TCS1-IN: Maze generator

Author : Christophe Garion
Audience : SUPAERO 1A
Date : February 15, 2024

1 Objectives

The objectives of this project are the following:

- write an implementation of Prim's minimum spanning tree algorithm to generate a maze from a random map. This implementation will allow to load a map from a file and to generate a PPM graphic file with the maze computed from the map.
- use an efficient data structure to accelerate the generation of the maze.

These two objectives are presented in the following.

2 A maze generation algorithm

Our first objective is to generate a maze. We will define a maze as a path on a 2D grid such that:

- each node on the grid can be reached from any other node using the path.
- there is no "loop" in the path, i.e. you cannot return to a starting node following the path.

There are lots of maze generation algorithms (cf [6] for instance) and you will use a simple one here: Prim's algorithm for computing a minimum spanning tree over a graph.
First, let us present which kind of graph we will consider. Remember that a graph $G = \langle V, E \rangle$ is a set of *vertices* V and a binary relation E on V representing *edges*, i.e. links between vertices. We are only interested here in graphs

- in which edges are labelled with integers representing the corresponding edge cost
- that are undirected, i.e. if there is a link from vertex v_1 to v_2 with label n, then the same link with label n exists between v_2 and v_1

Moreover, we will only consider graphs such that V and E represents what we call a *map*, i.e. a graph in which vertices are organized like a 2D mesh. Figure 1 presents such a graph. Notice how the vertices are numbered with consecutive red numbers to identify them. We call these numbers *IDs*.

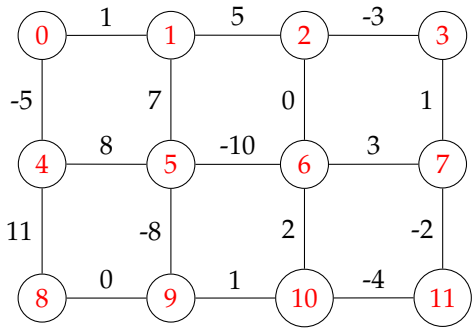

Figure 1: An example of a map, i.e. the kind of graphs we consider in the project

Let us now explain what is a minimum spanning tree (MST). A minimum spanning tree T on G respects the three following properties:

- T is a tree built on E, i.e T is a subset E' of E such that
  - there is no cycle in E' and

  - for every vertices v_1 and v_2 appearing in the edges of E', there is a path from v_1 to v_2 using the edges of E'.

- T is spanning, i.e. every vertex of G is the extremity of at least one edge in T (taking any vertices v_1 and v_2 of G, you can "go" from v_1 to v_2 using the edges in T).
- T is minimum, i.e. the sum S of the labels of the edges of E' is minimal (there is no other spanning tree on G such that the sum of the labels of its edges is strictly less than S).

For instance, Figure 2 shows a minimum spanning tree on the previous graph with a total cost of -25. Notice that there exist several minimum spanning trees on this graph (you can choose for instance the edge with cost 1 on the last row instead of the one with cost 1 on the last column for instance).

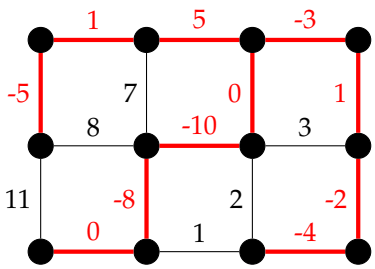

Figure 2: A minimum spanning tree on a simple graph (nodes ID removed for readability)

If we now consider a black background (with a border) and white squares for the graph vertices and the edges of the minimum spanning tree, we obtain Figure 3 which is a (very simple) maze! Notice that white cell correspond to the vertices of the graph, whereas gray cells are built from the edges selected in the minimum spanning tree.

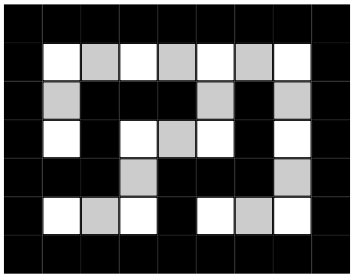

Figure 3: The maze generated using the previous minimum spanning tree (white cells represent vertices of the graph, gray cells represent the edges in the tree)

The idea here is therefore to start from a initial graph like the one presented in Figure 1 to generate a minimum spanning tree from this graph to obtain a maze.
Which algorithm will you use to compute a minimum spanning tree from the graph? You will implement Prim's algorithm [3, 8], an algorithm very similar to Dijstra's shortest path algorithm you should have seen during your previous studies. Like Dijkstra's algorithm, Prim's algorithm is a greedy algorithm that guarantees a global optimum by choosing at each turn a local optimum, here the edge with the minimal cost that adds a new node in the tree. In contrast with Dijsktra's algorithm, the starting node does not matter.
Prim's algorithm is presented on Algorithm 2.1. The minimum spanning tree can be built using the `predecessors` array. Table 1 presents the iterations of Prim's algorithm loop on the example presented on Figure 1. Each node is associated with a pair c/p where c is the current cost associated to the node and p is the current predecessor of the node in the tree. Only pairs needed to be updated are written on a particular iteration.
Notice that:

- on iteration 10, I have chosen node 7, but node 10 could also have been chosen as both have a score of 1.
- on iteration 9, node 8 is chosen and scores are not updated as edges going out of node 8 cannot decrease the cost of unexplored nodes.

Algorithm 2.1: Prim's MST algorithm

Input: a connected graph G with edges labelled with costs
Result: at the end of the algorithm, the MST can be built with the predecessors array

```
1  create an empty set N;
2  foreach vertex v of G do
3  |  cheapest[v] = ∞ ;
4  |  predecessors[v] = null_node ;
5  |  add v in N;
6  end
7  choose a node S;
8  cheapest[S] = 0;
9  while N is not empty do
10 |  v = the node in N with the minimal cost ;
11 |  remove v from N;
12 |  foreach neighbor n of v do
13 |  |  if n ∈ N then
14 |  |  |  if weight of (v, n) < cheapest[n] then
15 |  |  |  |  cheapest[n] = weight of (v, n);
16 |  |  |  |  predecessors[n] = v;
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

iteration	0	1	2	3	4	5	6	7	8	9	10	11	chosen
	$0/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	$\infty/\_$	
1		$1/0$			$-5/0$								0
2						$8/4$			$11/4$				4
3			$5/1$			$7/1$							1
4				$-3/2$			$0/2$						2
5								$1/3$					3
6						$-10/6$					$2/6$		6
7										$-8/5$			5
8									$0/9$		$1/9$		9
9													8
10												$-2/7$	7
11											$-4/11$		11
12													10

Table 1: Execution of Prim's algorithm on the previous example (starting from node 0)

3 Accelerating maze generation using a heap

The bottleneck of the algorithm is of course the search of the unexplored node with the minimal associated cost (line 10 of Algorithm 2.1). We therefore need an efficient data structure to store the nodes in order to accelerate this search. A priority queue is an Abstract Data Type similar to a queue but with a priority order associated to the elements that it stores. It has two operations:

- an `insert` operation that stores an element with an associated priority
- an `extract-min` operation that removes the element with the highest priority from the queue and returns it

You will implement a priority queue using a binary heap which guarantees a worst case execution time in $O(\log n)$ for both operations. See section 4.7 for details on the implementation.

The priority queue will store the nodes of the graph and the priority order will be given by the cost associated with the node: the highest priority element will be the one with the smallest cost. We will thus be able to retrieve the node with the minimal cost in $O(\log n)$.
Does it really accelerate Prim's algorithm execution? Table 2 shows the execution of Prim's algorithm on four maps using both a linear search or a binary heap (`map_W_H_R.txt` is a file representing a map with width `W`, height `H` and generated from a random seed `R`). Experiments have been made on a laptop equiped with a Intel i9 processor and executables have been generated using the `-O3` option of GCC.

file	time for linear search (s)	time for heap (s)
./data/map_10_8_76.txt	0.000057	0.000111
./data/map_100_100_42.txt	0.078898	0.003038
./data/map_200_200_42.txt	1.526626	0.013774
./data/map_400_400_42.txt	23.424665	0.063049

Table 2: Comparison of Prim's algorithm execution times using a linear search or a heap

4 Implementation details

4.1 The map

Maps used to generate the maze will be implemented by a `map` structure containing :

- the height and the width of the map
- a pointer to a `edges` structure

The `edges` structure represents the edges starting from a node. It contains:

- a field named `down` containing an **`int`** value representing the cost of the edge starting from the node and going to the node below it;
- a field named `right` containing an **`int`** value representing the cost of the edge starting from the node and going to the node at its right.

To avoid duplicating information, there are no fields for edges going upwards and leftwards from a node. To ease the implementation, nodes on the border of the map will have values for both the `down` and `right` fields even if these values are useless.
The pointer to `edges` in the `map` structure points to a *dynamically allocated* memory region that can be assimilated to an array of `edges` values. This region contains the width $\times$ height `edges` values representing the map. We suppose that the nodes are numbered starting from the upper-left node and going right, row by row as presented on Figure 1.
The declaration and implementation of the map structure and functions should be done respectively in the `map.h` and `map.c` files.

4.2 The map loader

The map loader reads a text file containing the map description and returns a structure representing the map (cf. section 4.1). You will find in the `data` subdirectory of your repository several file examples. For instance, `map_2_3_42.txt` (a simple map that can be used for testing purposes) is the following:

```
2 3
0 -14
-19 -19
18 -8
0 1
3 -5
3 14
```

- the first line contains the width and the height of the map separated by a space

- the other lines contains the edges cost for each node of the map with the convention described in section 4.1 (first value for "down" direction, then for "right" direction)

You have to implement a `load` function that takes a filename as parameter and returns the corresponding `map` structure in a `loader.c` file.
Look at appendix E to understand how to read data from a file (`"%s"` is the format string used in `scanf` to read a string). The `app-ex-loader.c` file contains a example program that reads a file given as input on the command line, gets the dimensions of the map and print them and print each line of the map. You can use it to write your `load` function.
You must provide a test for your loader in the `test-loader.c` file. This program should:

- read a file using `main argv` argument (see appendix D to understand how to pass arguments to `main`)
- build the corresponding `map` structure
- print the structure using a printing function defined in `map.c`

To test your implementation, simply try the previous program with different examples and verify that the output on the console is the same as the content of the file.

4.3 The `find_neighbors` function

In order to ease the development of your application, you will implement a `find_neighbors` function taking as parameter a map, a node number and returning a `neighbors` structure. The `neighbors` structure for a node n is composed of:

- a `nb` field containing the number of neighbors of n
- an array of 4 `int` values containing the numbers of the nodes that are neighbors of n
- an array of 4 `int` values containing the cost of the edges to go from n to its neighbors in the same order than the previous array

Of course, nodes may have 2, 3 or 4 neighbors depending on their position on the map.
The `test_neighbors.txt` file contains neighbors structures for some nodes of the map described in the `map_10_8_tests.txt` file. For instance, the `neighbors_node_59` structures contains the neigbors of node 59. **Neighbors are stored in the following order**: below, right, above, left.
You must write a `test-find-neighbors.c` test application that verifies the neighbors of all nodes as defined in `test_neighbors.txt`. Include the C definitions of `test_neighbors.txt` directly in `test-find-neighbors.c` to write your tests. All comparisons must be checked with the `assert` function (see appendix A).

4.4 The maze

The maze is represented by a `maze` structure containing the following fields:

- two fields with `int` values representing the width and the height of the map
- an array `predecessors` of `int` values such that `predecessors[i]` is the number of the node that is the predecessor of node `i` in the minimum spanning tree
- a field containing the total cost of the maze (i.e. the sum of all edges in the corresponding minimum spanning tree)

You should also implement a `print_maze` function that simply prints the maze on the console. Use a character like `'#'` to print walls. On Linux, you may also print the following string to print an inverse color square on the console: `"\033[7m \033[m"`. You may print two characters (spaces, `'#'` or spaces in inverse color) to have a better printing on the console.
Beware, the width and the height contained in the structure are these of the map. When printing the maze, you have to print an "extra" character between each node of the map, this character representing either a wall (if there is no path between the two nodes) or an empty space (if there is a path between the two nodes). You may implement the function:

- either by allocating memory for a matrix of **`char`** that represents the maze, assigning each of its components with the corresponding character (blank for a map node or a path between two nodes) and then print this matrix
- or by printing directly the maze without allocating extra memory. This solution is of course better.

4.5 Creating an image of the maze

We want to be able to generate pictures of the maze in the PBM format [7], a simple graphical format that uses text files.
A file containing an image in the PPM format is a text file with a `.pbm` extension and is defined as follows:

- the first line of the file must be `P1`.
- the second line contains the width and the height of the image separated by a space character.
- the following lines contains the list of pixels of the image from left to right, then from up to down. Each pixel is coded by only one value, either `0` for white or `1` for black. Each pixel separated from the next by a space character. You can have a line of text for each line of pixels, but it is not mandatory.

For instance,

```
P1
17 7
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1
1 1 0 1 1 0 1 1 1 0 1 0 1 0 1 1 1
1 1 0 1 1 0 0 0 1 0 0 0 1 0 0 0 1
1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 1 1
1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

will display the word ISAE.
You will find in your repository (in the `data` directory) the previous example in the `isae.ppm` PPM file. Do not hesitate to open the PPM file in VSCode.
You must provide a `generate_image` function in the `maze.c` file. This function takes a maze and a string representing the name of the file in which you should write the PPM image.

Do not add the PPM files in your repository! Normally, Git should ignore them.

4.6 Prim's algorithm

You have to implement a first version of Prim's algorithm in the `prim.c` file. You should follow Algorithm 2.1 and use a simple linear search to find the node with the minimum cost among the unexplored nodes. Use dynamically allocated arrays to store the current costs for each node and to represent the fact that a node has been explored or not.
To verify your implementation, the `tests_maze.txt` file in the `data` directory contains the C declarations of the mazes you should obtain for the `map_10_8_42.txt`, `map_10_8_76.txt`, `map_10_8_1024.txt` and `map_10_8_2048.txt` map files. Write a `test-prim.c` application that verifies that you obtain the same maze as the ones described in the `tests_maze.txt` file when applying your generating function on the corresponding files.
You also have to write a program in the `app-generate-maze.c` file. This program should take as argument on the command line the name of the file containing the map from which the user wants to generate the maze. The program should generate an image of the maze in PPM format as explained in the previous section.

4.7 Binary heap implementation

In order to accelerate Prim's algorithm, we will use a *priority queue* to find the node with the minimum score in the set on unexplored nodes, cf. section 3.

We will consider in the following priority queues containing `int` values and that the `extract` operation is a `extract_min` operation (we are interested in the element with the minimum value in the priority queue).
Binary heaps are an implementation of priority queues that use a binary tree. They respect the following two properties:

- they are *complete* trees, i.e. all levels of the tree are full except the last one which is filled from the left
- each node of the tree has a lower value than all its sons

For instance, figure 4 is a binary heap: the tree is complete and all nodes respect the second property. Notice that the node (80) cannot be placed under the node (75) or the node (51) even if the second property is respected as the last level of the tree has to be filled from the left.

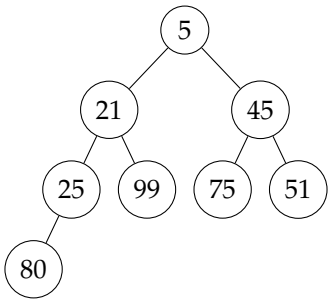

Figure 4: An example of binary heap

Of course, the minimum value is at the root of the tree and we can get it in $O(1)$, but how can we guarantee $\log n$ complexity for `extract_min` and `insert` operations?

- inserting a new node is done as follows: find the first free place in the tree, put a node with the new value here and make the node go up in the tree while its parent node is greater than it. For instance, the steps for inserting the value 17 in the previous binary heap are shown on figure 5.

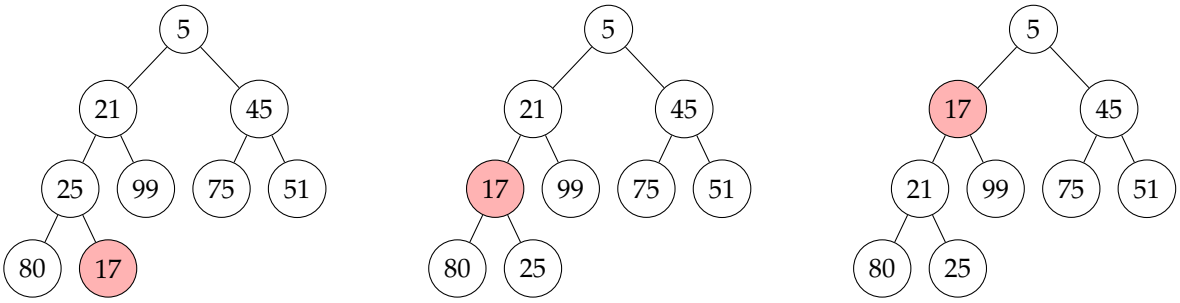

Figure 5: Inserting the value 17 in the previous binary heap

  As the tree is complete, the node will go up in at most $\log_2(n)$ steps.

  Notice that it should be a good idea for each node to know its parent node in order to ease the exchange.

- the `extract_min` operation works as follows: get the minimum value at the root of the binary heap, put the last value in the binary heap at the root and make it go down the tree while it is greater than at least one of its child (exchange it with the child with the lowest value). For instance, the steps for extracting the minimum value from the tree presented on figure 4 are shown on figure 6.

Of course, to implement these algorithms, you should know where is the last node in the binary heap or where to insert a new node! It is rather simple. Let us look at an example. Suppose that the nodes are numbered as in figure 7 (this numbering is rather natural). Consider now the 9^{th} node in the heap. 9 in binary notation is `1001`. Remove the leading `1`, start from the root and consider that `0` means "left" and `1` means "right". You arrive exactly at the ninth node! Try it on several examples, it works 😁!
Algorithm 4.1 presents the algorithm to find the father node of node numbered n in a binary heap. Notice that:

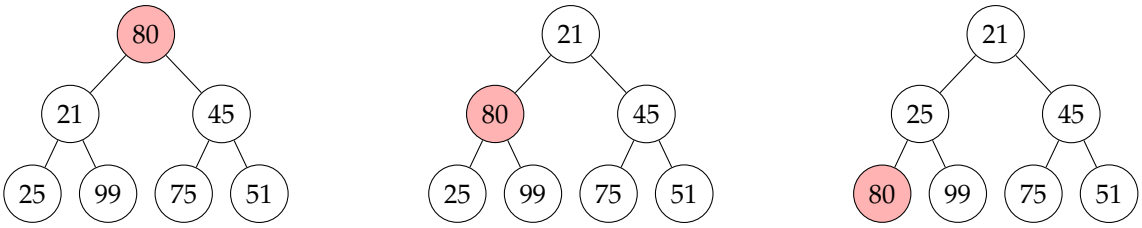

Figure 6: Extracting the minimum value in the previous binary heap

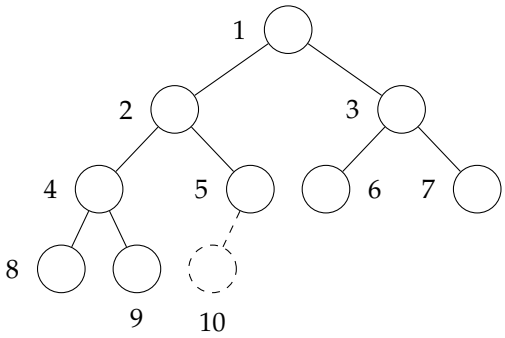

Figure 7: Numbering the nodes of a binary heap

- >> is a bit shifting operation and & is a bitwise AND operator (see bitwise operations in C)
- $\log_2$ is implemented as the `log2` operation in the `math.h` library. To use it in C, you must cast the result to **`int`**. For instance

```
int depth = (int) log2(n);
```

Algorithm 4.1: finding the father of node n $(n > 1)$

```
depth = log2(n);
initialize p_father_node with a pointer to the root of the tree;
for i ∈ {depth −1, ... , 1} do
    if (n >> i) & 1 = 0 then
        p_father_node := pointer to left child of p_father_node;
    else
        p_father_node := pointer to right child of p_father_node;
    end
end
return p_father_node;
```

4.7.1 Structures needed to represent a binary heap

In order to represent a binary heap in C, you will need:

- a node structure containing an **`int`** value representing the ID of the node, an **`int`** value representing the current cost of the node, a pointer to its left child (possibly `NULL`), a pointer to its right child (possibly `NULL`) and a pointer to its parent node (possibly `NULL` for the root of the binary heap)
- a structure representing the heap containing the number of nodes in the heap and a pointer to the root node of the heap

For instance, the heap presented in figure 4 could be represented in memory as in figure 8 (only the cost are represented in the figure for readability, the node identifiers are not represented).

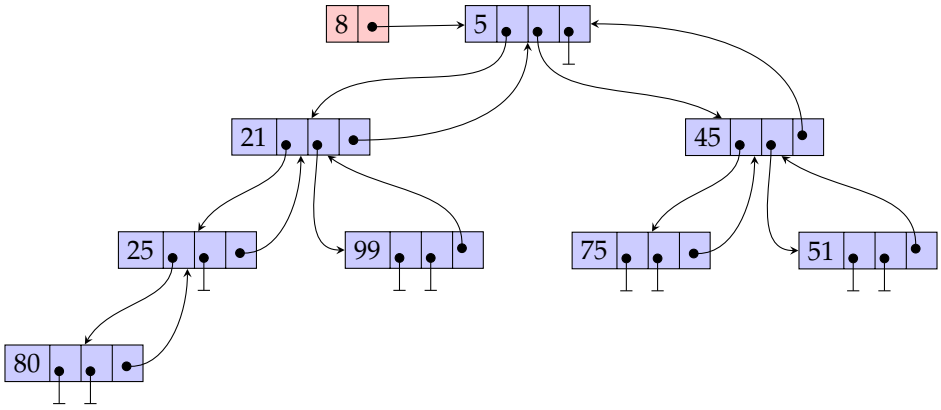

Figure 8: A possible memory representation for binary heap from figure 4

4.7.2 Binary heap validation

In order to validate your implementation, you will produce two programs:

- a `test-heap-correctness.c` file that contains a program showing that your heap is correct. In this program, you will insert 50 nodes with cost from 1 to 50 in a heap and verify that when you call `extract-min` 50 times on the heap you get the nodes in the right order.
- a `test-heap-complexity.c` file that contains a program showing that the operations `insert` and `extract-min` have the right complexity. In order to do so, you will measure the time needed to insert n values and to call `extract-min` n times, with n starting at 20 and finishing at 10000000 for instance (simply double n at each experiment).

  To measure time, use the `clock` function declared in `time.h` (you should therefore include this file). For instance, here is a snippet of code measuring the time to call a function `foo` n times:

```c
clock_t start = clock();

for (int i = 0; i < n; i++) {
    foo();
}

clock_t end = clock();

double elapsed_time = (double) (end - start) / CLOCKS_PER_SEC;
```

  You will write the results in a `data_complexity_heap.csv` file. Each line of the file will have the following format:

```
nb,time_to_insert,time_to_extract
```

  where `nb` is the number of values inserted and extracted. Refer to appendix E to understand how to write data to a file. The Python script `plot_heap_complexity.py` in the `scripts` directory will plot your data.

4.7.3 Maze generation with binary heaps

You should write a version of Prim's algorithm that uses a binary heap to find the node with the minimal cost amongst the unexplored nodes. This algorithm must be implemented in a `prim_bh.c` file.
There is although a tricky point: the cost of a node changes during the algorithm execution! For instance, the node with ID `5` may have a cost of `15` and then `7` when one of its neighbors is added in the MST.
You may implement cost updating in two ways:

1. as the cost of the nodes can only decrease during the execution of the algorithm, you may simply add a new node in the heap with the same ID and the new cost. Let us suppose for instance that the node with ID `5` is stored in the heap with a cost of `15` and that you want to update this cost to `7`. If you add a new node in the

heap with ID 5 and cost 7, then this node will be removed from the tree before the one with ID 5 and cost 15 (due to the second property of binary heaps).

Therefore, when you remove the minimal cost node in the heap, you have to check if the node with the corresponding ID has already been removed from the heap. In this case, it means that you already have encountered the node but with a better cost. On the other hand, if a node with this ID has not been already removed, then the cost of the removed node is the minimal one.

2. (more difficult) you maintain an array `pos` of pointers to nodes of the heap, `pos[i]` being a pointer to the node in the heap representing node `i` of the map. When updating the cost of a node, you balance the heap by making the node going upwards the tree to respect the heap property.

You should write a program in `app-generate-maze-bh.c` that generates a maze using an implementation of Prim's algorithm using binary heaps with the same guidelines as these given in section 4.6.

4.8 Development plan

You may apply the following development plan:

1. `map.h` and `map.c`: define necessary structures and the `print_map` function.

   Notice that you may test your print function by defining a map structure directly in a program (i.e. without loading a file).

2. `loader.h`, `loader.c` and `test-loader.c`: define the loader and test it on several examples.
3. `map.h`, `map.c` and `test-find-neighbors.c`: define the `find_neighbors` function and test it.
4. `maze.h` and `maze.c`: define necessary structures for the maze representation and a `print_maze` function.
5. `maze.h` and `maze.c`: add PPM generating function.
6. `prim.h`, `prim.c` and `test-prim.c`: implement Prim's algorithm and test it.
7. `app-generate-maze.c`: define an application that takes a map filename on the command line and generates the corresponding maze in a PPM file.
8. `heap.h`, `heap.c`, `test-heap-correctness.c` and `test-heap-complexity.c`: define a binary heap for nodes and test it.
9. `prim-bh.h`, `prim-bh.c` and `test-prim-bh.c`: implement Prim's algorithm using a binary heap and test it.
10. `app-generate-maze.c`: define an application that takes a map filename on the command line and generates the corresponding maze in a PPM file using binary heap version of Prim's algorithm.

Do not try to implement several functionalities at once, it will not work! Implement **and test** your code incrementally.
See section 6 to see how you will be graded.

Notice that:

- you will have all notions to implement the map, the loader, the maze and Prim's algorithm after lab session M8 (1 March 2024). Of course, you may begin before M8!
- you will have all notions to implement the heap after session M10 (13 March 2024).

4.9 Summary of files to produce

	specification	implementation
map	`map.h`	`map.c`
loader	`loader.h`	`loader.c`
maze	`maze.h`	`maze.c`
test neighbors		`test-find-neigbors.c`
test loader		`test-loader.c`
Prim's algorithm (naive)	`prim.h`	`prim.c`
app to generate maze (naive)		`app-generate-maze.c`
test Prim's algorithm (naive)		`test-prim.c`
heap implementation	`heap.h`	`heap.c`
heap correctness		`test-heap-correctness.c`
heap complexity		`test-heap-complexity.c`
Prim's algorithm (with heap)	`prim_bh.h`	`prim_bh.c`
app to generate maze (with heap)		`app-generate-maze-bh.c`
test Prim's algorithm (with heap)		`test-prim-bh.c`

5 Work to do and requirements

What you have to do and the project requirements are summarized in the following points. The project is due **3 April 2024 23h59**. **The code retrieved from your repository at this date will be considered as the final version of your project.**
You will have to commit working code for the maze generator (without heap) on **15 March 2024 23h59**. If not, penalty will be applied on your final grade.

5.1 Requirements about functionalities

[R1] you must implement the following functionalities: a program generating a maze from a map loaded from a file, an optimized version of the generator and tests as defined in section 4.

[R2] you may implement one of the following extensions: use a Fibonacci heap implementation to verify if it is better than the "classic" one, implement Wilson's algorithm [5], another maze generation algorithm, with an efficient solution to detect cycles. You may also find yourself an interesting extension. These extensions will be taken into account in your final grade only if the basic functionalities are correctly working.

5.2 Requirements about implementation

[R3] your implementation must be written in C.

[R4] your code must compile with the `-std=c99 -Wall -Werror` options and at least GCC version 10.

[R5] your must use the provided Makefile (see appendix B). The rule `compile-all` should compile all your executables and tests. The rule `launch-tests` should run all your tests. See section B for more details.

[R6] your code must work on the ISAE computers under Linux.

[R7] you must write

- a program in the `app-generate-maze.c` file that allows to generate a maze from a map file using Prim's algorithm.
- a program in the `app-generate-maze-bh.c` file that allows to generate a maze from a map file using a heap version of Prim's algorithm.

If you have not implemented one functionality, do not code the corresponding part of the program.

[R8] you must write the test files as defined in section 4.

[R9] you may add programs to test your implementation. They must be located in files beginning with `app-`.

[R10] your data files (maps for instance) should be located in the `data` directory and have a `.txt` suffix.

[R11] your code must not produce errors when using the Valgrind tool.

[R12] you may reuse code **YOU** have implemented during the lab sessions.

5.3 Requirements about code conventions

[**R13**] your header files must be documented, possibly with the Doxygen tool (look at lab sessions and the corresponding refcard on LMS). Do not document preconditions or postconditions, only functions behavior, their returns and their parameters. You should add normal C comments in your code to explain your implementation if they are relevant.

[**R14**] your C code must follow the code conventions explained during the lecture.

5.4 Requirements about your repository

[**R15**] you must commit your work at each successful functionality implementation. An intelligible message should be added to the commit.

[**R16**] you must follow the following directories convention: your `.c` files must be in the `src` directory or in the `tests` directory for tests and your `.h` files in the `include` directory. You may put the `.o` and executable files wherever you want. See section C for more details.

6 Grading

Your final grade will be calculated as presented on table 3. **Beware, your tests will be used to evaluate your work!**

Part	Details	Grade	Comments
Basic functionalities			
	map management	2	
	loader	1	
	maze	1	
	maze printer/ PPM printer	2	
	Prim's algorithm and app.	3	
Using a heap			
	heap definition and tests	3	
	Prim's algorithm using a heap	2	
Code quality			
	overall quality	4	variables naming, documentation etc.
	tests quality	2	
	your code does not compile	-5	
	your code produces errors with valgrind	-2	maximum malus, it depends on the number of errors
	incorrect use of Git	-1	not useful commit messages etc.

Table 3: Grade details

A successful extension implementation will give you up to 4 points **if the basic functionalities are correctly implemented**.
Notice that you may have a really good grade even if you do not implement all the functionalities: **the quality of your code is really important!**

You are not allowed to use chatbots like ChatGPT or Copilot to generate code or tests for your project.
Your code will also be analyzed by JPlag [2], a code plagiarism detector. DO NOT TAKE CODE FROM ANOTHER STUDENT, EVEN IF YOU TRY TO DISGUISE IT.
If you are convinced of plagiarism or use of forbidden software, you will obtain the "R" grade for the whole course.

A Testing functions

In this section, I will present how to write tests for your project. If you want to test a function or a property for the project, you should

- create a file whose name begins with `test-` in the `tests` directory
- add a rule to build the corresponding executable in your Makefile and add the test to the global `launch-tests` variable (see section B)
- use the C `assert` macro to verify properties
- define one or several functions, each one corresponding to a specific test or property
- call these functions into your `main` function
- use simple `printf` commands to indicate that the tests are satisfied

You will find in your `tests` directory a `test-dummy.c` file with two dummy tests about integer arithmetics that can serve as an example. There is a corresponding rule in your Makefile to build the executable.
If your test needs external information such as a filename, use `main` arguments (cf. section D). Do not add the test to the `launch-tests` variable of your Makefile.

B Makefile

In order to compile your applications and tests, you should use the Makefile provided in your repository. Here are some explanations:

- the `doc` rule generates the Doxygen documentation from your `.h` files. The documentation is located in the `doc` directory
- the `clean` rule removes all generated `.o` files and executables
- the C files in the `src` and `tests` directories can be automatically compiled. For instance, if you want to produce the `.o` file from `src/my_file.c`, just execute the following command:

  ```
  make my_file.o
  ```

- you should put all the `.o` file names that can be generated from your `.c` files as prerequisites of the `check-syntax` rule. Therefore, executing `make check-syntax` should compile all your C source files
- you should create a compilation rule for each of your applications or tests (see examples). You can use the `.o` files as prerequisites as the provided rules generate them from your `.c` files. Do not forget to add the needed header files as prerequisites
- you should add all your executables (applications and tests) as prerequisites of the `compile-all` rule. Therefore, executing `make compile-all` should generate all your applications and tests
- you should add all your test executables in the `ALL_TESTS` variables. Therefore, executing `make launch-tests` should execute all your tests.

  Beware: do not add tests that need an argument on the command line

- the `OPT` variable can be used to add the `-O3 -flto` options to GCC (and thus optimize aggressively your code). To use it, call `make` with `OPT=1` as for instance

  ```
  make compile-all OPT=1
  ```

- you can put debug messages in your code using `printf`. In order to avoid "polluting" the execution of your applications, you can encapsulate these messages in a preprocessor directive:

  ```
  #ifdef DEBUG
  printf("this is a debug message...\n");
  #endif
  ```

When compiling your applications, all code between the `#ifdef DEBUG` and `#endif` directives will be wiped except if you use `make` with `DEBUG=1` as for instance

```
make compile-all DEBUG=1
```

- the `git-add-all-files` rule add all `.c`, `.h` and `.txt` files that are in the expected directories. Do not hesitate to use it!

C Repository organization

Your repository is organized as follows:

- the `include` directory contains your header files (`.h` files)
- the `src` directory contains your C source files, except tests
- the `tests` directory contains your C test files
- the `doc` directory contains the documentation generated by Doxygen

You should compile your files using the provided Makefile (see section B).
You must include a `README.txt` file at the root of your repository to give some explanations on your project, particularly how to run executables that need command-line arguments.

D Passing parameters to a program from command line

You have seen in session M0 that you can pass parameters to a Python program using the `sys.argv` list. There is a similar mechanism in C when you use the following signature for `main`:

```
int main(int argc, char *argv[])
```

`argc` represents the length of `argv` and `argv` is an array of strings containing the parameters given on the command line and the name of the executable in position 0.
Let us take an example: let us suppose that you have such a `main` function in a C file compiled to a `example-main` executable. Then, when executing

```
./example-main hello world
```

- `argc` has the value 3, as there is two parameters
- `argv[0]` is `./example-main`, the name of the executable
- `argv[1]` is `hello`
- `argv[2]` is `world`

You will find in your `src` directory an `example-main.c` file that prints these informations when executing the corresponding executable.

E File Input/Output in C

If you want to read or write data to a file in C, you should use the structures and functions provided by the `stdio.h` header file. We will present here only simple cases when wanting to read or write formatted data from or to a text file. You may find more information in [4] or manual pages for the functions that are described in the following.

E.1 Opening and closing files

In order to read from files (or write to files), you must first open them. `stdio.h` provides a type, `FILE`, to represent a data stream from or/and to a particular file. Let us suppose that we want to read the content of a file `data.txt`. To open the file, we will use the `fopen` function and get a pointer to a `FILE` object:

```
FILE *p_file = fopen("data.txt", "r+");
```

Some remarks:

- `"data.txt"` represents the path to the file, it is here a relative path (the file `data.txt` must be in the current directory).
- `"r"` is the mode of the stream and specifies what you want to do with the file. You will mainly use the following modes:
  - `"r"` to read a file
  - `"w"` to write to a file
  - `"r+"` to read and write to a file
- if there are some errors when opening the file, the returned pointer is `NULL`. You should therefore test `p_file`.

When you have finished working with a file, **you should close it.** This is particularly true when writing to a file, as the file may have been put in central memory and the modifications you have done to the file might not be commited by the operating system (if you want to know more, you may refer to [1]). To close a file, simply call the `fclose` function on your pointer:

```
fclose(p_file);
```

E.2 Reading from files

E.2.1 Reading formatted input

If your file contains only **readable characters** and you know that data will follow a given format, then you may use the `fscanf` function. `fscanf` behaves like the `scanf` function we have seen in lab M1, but it works on files instead of standard input.
Let us look at the `fscanf` signature. It needs:

- a `FILE *` pointer to the file you want to read from
- a **format string** in the same format than the `printf` function
- several pointer variables to store what you read

Let us suppose that we want to read **`int`** values from a file with the following format:

```
2 - 4
4 - 12
5 - 42
13 - 2323
```

Each line of the file has two **`int`** values separated by the string `" - "`. Let us suppose that `first_value` and `second_value` are **`int`** variables in which we want to store the read values, then the call to `fscanf` may be:

```
fscanf(p_file, "%d - %d", &first_value, &second_value);
```

The format string is `"%d - %d"`: we expect an integer, then a space, then `-`, then a space, then another integer.
`fscanf` will return the number of input items successfully matched and assigned to the variables. Therefore, if `fscanf` returns 2, then the line has been correctly read and the variables assigned.

You may wonder how you may know when the file has been completely read. The `EOF` special value is returned by `fscanf` when it encounters the end of the file.

When using `fscanf`, the pointer `p_file` changes! After calling for instance the previous instruction, `p_file` will then "point" after the two integers. Therefore, you should not reuse directly `p_file` supposing that it points at the beginning of the file.

You will find in the `src` directory of your repository a simple program in the `read-file-formatted.c` file that tries to print all lines of a file respecting the previous syntax (cf. listing 1). You can compile it with the following Makefile command:

```
make read-file-formatted
```

You can execute it on the file `data-toread-formatted.txt` from your `data` directory with the following command[1]:

```
./read-file-formatted data/data-toread-formatted.txt
```

Try it on different inputs to verify that it works correctly, particularly when there are errors in the data file (see `data-toread-error.txt` in the `data` directory).

Listing 1: A simple program to read formatted text files

```c
/**
 * @file read-file-formatted.c
 *
 * @brief Simple program to explain how to read
 *        formatted data from file
 *
 * @author C. Garion
 *
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    int first_int     = 0;
    int second_int    = 0;
    int line_nb       = 1;
    int fscanf_result = 0;

    fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);

    while (fscanf_result != EOF) {
        if (fscanf_result != 2) {
            fprintf(stderr, "Line number %d is not syntactically correct!\n",
                    line_nb);
            exit(EXIT_FAILURE);
        }
```

[1] `data-toread-formatted.txt` must be in the `data` directory in the command, but you can use relative or absolute paths to call `read-file-formatted` on files that are not in the current directory.

```
        printf("first value at line %d: %d\n", line_nb, first_int);
        printf("second value at line %d: %d\n", line_nb, second_int);

        line_nb = line_nb + 1;
        fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

E.2.2 Reading text files

If you want to read simple text files[2], do not use `fscanf` but rather the `fgets` function. `fgets` takes as input:

- a `char *` string `s` that is used as a buffer, i.e. `s` will contain after the call to `fgets`
- an `int` value `size` that indicates the number of characters to be *at most* read
- a `FILE *` argument that represents the file to be read

Notice that `fgets` stops when encountering a newline or the `EOF` character representing the end of the file. Notice also that you should know the maximum length of the lines of the file in order to have correct `s` and `size` arguments. `fgets` will return NULL when encountering the end of the file.
Let us take an example and consider the following file:

```
Sing, O goddess, the anger of Achilles son of Peleus, that brought
countless ills upon the Achaeans. Many a brave soul did it send
hurrying down to Hades, and many a hero did it yield a prey to dogs
and vultures, for so were the counsels of Jove fulfilled from the day
on which the son of Atreus, king of men, and great Achilles, first
fell out with one another.
```

We know that the length of each line will not be more than 80 characters, therefore we could read the file and prints each line as presented on listing 2. Compile the executable with the corresponding Makefile rule and use it on the `data/iliad.txt` file in your repository.

Listing 2: A simple program to read text files

```
/**
 * @file read-file-text.c
 *
 * @brief Simple program to explain how to read
 *        text lines from file
 *
 * @author C. Garion
 *
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");
```

[2] Or part of a file that contains only text

```
    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    // declaration of a buffer of 81 chars: we suppose that each line
    // of the file has no more than 80 chars and we add 1 char for the
    // final '\0' character
    char buffer[81];
    int  line = 0;

    // while not having encountered end of file, read lines
    while (fgets(buffer, 80, p_file) != NULL) {
        line++;

        printf("%2d: %s", line, buffer);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

E.3 Writing to files

If you want to write formatted content to a file, you must first open the file with the `"r+"` or `"w"` modes with `fopen`. You then use the `fprintf` function. It behaves like the `printf` function, but it takes as first argument a `FILE *` pointer to the file you want to write to.
For instance, the program presented on listing 3 writes the factorial of the 10 first natural numbers to the file `fact.txt`. The program is available in the `src` directory of your repository and a compilation target is provided in your Makefile.

Listing 3: A simple program to write some computations in a text file

```
/**
 * @file write-fact.c
 *
 * @brief Simple program to explain how to write
 *        text into a file
 *
 * @author C. Garion
 *
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *p_file = NULL;

    p_file = fopen("fact.txt", "w");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot write to file fact.txt!\n");
        exit(EXIT_FAILURE);
    }
```

```c
    int fact = 1;

    for (int i = 0; i < 10; i++) {
        fprintf(p_file, "%d! = %d\n", i, fact);

        fact = fact * (i + 1);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

References

[1] Ulrich Drepper. "What Every Programmer Should Know About Memory". Nov. 21, 2007. URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[2] Karlsruhe Institute of Technology. *JPlag – Detecting Software Plagiarism*. 2016. URL: https://jplag.ipd.kit.edu/.

[3] R. C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[4] Wikibooks, ed. *C Programming, File IO — Wikibooks, The Free Textbook Project*. Wikimedia Foundation. Apr. 16, 2020. URL: https://en.wikibooks.org/wiki/C_Programming/File_IO (visited on 11/29/2023).

[5] Wikipedia, The Free Encyclopedia, ed. *Loop-erased random walk*. Wikimedia Foundation. Jan. 4, 2024. URL: https://en.wikipedia.org/wiki/Loop-erased_random_walk (visited on 02/02/2024).

[6] Wikipedia, The Free Encyclopedia, ed. *Maze generation algorithm*. Wikimedia Foundation. Jan. 3, 2024. URL: https://en.wikipedia.org/wiki/Maze_generation_algorithm (visited on 12/04/2023).

[7] Wikipedia, The Free Encyclopedia, ed. *Netpbm format*. Wikimedia Foundation. Jan. 29, 2024. URL: https://en.wikipedia.org/wiki/Netpbm_format (visited on 11/29/2023).

[8] Wikipedia, The Free Encyclopedia, ed. *Prim's algorithm*. Wikimedia Foundation. Nov. 15, 2023. URL: https://en.wikipedia.org/wiki/Prim's_algorithm (visited on 12/04/2023).

License CC BY-NC-SA 3.0