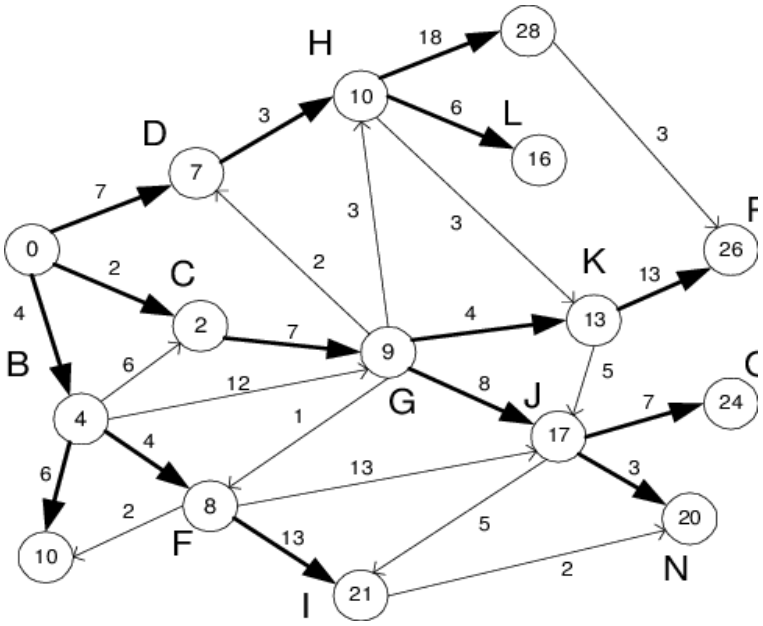
Ecole Polytechnique - Bachelor of Science

CSE305 Project Report

Parallel shortest paths



By Clémence Mottez and Garance Perrot

Supervised by:

Gleb Pogudin

Eric Goubault

May - June 2024

1 Introduction and research aim

Introduction Finding shortest distances in a graph is one the fundamental problems in computer science with numerous applications (route planning by CityMapper and Google/Yandex-maps, for example). We have already seen classical algorithms (BFS, Dijkstra, etc) for this task in our algorithm courses. However, these algorithms are inherently sequential and hard to parallelize (although parallel versions exist).

Aim Consider a graph $G = (V, E, c)$, with n vertices ($n = |V|$), m edges ($m = |E|$) where each edge is associated to a weight from the function $c : e \to R^+$ where $e \in E$. In our implementation, edge weights can be either integers or positive real-valued (respectively *int* or *double*, both designated by the typename T). A source node s is designated. The objective of the SSSP is to compute, for each vertex v of the graph, the weight of a minimum-weight ("shortest") path from s to v, denoted by $dist(s, v)$ and abbreviated to $dist(v)$. The weight of a path is the sum of the weights of its edges and we set $dist(u, v) := \infty$ if v is unreachable from u. Sequential shortest path algorithms commonly apply iterative labeling methods based on maintaining a tentative distance for all nodes: $tent(v)$ is always ∞ or the weight of some path from s to v and hence an upper bound on $dist(v)$ [1].

In this project, we implement and benchmark one of the most standard shortest path algorithms, namely the Δ-stepping algorithm. We implement it, run it on a set of benchmarks, and compare it with non-parallel algorithms including single-thread Δ-stepping and Dijkstra to evaluate the benefits of parallelization.

2 Different algorithms implemented

Dijkstra Dijkstra's algorithm is a popular method for solving the SSSP problem and widely used due to its efficiency and simplicity.

The algorithm is straightforward to understand and implement. It works by iteratively selecting the vertex with the minimum known distance from the source and updating the shortest path distances to its adjacent vertices. It uses a priority queue to efficiently fetch the next vertex with the smallest distance and updates distances for its neighbors if a shorter path is found through this vertex. The process repeats until all vertices have been visited and the shortest paths from the source to all other vertices are established. Yet, the sequential nature of the algorithm makes it hard to parallelize , which can be a limitation for large-scale graphs or applications needing parallel processing. Dijkstra's algorithm is optimal in the sense that it guarantees the shortest path, contrary to other SSSP algorithms such as Nearest Neighbor ([2]) that can lead to suboptimal paths by getting stuck in local minima.

Dijkstra's algorithm demonstrates a runtime complexity of $O((m+n)log(n))$. Indeed, adding n vertices to the priority queue as well as extracting the minimum element (and updating its neighbors) at most n times both cost $O(nlog(n))$, and checking edges between connected vertices (in the worst case all m edges, but typically less in sparse graphs) adds $O(mlog(n))$ to the complexity. Dijkstra is particularly efficient for sparse graphs which have relatively few edges compared to the number of vertices ($m \ll n$).

However, as the number of edges m approaches n^2 (a dense graph), the benefit of the priority queue diminishes. In the worst case, where all vertices are connected, it might even become comparable to the basic $O(n^2)$ approach. Other limitations of this algorithm are its incapacity to handle graphs with negative edge weights (for those graphs, the Bellman-Ford algorithm ([3]) is more appropriate), and its need to maintain data structures for the graph can be memory-intensive for very large graphs.

Single thread Delta stepping Δ-stepping algorithm is a parallelizable approach to solving the SSSP problem for graphs with positive edge weights. It is a distance-correcting algorithm, meaning it iteratively refines tentative distances for vertices until they converge to the actual shortest paths, making it an optimal algorithm like Dijkstra's algorithm.

The algorithm segments the graph's vertices into "buckets" based on their tentative distance from the source, using a parameter Δ to define each bucket's range. The process involves repeatedly selecting and processing the bucket with the smallest current tentative distances, updating the distances of neighboring vertices if a shorter path is found. Specifically (see Fig.1 for reference), at initialization, an infinite distance is assigned to all nodes except the source node, which gets a distance of 0. Buckets are created to hold nodes with tentative distances within a range of $[dist(v), dist(v) + \Delta]$. In each iteration, the first non-empty bucket is processed. The outgoing edges of all nodes in that bucket are considered for relaxation, meaning each neighbor's tentative distance is updated if the path through the current node is shorter. This update may move the neighbor's tentative distance to a different bucket range, marking it for future processing. The algorithm continues until all buckets are empty, indicating that no further distance updates are needed and the shortest paths have been found.

The main advantage of this algorithm is its ability to process nodes from a bucket independently, leading to faster computation on multi-core or distributed systems. We explain our parallelization method in the next section.

In general, Δ-stepping is an efficient algorithm with complexity influenced by multiple factors, providing good average-case performance. In the worst case, the number of iterations might be proportional to the number of edges in the graph ($O(m)$). As the number of edges increases, more iterations may be needed due to frequent distance updates and shifts of eligible nodes between buckets, increasing runtime. The parameter Δ also affects complexity: a smaller Δ creates more buckets and potentially more accurate updates but increases the number of iterations, while a larger Δ reduces iterations but might result in less precise initial distance estimates, requiring more overall work. The algorithm's complexity also depends on the type of edge weights. It performs well on graphs with random edge weights, typically assigned from distributions like uniform or normal distribution in our implementation. Random edge weights reduce the likelihood of chain reactions of updates, which is crucial for parallel processing.

However, Δ-stepping algorithm has some limitations. Its complexity is less predictable than Dijkstra's, and it is not suitable for dense graphs. Finding the optimal value of the Δ parameter can be challenging as it heavily depends on the graph structure. Additionally, the algorithm performs best on random edge weights, which might not always represent real-world scenarios (e.g., road networks often have distance-based weights), and it does not support negative edge weights.

Parallelized Delta stepping Parallelization can be achieved in various ways in the Δ-stepping algorithm, generally reducing runtime. For graphs with random edge weights, theoretical results suggest an expected time complexity of $O(log^3 n)$ using the Parallel Random Access Machine (PRAM) model. This means the number of iterations grows slowly with the number of vertices in the graph, even for large graphs, as noted in [1]. The authors also suggest that deletion and edge relaxation for an entire bucket can be done in parallel and in arbitrary order, provided that each individual relaxation is atomic (i.e., the relaxations for a particular node are done sequentially). After experimenting with multiple versions of parallel implementations, we decided to implement a dynamic workload assignment where nodes are dynamically allocated to threads during each bucket processing. More specifically, in general the following tasks were carried out in parallel:
- Generating light and heavy request sets for each node u in a bucket as a set of pairs $(v, w) \in E$ consisting of the neighbour v and the connecting edge weight w, used later for edge relaxation, i.e the algorithm sets $tent(w) := min\{tent(w), tent(v) + c(v, w)\}$.
- Relaxing light edges, i.e edges such that $c(e) \leq \Delta$ where $e \in E$.
- Relaxing heavy edges, i.e edges such that $c(e) > \Delta$.

In the *findRequestsPar()* routine, the bucket is divided into chunks processed by multiple threads. We use a single mutex to protect shared data across threads, specifically the current bucket. Similarly, the *relaxRequestsPar()* function relies on multiple threads, each managing a portion of the requests, with a mutex safeguarding concurrent access to buckets.

However, creating threads is resource-intensive and sometimes inefficient when the bucket has low occupancy. Therefore, we included cases to adjust the number of threads based on the current bucket's size for *findRequestsPar()* and the total number of buckets for *relaxRequestsPar()*. For instance, if the current bucket's size is less than $10 \times nb\_threads$, finding requests is not parallelized. Additionally, if there are at most two buckets in total, relaxing requests is performed with a single thread. For the current bucket's processing, the number of threads is limited to the bucket's size, and similarly, for edge relaxation, the number of threads is adjusted to be at most the total number of buckets. The processes of finding and relaxing requests are managed independently, as well as operations on light and heavy edges, allowing an optimal combination of thread use and their number for the overall algorithm.

```
foreach v ∈ V do tent(v) := ∞
relax(s, 0);                                          (* Insert source node with distance 0      *)
while ¬isEmpty(B) do                                  (* A phase: Some queued nodes left (a) *)
    i := min{j ⩾ 0: B[j] ≠ ∅}                         (* Smallest nonempty bucket (b) *)
    R := ∅                                            (* No nodes deleted for bucket B[i] yet    *)
    while B[i] ≠ ∅ do                                 (* New phase (c) *)
        Req := findRequests(B[i], light)  -> parallelization   (* Create requests for light edges (d) *)
        R := R ∪ B[i]                                 (* Remember deleted nodes (e) *)
        B[i] := ∅                                     (* Current bucket empty    *)
        relaxRequests(Req)  -> parallelization        (* Do relaxations, nodes may (re)enter B[i] (f) *)
    Req := findRequests(R, heavy)  -> parallelization (* Create requests for heavy edges (g) *)
    relaxRequests(Req)  -> parallelization            (* Relaxations will not refill B[i] (h) *)

Function findRequests(V', kind : {light, heavy}) : set of Request   -> V' divided into chunks each processed by a dif. thread
                                                                       one mutex shared by all threads
    return {(w, tent(v) + c(v, w)): v ∈ V' ∧ (v, w) ∈ E_kind)}

Procedure relaxRequests(Req)  -> Req divided into chunks each processed by a different thread, one mutex shared by all threads
    foreach (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                                 (* Insert or move w in B if x < tent(w) *)
    if x < tent(w) then
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}         (* If in, remove from old bucket *)
        B[⌊x /Δ⌋] := B[⌊x /Δ⌋] ∪{w}                   (* Insert into new bucket *)
        tent(w) := x
```

Figure 1: Pseudo-code of our implementation of Δ-stepping following paper [1], along with our parallelization operations

In order to reduce the computation cost, we have attempted to implement fine-grained granularity for edge relaxation in parallel. Instead of locking the entire *dist* and *buckets* structures, we have considered finer-grained locks by the means of multiple mutexes for updating distances. Indeed, if a single mutex locks the entire distance vector for any update, there is a risk of bottlenecks if multiple threads try to access the same large chunk of data concurrently. With multiple mutexes based on buckets, only a portion of the data (distances within a specific range) would get locked, minimizing the impact on other threads working on different parts of the vector. This would improve efficiency and reduces unnecessary waiting for unrelated updates. However, an attempt of this approach required the use of a vector of mutexes which size was difficult to handle. Allocating a large number of mutexes slowed down the program due to memory overhead, but it crashed if there weren't enough mutexes.

In the end, using one mutex per bucket improved the runtime for some graphs but failed for other graphs, so we prefered to keep a coarse-grained implementation for Δ-Stepping in Parallel.

Another attempt of improvement was to implement a static workload assignment where the nodes are assigned to threads at the beginning of the algorithm and only the assigned thread can relax edges leading to a node. It has been proven to be more efficient than dynamic workload assignment that we implemented, according to paper [4]. The program managed to generate light and heavy requests but crashed during edge relaxation due to a misuse of thread management, which we failed to debug.

3 Comparison of the algorithms

Effect of delta The Delta Stepping algorithm's performance can significantly vary based on the choice of the delta parameter and the weight distribution in the graph. To recall, this parameter defines the bucket width. Each bucket contains nodes whose tentative distance from the source lies within the range of previous bucket's maximum distance and the maximum distance plus delta. Nodes are moved from bucket to bucket as shorter paths to them are discovered.

A smaller delta results in more buckets, which can mean more iterations and a larger delta reduces the number of buckets but increases the range of weights in each bucket, potentially increasing the work per bucket.

We studied the impact of delta for different weight ranges in the context of a graph with 10,000 nodes and 100,000 edges. The results are presented in Fig.2.

With weights uniformly distributed between 1 and 5, we can see that a small delta (around 3) is the most effective here. This allows for closely grouping nodes with similar path lengths, making the algorithm efficient by minimizing the number of relaxations per node.

With weights uniformly distributed between 1 and 50, a moderate delta (around 15) could be a balance between the number of buckets and the range of weights in each bucket.

With weights uniformly distributed between 1 and 100, a large delta (around 25) is needed because of the broader range of edge weights.

The uniformity of the weight distribution also has an impact. For example, when the weights are normally distributed with a mean of 25 and a standard deviation of 1, most edge weights cluster very closely around the mean. Choosing a delta close to the mean, like 25, aligns well with the predominant weight distribution, which allows each bucket to efficiently process edges that have similar weights. This alignment reduces the need for frequent updates and reprocessings across buckets, optimizing the overall runtime of the delta stepping algorithm.

Effect of weights On the one hand, when all edge weights are the same, Dijkstra's algorithm effectively simplifies to Breadth-First Search (BFS). This scenario might reduce the number of updates required in the priority queue, thus potentially speeding up the execution slightly. However, the performance can degrade if the graph has highly varying edge weights. This is because the priority queue operations become more costly due to frequent updates of distances.

On the other hand, for the Parallel Delta Stepping algorithm, if the edge weights are uniform, the algorithm can achieve high parallelism because many vertices can be processed simultaneously without frequent synchronization. However, if the weights vary significantly, it might lead to performance issues due to load imbalance among processors. Some processors might end up processing more vertices (heavier buckets) while others are idle. The granularity of bucket size (delta) also becomes crucial; an inappropriate delta can either cause too much work per step or too many synchronization steps.

These thoughts are confirm by the following graphs, see Fig.3.

Effect of graph density We measured the graph density as the ratio of the number of edges to the number of vertices in the graph. The results are presented in Fig. 4.

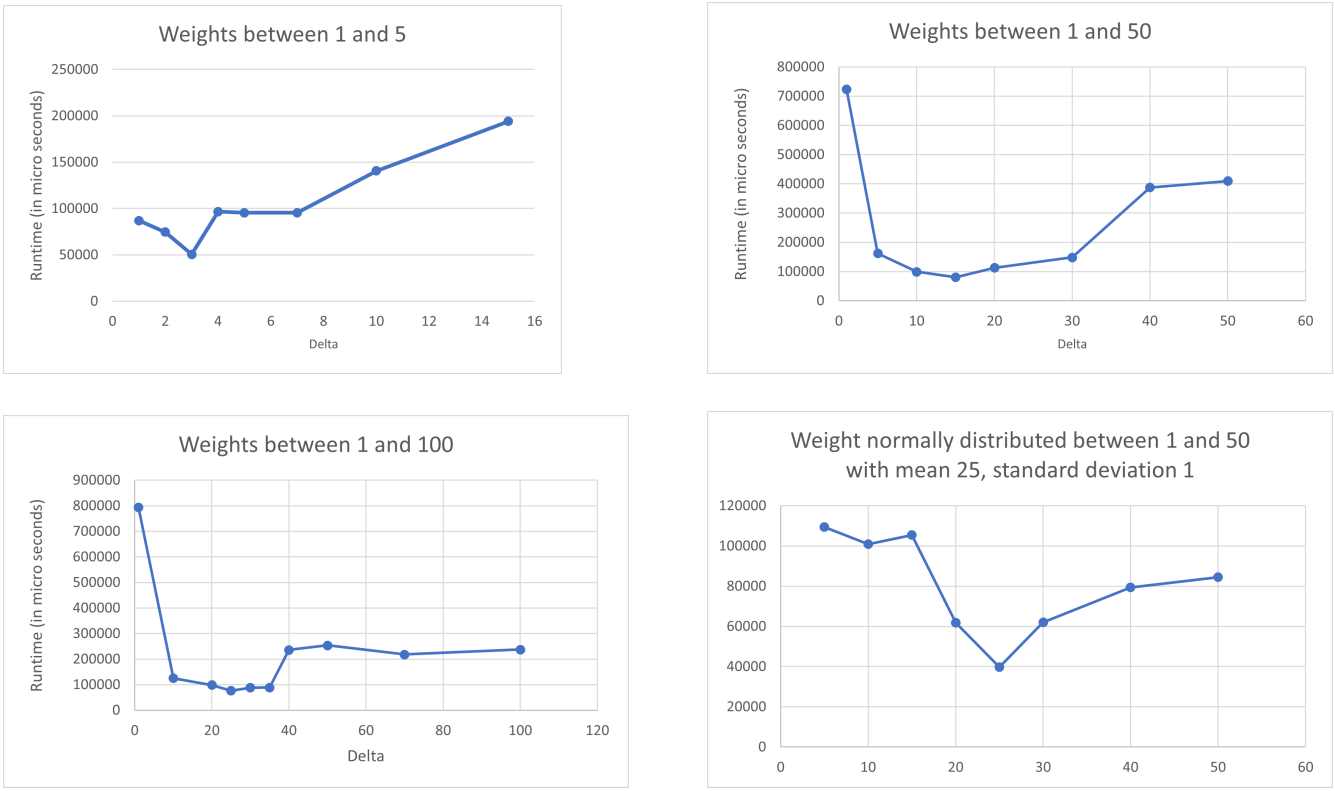


Figure 2: Effect of delta on runtime according to different weight distributions. The graphs are randomly generated with 10 000 nodes and 100 000 edges.

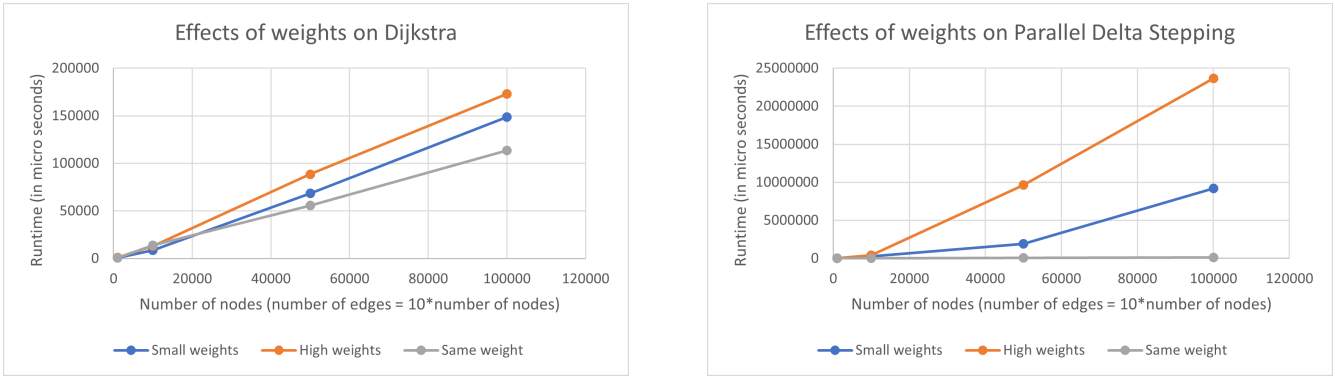
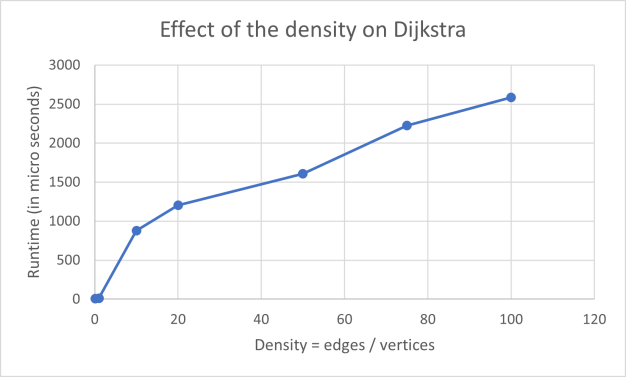


Figure 3: Effect of weights on runtime. Same weights: all the weights are equal to 1; Small weights: weights are randomly distributed between 1 and 5; Large weights: weights are randomly distributed between 1 and 100.

For Dijkstra's Algorithm, as the density increases, the runtime also increases. This pattern is expected because Dijkstra's algorithm has a time complexity that is influenced by both the number of vertices and the number of edges. It is using a min-priority queue, which complexity is $O((n+m)\log n)$. Therefore, as the number of edges increases, the amount of work required per vertex increases, leading to longer runtimes. The graph illustrates a nearly linear growth in runtime with density, showing that as edges increase, the computational load increases correspondingly. For Delta Stepping parallelized version, the runtime initially increases sharply as the density increases. However, beyond one point (around a density of 20), the increase in runtime stabilizes as density increases further. This indicates that the delta stepping algorithm, especially in its parallel form, manages to handle the increased edge count more efficiently than

the traditional Dijkstra's algorithm at higher densities. It is not surprising since the parallel processing enables handling multiple edges concurrently, which mitigates the impact of increased density on runtime.

On the one hand, Dijkstra's algorithm shows a consistent increase in runtime with increasing density, reflecting its sensitivity to the number of edges in the graph due to the higher computational cost per vertex as edges increase. On the other hand, Delta Stepping Parallel algorithm runtime stabilizes at higher densities, suggesting that parallel processing effectively counters the runtime penalty of a higher edge count. It highlights the benefits of using parallel algorithms for graph computations, particularly in dense graphs where the overhead associated with managing more connections can be distributed across multiple processors.



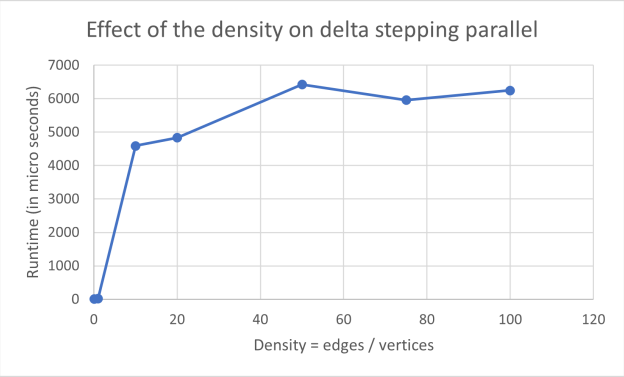


Figure 4: Effect of density on runtime. With number of nodes = 1000, density = number of edges / number of vertices.

Effect of number of threads We also studied the effect of the number of threads on the runtime performance of the parallel version of our Delta Stepping algorithm for large and dense graphs (1000 vertices for 100 000 edges) with different weight distributions. The results are presented in Fig. 5.

For each graph, we can see a sharp decrease in runtime as the number of threads increases from 0 to around 5-10 (the exact number of threads at which this occurs varies depending on the weight characteristics). After this initial decrease, the runtime stabilizes and shows minor fluctuations but generally maintains a consistent level up to 20 threads (and even more but it is not shown here). It suggests that the parallel version of Delta Stepping efficiently utilizes multiple threads to handle computations. The plateau indicates that adding more threads beyond one point does not significantly contribute to performance gains, likely due to overhead or limits in parallelism efficiency for this particular algorithm and graph structure.

4 Which algorithm to choose?

Dijkstra We initially compared the runtime performance of the three algorithms as a function of the number of nodes in the graph. The number of edges is scaled linearly with the number of nodes, (ten times the number of nodes), indicating a constant density (of 10) as the graph size increases. The results are presented in Fig. 6. For this type of graph, we can clearly see that Dijkstra's Algorithm exhibits by far the best performance among the three, indicating its suitability for graphs where the constant density does not excessively penalize its single-threaded approach. Moreover, the Delta Stepping parallel version significantly improves performance over the serial version, showcasing the benefits of parallel processing in handling larger graphs more efficiently. However, both Delta Stepping versions are less efficient than Dijkstra's algorithm in this test scenario, highlighting the importance of algorithm choice based on graph characteristics.

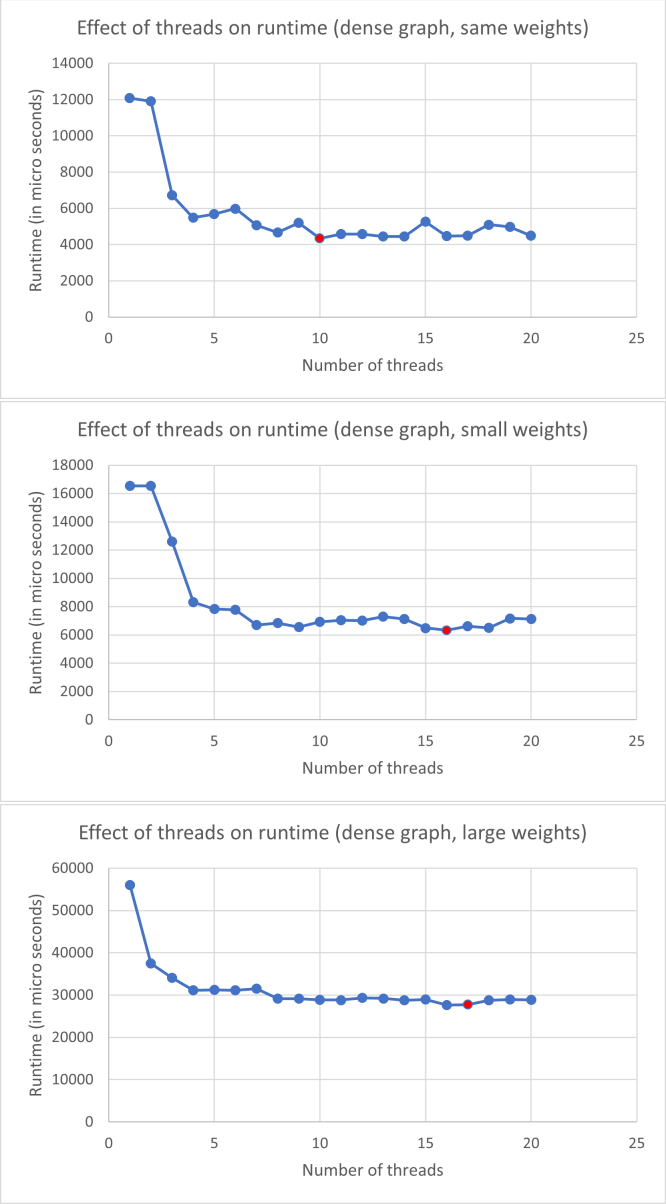


Figure 5: Effect of the number of threads on runtime according to different weight distributions. The graphs are randomly generated with 1 000 vertices and 100 000 edges. The red dot indicates the lowest runtime.

Parallel Delta Stepping We unfortunately did not manage to beat the Dijkstra algorithm (or only on very rare random graphs). However, we did manage to approach its runtime closely. Based on our previous analysis, we conclude that the parallel version of the Delta Stepping algorithm excels in handling high-density graphs, especially those with uniform or minimal variations in edge weights, and having a large number of vertices.

For instance, we studied a graph having 5 000 vertices and 500 000 edges with all edge weights set to 1. (You can run *"./test 2 5000 500000 1 1 0 0 0 1000 0 0"* if you want to test it). Here, we generally achieve a speedup of 0.8 relative to Dijkstra's algorithm. While this performance is slightly inferior, it is relatively close! And reaches more than 1 for some runs! Moreover, we observed a significant speedup of around 2.7 compared to the sequential version of Delta Stepping. This demonstrates that while the parallel Delta Stepping may not always surpass Dijkstra's efficiency, it significantly outperforms its sequential counterpart and shows potential in specific graph scenarios. For the same graph but with

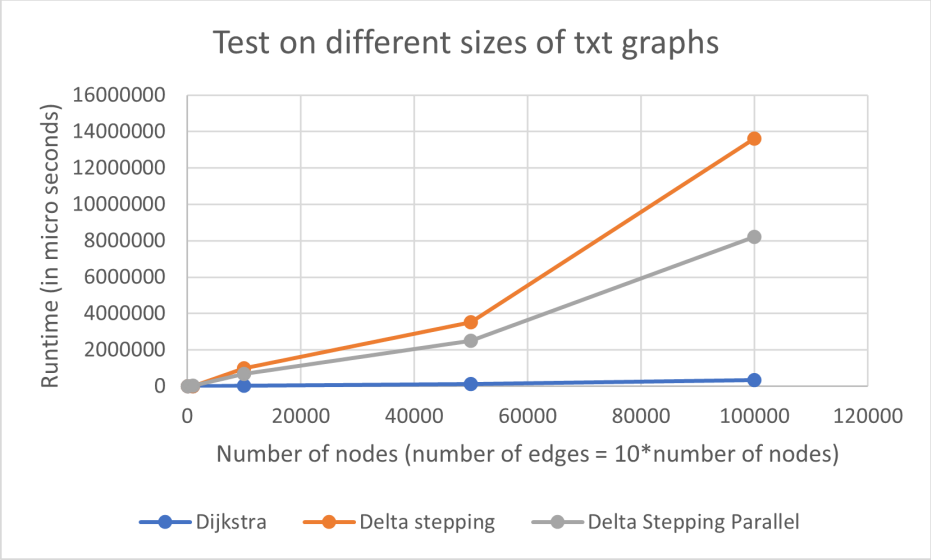


Figure 6: Scenario where Dijkstra is better

weights randomly distributed between 1 and 5, we generally only have a speed up of 0.3 compared to Dijkstra, and of 1.7 compared to the sequential version of Delta stepping.

5 Conclusion

In this research study, we have conducted a thorough comparison of Dijkstra's algorithm, the sequential version of Delta Stepping, and its parallel counterpart across various graph configurations. The findings reinforce the notion that the parallel version of Delta Stepping consistently outperforms the sequential version in all tested scenarios. Specifically, we observed that the runtime decreases progressively as the number of threads increases, highlighting the benefits of parallel computation in efficiently managing the computational demands of larger graphs.

While Dijkstra's algorithm remains the superior choice for most graph types due to its robust performance across a wide range of conditions, the parallel Delta Stepping algorithm shows promise under specific circumstances. It sometimes matches or even surpasses Dijkstra's performance, particularly in graphs that are large, densely connected, and have uniformly low weights due to handling multiple nodes concurrently.

6 Limitations

Practically, as mentioned in Section 2, many parameters can significantly affect the algorithm's performance. For instance, the best choice of Δ, the optimal number of buckets, and the optimal number of threads depend on the graph's structure, weight distribution, and the implementation itself. To address this, our implementation includes functions to determine these values: *graph.findDelta()* , *graph.nb_buckets(delta)*, and *graph.suggestOptimalNumberOfThreads()*.

Another limitation is that the runtimes used for analysis in Sections 2 and 3 depend on the machine on which the algorithms are run. The data was gathered on our local machine (Config A), but we obtained slightly better results on the machines in the computer rooms (Config B).
Config A Intel(R) Core(TM) i3-1005G1 processor with 1200MHz CPU, 7.8 GoB memory, 1 NUMA node, 1 socket with 2 cores.

Config B 13th Gen Intel(R) Core(TM) i7-13700K processor with 5400MHz CPU, 124 GiB (130 GoB) memory, 1 NUMA node, 1 socket with 16 cores.

In the following, we describe the problems that we have encountered during our implementation.

Logical Complications First, the definition of the discrimination between light and heavy edges varies across research papers. In paper [1], for $e \in E$ light edges have weight at most Δ ($c(e) \leq \Delta$) and heavy edges are such that $c(e) > \Delta$, while in paper [4] light edges are only those with $c(e) < \Delta$ and the remaining edges are heavy. We have chosen to work with the first definition.

In our implementation, we offer the option to launch algorithms on random graphs generated by the *gen_random_graph()* function. Initially, this function allowed the generation of cyclic graphs, which caused several issues. SSSP algorithms rely on the assumption that all nodes will eventually be reachable with a finite distance from the source node. In a cyclic graph, however, a node may be part of a cycle where edges repeatedly connect nodes, leading the algorithm to revisit the same nodes indefinitely. This prevents the algorithm from finalizing distances for all nodes.The algorithm might become stuck in an infinite loop, continuously relaxing distances on the cycle without ever converging to a solution, or it might terminate with inaccurate distance values. Therefore, we revised our function to prevent the generation of cyclic graphs.

Coding Obstacles It is crucial to choose appropriate data structures for storing necessary information. After various trials, we opted for the *std::deque* data structure to represent buckets. This choice enables efficient insertion and removal of elements at both ends (front and back) without requiring memory reallocation for the entire structure, which is important for the algorithm. Specifically, in *findRequests()*, new nodes with updated distances need to be added to the back of a temporary container for later processing. Using a standard vector would be less efficient because inserting at the back necessitates shifting all subsequent elements, becoming increasingly costly as the vector's size grows. A deque also benefits from dynamic resizing, unlike fixed-size lists. An earlier version of the code used lists to represent buckets, but resizing them increased computation costs, especially when the optimal number of buckets was not used. For example, having too few buckets could lead to overflow, leadind to frequent resizing.

In some situations, theoretically allowed operations do not behave correctly in practical programming. A notable example is the handling of infinite distances in the *relax* routine. In an earlier version of the code, to relax node v with its neighbor u, we created a variable $newDist = dist(u) + c(u, v)$ before comparing it to $dist(v)$ to decide which distance to keep (the smallest one). However, adding a positive number to INT_MAX (which represents ∞ in the code) results in a negative number due to the fixed-size, two's complement integer representation in C++. Consequently, the distance comparison behaved oppositely, causing the program to crash for some graphs. This problem was resolved by adding a control line $if(dist(u) \neq INT\_MAX)$ before conducting the comparisons.

7 Github

The source code of this project is published at this GitHub: `https://github.com/clemence-mottez/CSE305-Project-Parallel-shortest-paths`.

8 Contribution of each team member

We mostly worked together on both the code and the report. You can find specific contributions bellow:

Clémence Mottez Set up the Graph class, wrote the initial single thread delta stepping function, parallelize delta stepping (fine grained granularity and adaptation of the number of threads according to the emptiness of the buckets), wrote a function to find the number of buckets, find the optimal delta and the optimal number of threads. Introduction, comparison part, "which algorithm to choose" part in the report.

Garance Perrot Implemented Dijkstra, a function to compare the different algorithms, parallelize delta stepping, worked on integer and double weights, modified delta stepping algorithms so that they correspond exactly to the pseudo-code from paper [1]. Description part of the algorithms and limitations in the report.

REFERENCES

[1] P. Sanders U. Meyer. Delta-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, August 2003.

[2] A. Vandana N. Bhatia. Survey of nearest neighbor techniques. *(IJCSIS) International Journal of Computer Science and Information Security*, 8(2), 2010.

[3] Prof. C. Baggar V. Patel. A survey paper of bellman-ford algorithm and dijkstra algorithm for finding shortest path in gis application. *International Journal of P2P Network Trends and Technology (IJPTT)*, 4(1), Feb. 2014.

[4] Erika Duriakova, Deepak Ajwani, and Neil Hurley. Engineering a parallel -stepping algorithm. pages 609–616, 2019.