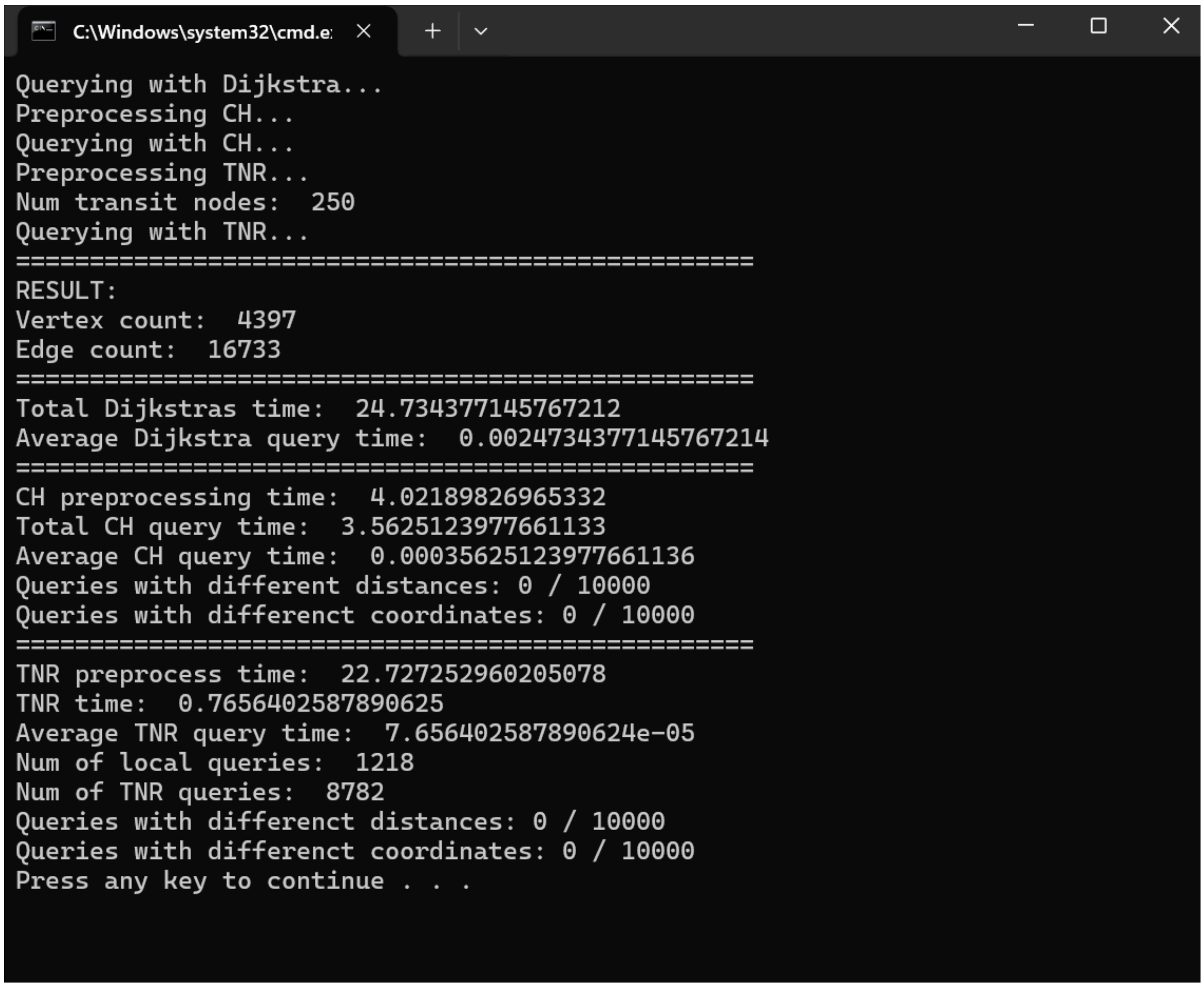
```
Querying with Dijkstra...
Preprocessing CH...
Querying with CH...
Preprocessing TNR...
Num transit nodes:  250
Querying with TNR...
==================================================
RESULT:
Vertex count:  4397
Edge count:  16733
==================================================
Total Dijkstras time:  25.663999795913696
Average Dijkstra query time:  0.0025663999795913694
==================================================
CH preprocessing time:  3.7663636207580566
Total CH query time:  3.453118324279785
Average CH query time:  0.0003453118324279785
Queries with different distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
==================================================
TNR preprocess time:  24.87816286087036
TNR time:  0.8918702602386475
Average TNR query time:  8.918702602386474e-05
Num of local queries:  1154
Num of TNR queries:  8846
Queries with differenct distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
Press any key to continue . . .
```



```
Querying with Dijkstra...
Preprocessing CH...
Querying with CH...
Preprocessing TNR...
Num transit nodes:  250
Querying with TNR...
==================================================
RESULT:
Vertex count:  4397
Edge count:  16733
==================================================
Total Dijkstras time:  24.941375017166138
Average Dijkstra query time:  0.0024941375017166136
==================================================
CH preprocessing time:  3.8906314373016357
Total CH query time:  3.312488555908203
Average CH query time:  0.00033124885559082033
Queries with different distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
==================================================
TNR preprocess time:  25.718737602233887
TNR time:  0.7656331062316895
Average TNR query time:  7.656331062316894e-05
Num of local queries:  1135
Num of TNR queries:  8865
Queries with differenct distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
Press any key to continue . . .
```



```
Querying with Dijkstra...
Preprocessing CH...
Querying with CH...
Preprocessing TNR...
Num transit nodes:  250
Querying with TNR...
==================================================
RESULT:
Vertex count:  4397
Edge count:  16733
==================================================
Total Dijkstras time:  25.550498247146606
Average Dijkstra query time:  0.0025550498247146605
==================================================
CH preprocessing time:  3.6875123977661133
Total CH query time:  3.2031192779541016
Average CH query time:  0.00032031192779541015
Queries with different distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
==================================================
TNR preprocess time:  23.971205711364746
TNR time:  0.812502384185791
Average TNR query time:  8.12502384185791e-05
Num of local queries:  1174
Num of TNR queries:  8826
Queries with differenct distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
Press any key to continue . . .
```



```
Querying with Dijkstra...
Preprocessing CH...
Querying with CH...
Preprocessing TNR...
Num transit nodes:  250
Querying with TNR...
==================================================
RESULT:
Vertex count:  4397
Edge count:  16733
==================================================
Total Dijkstras time:  24.734377145767212
Average Dijkstra query time:  0.0024734377145767214
==================================================
CH preprocessing time:  4.02189826965332
Total CH query time:  3.5625123977661133
Average CH query time:  0.00035625123977661136
Queries with different distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
==================================================
TNR preprocess time:  22.727252960205078
TNR time:  0.7656402587890625
Average TNR query time:  7.656402587890624e-05
Num of local queries:  1218
Num of TNR queries:  8782
Queries with differenct distances: 0 / 10000
Queries with differenct coordinates: 0 / 10000
Press any key to continue . . .
```

*Image. 5 tests for query time*

Notice that the number of local queries are about 1,000 out of 10,000 queries we performed, which is good since most of the queries will make use of the transit node paths we already calculated.

I also compared the distances and path coordinates that CH and TNR gave to Dijkstra's to verify the correctness of our methods. (As you can see out of 10,000 queries, none gave a different distance or coordinate for both speed-up methods).

The tests allow me to calculate the average runtime and preprocessing time of 1 query for each method with acceptable margin of error. I used a calculator to get the average calculations in the table below:

| | **Average runtime (ms)** | **Average preprocessing time (s)** |
|---|---|---|
| **Dijkstra's algorithm** | 2,02 ms | 0 s |
| **Contraction Hierarchies** | 0,27 ms | 3,07 s |
| **Transit Node Routing** | 0,065 ms | 19.45 s |

*Table. Comparison between 3 methods on average runtime and average preprocessing time*

As can be seen from the table, Contraction Hierarchies already performs pretty well when it reduces the query time to 13% of that of normal Dijkstra's, with a trivial preprocessing time of just over 3 seconds.

Meanwhile, the preprocessing time for TNR is quite large, almost 20 seconds, but this extra time is inconsequential when we acknowledge the fact that our query time is now the much-desired 3% of that normal Dijkstra's (0.065 millisecond).

We can see that TNR and CH both worked as intended and brought in the results we wanted.

## References

Transit Node Routing Reconsidered (arxiv.org)
Contraction Hierarchies Guide (jlazarsfeld.github.io)
Routing Faster Than Dijkstra Thanks to Contraction Hierarchies | by Jean-Sébastien Gonsette | Medium