Graphs and shortest distance in C

Sam Serbouti serbouti@kth.se

October 27, 2024

In this assignment, I will build a graph in C. They can be seen as a set of edges nodes linked by edges. In my case, this assignment is about Swedish cities linked by minutes of travel. There are 75 connections and 52 cities in total. graphs are a non-linear structure. On paper they make sense but to build one, I will use linked lists.

1 Building the graph

Choosing the mod value and how to handle collisions

For hashing using modulo, it is best to have a prime number as the mod. Let's take 47, 61, 97: respectively, they have 48, 45 and 41 collisions. Choosing a large prime like 97 reduces the number of collisions but for only 7 of them, that's not so much of a gain... I will choose **mod=47**

	pros	cons
bucket	simpler to implement; small nb of unique cities $\rightarrow$ won't impact performance that much	longer lookup times; overhead for pointers
open-addressing	memory friendly	mod value $\approx$ number of cities $\rightarrow$ more collisions

Since I am working with a small number of cities, **buckets** will be more efficient.

Data structure

The index to locate a city's right bucket in the hash table works by looping through its characters (of its name), multiplying the hash by a prime number (31), adding the character's ASCII code and taking it modulo the hash table's size.

This is an edge connecting two cities by time (in minutes):

```
typedef struct connection {
    struct city *destination;
    int time;
    struct connection *next;
} connection;

typedef struct city {
    char *name;
    connection *connections;
} city;
```

This is the hash table that stores the cities by their name:

```
typedef struct codes {
    bucket *buckets;
    int n;//nb of buckets
} codes;

typedef struct bucket {
    city *city_data; //to first city in list
    struct bucket *next;
} bucket;
```

Adding a city to the hash table

`insert_city(codes *cities, char *name)` adds a new city to the map using a quick lookup table. It hashes the city's name to find a specific "bucket" like a folder in a filing cabinet to store the city data : `bucket *b = cities->buckets[hash(name, cities->n)]`. If the bucket is empty (it means there is no city at this location), it allocates memory for a city object and then copies the city name, sets up a blank list of connections (no roads yet)[1], and notes that there's now one city in this bucket. If there's already a city in the bucket, it creates a new bucket node to hold the new city. This new node is linked into the chain, handling the collision by forming a linked list in the bucket: `new_bucket->next=b->next; b->next=new_bucket;`

Adding a connection between two cities

`add_connection(city *src, city *dst, int time` links two cities with a road, adding the destination and time to the source city's adjacency list. First, it checks if either the origin or target city is NULL. If either is missing, it prints an error and exits. But if both cities exist, it allocates a new connection object, setting the destination city and travel time and then inserts this new connection at the start of `src`'s list of connections, making it the first road in the list.

Building that map!

The function `build_graph(file)` creates a codes structure to represent the graph, setting up an array of empty buckets with size MOD (presumably 47) for storing cities. It first tries to open the file. [2]. For each line in the file: It splits the line by commas to get the from city, to city, and time between them and converts the travel time from a string to an integer. THen it checks if a city is already in the graph, if not, it adds it with `insert_city` and looks

[1]using the null pointer : `connections = NULL`

[2]If it can't, it prints an error and exits

it up again to get the reference. Once both cities are in the graph, it calls `add_connection` both directions[3].Finally, it closes the file, frees temporary memory, and returns the complete city network.

Search algorithm

Now that all of our pieces are made and connected, I want to be able to find a city by name. `lookup_city()` must use hashing and handle collisions by checking linked elements in the buckets.

```
city* lookup_city(codes *cities, char *name) {
    int index = hash(name, cities->n); //hash the name
    bucket *b = &cities->buckets[index]; //locate likely folder
    while (b != NULL) {
        if (b->city_data != NULL && strcmp(b->city_data->name, name)==0)
            {return b->city_data;} //if it's a match returnn the city's data
        b = b->next; //if not go through the folder
    }
    return NULL; //not found
}
```

2 Find the shortest path between two cities

2.1 Depth first strategy (DFS)

I will start by making a naive and recursive search algorithm that takes 3 arguments: the starting city, the destination city, and a limit that bounds the path exploration (this will prevent segmetnation faults). It goes through all paths from the starting point to the target city and keeps track of the total travel time.

The best case happens when the current city is the target, it adds the city to the path and returns the total travel time so far. OTher than that, it records the current city's name in the path array to keep track of the route and loops through all connections from the current city:it calls dfs on this connected city, reduces limit by 1 and adds the travel time to the current time.

In the case of Stockholm to Sundsvall, the chosen path is:

```
./main Stockholm Sundsvall 5000
Shortest path found in 327 minutes in 0.96ms
Path: Stockholm -> S dert lje -> Norrk ping -> Link ping ->
     Mj lby -> N ssj  -> Alvesta -> H ssleholm ->
    Kristianstad -> Karlskrona -> Varberg -> G teborg ->
    Herrljunga -> Trollh ttan -> Uddevalla -> Str mstad ->
    Str mstad -> Kalmar -> Kalmar -> Kalmar -> Varberg ->
    G teborg -> Uddevalla -> Str mstad -> Str mstad ->
    Str mstad
```

[3]because I am building an undirected graph

It doesn't make much sense but this comes from the fact that DFS is unable to track "visited" cities so far. We could modify it to prevent it from visiting a city twice or more.

This idea of limiting the depth of the depth-first search (DFS) by setting a maximum allowable time constraint (instead of depth) is a smart way to avoid infinite looping especially in my scenario where I have loops in my map.

3 Tests and benchmarks

For testing if the shortest path algorithm works and how well (ie fast) it works, I will test them for 9 travel destinations and measure the time it takes to find it. Both algorithm have a 83 hours travel time limit (5000 minutes) give me the travel time in minutes and the shortest path (city by city) but because I didn't have enough place in my report, I won't write it here.

	travel time	computing time
Malmö to Göteborg	153	0.12
Göteborg to Stockholm	373	0.02
Malmö to Stockholm	368	0.24
Stockholm to Sundsvall	327	0.96
Stockholm to Umeå	517	1.00
Göteborg to Sundsvall	538	0.19
Sundsvall to Umeå	190	1.48
Umeå to Göteborg	no path under 83h	-
Göteborg to Umeå	no path under 83h	-

Table 1: Computing time in ms, Travel time in minutes

4 Let's avoid loops

Now i want to find the shortest path using depth-first search with a dynamic pruning strategy :a `dynamic_max` variable scratches any connections that goes over the known shortest path so far,to avoid inefficient routes and as shorter paths are discovered, `dynamic_max` is updated. But I also want the search to check for cycles to prevent infinite loops.

It goes as follows:

1. starts by checking if the recusrion level `k` hasn't exceeded a `MAX_RECURSION_DEPTH`. If it has, it returns -1 and stops further exploration down that path.
2. if source == target, it returns 0 as there is no further travel time required to reach the destination.

3. it checks if the current city is alraedy in the path, which would indicate a loop $\rightarrow$ it returns -1, making this path unusable
4. records the current city in the path array
5. it then goes over all connections from the current point for the connections with a time less than or equal to dynamic_max (it delets longer paths)
6. recursive calls for every connection's destination city and adds the travel time to the total time so far ; once all connections have been explored, the function removes the current city from path to let other routes to be traversed

Conclusion

I was very weird building a data structure that i couldn't quite draw in my head. I ended up having to write many errors signals to print on the terminal, a print_graph function and not knowing Swedish cities well, debugging was painful. When implementing the solution with loop tracking, I ran into segmentation faults over and over again. I learned to use the debugger GDB to be able to track what went wrong