Dijkstra to our rescue in C

Ying Pei Lin

Fall 2024

The City structure

The city is minimum unit of the map. We store the city in the hash table, a dynamic array in map struct. The name of the city is used as the key to access the city in the map struct.

The id provide another way to access the city, we assign the id to the city when we add the city to the map. The value of id will be the same as the size of the cities array in the map struct at the time of adding the city to the map. This ensures that the ids are continuous and there are no duplicates. We use the id to access the city in the path struct.

The size and capacity are used to keep track of the number of connections and the capacity of the connections array.

```
typedef struct City {
  char *name;
  int id;
  Connection *connections; // Dynamic array of connections
  int size;                // Number of connections
  int capacity;            // Capacity of connections
} City;
```

The Connection structures

The connection struct is stored in the connections array in the city struct. It stores the destination city and the time taken to travel from the source city to the destination city.

```
typedef struct Connection {
  City *dst;
  int time;
} Connection;
```

The Map structures

The map struct is used to store the cities in the map. The cities array is a dynamic array of cities, which is implemented as a hash table, using the

name of the city as the key to access the city in the map struct. The size and capacity are used to keep track of the number of cities and the capacity of the cities array.

```
typedef struct Map {
  City *cities;             // Dynamic array of cities, Hash table
  int size;                 // Number of cities
  int capacity;             // Capacity of cities
} Map;
```

The Path structures

The path array is used to store the path from the source city to the destination city. Each path struct stores a city, the total time taken to travel from the source city to the current city. The prev pointer is used to store the previous city in the path.

If we keep the path to be the shortest path at each stage by only update the path when we find a shorter path, then after we reach the destination city, the path array will store the shortest path from the source city to the destination city.

```
typedef struct Path {
  City *city;
  int total_time;
  struct Path *prev;
} Path;
```

The PriorityQueue structures

The priority queue stores the paths in the order of the total time taken to travel from the source city to the current city. We enqueue the paths when we find a shorter path from the current city and pop the path with the shortest time when we finish exploring the current city.

```
typedef struct PriorityQueue {
  Path **paths;
  int size;
  int capacity;
} PriorityQueue;
```

The Dijkstra algorithm

While exploring the neighbors of the current city, we only add the city to the queue if the city is the clostest unvisited city to the source city.

The way we check if the city is the clostest city is by checking if the time from the beginning to its neighbor is shorter than the time we have already

stored in the path plus the time from the current city to the neighbor. The algorithm will keep running until we reach the destination city or every city has been visited.

Since we always ensure that the path is the shortest path at each stage, we can be sure that the path we find at the end will also be the shortest path. It will be more efficient to BFS the graph since we won't check the same city multiple times. The code below shows the implementation of the Dijkstra algorithm.

```
Path *dijsktra(Map *map, City *from, City *to) {
  PriorityQueue *pq = new_priority_queue();
  Path **done = init_done(map);

  done[from->id]->city = from;
  done[from->id]->total_time = 0;
  push(pq, done[from->id]);

  while(pq->size > 0) {
    Path *p = pop(pq);
    City *c = p->city;
    if(c == to) return p; // Found the destination

    // Check all connections
    done[c->id] = p;
    for(int i = 0; i < c->size; i++) {
      Connection *nxt = &c->connections[i];

      // Skip it, if the city is already in the path,
      // or if the time is not better.
      if(done[nxt->dst->id]->city != NULL ||
          (done[nxt->dst->id]->total_time >= p->total_time + nxt->time)) {
        continue;
      }

      // Add the city to the queue
      Path *new = (Path*)malloc(sizeof(Path));
      new->city = nxt->dst;
      new->total_time = p->total_time + nxt->time;
      new->prev = p;
      push(pq, new);
    }
  }
  return NULL;
}
```

Benchmarks

First, we compare the compute time of the Dijkstra algorithm with BFS. For the same route from Malmö to Kiruna, Dijkstra is much faster than BFS, as we can see from the Table 1

Algorithm	**Compute Time (ns)**
BFS	569736000
Dijkstra	108900

Table 1: Compute Time Comparison from Malmö to Kiruna

Next, we compare the compute time of the Dijkstra algorithm for different number of nodes in the done array. We select twelve cities from the data and calculate the shortest path from Malmö to each city. The results are shown in Table 2 and Figure 1.

Nodes in Done Array	Destination City	Compute Time (ns)
115	Kiruna	119900
34	Stockholm	9420
18	Göteborg	4660
42	Uppsala	8920
33	Örebro	5630
40	Karlstad	8460
64	Sundsvall	22600
82	Umeå	43700
98	Luleå	76990
50	Gävle	10840
22	Linköping	4490
5	Helsingborg	3550

Table 2: Benchmark Results for Shortest Path from Malmö

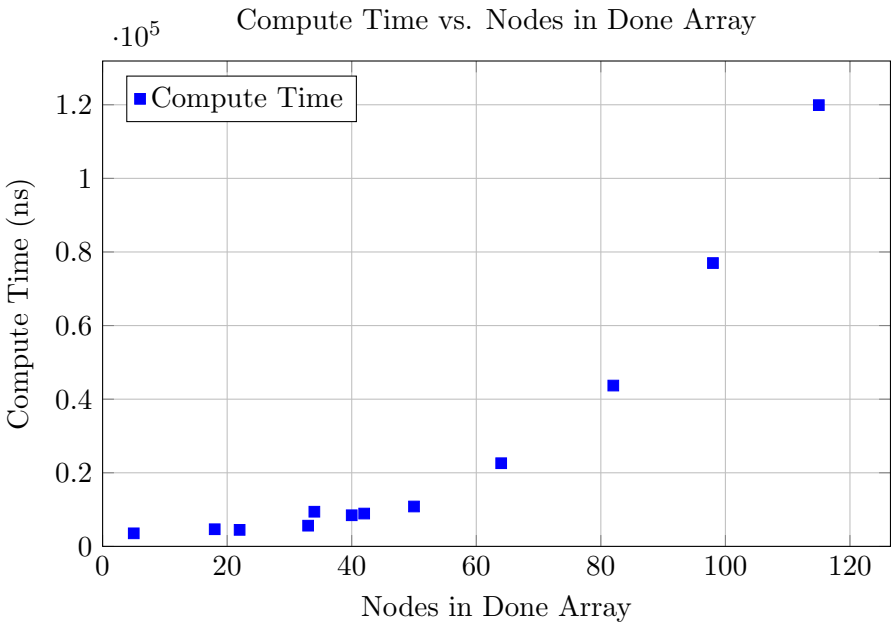

Figure 1: Compute Time vs. Nodes in Done Array for Shortest Paths from Malmö

Assume that there are n cities in the map, and the average number of connections for each city is m. To reason about the time complexity of the Dijkstra algorithm, we first look at the time complexity of the priority queue. The time complexity of the priority queue is $O(n)$ for pop operations and $O(1)$ for push operations (I didn't implement the priority queue with a heap, otherwise the time complexity for both push and pop operations would be $O(log(n))$).

In each iteration, we pop the path with the shortest time from the priority queue ($O(n)$). Then, we check all the connections of the current city ($O(m)$). If the city is not in the path, and the time is better, we add the city to the queue ($O(1)$). The total connections we check throughout the loop is $O(nm)$. Therefore, we perform $O(n)$ pop operations and $O(nm)$ push operations, which gives us a time complexity of $O(n^2 + nm)$ for the Dijkstra algorithm. Since m is usually much smaller than n, e.g., $avg(m) = 1.5$ for the data we have, we can simplify the time complexity to $O(n^2)$.

To check if this fit the results we have, we check the lineariy of the time and the square of the nodes in the done array. We do another benchmark to get more data points. The results are shown in Figure 2.

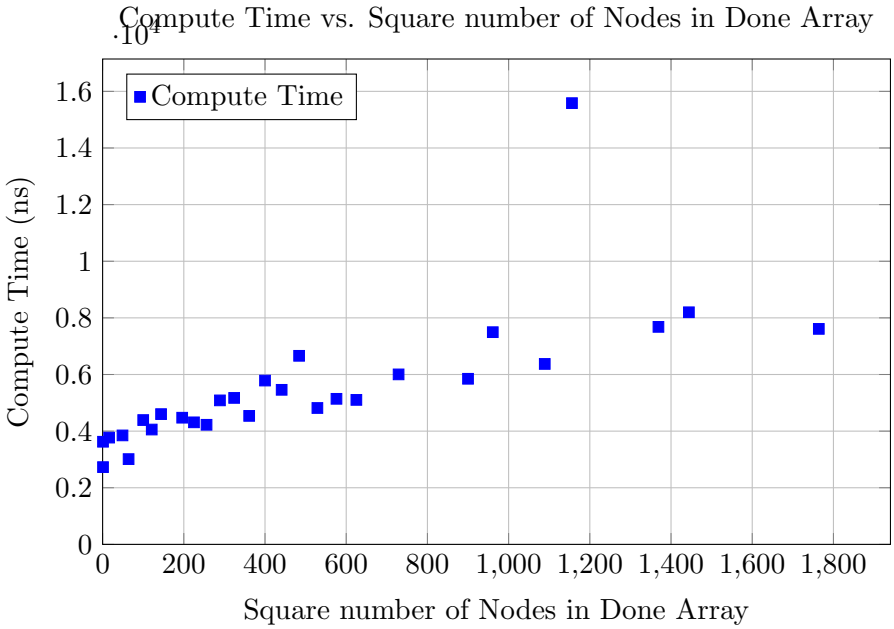

Figure 2: Compute Time vs. Square number of Nodes in Done Array for Shortest Paths from Malmö