Projet 3 - Groupe A1

Aydin Matya 37592100
Dallemagne Brieuc 77122100
Lebras Floriane 35022100
Mounir Samy 46422100
Van Hees Charles 35562100
Verschuere Louise 33772100

I. Introduction

Within the framework of project P3, we were tasked with implementing, in the C language and based on existing Python code, the Bellman-Ford shortest path algorithm in a weighted directed graph. Initially, this was done in a *single-threaded* manner. We now present our implementation in a *multi-threaded* environment.

This report begins with a commentary on the program we implemented. Following that, we compare the time, memory, and energy consumption of our C language implementation with the provided Python code. After this comprehensive overview, we present the enhancements made to the algorithm to reduce instant memory and time consumption.

II. Description of the C Code

A. Code Structure

The code was developed using four modules :

— `sp` : contains the `main` function as well as functions and structures related to the program's input arguments ;
— `file` : contains functions related to file reading and closing ;
— `graph` : contains functions related to the Bellman-Ford algorithm ;
— `thread` : contains functions related to threads.

The source files for these modules are contained in the `src` directory, and the corresponding headers are in the `include` folder. In the `include` folder, three other headers are also present : `portable_semaphore.h` and `portable_endian.h` are necessary for the code to function on Apple, and `include_and_struct.h` imports libraries and defines structures used in multiple files. This last header is imported into the other files.
The `tests` directory contains the graphs used for testing, as well as the test code in the `tests.c` file.

B. Data Structure

Given that the C language does not allow a function to return multiple elements, we have defined structures in which we encapsulate the function's results. For example, the `bellman_ford` function takes a pointer to a `result_bellman_ford_t` structure as an argument, whose variables are `source, dist, path`, with `dist` and `path` being the results of the Bellman-Ford algorithm applied to the node `source`. We have followed a similar approach for `get_path` and `get_max`.

C. Multithreading

Initially, we opted for a producer-consumer model with a fixed-size buffer. Ultimately, we shifted to a model that uses a linked data structure of type *queue* because its operation aligns with the aspect we aimed to optimize : memory. Two semaphores are used to establish communication between the producer and the consumer. The first semaphore makes the consumer wait until there is an element in the queue. The second limits the size of the queue to restrict instant memory usage, as explained in the code improvement section. If an error is detected, all threads terminate, and memory is released.

1) Producer: The "producer" threads execute the Bellman-Ford algorithm from each source node, store the result in a node, and place it in the *queue*. Mutexes are used to ensure that all source nodes are processed only once and to prevent conflicts during node additions to the queue.

2) Consumer: The consumer thread executes a `dequeue` function that removes a node from the *queue*, applies the `get_path` and `get_max` functions to the node's content, and writes the results to the output file. Mutexes are used to prevent conflicts when taking an element from the queue, to determine when all source nodes have been processed, and to avoid overlapping results from different nodes during the output file writing.

$$\begin{tikzcd}
 & & T_1 \arrow[dr] & & \\
\boxed{\text{Entry}} \arrow[r] & \boxed{\text{Reading}} \arrow[ur] \arrow[r] \arrow[dr] & T_2 \arrow[r] & \boxed{\text{Queue}} \arrow[r] & \boxed{\text{Gp,Gm,write}} \\
 & & T_n \arrow[ur] & &
\end{tikzcd}$$

The writing of the output file must be done sequentially, so we chose to assign only one thread to this function. While we could have used multiple threads to execute the `get_path` and `get_max` functions, these two functions have minimal execution time compared to the rest of the code, and the execution time gain is not significant. The second mutex, which prevents conflicts during the writing of the output file, is therefore unnecessary, but we have kept it to easily increase the number of threads in this part if desired.

III. Unit Tests and Analysis Tools

A. Unit Tests

We used four graphs to verify the proper functioning of our code : a normal graph (without any particularities), a non-connected graph, a graph with a negative cycle, and a graph with all nodes originating from the same point. This last graph allows us to test that the lowest index is taken if multiple nodes are at the same smallest distance and to ensure that the edge with the lowest cost is chosen if two edges have the same sources and destinations. These four graphs together cover all possible situations. For some functions, it is not even necessary to check the correct operation on all graphs.

The four graphs are useful for the `bellman_ford` function. For the `get_path` function, the normal and non-connected graphs verify the result in a standard case and in a case where no node is accessible. For the `get_max` function, the normal graph and the one where all edges originate from the same node are sufficient to test a normal case, a case where a node is not accessible, and a case where multiple nodes are at the same greatest distance.

For the functions of the `file` module, we verify on a graph that the reading and writing of files are done correctly. Since these operations are independent of the graph, a single test is sufficient. Testing the output file also allows us to verify that the entire program execution works.

B. Valgrind

The Valgrind tool allowed us to verify that there are no memory leaks when we run the algorithm on each of the graphs, whether the result is displayed on the screen or saved in an output file. We also ensured that in the event of an error during a function call (malloc, fread, etc.), there is no memory loss.

C. Cppcheck

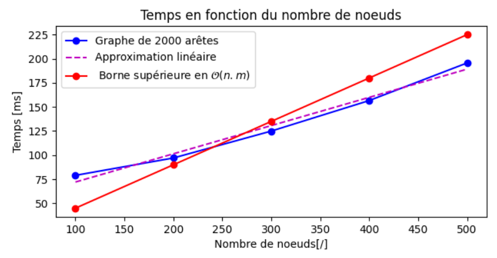
Cppcheck helped us improve the code for variables that might be used without always being initialized and to minimize the scope of variables as much as possible.

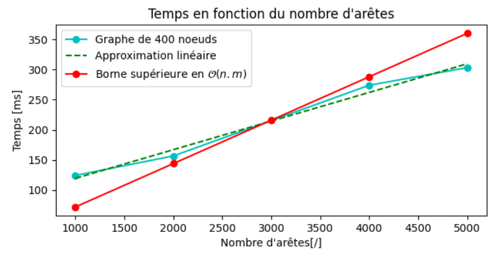
IV. Performance Analysis

A. C Code Compared to Python Code

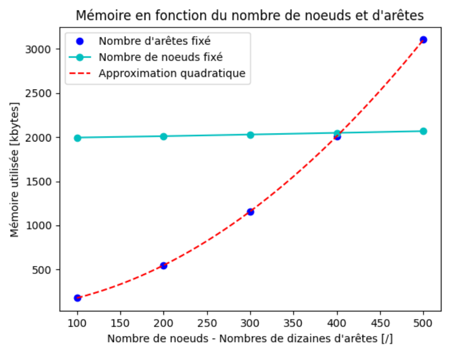
1) Execution Time: For a graph with 400 nodes and 2000 edges, the execution times are $57s$ for the Python code and $80ms$ for the C code. In the case of a graph of the same size but containing a negative cycle, the execution times are $66.32s$ and $77ms$ respectively. The execution of the C code is significantly faster.This can be explained by multithreading, the improvement made to the code described in the code improvement section, and the nature of the C language.

Note that the algorithm is in $\mathcal{O}(n.m)$ with n being the number of nodes and m being the number of edges (due to the iterations of the Bellman-Ford algorithm)





2) Memory Consumption: For a graph with 400 nodes and 2000 edges, the C code allocates $2020.59kB$, while the Python code uses $343.96kB$. The C code, therefore, allocates much more memory. However, as explained in the code improvement section, we decided to enhance the instant memory of the code. We could have reduced the total allocated memory by, for example, allocating memory to the `result_get_path_t` structure only once instead of allocating it at each iteration, but we preferred this second option to achieve our goal.



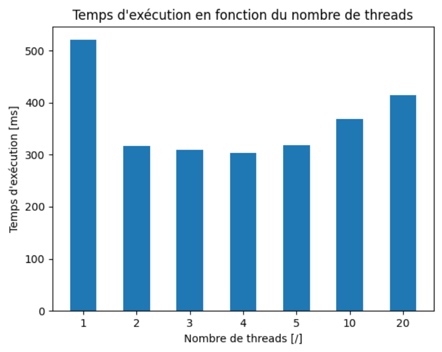
Note that memory evolves in $O(n^2)$ with n being the number of nodes. This quadratic evolution is due to the fact that the Bellman-Ford algorithm stores its results in two arrays of size n, and this algorithm is executed for each source node (n times). Edges are only stored in the graph, thus linearly affecting memory.

3) Energy Consumption: For a graph with 400 nodes and 2000 edges without a negative cycle, the power values are $3.4W$ and $2.6W$ respectively for the Python and C codes.

For a graph of the same size but with a negative cycle, the powers are $3.4W$ and $2.65W$. The execution of the algorithm in Python therefore uses approximately 24.5% more power. The execution of the Python code lasted 57 seconds and thus consumed $57 * 3.4 = 193.8J$. The execution of the C code lasted only 0.08 seconds and therefore consumed only $0.08 * 2.65 = 0.212J$. The C code consumes much less energy because its execution time is shorter.

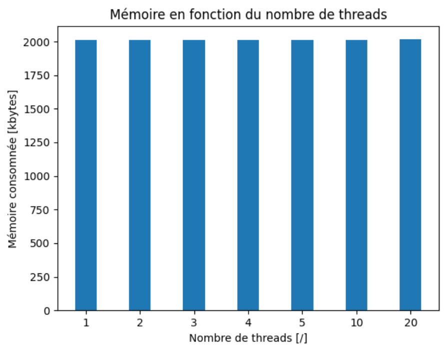
B. Evolution in Function of the Number of Threads

1) Execution Time:



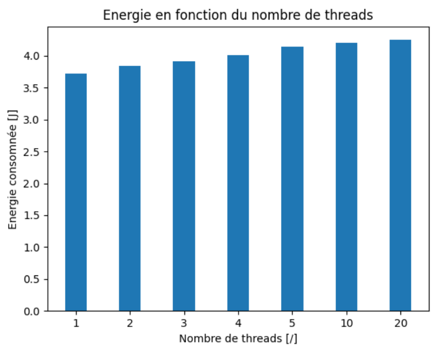
Going from one to two threads to execute the Bellman-Ford algorithm makes the execution approximately 1.72 times faster. However, adding too many threads decreases the execution speed. Indeed, the Raspberry Pi has only 4 cores. Above 4 threads, they must share the cores using a non-deterministic scheduler. Therefore, some threads are idle, waiting for their turn. The time spent creating a thread is not "profitable".

2) Memory Consumption:



As visible in the graph above, the total allocated memory during execution varies very little with the number of threads. Indeed, if we add a thread, only the memory allocated for the creation of that thread is added to the total memory. However, we will note in the code improvement section that the instant memory varies depending on the number of threads.

3) Energy Consumption:



Energy consumption grows little with the number of threads. Indeed, the execution time decreases, but the power increases, stabilizing the product. However, similarly to execution time, adding too many threads leads to overconsumption. The power increases, but the execution time no longer decreases.

V. Code Improvement

A. Instantly Allocated Memory

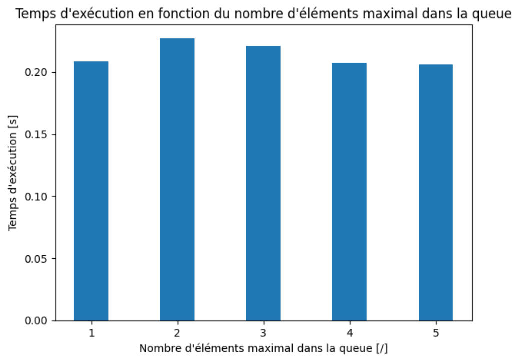
First, we optimized the instantly allocated memory. This means that we made the memory allocated by the code as minimal as possible at any point during execution. To achieve this, we used as many pointers as possible to avoid copying data. Furthermore, we only allocate memory when it becomes necessary and free it as soon as it is no longer needed. This slightly slows down the code due to numerous system calls, but it was the price to pay for having the most optimal instant memory.

The use of a queue between the producer and the consumer allows for a maximum of $n + 1 + 1$ `result_bellman_ford_t` structures to be allocated at the same time (with n being the number of threads executing the Bellman-Ford algorithm). The two 1s correspond to the `result_bellman_ford_t` structure stored in the unique node of the queue and the `result_bellman_ford_t` structure on which the thread responsible for `get_max`, `get_path`, and writing to the output file is working.

For each `result_bellman_ford_t` structure, there may also be memory allocated to a node because the memory of a node is allocated before knowing if there is space available in the queue and is freed after signaling that space is available in the queue. Another node may therefore enter the queue before the memory of the previous node is freed.

At any given moment, there may also be memory allocated to a maximum of one `result_get_max_t` structure and one `result_get_path_t` structure because there is only one thread that handles this part of the code. Additionally, memory is also allocated for the graph and threads, which are used in the majority of the code. Finally, memory is sometimes allocated for a short moment during file reading and writing. Instantly allocated memory thus evolves linearly with the number of threads.

We decided to limit the size of the queue to one element, but we tested what would happen if we allowed more nodes in the queue. As visible in the figure below, the execution time is approximately the same regardless of the queue size limit. This means that even if we allow the queue to have an infinite size, it remains close to one node. It is possible that two nodes enter the queue, but it is rare. The quantity of memory described above is indeed an upper bound on instant memory, and it is possible that this bound is not always reached. Limiting to one node in the queue is therefore not a significant restriction but provides an idea of the memory allocated instantly.

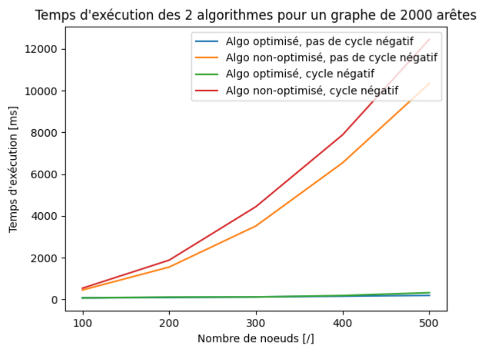


B. *Execution Time*

Subsequently, we also decided to improve the execution time of our algorithm. We made two modifications to the Bellman-Ford algorithm :

- For graphs without a negative cycle, we stop the iterations of `bellman_ford` if, during an iteration, the minimal cost at each node has not changed between the beginning and the end of the iteration.
- For graphs with a negative cycle, we stop the iterations of `bellman_ford` if the distance to the source node becomes negative at some point. Indeed, if it is negative, there is a negative cycle.

The complexity of the algorithm remains in O(n.m), but these two modifications help reduce the execution time in many "simpler" problems. The complexity provides an upper bound on the worst-case scenario.
The figure below compares the execution time between optimized and non-optimized algorithms, in the case of a graph without and with a negative cycle.



VI. Conclusion

This project has taught us C programming, both in single and multi-threading, as well as the use of tools such as Git and getting started with a Raspberry Pi.
The results of our measurements are well supported by theory, especially in terms of memory consumption and execution time. Our implementation of the Bellman-Ford algorithm in C is significantly faster than the one in Python, and we have optimized the instantly allocated memory.