# CSE305 FINAL PROJECT REPORT

Marcel Chwiałkowski, Anca Sfia, Bruno Iorio

In this project we explore the Delta-stepping algorithm for solving the single-source shortest path (SSSP) problem in graphs. The report firstly describes the basic Delta-stepping algorithm and its variations (static and dynamic implementations), and presents certain established results on its performance. Later, we take a look at the structure of our project – the algorithms that we implemented, helper functions for generating random graphs and scripts for analyzing results. Finally, we present the results of our experiments on the performance of different variants of Delta-stepping on different graphs and parameters, and compare it to regular serial Dijkstra.

**Contributions to the project were distributed among the team members as follows:**

1. Marcel Chwiałkowski – Dijkstra, serial Delta-Stepping, parallel Delta-stepping (static variant)
2. Anca Sfia – Serial and parallel Delta-stepping (dynamic variant)
3. Bruno Iorio – Benchmarking

## 1 Introduction

The Delta-stepping algorithm is an easily parallelizable solution to the SSSP problem. It principle, it works similarly to Dijkstra's algorithm - both algorithms maintain an array `dist` of tentative distances, which keeps the current distance of each vertex from the source throughout the runtime of the algorithm. At initialisation, each distance is set to $+\infty$, and throughout the runtime, as the algorithms scan through edges, new shorter paths are found and the tentative distances relax, finally yielding an array of exact shortest distances between vertices and the source. However, Dijkstra's algorithm goes vertex-by-vertex: it keeps a priority queue of unvisited nodes sorted by the lowest tentative distance, and at each iteration it pops the first node of the queue, and relaxes all the edges going out from it. This presents a challenge for parallelization, since subsequent relaxations depend on each other, which makes it difficult to perform several relaxations at once,

In comparison, Delta stepping maintains a set of buckets $[B_1, B_2, \dots]$ such that the bucket $B_i$ holds unvisited vertices with tentative distance between $[i \times \Delta, (i+1) \times \Delta)$. Then, the buckets are emptied sequentially: starting from the non-empty bucket with the smallest index, all the edges coming out of vertices from this bucket are relaxed until the bucket is empty. This involves dividing the edges into heavy and light edges so that each light edge has weight

less than $\Delta$, and each heavy edge has weight of $\Delta$ or more. For a bucket $B_i$, firstly, all the light edges are relaxed at once - but since they have a weight of less than $\Delta$, this means that some vertices can be inserted into $B_i$, thus this step is reiterated until $B_i$ is empty. Afterwards, all heavy edges are relaxed at once - since their weight is more than $\Delta$, the relaxed vertices will only be inserted to buckets with index larger than $i$.

This phase is repeated, at each time emptying the non-empty bucket with the smallest index - this smallest index grows at each step, as it is impossible to for example when processing $B_i$, insert a vertex into $B_{i-1}$. The algorithm eventually terminates when all the buckets are empty.

Pseudocode for the $\Delta$-stepping algorithm (taken from CITE HERE) can be seen below. Each bucket emptying phase consists of three loops: Loop 1 empties the current bucket, while generating requests for relaxations of heavy and light edges (that is, saving to corresponding buffers, that a vertex should be updated to a different tentative dstance). Loop 2 reads the relaxation requests for light edges, and performs them. Both 1 and 2 are locked in a while loop, as relaxations to light edges can result in reinsertions to the current bucket, so the while loop ensures that in the end the bucket is empty. Afterwards, loop 3 performs relaxation requests for heavy edges.

**Algorithm 1** $\Delta$-stepping SSSP

**Require:** $G(V, E)$, source $s$, bucket width $\Delta$

**for all** $v \in V$ **do**
  define $N_v^\ell$ and $N_v^h$
  $d^\star(v) \leftarrow \infty, \quad B_\infty \leftarrow \{v\}$
**end for**
$B_\infty \leftarrow B_\infty \setminus \{s\},\ B_0 \leftarrow B_0 \cup \{s\},\ d^\star(s) \leftarrow 0$
$k \leftarrow 0$
**while** $k < \infty$ **do**
  $R_h \leftarrow \emptyset,\ R_\ell \leftarrow \emptyset$
  **while** $B_k \neq \emptyset$ **do** ▷ **Emptying buckets**
    **for all** $v \in B_k$ **do** (sequentially or in parallel) ▷ Loop 1
      $B_k \leftarrow B_k \setminus \{v\}$
      $R_\ell \leftarrow \textsc{GenRequests}(R_\ell, N_v^\ell)$
      $R_h \leftarrow \textsc{GenRequests}(R_h, N_v^h)$
    **end for**
    **for all** $(v, u, w) \in R_\ell$ **do** (sequentially or in parallel) ▷ Loop 2
      $R_\ell \leftarrow R_\ell \setminus \{(v, u, w)\}$
      $\textsc{Relax}(v, u, w)$
    **end for**
  **end while**
  **for all** $(v, u, w) \in R_h$ **do** (sequentially or in parallel) ▷ Loop 3
    $R_h \leftarrow R_h \setminus \{(v, u, w)\}$
    $\textsc{Relax}(v, u, w)$
  **end for**
  $k \leftarrow \min_i\{\, i : B_i \neq \emptyset \,\}$
**end while**
**function** $\textsc{GenRequests}(R, N_v)$
  **for all** $u \in N_v$ **do**
    $R \leftarrow R \cup \{(v, u, w(v, u))\}$
  **end for**
  **return** $R$
**end function**
**function** $\textsc{Relax}(v, u \in V,\ w \in \mathbb{R})$
  **if** $d^\star(v) + w < d^\star(u)$ **then**
    $i \leftarrow \left\lfloor \frac{d^\star(u)}{\Delta} \right\rfloor, \quad j \leftarrow \left\lfloor \frac{d^\star(v)+w}{\Delta} \right\rfloor$
    $B_i \leftarrow B_i \setminus \{u\},\ B_j \leftarrow B_j \cup \{u\}$
    $d^\star(u) \leftarrow d^\star(v) + w,\ p(u) \leftarrow v$
  **end if**
**end function**

**Ensure:** $d^\star(v),\ p(v) \quad \forall\, v \in V$

In contrast to Dijkstra, $\Delta$-stepping is easily parallelizable - when emptying each bucket, the order of relaxation doesn't matter, so each loop can be executed by multiple threads.

## 1.1 When does Δ-stepping excel?

Δ-stepping is especially effective when:

- The graph is large, so synchronization overhead is amortized.
- Edge weights are uniformly distributed, enabling large buckets and high parallelism.
- Graph degrees have a balanced distribution, so that each thread has a similar amount of work to process in each loop - this prevents other threads from waiting for one slow one to finish).
- The graph diameter is low, so few buckets must be processed and threads do not idle.

## 1.2 Theoretical guarantees

The actual work depends heavily on the choice of $\Delta$ and the edge weight distribution. As $\Delta \to 0$, Δ-stepping becomes more sequential (like Dijkstra), and as $\Delta \to \infty$, it approaches Bellman-Ford (more parallelism, but greater total work).

Let $n$ be the number of vertices, $m$ the number of edges, $L$ the maximum shortest-path distance from the source, and $\delta$ the bucket width parameter. Define:

- $n_\delta = |C_\delta|$: number of pairs of nodes connected by a $\delta$-path.
- $m_\delta = |C_{\delta+}|$: number of triples $(u, v, w)$ where $u, v$ are connected by a $\delta$-path and $(v, w)$ is a light edge.
- $d$: maximum degree.
- $\ell_\delta$: maximum number of edges in a $\delta$-path plus one.

For an arbitrary graph with arbitrary positive edge weights:

$$\text{Time} = O\left(n + m + \frac{L}{\delta} + n_\delta + m_\delta\right)$$

For arbitrary graphs with random positive edge weights and $\delta = O(1/d)$:

$$\text{Time} = O(n + m + dL)$$

## 1.3 Challenges

### 1.3.1 Choosing the optimal Δ

Choosing the right value of $\Delta$ has a huge impact on the performance of our algorithm. Notice that for graphs with integer weights, choosing $\Delta = 1$ effectively makes the Δ-stepping perform Dijkstra's algorithm - with no reinsertions to the same bucket, but little room for parallelism. Similarly, choosing a large $\Delta$, makes the algorithm equivalent to Bellman-Ford - which is easily parallelizable, however it works in $O(mn)$. Therefore, $\Delta$ should be chosen to balance between available parallelism and number of reinsertions to one bucket. Paper CITE HERE shows that in certain cases, for example for random graphs with edge weights distributed uniformly, choosing $\Delta$ proportional to $\frac{1}{\text{Maximal degree}}$ is optimal, however in practice finding the right $\Delta$ is non-trivial.

#### 1.3.2 Choosing the right number of threads

Ideally, choosing the number of threads close to our device's number of cores should be optimal - however, as we will see in the experiments, this is not always the case. For smaller graphs or graphs with skewed degree distributions (like road graphs), running more threads is less efficient, as each thread has less operations to process, or an imbalanced number of operations to process - and this overhead cancels out the benefits of parallelization.

# 2 Our implementations

In our project, we implement:

1. Two variants of parallel $\Delta$-stepping: parallel $\Delta$-stepping with static node partitioning and parallel $\Delta$-stepping with static node partitioning and adaptive $\Delta$ (we call it dynamic $\Delta$-stepping, however it is different from the dynamic node partitioning mentioned CITE HERE).

2. serial Dijkstra for comparison.

## 2.1 Static Parallelization - Invariant $\Delta$

In this approach, upon initialization, the graph nodes are assigned randomly to the threads. At each execution of `loop 1`, `loop 2`, `loop 3`, each thread processes only these vertices which are assigned to it - and the lists of requests are stored in 2-dimensional buffers, such that the requests at $R[fromID][toID]$ are written by thread $fromID$ in `Loop 1`, and read by thread $toID$ in `Loop 2` and `Loop 3`. This configuration avoids any race conditions - firstly, writing and reading are never occuring at the same time, and secondly when writing, each thread accesses the $R$ buffer at a different $fromID$ and when reading, each accesses the $R$ buffer at a different $toID$. This allows us to implement $\Delta$-stepping lock-free.

## 2.2 Dynamic $\Delta$

In the dynamic variant, the workload is distributed in the same way between the threads. However, $\Delta$ can be updated throughout the runtime of the algorithm.

Similar to the static version, vertices are grouped into buckets based on their tentative distances. For each bucket, light edges are processed first, repeatedly relaxing them. If the algorithm spends too many relaxations on light edges in the same bucket, it is interpreted as $\Delta$ being too small. The width of the buckets is doubled and all the buckets are rebuilt, automatically adapting $\Delta$ based on the observations made.

The smallest non-empty bucket index is maintained in a min-heap for efficiency when rebuilding buckets. The buckets allow for fast insert/erase.

## 2.3 Design choices

Here we describe and justify some of the non-obvious design choices that we made when implementing the algorithms:

1. Thee requests buffers are defined as:
`std::vector<std::vector<std::vector<std::pair<int, double>>> ReqLight, ReqHeavy;`
Thus, each list of requests passed from thread $A$ to thread $B$ is a `std::vector` - as we only need to iterate through it in any order, we don't need a different data structure.

2. The buckets are defined as:
`std::vector<std::vector<std::unordered_set<int>> buckets;` where each bucket is distributed between the threads. The portions of the buckets belonging to each thread are implemented as `std::unordered_set`, as we need $O(1)$ insertion and deletion (not necessarily from the back!). In paper CITE HERE this is implemented with doubly linked lists, however due to time constraints we opt for sets, which have the same theoretical complexity.

3. The threads contain elements from `C++20`, namely `std::barrier`. We found it to be the easiest way to parallelize $\Delta$-stepping: instead of recreating every thread at each execution of `Loop 1`, `Loop 2` and `Loop 3`, we crete each thread at the beginning of the algorithm, and synchronize them at each phase with the barrier - for example, main cannot start `Loop 2` until each thread finishes `Loop 1`.

# 3 Benchmarking

## 3.1 Dataset

As part of our investigation, we benchmarked the performance of the delta stepping algorithm on a collection of road networks of large cities in the United States. These tests may provide us information of real applicabilities and limitations of choice of implementation of our algorithms

## 3.2 Random Graph Generators

We use a Random Graph Generator selects $m$ edges by uniformily choosing 2 different nodes $m$ times, in a graph with $n$ nodes. As it is shown in the code below.

```
void wideWeightRandomGraph(int n, int m, double min_w, double max_w,
   Graph& G) {
    G.n = n;
    G.adj_lists.assign(n, {});
    G.maxDist = 0;
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<int> node_dist(0, n-1);
    std::uniform_real_distribution<double> weight_dist(min_w, max_w);
    std::set<std::pair<int, double>> edges;
    while (edges.size() < (size_t)m) {
        int u = node_dist(gen), v = node_dist(gen);
        if (u == v) continue;
        int a = std::min(u, v), b = std::max(u, v);
```

```
14          if (!edges.insert({a, b}).second) continue;
15          double w = weight_dist(gen);
16          G.adj_lists[a].emplace_back(b, w);
17          G.adj_lists[b].emplace_back(a, w);
18          G.maxDist += w;
19      }
20  }
```

We will consider for most of our tests graphs of size $2^{17}$, and density $c = \frac{m}{n} = 32$. Note that as $c$ grows, the graph becomes less sparse. We denote a random graph with $n$ nodes and density $c$ as $RG(n, c)$.

### 3.3 Experimental Setup

For this project, we used Ecole Polytechnique computers to run our tests. Specifically, all tests were ran in the `ain.polytechnique.fr` login.

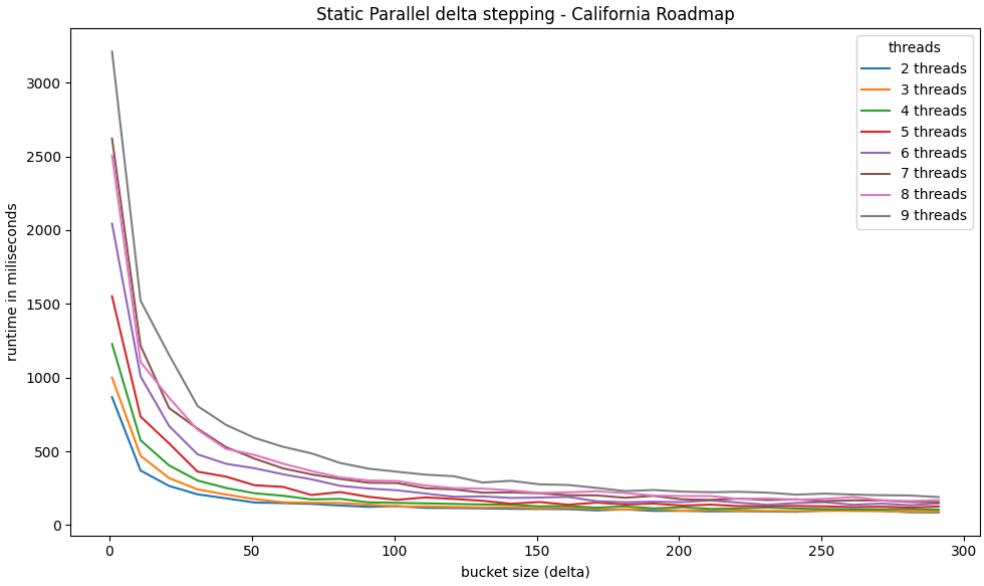
We perform the following tests for both road network graphs and $RG$s for our algorithms:

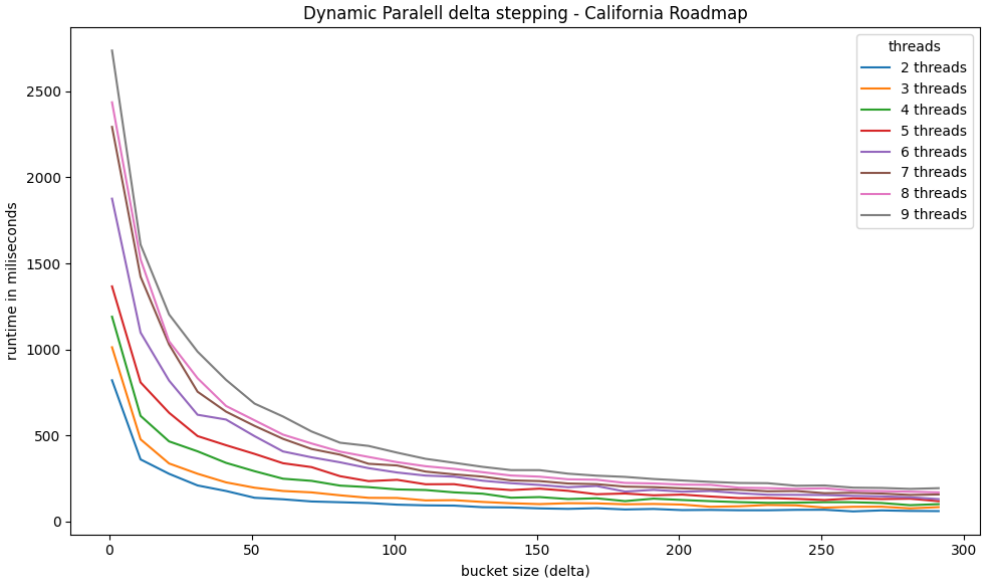
- tests over the values of delta
- tests over the number of threads
- tests over the density of the graph (only for $RG$)

We finally, will compare the performance of each implementation of our algorithm with Dijkstra's algorithm.

## 4 Results

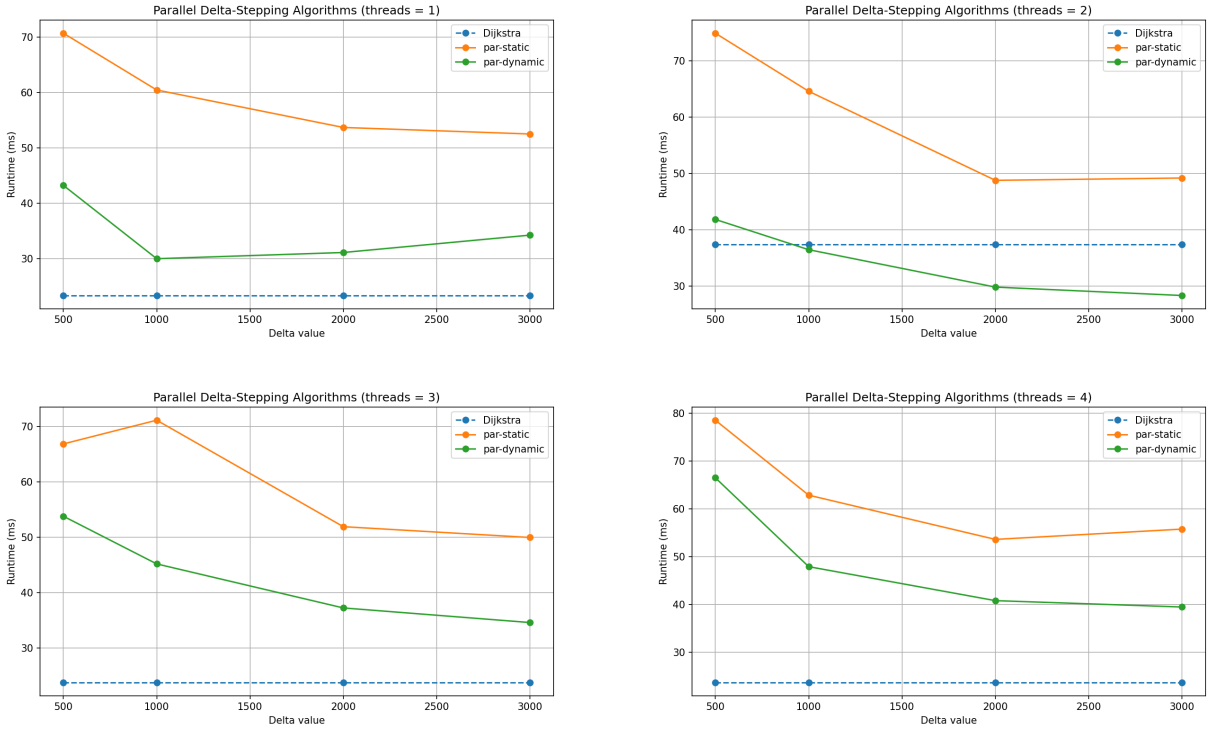
Firstly, we will present the performance of each Algorithm under the road network of the state of California. This network contains 1,890,815 nodes and 4,657,742 edges.





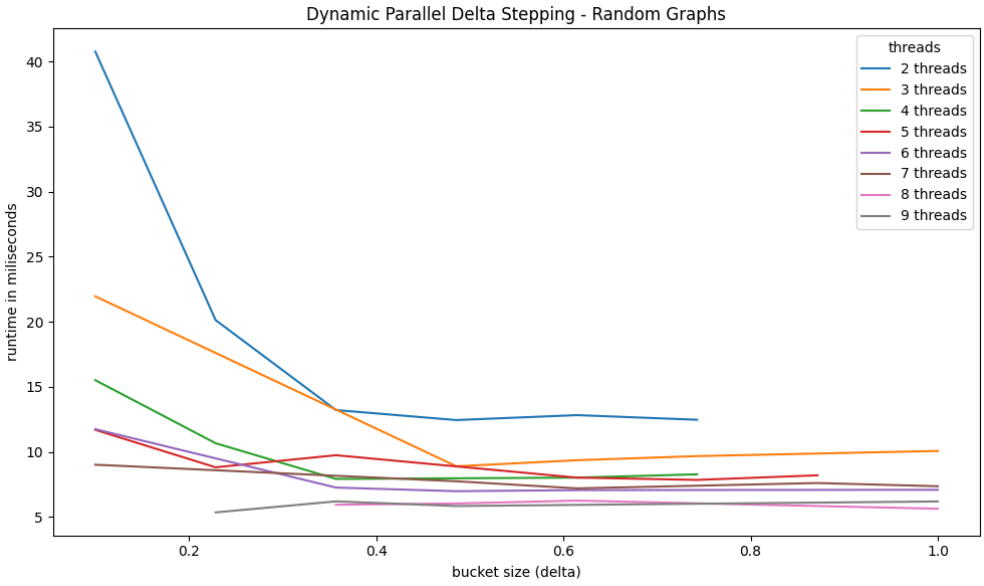

Rysunek 1: Delta stepping on Road network

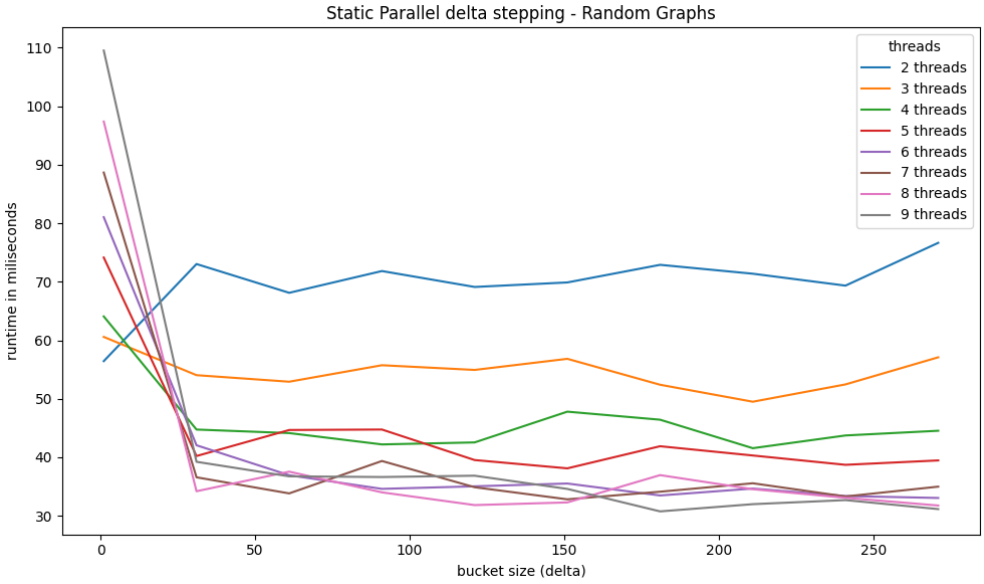
For the road network graph consisting of the bay area, the delta stepping algorithms best perform on 2 threads likely due to synchronization overload since the planar graph format does not allow for good parallelism. Notably, the dynamic delta stepping algorithm can outperform Dijkstra in the best case.



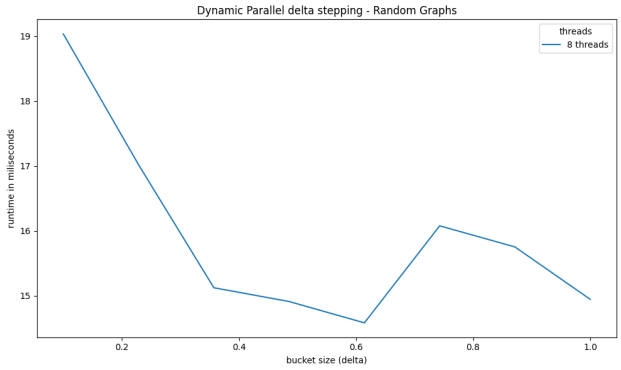

Rysunek 2: Performance on the BAY area graph (N = 321270, M = 800172)

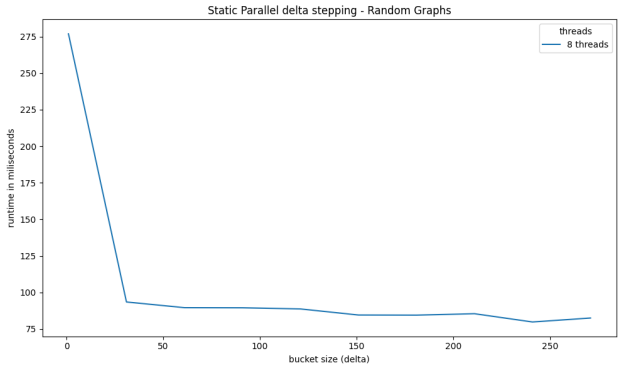
Now considering $RG(n, c)$, where $n = 2^{17}$ and $c = 32$. We note a very different trend on the performance of our algorithms.




In this case, more threads actually were positively affecting the performance of the model, in terms of speed. However, the static variant required large values of delta, whereas the dynamic variant required values between 0 and 1. So, we considered 8 threads for both.




We found $\Delta = 0.6$ in the case of the dynamic variant and $\Delta = 200$ in the case of the static variant. Under these ideal parameters, we evaluate the algorithms while varying the density of the graph $RG(2^{17}, c)$.