# Dijkstra's Algorithm for Shortest Path Calculation in C

Sam Serbouti serbouti@kth.se

October 27, 2024

This assignment completes the previous work of a depth-first search for finding the shortest path between cities. This time, I'll be using Dijkstra's algorithm. DFS had limits (for large graphs and re-exploring the same cities multiple times). Dijkstra tracks the shortest known paths to cities and widens the search step by step from the closest city.

## 1 Data structure

### The graph

The graph is built with an adjacency list, where each city has a list of connections to its neighboring cities [1]. However, now I need my `struc city` to have a unique ID

### THe path

In Dijkstra's algorithm, a path is represented as:

```
typedef struct path_entry {
    city *city_data;
    int total_time;
    city *prev_city;
} path_entry;
```

This allows Dijkstra to maintain both the shortest path to a city **and** the ability to reconstruct the path by following the prev_city pointers.

### The priority queue

A priority queue is used to always widen the shortest known path. The priority queue orders path entries by the cumulative travel time (total_time) to ensure that the next city to be processed is always the closest one.

[1] This was reused from the previous implementation

```c
typedef struct priority_queue {
    path_entry **data;
    int size; //initially 0
    int capacity; //initially the number of cities
} priority_queue;
```

## 2 Using Dijkstra's algorithm

The goal of Dijkstra's algorithm is to incrementally explore the shortest path from the source city to other cities.

A `done` array stores the current shortest path(s) for each city. When a city is encountered for the first time, it gets added to this array, so that it is not revisited. The algorithm continues until the target city is reached:

- adds the source city to the priority queue [2] with a travel time of 0
- the city with the shortest travel time is removed from the queueif it's the target, the algorithm terminates and the path is reconstructed; if it isn't, the city's neighbours are searched, and if a shorter path is found. their sumed travel time is updated
- if a shorter path is discovered a new path is built and the neighbor is added to the priority queue but the previous city in the path is **also** updated because I'll need to write the path in the end!

### Implementation

```c
void dijkstra(codes *cities, city *source, city *target) {
    int n=cities->city_count;
    path_entry **done = (path_entry **)calloc(n,sizeof(path_entry *));
    for (int i = 0; i < n; i++) {done[i] = NULL;}
    priority_queue *pq = create_priority_queue(n);

    path_entry *start =(path_entry *)malloc(sizeof(path_entry));
    start->city_data =source;
    start->total_time =0;
    start->prev_city =NULL;
    push(pq, start); //push starting city to the queue

    while (pq->size>0//until there's nothing left in the queue
        path_entry *current = pop_min(pq);
        if (current->city_data==target) {//reached target :))}

        int current_id =current->city_data->id;
        if (done[current_id]==NULL) {
            done[current_id]=current;
            connection *conn=current->city_data->connections;
            while (conn!=NULL){//expolre neighbours
```

[2] the pq ensuring that the city with the shortest travel time is always processed next

```
                city *neighbor=conn->destination;
                int new_time =current->total_time+conn->time;
                if(done[neighbor->id]==NULL||new_time<done[neighbor->id]->total_time) {
                //need to update shortest path for neighbour?
                    path_entry *neighbor_entry =(path_entry*)malloc(sizeof(path_entry));
                    neighbor_entry->city_data=neighbor;
                    neighbor_entry->total_time= new_time;
                    neighbor_entry->prev_city= current->city_data;
                    push(pq, neighbor_entry);
                }
                conn=conn->next;
            }
        }
    }
    //clean with free()
}
```

# 3 Benchmarks

| destination | execution time in ns |
|---|---|
| Stockholm | 149930 |
| Göteborg | 160687 |
| Copenhagen | 152365 |
| Oslo | 165078 |
| Helsinki | 154922 |
| Berlin | 178140 |
| Amsterdam | 117756 |
| Paris | 148951 |
| Bruxelles | 147165 |
| Vienna | 150159 |
| Zurich | 202881 |
| London | 125311 |

**Table 1:** Malmö to ...