Universitat Autònoma de Barcelona

OPTIMIZATION 2ND DELIVERY

ASTAR ASSIGNMENT

A ROUTING PROBLEM

(15/10/2024)

Daniel Bedmar Romero (1565494), Joan Garcia i Masferrer (1563605), Daniel Sarrat Palau (1600419)

In this report, it will be explained the implementation that our group has done of an $A*$-type algorithm as a means to solve the problem posed in this assignment.

The objective in this project consists in finding an optimal path, within a map of Spain, between two given locations: from *Basílica de Santa Maria del Mar (Plaça de Santa Maria)* in Barcelona to the *Giralda (Calle Mateos Gago) in Sevilla*, according to the distance traveled. And the implementation has to be done via an $A*$ algorithm in the programming language *C*.

For the purpose of optimizing the code, as outlined in the assignment, our team developed two different programs. The first program processes the *.csv* file containing the map node information, generating a binary file that pre-establishes the connections between nodes, hence enabling faster file access. The second program reads the binary file and implements the $A*$ algorithm to efficiently compute the solution to the problem.

The report will be divided in three sections: one for each code developed, and one to explore potential improvements to our programs and the programs mentioned in the report can be found in this Github repository [1].

1 Writing of the binary file

structure of nodes, counting of nodes, reading of nodes, allocation of successors + use of realloc + how does it find the correct successors, allocation of names, writting of nodes.

To create the binary file, the code was developed starting from the program *readingmap2.c* (provided in the CV), with various modifications to achieve the desired results. The original code base did not address count with some important aspects: it did not treat successors and node names as pointers and imposed an arbitrary maximum of 9 successors using the *MAXSUCC* constant. Consequently, our first task was to address these limitations. Additionally, while *readingmap2.c* reads the *.csv* file, it does not generate a binary file. Therefore, this functionality was also incorporated into our code. In the following section, the different modifications will be explained.

1.1 Code details

The first relevant change made to the base code is the `node` structure, converting names and successors into pointers.

```
    typedef struct {
    unsigned long id;         // Node identification
    char *name;               // Name of the node (now a pointer)
    double lat, lon;          // Node coordinates
    unsigned short nsucc;     // Number of node successors
    unsigned long *successors; // Array of the successors (now a pointer)
} node;
```

Subsequently, we dynamically allocated memory for these data structures referenced by the pointers. In the case of the node names, in the given *.csv* files, several of the nodes do not have a name. Hence, when reading the information with `field = strsep(&tmpline, "|");`, it returns a null string for these cases, given by the terminator `'\0'`. Then, in order not to waste a lot of memory space, a different size of memory was allocated for each case: saving 1 byte when there is no name, and a default `CHAR_LENGTH = 200` bytes when there is one.

```
 if (field != NULL && *field != '\0') // Check if the node has a name
{
    nodes[index].name = (char *)malloc( CHAR_LENGTH * sizeof(char) );
    if (nodes[index].name == NULL){
        printf("Error while allocating memory for the names.\n");
        return 2;
    }
    strcpy(nodes[index].name,field);
}
else{
    nodes[index].name = (char *)malloc( sizeof(char) );
    if (nodes[index].name == NULL){
        printf("Error while allocating memory for the names.\n");
        return 2;
    }
    nodes[index].name[0] = '\0'; // Save an empty string}
```

To manage the successors, it is started by dynamically allocating 1 space of the size of `unsigned long` for each node (since it is storing the corresponding indexes of the successors), but this space proved insufficient, since on average the nodes have a valence of 1.933.

To address this limitation, the `realloc` function was introduced to dynamically expand the allocated memory when necessary. Specifically, within the loop that iterates through all the IDs in the `@way` data from the *.csv* file, additional memory is allocated each time a successor is added to a node. This ensures that the array has adequate space for storing the newly added elements before insertion occurs.

The following is an example of this implementation:

```
if(newdest){
    if (nodes[origin].nsucc > 0){
        unsigned long *temp_orig;
        temp_orig = (unsigned long*)realloc(nodes[origin].successors,
        (nodes[origin].nsucc + 1)*sizeof(unsigned long));

        if (temp_orig == NULL){
            printf("Error while reallocating memory for the successors.\n");
            free(nodes[origin].successors);
            return 10;
        }

        nodes[origin].successors = temp_orig;
    }

    nodes[origin].successors[nodes[origin].nsucc] = dest;
    nodes[origin].nsucc++;
    nedges++;
}
```

Finally, the binary file is created. The file begins with a header that states the total number of nodes and successors (`nnodes` and `nedges` respectively). This information is crucial when reading the binary file, as it determines how much memory needs to be allocated. Following the header, the binary file contains two distinct blocks: one for the vector representing the nodes and another for the successors of these nodes. This information is written while error-checking with the function `ExitError`. The specifics of the code are of the same nature as in the assignment description.

As a final step, all dynamically allocated memory is freed to prevent memory leaks and ensure proper resource management.

1.2 Compilation and execution instructions

Compilation: For compiling the first code, it is required to type into the terminal (considering the name of the C file remains as when delivered):

```
gcc Astar_binaryfile_write.c -o Astar_binaryfile_write
```

Execution: The name of the *.csv* file from which the node and way information is obtained must be given at runtime. For a certain "name.csv" file, it generates a binary file named "name.csv.bin". If an argument is no given, it will automatically search for a file named *andorra.csv* (which corresponds to the map of Andorra).

```
./Astar_binaryfile_write <name.csv>
```

2 Reading of the binary file and application of the A* algorithm

The A* algorithm has been implemented, following the course indications, by initializing an empty priority queue (`Open`, corresponding to unexplored nodes), setting all the nodes cost to infinite and all of them neither in the open nor `Closed` list (an `AstarPath` structure with `IsOpen` as false).

Following up, the first node is set (with the previously introduced ID) as start, we set its cost to zero and its parent as `UINT_MAX` (a way of setting to infinite, determining that it does not exist and for avoiding segmentation faults) and we compute the distance between this point to the target using the *Haversine Formula*.

Once everything is initialized, we start the main A* loop, that will be executing itself until we arrive to the goal node, or all elements have been explored.

First, we chose what node will we be working with, to chose that node, we check the node with the minimum total cost function value in the Open List. We first check if the current node is the goal. If it is, the loop ends, if not, we explore each successor of the current node, compute its distance to the current node and its the aggregated cost to reach the successor.

Then, if the new calculated total cost is less than the previous cost to get to a particular successor, we change the total cost wit the new aggregated cost plus the heuristic function. Once computed the estimated cost, if the node has already been explored, we have to reorder it (with its new cost) in the priority queue and if it has not been explored before, we simply add it to this queue. Once this is done, we set the current node out of the open list. Afterwards, a new iteration of the main loop will begin.

The program will generate a txt file "path.txt" that contains the list of nodes that compose the optimal path. This file can be passed as an argument to the file `Map_Plot.py`, provided in the assigment, in order to visualize the optimal path in a map.

2.1 Code details

First, a custom boolean type is defined: `bool`. Following up, the structures defined in the program are `node` which contains all the information about each node in the map that is read, and it preserves the same form as in the "writing" code, `AstarPath` which is a structure that stores the minimal path data and is used to compare node values in the Astar function), `AstarControlData` which is a data structure that keeps track of each node's estimated total cost and indicates whether a node has been explored. And finally, the `QueueElement` and `PriorityQueue` structures used to build the graph to explore.

Our "Astar" code main function is divided in three main parts: the reading of a map in binary format (requested as input), the execution of the A* algorithm between two previously set nodes (both requested as input), and the writing of a ".txt" file with the minimal path nodes information in the requested format to subsequently visualize it in a map.

The functions that we have used are a function `ExitError` to terminate the program when an error occurs, `searchNode` that traverses the nodes graph doing a binary search to retrieve a node using its ID, `IsEmpty` that checks the status of a queue (used in the A*), `extract_min` that returns the minimal element of the previously sorted linked list, and the functions `add_with_priority` and `requeue_with_priority` which add and reorder a given node in the set priority queue based on the cost function of each node in the list.

Finally, the last two functions used in the program are the `Astar` function and the `heuristic` function used as a heuristic in the A* algorithm and, in our case, to compute the distance between nodes. The heuristic function we have used is the *Haversine Formula*; a function that given two latitudes and longitudes computes the "great-circle" distance between them, considering the earth a sphere.

2.2 Compilation and execution instructions

In this section we will assume that the *C* file is named as *Astar.c* and we want an executable named *Astar*. In case we wanted to change these names, there would not be any problems as long as consistency is kept.

Compilation: We recommend compiling this program as:

```
gcc -Ofast Astar.c -o Astar
```

Moreover, depending on the configuration of the compiler it might be necessary to add the `-lm` flag to the compilation command [1].

Execution: For executing the program we have to use the command:

```
./Astar <binary_file_map> <start_id> <target_id>
```

Where `binary_file_map` is the previously generated binary file containing the map information, `start_id` is the ID of node that is the starting point of the route and `target_id` is the ID of the node we want to arrive to.

We have introduced some complementary code to the one that tries to work with the Andorra's map if the incorrect amount of arguments is passed. What this code will do is:

- If at least one argument, but not three, have been passed: it will assume the first argument you have passed is a valid map file name and try to compute a demo using that map.
- If no arguments are passed: it will try to work with the Andorra's map and compute a route between two arbitrary nodes in the map.

Note that if the number of arguments is different than 3, only *andorra.csv.bin*, *catalunya.csv.bin* and *spain.csv.bin* will be considered valid and tried as map file names. If these files don't exist, or they contain different nodes than the ones provided in the Assignment, the computation will fail.

[1] When executing the program with the *-Wall* flag it may appear the warning "Astar.c:9:9: warning: 'M_PI' macro redefined [-Wmacro-redefined]". This warning will appear depending on the configuration of the compiler but it does not affect on the performance of the program, it is just informing that the constant PI has been redefined. Redefining this macro is necessary for some compiler configurations.

3 Further improvements

Even-though we have implemented a solution that is able to write the map in a binary file in 23.8 seconds and then read and find the optimal path in 4.6 seconds with normal laptops [2], further improvements could be implemented to build a more optimized program for potential applications in bigger maps or simpler devices.

Binary Heap implementation
In the current implementation, the Open and Closed list are linked lists . In one hand, this approach is very simple, both to implement and debug. But, in the other hand, binary trees are more efficient in managing data than linked list, specially if we have to manage big volumes of data or if we have to add new data or rearrange the order of the data a lot of times, as it can be seen in Table 1.

Therefore, to get a faster program, we would add Binary Trees as a data structure for managing the necessary Lists in the A* implementation.

Operation	Binary Heap	Linked List
Insert element	O(log n)	O(n)
Extract the smallest element	O(log n)	O(1)
Requeue element	O(log n)	O(n)

Table 1: In this table we compare the complexity of the operations of interest between storing data in a Binary Tree and in a Linked List.

Memory Handling Improvement
In the current implementation, a lot of memory is allocated to initialize the Open and Closed list, as well as the list that keeps track in which list each element is. In fact, it allocates as many elements as numbers of nodes are in the map. We decided to keep this approach because since we have the nodes in the structure of a linked list, being able to also store them in a linked list and in the same position makes the code much clear and easy to implement and debug.

Although the ease of understanding and working with this approach, in order to build a real optimized program we would need to dynamically allocate the size of the lists as we add elements to them, as the difference in size of the total of nodes in the map and the nodes we explore grows with the size of the map.

References

[1] "Github repository with the material generated." `https://github.com/danielsarrat/optimization_A-A/tree/main`.

[2] In particular, these times were obtained with a MacBook Air from 2018. 1,6 GHz Intel Core i5 of double core processor and 8GB of RAM.