Graphs in C

Ying Pei Lin

Fall 2024

The graph

To build the graph of the cities in this assignment, I use the provided `*graph(char *file)` function to read the file. After the file is read, the function will extract the cities and the time to travel between the cities in each line of the csv file and store them in the following structs.

```
typedef struct connection {
  city *dst;
  int time;
} connection;

typedef struct city {
  char *name;
  connection *connections;        // Dynamic array of connections, Hash table
  connection *recent_connections; // Array of recent connections
  int n;                          // Number of connections
  int capacity;                   // Capacity of connections
} city;

typedef struct map {
  city *cities; // Dynamic array of cities
  int n;        // Number of cities
  int capacity; // Capacity of cities
} map;
```

Then, I use the `connect_cities(map *m, char *city1, char *city2, int time)` function to connect the cities with the time to travel between them. It is two-way connection, which means I need to set up the connection elements for both cities. Down below is the code snippet to connect the cities.

```
void connect(city *src, city *dst, int time) {
  connection *c = (connection*)malloc(sizeof(connection));
```

```
  int index; // For the hash function
  c->time = time;

  // Set connection at city src
  c->dst = dst;
  if(src->n >= src->capacity) {
    src->capacity *= 2;
    src->connections = realloc(src->connections, src->capacity * sizeof(connection))
  }

  index = hash(dst->name, src->capacity);
  while(src->connections[index].dst != NULL) {
    index = (index + 1) % src->capacity;
  }
  src->connections[index] = *c;
  src->recent_connections[src->n] = *c;
  src->n++;

  // Set connection at city dst
  c->dst = src;
  if(dst->n >= dst->capacity) {
    dst->capacity *= 2;
    dst->connections = realloc(dst->connections, dst->capacity * sizeof(connection))
  }

  index = hash(src->name, dst->capacity);
  while(dst->connections[index].dst != NULL) {
    index = (index + 1) % dst->capacity;
  }
  dst->connections[index] = *c;
  dst->recent_connections[dst->n] = *c;
  dst->n++;
}
```

I store the connection in `*connections` and `*recent_connections` in the city struct. The `*connections` is a hash table using the destination city name as the key and it is used when we want to find the connection between two cities by name. The `*recent_connections` on the other hand is an array of connections and it is used when we want to iterate through all the connections of a city. It is also used in the depth-first search algorithm.

Depth-first search

One way to find the shortest path between two cities is to use the depth-first search algorithm. The depth-first search algorithm is implemented in the `shortest(city *from, city *to, int left)`. The function will recursively call itself to find the destination city. If the destination city is reached, it will return 0. If the destination city is not reached, it will iterate through all the connections of the current city and recursively call the function with the time left to travel. If we find a path that is shorter than the previous path, we will update the time.

```
int shortest(city *from, city *to, int left) {
  // Reached destination
  if (from == to) {
    return 0;
  }

  int sofar = -1; // Time to destination
  int update = 0; // Update path

  // Check all connections
  for(int i = 0; i < from->n; i++) {
    connection *c = &from->recent_connections[i];

    // If there is time left, try to reach the destination
    if (c->time <= left) {
      left -= c->time;

      // Recursively call the function
      // check children
      int d = shortest(c->dst, to, left);

      // If the destination is reached or the time is less than the previous time
      // update the time
      if (d >= 0 && ((sofar == -1) || (d + c->time) < sofar)) {
        sofar = (d + c->time);
      }
    }
  }
  return sofar;
}
```

Some benchmarks

Table 1 shows the results of the shortest path between two cities and the computation time.

Table 1: Shortest Path Results and Computation Time

Start	**Destination**	**Total Time (min)**	**Computation Time (ms)**
Malmö	Göteborg	153	1.3
Göteborg	Stockholm	211	18.1
Malmö	Stockholm	273	151
Stockholm	Sundsvall	327	28800
Stockholm	Umeå	517	8840000
Göteborg	Sundsvall	515	1120000
Sundsvall	Umeå	190	1.5
Umeå	Göteborg	728	143000
Göteborg	Umeå	705	2750000000000

While doing the benchmarks, I found that the left value will affect the result and the computation time. If it is too small, the function will not be able to find the destination city. If it is not big enough, the function will not be able to find the shortest path. If it is too big, the function will take a long time to compute. This is why the result of the time from Göteborg to Umeå is 705 minutes, which is different from the result of the time from Umeå to Göteborg, which is 728 minutes. The left value should be handled carefully to get the correct result.

Improvements

The current implementation of the depth-first search algorithm is not efficient because it will visit the same city multiple times, also it won't be able to find the shortest path if the left value is not big enough. To improve the algorithm, we can use an array to store the cities that have been visited and we can skip the city if it has been visited. By doing this, we can also solve the problem of the left value because we can stop the function if the destination city has been visited. Below is the code snippet of the improved depth-first search algorithm.

```
int shortest_path(city *from, city *to, city **path, int k) {
  if (from == to) return 0;
```

```
  int sofar = -1;
  for(int i = 0; i < from->n; i++) {
    connection *nxt = &from->recent_connections[i];
    if (!loop(path, k, nxt->dst)) {
      path[k] = nxt->dst;
      int d = shortest_path(nxt->dst, to, path, k+1);
      if (d >= 0 && ((sofar == -1 ) || (d + nxt->time) < sofar)) {
        sofar = (d + nxt->time);
      }
    }
  }
  return sofar;
}

int loop(city **path, int k, city *dst) {
  for (int i = 0; i < k; i++) {
    if (path[i] == dst) {
      return 1;  // City found in path, loop detected
    }
  }
  return 0;  // No loop detected
}
```

And the results of the improved depth-first search algorithm are shown in Table 2. We can see that although the computation time is longer than the previous implementation in some early cases, the computation time is much shorter in the harder cases.

Table 2: Shortest Path Results and Computation Time

Start	**Destination**	**Total Time (min)**	**Computation Time (ms)**
Malmö	Göteborg	153	149800
Göteborg	Stockholm	211	386000
Malmö	Stockholm	273	140000
Stockholm	Sundsvall	327	377000
Stockholm	Umeå	517	528000
Göteborg	Sundsvall	515	666000
Sundsvall	Umeå	190	527000
Umeå	Göteborg	705	234000
Göteborg	Umeå	705	968000