VIETNAM NATIONAL UNIVERSITY, HCMC - UNIVERSITY OF SCIENCE

FACULTY OF INFORMATICS TECHNOLOGY
ADVANCED PROGRAM IN COMPUTER SCIENCE

DATA STRUCTURES - CS163

Technical Report

Task 2.5

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

§1.1 Problem Statement

Given a bus transportation network in Ho Chi Minh City, the task is to find out the stress centrality of all bus stops in the network. The shortest path between any two bus stops is determined by considering the number of transfers between bus routes, travel time, and departure and arrival times respectively.

§1.2 Conceptual Approach

The general idea is using brute force to calculate the shortest paths between all pairs of bus stops. For each bus stop, the shortest paths to all other stops are determined using a modified version of Dijkstra's algorithm (details are provided in the Implementation section). The reason for this idea is is to make the timestamp of the shortest path be consecutive. To enhance performance, the process is parallelized using multiprocessing.

§2 Implementation

§2.1 Graph Construction

§2.1.1 Conceptual Idea

In the given bus transportation network of Ho Chi Minh City, there are multiple routes a variety of bus stops. Each bus is identified by a unique *stop_ id* and is associated with a list of timestamps, reflecting the different times buses (or vehicles, as indicated by the *vehicle_ number* in the provided .csv files) pass through the stop. A single bus stop can be part of many routes or different routes intersect at the same stop.

Each node is characterized by following attributes (*route_ id, var_ id, stop_ id, time_ stamp, node_ type, latx, lngy*), all of which are extracted from the provided .csv files. Consequently, a bus stop (stop_id) correspond to many nodes, each representing different routes and timestamps.

Each edge is established between two nodes with weight represented as a tuple of (number of transfers, *time_ diff*). While *time_ diff* is derived directly from the file, number of transfer, which is more significant, depends on type of edge. There are two provided files: `type12.csv` and `type34.csv`

- `type12.csv` contains 500,000 edges where two nodes of the edge have the same routeid and varid, so no transfer is made ⇒ number of transfers = 0
- `type34.csv` contains 500,000,000 edges where nodes have different routeid and varid, so there is a route transfer ⇒ number of transfers = 1

Based on these information, the graph is contructed.

§2.1.2 Implementation Code

To handle the graph, I use **NetworkX** library and define all functions within `Graph` class in the `graph.py` file.

For the contruction function, I initialize three components:

- The graph `G` is defined as `DiGraph()` since all edges are directed and weighted.
- A dict of set named `stop_id_to_node` is defined to store all nodes corresponding to each stop_id.
- A set named `node_ids` stores all unique stop_ids.

Below is the `__init__` function:

```python
def __init__(self):
    self.G = nx.DiGraph()
    self.stop_id_to_node = defaultdict(set)
    self.node_ids = set()
```

To read the provided .csv files, I use **csv** library, processing the files line by line to add edges and nodes to the graph. Below is the `build_bus_graph` function:

```python
def build_bus_graph(self):
    print("Starting to build the bus graph...")
    csv_files = ['type12.csv', 'type34.csv']
    for csv_file_path in csv_files:
        print(f"Processing file: {csv_file_path}")

        with open(csv_file_path, mode='r') as file:
            total_rows = sum(1 for row in file)

        with open(csv_file_path, mode='r') as file:
            reader = csv.reader(file)

            for row in tqdm(reader, total=total_rows, desc=f"Building Graph from
            ↪ {csv_file_path}", unit="row"):
                stop_id1, route_id1, var_id1 = map(int, row[:3])
                stop_id2, route_id2, var_id2 = map(int, row[4:7])
                timestamp1, timestamp2, time_diff = map(lambda x: int(float(x)),
                ↪ (row[3], row[7], row[8]))
                latx1, lngy1, latx2, lngy2 = map(float, row[9:13])
                node_type1, node_type2, node_pos1,node_pos2,edge_pos,edge_type =
                ↪ map(int, row[15:21])

                node1 = (stop_id1, route_id1, var_id1, timestamp1, node_type1,
                ↪ latx1, lngy1)
                node2 = (stop_id2, route_id2, var_id2, timestamp2, node_type2,
                ↪ latx2, lngy2)

                self.stop_id_to_node[stop_id1].add(node1)
                self.stop_id_to_node[stop_id2].add(node2)

                self.node_ids.add(stop_id1)
                self.node_ids.add(stop_id2)

```

```
                    num_transfers = 0 if edge_type in {1, 2} else 1
                    weight = (num_transfers, time_diff)
                    self.G.add_edge(node1, node2, weight=weight)

        print(f"Graph building completed. {len(self.G.nodes)} nodes and
    ↪   {len(self.G.edges)} edges have been added.")
```

§2.1.3 Complexity Analysis and Result

Time Complexity: $O(n)$ with n is the number of rows in all CSV files combined. This accounts for the time required to read and process each row in the CSV files.

Space Complexity of the Graph: $O(V + E)$, where V is the number of nodes and E is the number of edges. This includes space for storing nodes, edges, and edge weights.

Results: After executing the function, the resulting graph contains 768,211 nodes and 5,497,345 edges. The process completed in approximately 1 minute and 30 seconds.

§2.2 Dijkstra's algorithm

§2.2.1 Conceptual Idea

From a given source bus stop or stop_id, the task is to find the shortest paths to all other stop_ids. Since each stop_id corresponds to multiple nodes, a standard implementation of Dijkstra's algorithm, which operates from a single node, requires adaptation. Specifically, the algorithm must be executed multiple times, once for each node associated with the source stop_id.

For each run, Dijkstra's algorithm starts from one of the source nodes. During the process, whenever it reaches a node with a target stop_id that has not been previously visited, the shortest path from the source stop_id to that target stop_id is updated. This method ensures that all potential paths are considered, and the shortest path is accurately determined for each source-target pair.

However, running Dijkstra's algorithm individually for each source node is computationally expensive and can significantly slow down the process. To optimize this, I perform a single run of Dijkstra's algorithm from all source nodes simultaneously. During this run, the shortest path to each target stop_id is saved as soon as a node with that target stop_id is reached. This approach reduces the number of algorithm executions and speeds up the overall process.

§2.2.2 Implementation Code

Firstly, I initialize the least cost from all nodes associated with the source stop_id is (0, 0), (number of transfers, time_diff). Each element in the priority queue includes the number of transfers, time_diff, the source node, and the current node.

Additionally, I maintain two dictionaries: `target_node` and `source_node` . The `target_node` dictionary stores the target node corresponding to the shortest path from the source stop_id to the target stop_id. The `source_node` dictionary records the source node for each target node.

Below is the source code of the modified Dijkstra's algorithm:

```python
def dijkstra(self, source_id):
    target_node = {}
    source_node = {}
    cost = {node: (float('inf'), float('inf')) for node in self.G.nodes}
    predecessors = {node: None for node in self.G.nodes}
    priority_queue = []

    # Initialize cost for all source nodes
    for source in self.stop_id_to_node[source_id]:
        cost[source] = (0, 0)  # (transfers, time)
        heapq.heappush(priority_queue, (0, 0, source, source))  # (transfers,
    ↪ time, source, target)

    while priority_queue:
        current_transfers, current_time, source, u =
    ↪ heapq.heappop(priority_queue)

        if (current_transfers, current_time) > cost[u]:
            continue
        if u[0] != source_id:
            source_node[u] = source
            node_id = u[0]
            if node_id not in target_node:
                target_node[node_id] = u
            else:
                target = target_node[node_id]
                if cost[target] == cost[u]:
                    current = (source_node[u][3], u[3])
                    best = (source_node[target][3], target[3])
                    if current < best:
                        target_node[node_id] = u

        for v, edge_data in self.G[u].items():
            new_transfers = current_transfers + edge_data['weight'][0]
            new_time = current_time + edge_data['weight'][1]

            if (new_transfers, new_time) < cost[v]:
                cost[v] = (new_transfers, new_time)
                predecessors[v] = u
                heapq.heappush(priority_queue, (new_transfers, new_time, source,
        ↪ v))

    return predecessors, target_node
```

§2.2.3 Complexity Analysis and Result

Time Complexity: $O((V + E) \log V)$

Space Complexity: $O(V + E)$

This analysis accounts for the typical behavior of Dijkstra's algorithm when implemented with a priority queue (heap) and supports the efficient computation of shortest paths in weighted directed graphs.

§2.3 Top K Most Important Bus Stops

§2.3.1 Conceptual Idea

To determine the shortest paths between all pairs of bus stops, I execute Dijkstra's algorithm starting from each source stop_id. After completing each execution of Dijkstra's algorithm, I trace back the shortest paths from the target stop_ids to the source stop_id. During this trace-back process, I accumulate the frequency of visits to each stop_id along the path until reaching the original source stop_id.

After completing the path tracing and frequency accumulation, I sort the frequencies of all stop_ids in descending order. From this sorted list, I extract the top k bus stops with the highest frequencies and save stop_ids and corresponding frequencies to .json file.

§2.3.2 Implementation Code

```python
def get_k_important_stops(self, k=10):
    print("Calculating vertex importance...")
    count = {v: 0 for v in self.node_ids}
    for source_id in tqdm(self.node_ids, desc="Processing nodes", unit='node'):
        predecessors, target_node = self.dijkstra(source_id)
        for target_id in self.node_ids:
            print(source_id, target_id)
            if source_id == target_id or target_id not in target_node:
                continue
            current_node = target_id
            count[current_node] += 1
            while current_node != source_id:
                print(current_node)
                current_node = predecessors[current_node]
                count[current_node] += 1

    sorted_count = dict(sorted(count.items(), key=lambda item: item[1],
↪   reverse=True))
    top_stops = list(sorted_count.keys())[:k]
    print(f"Top {k} most important bus stops found.")
    with open('important_bus_stops.json', 'w') as file:
        json.dump(sorted_count, file, indent=4)
    return top_stops
```

§2.3.3 Complexity Analysis and Result

Time Complexity: $O(S \cdot ((V+E)\log V + V^2))$. This is because for each source stop_id, the algorithm runs Dijkstra's algorithm once and performs path tracing for all target stop_ids once.

- **Time Complexity for each Dijkstra's Call:** $O((V+E)\log V)$.
- **Time Complexity for Path Tracing:** Since this path tracing is done for each target node, the time complexity is $O(V^2)$, assuming each path traversal operation is $O(V)$.

Space Complexity: $O(V)$ because we have a dict store frequencies of all stop_ids.

With 4162 stop_ids, the function takes roughly 10 hours to execute.

§3 Optimization - Multiprocess

§3.1 Multiprocessing vs. Multithreading

- **Multiprocessing:** Runs multiple processes on different CPU cores, allowing true parallelism. Each process has its own memory space, making it ideal for CPU-bound tasks, avoiding Python's Global Interpreter Lock (GIL). It has higher memory overhead due to separate memory spaces.
- **Multithreading:** Runs multiple threads within the same process, sharing memory. Suitable for I/O-bound tasks but limited by the GIL in Python for CPU-bound tasks, resulting in reduced performance.

Why Multiprocessing Was Chosen: Multiprocessing was chosen because the stress centrality calculation is CPU-bound, and multiprocessing allows parallel execution without GIL interference, significantly reducing runtime.

§3.2 Conceptual Idea

Observing the process of identifying the top k most important bus stops, I noted that the memory usage peaks at only 9 GB out of the available 16 GB of RAM. To optimize performance, I divided the 4162 stop_ids into smaller chunks and processed them concurrently.

My laptop is equipped with a **12th Gen Intel(R) Core(TM) i7-1255U** processor with 10 physical cores, allowing for a maximum of 10 simultaneous processes. Based on experimental results, I found that the most efficient configuration with 16 GB of RAM involves running 3 processes in parallel.

§3.3 Implementation Code

```python
def get_k_important_stops_parallel(self, num_processes=10, k=10):
    # Set environment variable to control Intel's thread affinity
    os.environ["KMP_AFFINITY"] = "granularity=fine,compact,1,0"
    self.node_ids = list(self.node_ids)

    with multiprocessing.Pool(processes=num_processes) as pool:
        results = pool.imap(self.process_source_chunk, np.array_split(self.node_ids, num_processes))

    count = defaultdict(int)
    for chunk_counts in results:
        for node, node_count in chunk_counts.items():
            count[node] += node_count

    sorted_count = dict(sorted(count.items(), key=lambda item: item[1], reverse=True))
    top_stops = list(sorted_count.keys())[:k]
    print(f"Top {k} most important bus stops found.")
    with open('important_bus_stops.json', 'w') as file:
        json.dump(sorted_count, file, indent=4)
    return top_stops

def process_source_chunk(self, source_chunk):
    chunk_count = defaultdict(int)
    for source_id in source_chunk:
        predecessors, target_node = self.dijkstra(source_id)

        for target_id in self.node_ids:
            if source_id == target_id or target_id not in target_node:
                continue

            current_node = target_node[target_id]
            chunk_count[current_node[0]] += 1

            while current_node[0] != source_id:
                prev_node = current_node
                current_node = predecessors[current_node]
                if current_node[0] != prev_node[0]:
                    chunk_count[current_node[0]] += 1
    return chunk_count
```

§3.4 Result

By running multiple processes simultaneously, while maintaining the same computational complexity, the time required to identify the top k most important bus stops was reduced to just 4 hours.

§4 Reference

Tất tần tật về Multi-processing trong Python

Dijkstra's Shortest Path Algorithm using *priority_queue* of STL

This is the link to my Github Repository.