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));
```

```c
  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. To record the path, I create a struct called `path` to store the cities and the time to travel between the cities. The struct is defined as follows.

```
typedef struct path {
  city *cities;  // Array of cities
  int *times;    // Array of times
  int n;         // Number of cities
} path;
```

The depth-first search algorithm is implemented in the `shortest(city *from, city *to, int left, path *path)`. 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 path.

```
int shortest(city *from, city *to, int left, path *path) {
  // Reached destination
  if (from == to) {
    path->n = 0;
    path->cities[path->n] = *from;
    path->times[path->n] = 0;
    path->n++;
    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, path);
```

```
        // If the destination is reached or the time is less than the previous time
        // update the path
        if (d >= 0 && ((sofar == -1) || (d + c->time) < sofar)) {
          sofar = (d + c->time);
          path->cities[path->n] = *from;
          path->times[path->n] = c->time;
          update = 1;
        }
      }
    }

    // After check all connections, if the path is updated,
    // increase the number of cities
    if (update)
      path->n++;

    return sofar;
  }
```

Some benchmarks

Table 1 shows the results of the shortest path between two cities and the computation time. Table 2 shows the results of the shortest path between two cities with the travel times.

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	330	6.8

Table 2: Shortest Path Results with Travel Times

Start	**Destination**	**Path (with travel times)**	**Total Time (min)**
Malmö	Göteborg	Malmö → Lund (13) → Åstorp (36) → Halmstad (36) → Varberg (29) → Göteborg (39)	153
Göteborg	Stockholm	Göteborg → Herrljunga (39) → Falköping (15) → Skövde (16) → Hallsberg (42) → Katrineholm (31) → Södertälje (47) → Stockholm (21)	211
Malmö	Stockholm	Malmö → Lund (13) → Helsingborg (30) → Lund (35) → Hässleholm (30) → Alvesta (38) → Nässjö (33) → Mjölby (39) → Linköping (16) → Norrköping (24) → Södertälje (59) → Stockholm (21)	273
Stockholm	Sundsvall	Stockholm → Uppsala (35) → Sundsvall (120)	155
Stockholm	Umeå	Stockholm → Gävle (60) → Umeå (210)	270
Göteborg	Sundsvall	Göteborg → Örebro (140) → Sundsvall (140)	280
Sundsvall	Umeå	Sundsvall → Härnösand (35) → Umeå (160)	195
Umeå	Göteborg	Umeå → Sundsvall (160) → Göteborg (180)	340
Göteborg	Umeå	Göteborg → Örebro (140) → Umeå (190)	330

Improvements

Until I entered this section, I realized that I could use the path to avoid the repeated cities in the path. Referring to the code provided in the assignment, I abondoned the original path struct and use the city struct array to store the path. Below is the new code snippet for the depth-first search algorithm.