Università di Genova | **DIBRIS** DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

Parallel solvers for shortest paths and distance fields on graphs

Nicholas Gazzo

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy
http://www.dibris.unige.it/

Master Thesis

Università di Genova

Laurea Magistrale in Computer Science
MSc in Computer Science
Data Science and Engineering Curriculum

Parallel solvers for shortest paths and distance fields on graphs

Nicholas Gazzo

Advisor: Prof. Enrico Puppo,
Prof. Daniele D'Agostino

Examiner: Prof. Giorgio Delzanno

March, 2023

Abstract – Finding the shortest path and/or distances from single source vertex to all the vertices in a graph is a common problem in graph analytics. The Dijkstra's algorithm, a known reference approach for these problems, offers the most efficient computation complexity, however it doesn't expose any kind of parallelism across vertices. The Bellman-Ford algorithm instead, despite a more redundant work, offers a high degree of parallelism, which can be exploited by modern GPU architectures to improve the performances on large graph structures.

This thesis presents a comparison, at the state-of-art, between the available GPU graph analytics libraries, and their parallel implementation of the Bellman-Ford algorithm, over the sequential algorithms that runs on the CPU. Our experiments focus on monitoring how the speedup of the parallel algorithm is affected by changes in the graph metrics (e.g., a higher average degree/graph density) for specific groups of graph datasets, such as: random generated graphs and mesh graphs. The experimental results show that, except for particular cases, the most notable speedup values are always obtained on larger graph structures.

Table of Contents

Chapter 1 Introduction

Graphs are everywhere, many systems of interest are composed of parts or components connected in some way. Examples can vary from: technological networks (the Internet, the telephone network, power grids, roads, and rail networks), social networks (people connected by acquaintance or social interaction), information networks (the World Wide Web, citation networks), biological networks (biochemical networks, neural networks) and many others.

Recently the attention on these structures has been focused on their use for machine learning, where graph neural networks (GNN), which are neural networks that directly operates on a graph structure, are used for applications like node classification.

However, many aspects and problems of these systems are worthy studying, one of which includes rapidly compute shortest paths and/or distances in the most efficient way possible. For example, we may want to calculate the shortest driving route from a point A to a point B via a road network, or like many routers do, we might want to calculate the route across the Internet that gets out our data packages to destination in the shortest time.

The goal of this thesis is to evaluate the performances at the state-of-the-art of the parallel algorithms which use modern GPUs on this field. It can seem odd to talk about GPUs associated with irregular data structures like graphs, however if we think about what the most common graph representation on computers are, sparse adjacency matrices, we can realize why GPUs are such a useful choice for the topic.

1.1 Thesis's Structure

The thesis is structured as follows:

- On **Chapter 2***,* we start by covering all the theoretical notions, regarding graph structures, needed to understand the consequent experiments. We then continue by exploring what are the most common graph representations for computers. Followed by the shortest path problem definition and the most notable algorithms used for its solution.
- On **Chapter 3**, we cover the related work done by others on the field of parallel solutions for the shortest path problem.
- On **Chapter 4**, we discuss the libraries that we took in consideration during the experiment. Covering for each of them the most relevant information on the programming model and the implemented graph primitives.

- On **Chapter 5**, we cover a more technical description on how the environment for the experiments was prepared. Covering all the software installation needed and how they were performed.
- On **Chapter 6***,* we provide a description of the scripts developed during the thesis, while also going over the most relevant parts of the code used for the testing during the experiments.
- On **Chapter 7***,* we cover all the experimentation done with the libraries, discussing what graph type we choose, what graph metrics we considered, and the results obtained.
- Lastly, on **Chapter 8** we give our conclusions and covers the future work.

Chapter 2 Background

2.1 Graph Theory

2.1.1 Directed and undirected graphs

A graph (also called network) is, in its simplest form, a collection of objects where some of the pairs are "related" in some way. In graph theory these objects are referred to as *vertices* or *nodes* and the connection between each pair of vertices is referred to as an *edge*.

Mathematically speaking we can define a graph as a pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \{(i,j) | (i,j) \in V^2 \text{ and } i \neq j\}$ is a set of edges, defined as pairs of distinct vertices. This first definition only takes into consideration what we usually call *simple directed graphs*. *Directed* because we consider the edge (i,j) directed from the vertex i to the vertex j and different from its inverted edge (j,i). And *simple* because it does not allow *multi-edges*, which are multiple edges starting from the same vertex i and directed to the same vertex j, nor it allows *self-edges*, which are edges starting from a vertex i and directed to the same vertex i.

Undirect graphs instead are graphs where the edges are symmetric, meaning that an edge (i,j) represents both a connection from vertex i to vertex j as well as a connection from vertex j to vertex i.

2.1.2 Weighted graphs

In some situations, it is useful to represent edges as having a weight (also called strength or cost) value associated with them, which is usually a real number. This can be used to represent an additional layer of information, as for example in a social network environment, where the vertices are the users and edges represent their friendship, the weight associated with the vertices can be used to express the frequency of contact between the users. Formally we can express the weights associated with edges as a function $f: E \to \mathbb{R}$, that takes an edge and return a real number.

Weights are usually positive numbers, but nothing prevent the use of negative numbers. Keeping the previous example of a social network, positive weights associated with edges could be used to express a cordial relationship, where negative weights instead could represent animosity between the users.

2.1.3 Edge list and adjacency matrix

There are a lot of different ways of representing graphs mathematically. If we consider an undirected graph with n vertices (using integers label $1 \dots n$ to identify the different vertices) we could denote edges between two vertices i and j as the pair (i,j). Then using this we could express the entire graph just by giving the value n and list of all edges $(i_1, j_1), (i_2, j_2), \dots, (i_m, j_m)$. This is what we usually refer to as *edge list*.

Another common way of representing graph is through their *adjacency matrix* (or *adjacency lists*). An adjacency matrix is a square matrix A of size $n \times n$, where n is the number of vertices in the graph, such that its elements A_{ij} are:

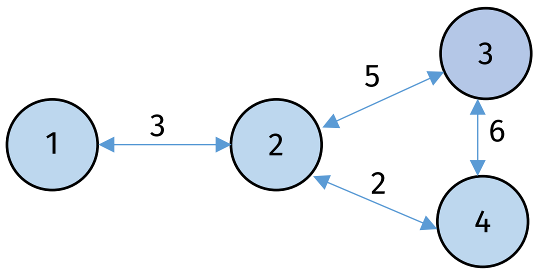
$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between vertices } i \text{ and } j \\ 0 & \text{otherwise} \end{cases}$$

Two things to notice about the adjacency matrix are that, at first, for a graph with no self-edges the diagonal matrix elements are zero, and second that for an *undirected graph* the adjacency matrix is symmetric, since if there is an edge between i and j there will also be an edge between j and i.

The adjacency matrices can also be used to represent multi-edges and self-edges. To represent a multi-edge, we proceed by setting the corresponding A_{ij} and A_{ji} elements equal to the multiplicity of the edge. To represent self-edges instead we need to set the corresponding diagonal element A_{ii} equal to 2, this because every self-edge has two ends, both of which are connected to the vertex i. On other terms, if we look at all the other non-self-edges on the graph, we will notice that they appear twice in the adjacency matrix, both on A_{ij} and A_{ji}. Where instead self-edges only appear once at A_{ii}. So, in order to count edges equally we need to record twice the number of edges into the single diagonal element at our disposal.

The adjacency matrix can be also used to represent *weighted graphs*, by giving the elements of the adjacency matrix values equal to the weight of the corresponding connection.

$$A_{ij} = \begin{cases} w & \text{if there is an edge between vertices } i \text{ and } j \text{ with weight } w \\ 0 & \text{otherwise} \end{cases}$$



$$A = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 3 & 0 & 5 & 2 \\ 0 & 5 & 0 & 6 \\ 0 & 2 & 6 & 0 \end{bmatrix}$$

Figure 2.1: On the left we have an undirected weighted graph with 4 vertices and 4 edges, on the right we have the adjacency matrix A corresponding to that graph.

2.1.4 Degree

In graph theory the degree of vertex is the number of edges that are connected to it. Exploiting the adjacency matrix, for an undirected graph of n vertices the degree of a node i, k_i, is defined as follow:

$$k_i = \sum_{j=1}^{n} A_{ij}$$

Usually, the number of edges m is equal to the sum of the degrees of all the vertices. But for the undirected case, since each edge represents two connections, the sum of the degrees of all vertices is instead equal to twice the number of edges in the graph.

$$2m = \sum_{i=1}^{n} k_i \text{ or } m = \frac{1}{2}\sum_{i=1}^{n} k_i$$

The mean degree of a vertex in an undirected graph is:

$$c = \frac{1}{n}\sum_{i=1}^{n} k_i = \frac{2m}{n}$$

For directed graphs instead each vertex has two degrees. The in-degree represents the number of ingoing edges connected to the vertex and the out-degree the number of outgoing edges. Keeping in mind the definition of the adjacency matrix in the directed case, we can write *in-degree* and *out-degree* as:

$$k_i^{out} = \sum_{j=1}^{n} A_{ij} \text{ and } k_j^{in} = \sum_{i=1}^{n} A_{ji}$$

For the directed case, the number of edges is equal to the sum of the outgoing or ingoing degrees of all the vertices:

$$m = \sum_{i=1}^{n} k_i^{out} = \sum_{j=1}^{n} k_j^{in} = \sum_{ij} A_{ij}$$

And the mean *in-degree* c_{in} and *out-degree* c_{ou} for directed graphs is equal to:

$$c_{in} = \frac{1}{n}\sum_{i=1}^{n} k_i^{out} = \frac{1}{n}\sum_{j=1}^{n} k_j^{in} = c_{out}$$
$$c_{in} = c_{out} = \frac{m}{n}$$

2.1.5 Density

The maximum number of edges in simple directed graph (so without self-edges or multi-edges) can be evaluated as $n(n-1)$ where n is the number of vertices. For a simple undirected graph instead since we only keep track of half of the edges the maximum number will be $\frac{1}{2}n(n-1)$. Using this we can introduce a new concept called *density* or *connectance*, which is the fraction of these edges that are present in the graph.

$$p = \frac{m}{n(n-1)} = \frac{c}{n-1}$$

The density of a graph lies strictly in the range $0 \leq p \leq 1$. With a density $p = 1$ we have a *fully connected graph*, where each vertex in the graph is connected to each other vertex. With a density $p = 0$ we have a graph where no vertex is connected to any other vertex, meaning that the edges set is empty.

Graphs with a density p that tends to a constant as $n \to \infty$ are said to be *dense*. Since in such graphs the fraction of non-zero elements in the adjacency matrix remains constant as the network becomes large. On the contrary, graphs in which $p = 0$ as $n \to \infty$ are said to be *sparse*. However, these definitions of dense and sparse graphs, only works for theoretical models and not in most practical situation since they can only be applied if one can actually take the limit $n \to \infty$.

2.1.6 Paths

A path is sequence of vertices such that every consecutive pair of vertices in the sequence is connected by and edge. This can be also seen as a route that run from vertex to vertex travelling along the edges of the network. Paths can be defined for both directed and undirected graphs. When we define a path on a directed graph each edge involved in the path must be traversed in the correct direction. On the contrary with undirected graphs each edge involved in the path can be traversed in either direction.

A path can intersect itself, by visiting a vertex that has been visited before or by traversing along an edge that has already been used. However, paths that do not intersect themselves are important in graph theory and are called *self-avoiding paths*. Among those paths we can find two special cases: Geodesic paths and Hamiltonian paths. The *length* of a path is the number of edges (not vertices) traversed along the way. When the same edge is traversed more than one time, it needs to be counted separately for each time.

Knowing that for both directed and undirected graph the element A_{ij} of the adjacency matrix is 1 if there is and edge between the vertex i and vertex j, and 0 otherwise. We know that the product $A_{ik}A_{kj}$ is 1 if there is a path of length 2 from vertex i to vertex j that goes through vertex k, and 0 otherwise. This mean that we can express the total number of paths of length 2 as:

$$N_{ij}^{2} = \sum_{k=1}^{n} A_{ik}A_{kj} = [A^2]_{ij}$$

Similarly, we can express the total number of paths of length 3 as product of three elements of the adjacency matrix:

$$N_{ij}^{3} = \sum_{k,l=1}^{n} A_{ik}A_{kl}A_{lj} = [A^3]_{ij}$$

And can be generalized to paths of length r as:

$$N_{ij}^{r} = [A^r]_{ij}$$

As subset of these are the paths of length r that start and end on the same vertex i. We call this special cases “cycles”, and we can calculate the total number L_r of loops of length r by summing the previous result for all the possible starting points i:

$$L_r = \sum_{i=1}^{n} [A^r]_{ij}$$

2.1.7 Geodesic Paths

A geodesic path, also known as shortest path, is a path between two vertices such that no shorter path exists. The length of a geodesic path is called geodesic distance or shortest distance. Mathematically speaking, the geodesic distance between two vertices i and j is the smallest value of r such that $[A^r]_{ij} > 0$. However, there are more efficient ways of calculating the geodesic distance than employing this formula and they will be explored in the *shortest path problem* section. Geodesic paths are intrinsically *self-avoiding*, since if a path would intersect itself creating a loop that same path could be shortened by removing the loop while still connecting the same start and end points.

It is possible for two vertices not to have a geodesic path between theme if those vertices are not connected by any route on the graph. Sometimes when this happen it is said that the geodesic distance between the two vertices is *infinite*, however it is mostly a convention and doesn't really mean much other that those vertices are not connected. We define the *diameter* of graph the length of the longest geodesic path between any pair of vertices in the graph for which a path exists.

2.1.8 Transitivity

One interesting metric in graph theory is transitivity. In mathematics a relation $\circ$ is said to be transitive if $a \circ b$ and $b \circ c$ implies $a \circ c$. In a graph environment there are different type of relationships between vertices, one of which is "being connected by an edge". If this type of relation were transitive it would mean that by knowing that a vertex i is connected to the vertex j, and a vertex j is connected to a vertex k, then we would also know that vertex i is connected to vertex k. Unfortunately, *perfect transitivity* like this is only possible with a fully connected graph, and so is pretty much useless in most scenarios.

It is however possible to estimate the *partial transitivity* of a graph, how much it is likely that if i is connected to j and j is connected to k then i is connected to k. To quantify the level of transitivity on the graph we can proceed as follows. Knowing that i is connected to j and that j is connected to k creates a path ijk of two edges. If i was also connected to k that would create a loop of length three, or a triangle, in the graph. We define the clustering coefficient to be the fraction of paths with length two in the graph that create a loop of length three.

$$C = \frac{(number\ of\ closed\ paths\ of\ length\ two)}{(number\ of\ paths\ of\ length\ two)}$$

Using the adjacency matrix, we can write that as:

$$C = \frac{\sum_{i,j,k} A_{ij}A_{jk}A_{ki}}{\sum_i k_i(k_i - 1)}$$

We can also define a clustering coefficient for a single vertex. For a vertex i, we have:

$$C_i = \frac{(number\ of\ pairs\ of\ neighbors\ of\ i\ that\ are\ connected)}{(number\ of pairs\ of\ neighbors\ i)}$$

Or using the adjacency matrix:

$$C_i = \frac{1}{k_i(k_i - 1)} \sum_{j,k}^{n} A_{ij} A_{jk} A_{ki}$$

2.1 Common graph representation on computers

The adjacency matrix also provides one of the simplest ways to represent a network on a computer. Most computer languages provide two-dimensional arrays that can be used to store an adjacency matrix directly in memory. However, when storing an adjacency matrix on a computer the storage space needed tends to grow fast compared to the number of vertices, since we need to store n elements for each vertex.

Fortunately, *sparse* graphs are characterized by a sparse adjacency matrix, in which most of the elements are zero. So, whenever we are dealing with *sparse* graphs, which are the most common one, we can substantially reduce the memory required by using a *sparse matrix representation* to store their adjacency matrices. Sparse matrix only represents the non-zero elements of the matrix while implicitly representing the zero elements. Depending on the number and distribution of non-zero elements, different data structure can be used and yield huge savings in memory.

The three most common format to store sparse adjacency matrix are: coordinate list (COO), compressed sparse row (CSR), compressed sparse column (CSC).

2.1.1 Coordinate list (COO)

Coordinate list stores a list of tuples with the format $(row, column, value)$. These are usually kept sorted by column index for a faster lookup. This format is good for incrementally constructing a sparse matrix in random order and is widely used on online repositories to store graphs using the Matrix market format. On the other hand, the coordinate list poorly performs when iterating the non-zero elements in lexicographical order, so is typically only used to construct the sparse matrix and then converted either into CSR format or CSC format.

2.1.2 Compressed sparse row (CSR)

Compressed sparse row (CSR) also known as Yale format represents the sparse matrix through three one dimensional arrays. One which contains the non-zero values, one which contains the column indices associated with the non-zero values and one that keep track of the extent of rows. The first two arrays have length equal to the number of non-zero elements, the third array instead has a length equal to $n + 1$ where n is the number of rows.
Here an example,

$$A = \begin{bmatrix} 4 & 0 & 2 & 0 \\ 0 & 4 & 0 & 8 \\ 6 & 5 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}$$

using the 4×4 matrix A with 7 non-zero elements we have:

$NONZERO\_VALUES = [4 \quad 2 \quad 4 \quad 8 \quad 6 \quad 5 \quad 3]$
$COLUMN\_INDICES = [0 \quad 2 \quad 1 \quad 3 \quad 0 \quad 1 \quad 2]$
$ROW\_OFFSETS = [0 \quad 2 \quad 4 \quad 6 \quad 7]$

To extract the non-zero values and column indices of a row r we just need to take the slice that goes from $ROW\_OFFSETS[r]$ to $ROW\_OFFSETS[r+1]$. For instance, using the previous example, we know that the values of the row zero are contained in the slice that goes from zero to two.

2.1.3 Compressed sparse column (CSC)

Compressed sparse column (CSC) works similarly to compressed sparse row, except that the values are first by column. Thus, in this format the second array contains the row indices associated with the non-zero elements and the third array keeps track of the extent of columns.

Here an example using the previous 4×4 matrix A with 7 non-zero elements:

$NONZERO\_VALUES = [4 \quad 6 \quad 4 \quad 5 \quad 2 \quad 3 \quad 8]$
$ROW\_INDICES = [0 \quad 2 \quad 1 \quad 2 \quad 0 \quad 3 \quad 1]$
$COLUMN\_OFFSETS = [0 \quad 2 \quad 4 \quad 6 \quad 7]$

2.2 Graph file formats

2.2.1 Matrix Market (.mtx)

The Matrix Market [5] (MM) format provide a simple and effective mechanism for exchanging matrix data. The objective of the format is to define a minimal base ASCII file format easy to explain and parse, but at the same time also easy to adapt and extend to applications with a more rigid structure.

Matrix Market specifies two different exchange formats for matrices:

- *Coordinate Format*, more suitable for the representation of general sparse matrices. This format only specifies nonzero entries, and explicitly give the coordinates of each nonzero entry.
- *Array Format*, more suitable for representing general dense matrices. All the entries in this format are provided in a pre-defined (column-oriented) order.

Here we present an example of the MM coordinate format:
The first line of the file always contains the header, which contains information regarding the matrix below.

```
%%MatrixMarket matrix coordinate real general
```

In this example, the header indicates that the object being represented is a matrix in coordinate format and that the numeric data following is real and represented in general form (meaning that the matrix format is not taking advantage of any symmetry properties). Variants of the numeric data are defined for matrices with complex and integer entries, as well as for those in which only the position of the nonzero entries is not defined. This can be indicated by changing the header line from `real` to `complex`, `integer`, or `pattern`.

Additional variants are defined for cases in which symmetries can be used to significantly reduce the size of the data: symmetric, skew-symmetric, and Hermitian. In these cases, only entries in the lower triangular portion need be supplied. In the skew-symmetric case the diagonal entries are zero, and hence they too are omitted. This can be indicated by changing the header line from `general` to `symmetric`, `skew-symmetric`, or `hermitian`.

The first (non-header and non-comment line) should contains: the number of rows, columns and entries in the matrix, separated by a space.

```
rows columns entries
```

All the other lines should either contains: a comment, that can be annotated at the start of the line with the `%` character, or a triple (or tuple for pattern matrices) containing the row index, column index and associated value at those coordinates.

```
row_index column_index value
```

2.3 Shortest path problem

The shortest path problem is defined as the problem of finding a path between two vertices in a graph such that the sum of the weights of its edges is minimized.

2.3.1 Definition

Given a graph $G = (V, E)$ and a real-valued function $f: E \to \mathbb{R}$, the shortest path from v to v' is the sequence of vertices $P = (v_1, v_2, \dots, v_n) \in V \times V \times \dots \times V$ with $v_1 = v$ and $v_n = v'$ such that v_i is adjacent to v_{i+1} for $1 \le i \le n$ and that minimizes the total cost $\sum_{i=1}^{n-1} f(e_{i,i+1})$ over all the possible such sequences.

When each edge in the graph has the same weight $f: E \to \{w\}$, the problem is equivalent to the problem of finding the path with the fewest edges.

The shortest path problem comes also with some variations:

- *Single-source shortest path* problem (SSSP), that is defined as the problem of finding the shortest path from a source vertex v to all the other vertices of the graph.
- *Multi-source shortest path problem* (MSSP), that is defined as the problem of finding the shortest paths for all the vertices of the graphs to the closest vertex among a collection of source vertices $(v_1, v_2, \dots, v_m)$.
- *All-pairs shortest path problem*, that is defined as the problem of finding the shortest paths between each pair of vertices v and v' in the graph.

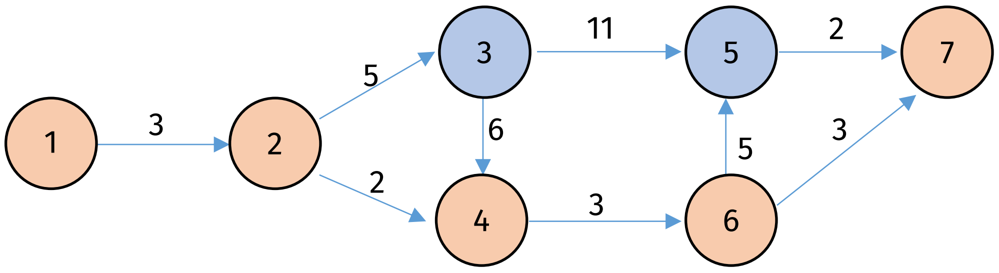


Figure 2.2: Shortest path $(v_1, v_2, v_4, v_6, v_7)$ between vertex 1 and vertex 7

2.4 Dijkstra's algorithm

Dijkstra's algorithm [4] is the most know and the de-facto reference approach to the shortest path problem. The algorithm has existed in many variations over the years, starting from the computation of the shortest path from two vertices to the more common variant that evaluates all the shortest paths from a fixed source. Dijkstra's algorithm proceeds by relaxation, where the approximated solution is replaced with better ones until the result is reached. To do so, the Dijkstra's algorithm uses a priority queue for storing and querying the partial solutions based on their distance from the source.

ALGORITHM 1: DIJKSTRA'S SINGLE-SOURCE SHORTEST PATH

```
Input: Graph G, Source S
Output: Distances D, Predecessors P
foreach vertex v of G do:
    D[v] = ∞
    P[v] = UNDEFINED
    Q.push(v,D[v])

D[source] = 0
Q.update(source,0)

while Q is not Empty do:
    u = Q.pop_min()
    foreach neighbor v of u do:
        $d_{new}$ = D[u] + G.edges(u,v)
        if $d_{new}$ < D[v] then:
            D[v] = $d_{new}$
            P[v] = u
            Q.update(v,$d_{new}$)
return D,P
```

Dijkstra's algorithm offers the most efficient computation complexity, that is almost linear to the number of vertices $\Theta(|E| + |V| \log|V|)$. However, the algorithm exposes no parallelism across vertices, and in application domains that involves graphs with millions of vertices the sequential implementation may become impractical.

2.4.1 Example

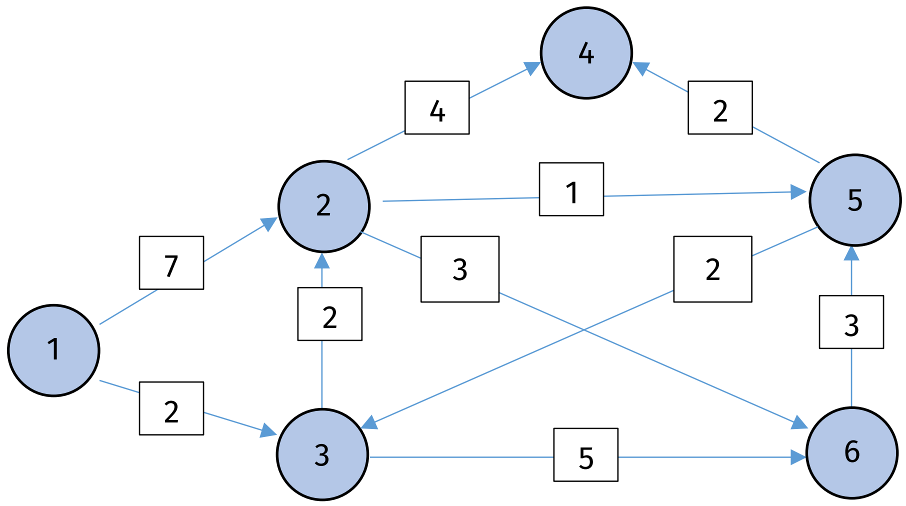


Figure 2.3: A directed weighted graph with 6 vertices and 10 edges

Here we provide an example of the execution of the Dijkstra's algorithm using the graph presented on Figure 2.3. We will use a table to mark the changes made by the algorithm after each visit. For each cell we will keep track of both the current minimum distance and the predecessor vertex. For the example we use vertex 1 as the source vertex.

VISITING	VERTEX 1	VERTEX 2	VERTEX 3	VERTEX 4	VERTEX 5	VERTEX 6	QUEUE
	$0_{(null)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	1,2,3,4,5,6
V=1	$0_{(null)}$	$7_{(1)}$	$2_{(1)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	2,3,4,5,6
V=3	$0_{(null)}$	$4_{(3)}$	$2_{(1)}$	$+\infty_{(null)}$	$+\infty_{(null)}$	$7_{(3)}$	2,4,5,6
V=2	$0_{(null)}$	$4_{(3)}$	$2_{(1)}$	$8_{(2)}$	$5_{(2)}$	$7_{(3)}$	4,5,6
V=5	$0_{(null)}$	$4_{(3)}$	$2_{(1)}$	$7_{(5)}$	$5_{(2)}$	$7_{(3)}$	4,6
V=4	$0_{(null)}$	$4_{(3)}$	$2_{(1)}$	$7_{(5)}$	$5_{(2)}$	$7_{(3)}$	6
V=6	$0_{(null)}$	$4_{(3)}$	$2_{(1)}$	$7_{(5)}$	$5_{(2)}$	$7_{(3)}$	
RESULT	$\mathbf{0}_{(\mathbf{null})}$	$\mathbf{4}_{(\mathbf{3})}$	$\mathbf{2}_{(\mathbf{1})}$	$\mathbf{7}_{(\mathbf{5})}$	$\mathbf{5}_{(\mathbf{2})}$	$\mathbf{7}_{(\mathbf{3})}$	

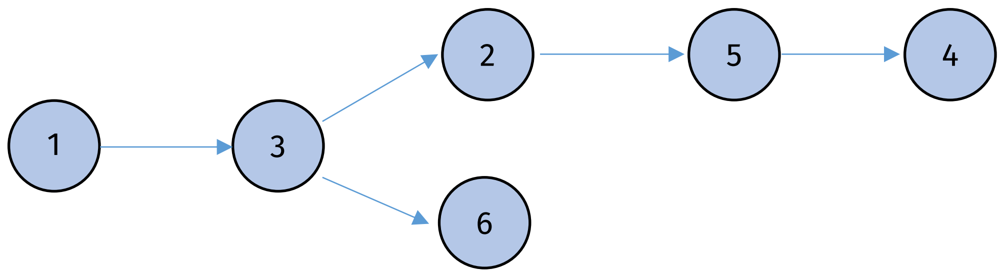


Figure 2.4: The resulting shortest path tree for the graph on Figure 2.3

2.5 Bellman-Ford algorithm

The Bellman-Ford algorithm [4], another known reference approach for the shortest path problem, follows a similar scheme using an iterative relaxation approach that relaxes all the edges in the graph, repeating the process $|V| - 1$ times. This number of iterations ensures that the evaluated distance for each vertex will be the correct one, since the length of the longest possible path (without cycles) consists of $|V| - 1$ edges.

ALGORITHM 2: BELLMAN-FORD SINGLE-SOURCE SHORTEST PATH

```
Input: Graph G, Source S
Output: Distances D, Predecessors P
foreach vertex v of G do:
    D[v] = ∞
    P[v] = UNDEFINED

D[source] = 0

repeat G.vertices() - 1 times:
    foreach edge (u,v) of G do:
        $d_{new}$ = D[u] + G.edges(u,v)
        if $d_{new}$ < D[v] then:
            D[v] = $d_{new}$
            P[v] = u
return D,P
```

This simpler procedure has a worse computational complexity of $\Theta(|E||V|)$ for the sequential implementation; however, it has shown to be easily parallelizable either by performing the search for the shortest path on each vertex independently or by parallelizing the relaxation operation on the edges.

2.5.1 Negative Cycles

Unlike the Dijkstra's algorithm, the Bellman Ford implementation can be used even with graphs where some of the edges are weighted with negative numbers. Negative weights in the shortest path problem are particularly problematic when they lead to the formation of a negative cycle, which is a cycle whose edges weights sum to a negative value.

When such cycle occurs, the graph has no cheapest path, since any path can become cheaper by traversing the negative cycle. The Bellman-Ford algorithm not only can work with negative weights, but it can also detect the presence of a negative cycle, interrupting the execution.

ALGORITHM 3: BELLMAN-FORD SSSP WITH NEGATIVE CYCLE CHECK

```
Input: Graph G, Source S
Output: Distances D, Predecessors P
foreach vertex v of G do:
    D[v] = ∞
    P[v] = UNDEFINED

D[source] = 0

repeat G.vertices() - 1 times:
    foreach edge (u,v) of G do:
        $d_{new}$ = D[u] + G.edges(u,v)
        if $d_{new}$ < D[v] then:
            D[v] = $d_{new}$
            P[v] = u

foreach edge (u,v) of G do:
    if D[v] > D[u] + G.edges(u,v) then:
        return "Negative cycle detected"
return D,P
```

2.5.2 Example

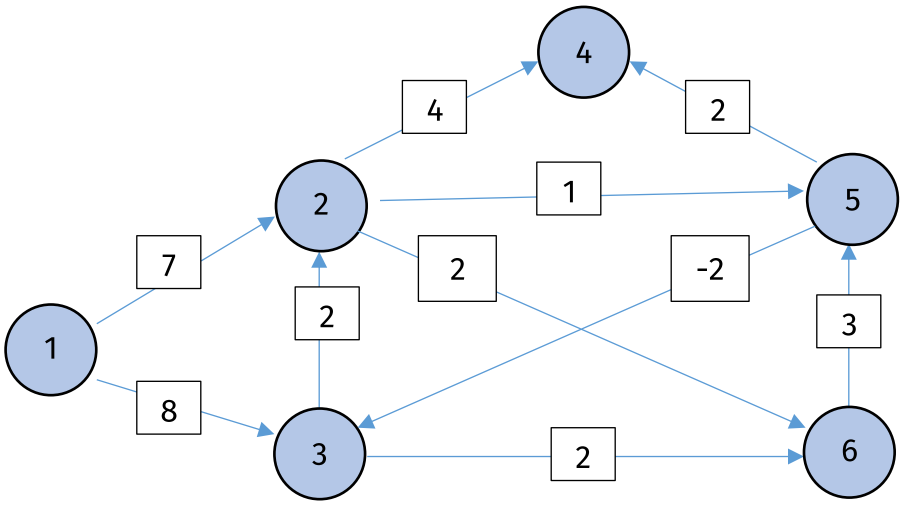


Figure 2.5: A directed weighted graph with 6 vertices and 10 edges

Here we provide an example on how the Bellman-Ford algorithm reach its final estimation using the graph presented on Figure 2.5. As before, we will use a table to mark the changes made by the algorithm at each iteration. Each cell will keep track of both the current minimum distance and the predecessor vertex. The example uses the vertex 1 as the source vertex.

ITERATION	VERTEX 1	VERTEX 2	VERTEX 3	VERTEX 4	VERTEX 5	VERTEX 6
0	$0_{(null)}$	$\infty_{(null)}$	$\infty_{(null)}$	$\infty_{(null)}$	$\infty_{(null)}$	$\infty_{(null)}$
1	$0_{(null)}$	$7_{(1)}$	$8_{(1)}$	$\infty_{(null)}$	$\infty_{(null)}$	$\infty_{(null)}$
2	$0_{(null)}$	$7_{(1)}$	$8_{(1)}$	$11_{(2)}$	$8_{(2)}$	$9_{(2)}$
3	$0_{(null)}$	$7_{(1)}$	$6_{(5)}$	$10_{(5)}$	$8_{(2)}$	$9_{(2)}$
4	$0_{(null)}$	$7_{(1)}$	$6_{(5)}$	$10_{(5)}$	$8_{(2)}$	$8_{(3)}$
5	$0_{(null)}$	$7_{(1)}$	$6_{(5)}$	$10_{(5)}$	$8_{(2)}$	$8_{(3)}$
RESULT	$\mathbf{0}_{(\mathbf{null})}$	$\mathbf{7}_{(1)}$	$\mathbf{6}_{(5)}$	$\mathbf{10}_{(5)}$	$\mathbf{8}_{(2)}$	$\mathbf{8}_{(3)}$

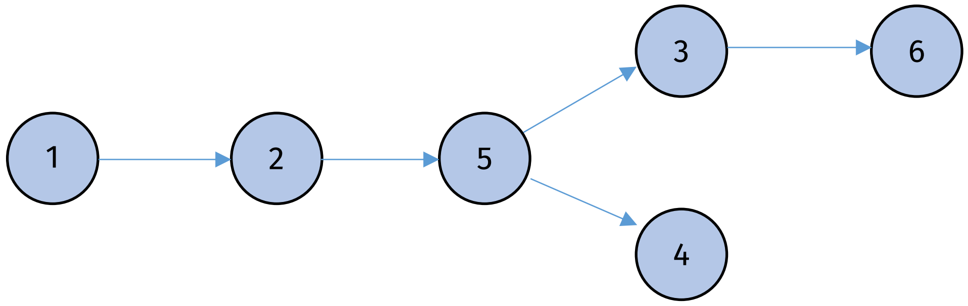


Figure 2.6: The resulting shortest path tree for the graph on Figure 2.5

2.6 SLF-LLL heuristics

Shortest path algorithms like the Dijkstra's algorithm are motivated by the idea that when the edge weights are nonnegative, the queue management strategy should prioritise computations for the vertices with small labels. To do so, the Dijkstra's algorithm uses a priority queue, retrieving the vertices with smallest label at each iteration, and thus resulting in a minimal number of iterations equal to the number of vertices. Following this same idea, others tried to emulate approximately this minimum label selection policy, while also trying data structure with smaller computational overheads.

A simple strategy to prioritise vertices with small labels is the so-called *Small Label First* method [4] (SLF). This method revolves around the idea of maintaining a double ended queue to keep track of vertices that need to be processed. At each iteration, whenever a vertex j enters the queue its label $D[i]$ is compared with the label $D[j]$ of the top vertex i of the queue. Then if $D[j] < D[i]$, the vertex enters at the top of the queue, otherwise it enters at the bottom of the queue.

This strategy provides a good mechanism for inserting the vertices in the queue, but it always processes the vertex on top, which might not be one of the smallest due to the updates done at each iteration. However, a more sophisticated method called *Large Label last* [4] (LLL), tries to mitigate that by providing a different removing strategy. At each iteration, when the label of the vertex j on top of the queue is larger than the average vertex label $\frac{1}{|Q|}\sum_{i\in Q} D[i]$, the vertex j on top is repositioned on the bottom of the queue.

The combination of the SLF queue insertion method and the LLL vertex removal strategy generates what we refer to as the *SLF-LLL heuristic*.

ALGORITHM 4: SLF-LLL HEURISTIC IN SINGLE-SOURCE SHORTEST PATH

```
Input: Graph G, Source S
Output: Distances D, Predecessors P
foreach vertex v of G do:
    D[v] = ∞
    P[v] = UNDEFINED

D[source] = 0
Q.push_front(source)
W = 0

while Q is not Empty do:
    $w_{avg}$ = w / Q.size()
    node = Q.pop_front()
    while D[node] > $w_{avg}$ do:
        Q.push_back(node)
        node = Q.pop_front()
    w = w - D[node]
    foreach neighbor neigh of node do:
        $d_{new}$ = D[node] + G.edges(node,neigh)
        if $d_{new}$ < D[node] then:
            if Q.contains(neigh) then:
                w = w - D[node] + $d_{new}$
            else:
                if $d_{new}$ < D[Q.front()] then:
                    Q.push_front(neighbor)
                else:
                    Q.push_back(neighbor)
                w = w + $d_{new}$
            D[node] = $d_{new}$
            P[node] = neigh
return D,P
```

The use of the Small Label First and Large Label Last as search heuristics [5], has shown to perform better than more standard propagation method like Dijkstra, on graphs of low density (few neighbors per node). That's because even though the Dijkstra algorithm visits the same node as few times as possible it requires to maintain a priority queue, where the SLF-LLL search heuristics only need to maintain a double-ended queue.

Chapter 3 Related Work

Here we will discuss some of the work that has been done by others on the field of parallel solvers for the shortest path problem, including some of the algorithms that are currently implemented by the software libraries that we have tested.

3.1 Δ-Stepping algorithm

The delta stepping algorithm proposed by Meyer and Sanders (2003) [3] revolves around the tunable parameter Δ. By varying the Δ parameter in the range $[1, \infty]$ we can tune the amount of parallelism desired in the algorithm, which results into a whole class of different algorithms. Looking at the boundaries of the parameter, when $\Delta = 1$ the algorithm exposes no parallelism acting as a variant of the Dijkstra's algorithm, where with $\Delta = \infty$ we have the Bellman-Ford algorithm.

ALGORITHM 5: Δ-STEPPING SINGLE-SOURCE SHORTEST PATH

```
Input: Graph G, Source S
Output: Distances D
foreach vertex v of G do:
     D[v] = ∞

Relax(s,0)

while B is not Empty do:
     i = B.get_smallest()
     R = Set()
     while B[i] is not Empty do:
          foreach edge (u,v) of B[i] do:
               R.add(u)
               if G.edges(u,v) <= Δ then:
                    Relax(u,G.edges(u,v))
     foreach edge (u,v) of R do:
          if G.edges(u,v) > Δ then:
               Relax(u,G.edges(u,v))
return D,P

fun Relax(v,w):
     if w < D[v] then:
          B[D[v]/Δ].remove(v)
          B[w/Δ].add(v)
          D[v] = w
```

In delta stepping the vertices are grouped into *buckets*, depending on their distance from the source vertex, meaning that vertices that lie in a specific range share the same bucket. Edges associated with vertices are grouped into two categories: *light edges* and *heavy edges*. For light edges, the updated distance of the destination vertex lies within the same bucket as the source vertex. On the other hand, destination vertices attached to heavy edges lie outside this range and inside another bucket.

Delta-stepping start approaching the *light edges* in a bucket and processes them simultaneously and in parallel until the entire bucket is emptied. Then it continues with the heavy edges, moving those vertices that result in a smaller Δ to a closer bucket (like relabeling in Dijkstra's method).

The main drawbacks of this approach are related to the structure of the buckets. Firstly, the delta-stepping's bucket implementation requires a dynamic structure that can be resized in parallel. Secondly, the moving of vertices between different buckets is a difficult problem to parallelize requiring atomics that reduce the amount of concurrency.

Note that this algorithm is not currently implemented in any of the library that we took in consideration, however it is suitable for representing what a trade-off between the two extremes of Dijkstra and Bellman-Ford would look like. And is also one of the first approaches at the parallelization of the shortest path problem.

3.1 Workfront sweep and Near-Far pile

In 2014 Davidson et al. [2] proposed different parallel solutions to improve the performances of single-source shortest path algorithms over the GPU.

3.1.1 Workfront sweep

The first, called *Workfront sweep,* revolves around the idea of using the mentioned Bellman-Ford algorithm combined with a queue that keep tracks of the vertices whose computation has not changed from the previous iteration. Then at each iteration, instead of proceeding by relaxing all the edges, the algorithm will only process the edges contained into the queue.

To ensure a proper behavior the use of the *Workfront sweep* also requires additional care compared to the usual Bellman-Ford. Firstly, since the edges will be processed in non-deterministic order, the update of the distances requires either atomic operations or an additional processing at the end of each Workfront expansion. Secondly, duplicates in the vertices queue should be avoided or removed since that could affect the overall performances due to redundant computations.

The aim of the *Workfront sweep* method as shown is to reduce the amount of redundant work done by the algorithm, by pruning the list of edges to avoid reprocessing data that has not changed from the previous iteration. However, the use of a queue can also be used to check for early termination conditions, since the Bellman-Ford can stop earlier whenever there is an iteration of the main loop that terminates without making any changes.

3.1.2 Near-Far pile

The *Near-Far pile* starts from the idea of prioritizing, based on some scoring heuristic, a subset of the vertices contained inside the *Workfront sweep* queue. The heuristic used for this method involves selecting a splitting distance and then process only those vertices less than that distance. This will split the work queue into two subsets: the Near set with distance less than the split distance, that will be processed next and the Far pile with distance greater than the split distance, which will be processed later.

The aim of the Near-Far pile is to increase the efficiency of the relaxation process, since in many cases the unprocessed vertices in the Far pile can be discarded as the closer vertices as processed, therefor minimizing the overall number of relax operations. The *Near-Far pile* method has proven to be able to reduce the amount of redundant work far more than the Workfront sweep one, however at the same time it introduces overheads for handling more complex data structure.

3.1.3 Bucketing with Far pile

Davidson et al. also proposed the *bucketing* method to implement the Δ-Stepping algorithm. As presented before, the original version of the Δ-Stepping algorithm used many finer-grained buckets, considering the smallest bucket on each iteration. The implementation proposed by Davidson et al. instead partition the active elements into a small number of buckets A, while also maintaining a far pile for vertices that fall beyond the last bucket, similarly to the use of the far pile in the previous method.

One of main disadvantages of the original Δ-Stepping algorithm was the difficulty of moving vertices between different buckets. The implementation of Davidson et al. instead focused on a sorting procedure that, at each step, emulate the bucket structure. More in details, the proposed implementation uses Thrust's radix sort to organize buckets based on distance. Then it proceeds by processing the smallest bucket. The resulting output gets spilt into two sets: those inside the bucket range, and those outside. The vertices inside the bucket range gets sorted and then merged again into the bucket array, appending the other vertices to the far pile. Once all the buckets are processed, the far pile gets split into A buckets and a far pile, then the process gets repeated.

The Bucketing method proved to be faster than Dijkstra and Bellman-Ford on higher-degree graphs, but slower than the previous two strategies. This due to the overheads that introduces at each iteration, requiring both sorting on the incoming vertices, and a merge procedure into the Workfront.

Chapter 4 GPU libraries for graphs

4.1 CUDA

Compute Unified Device Architecture (CUDA) is a framework developed by NVIDIA to provide a programming interface to GPU devices using the C and C++ languages. The host CPU is responsible for starting the main program and executing serial code, while delegating parallel execution of compute-intensive tasks to the GPU device.

CUDA programming extends C++ by allowing the programmer to define functions, called kernels, which are executed in parallel N times by N different GPU threads. Each thread run the same kernel function concurrently and is associated with a unique thread ID accessible for each thread through built-in variables.

Threads are arranged into *block of threads* that can be one-dimensional, two-dimensional or three-dimensional, providing a simple way to invoke computation across elements in vectors, matrices or volumes. Threads inside a block can cooperate by sharing data through *shared memory* and by synchronizing their execution with the use of barriers, at which all threads in the block must wait before any is allowed to proceed. Since all of them are expected share the memory resources of the block and to reside on the same streaming multiprocessor core, each block has a limit to the number of threads per block it can contains (up to 1024 threads per block).

However, to overcome this limitation, kernels can be executed by multiple equally shaped thread blocks, increasing the number of total threads that are executing the function. When doing so blocks of threads are arranged into a *grid* of thread block that can be one-dimensional, two-dimensional or three-dimensional, where each block is associated with a unique block ID accessible to each thread.

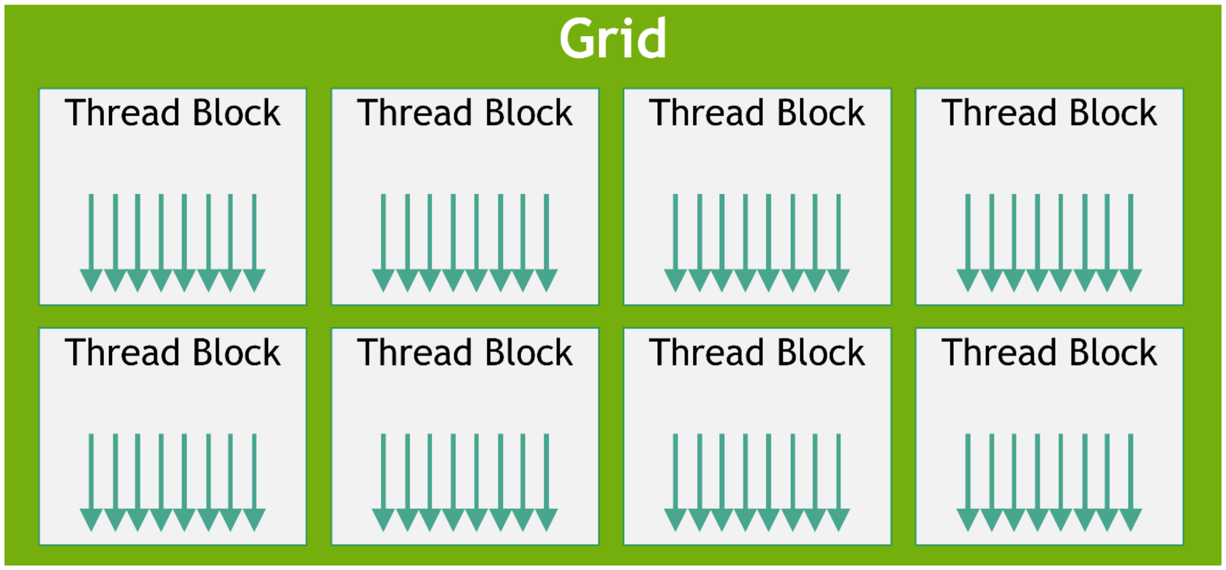


Figure 4.1: An example with 64 CUDA threads arranged in a grid of (4,2) blocks

4.2 NvGraph Library

NvGraph is a high-performance graph analytics library written in CUDA and developed by NVIDIA. Compared to other libraries it views the graph analytics from the perspective of linear algebra and matrix computations which are problems easily and effectively transposable on GPUs. It uses semi-ring of sparse matrix vector multiplication operations (SpMV) to express graph computation and implements three of the most widely used algorithms: Page Rank, Single-Source Shortest Path, and Single-Source Widest Path.

Unfortunately, the development force has stopped a few years ago, and the library has not been updated since then. This was probably related to the development of the RAPIDS suite, which includes a new graph analytics library called CuGraph, developed by NVIDIA, and that contains most of the NvGraph algorithms. The legacy version of NvGraph is still available on a GitHub repository to provide a way for users to continue using it after the CUDA Toolkit stopped releasing it.

4.3 CuGraph library

CuGraph is a library of graph algorithms developed by NVIDIA which is part of the RAPIDS suite, a collection of software libraries with the aim of moving end-to-end data science and analytics pipelines on GPUs designed for data scientist working on Python. Even though the library is mostly designed for developers working on Python, as the other GPU libraries, CuGraph relies on CUDA primitives for the low-level computation which are contained on another library, called libcugraph, that can be used independently from the Python API.

As mentioned, the RAPIDS suite aim is to help moving the computation from the CPU to the GPU for all the data scientist that are working on Python. In order to do so the library provides different modules that can be used independently and can replace the most common libraries that are already used in the field. As an example: the commonly used library Pandas has its “GPU alternative”, called CuDf, contained on the RAPIDS suite, and the same goes for CuGraph which is an alternative to the more common NetworkX.

Regarding the choice of the algorithm for the single-source shortest path problem, CuGraph decided to implement a parallel Bellman-Ford following the Near-Far Pile idea proposed by Davidson. The library also includes a large range of primitives that space from: centrality measures, link analysis and predictions, to graph traversing and sampling.

4.4 Gunrock Essentials library

4.4.1 Programming model

Gunrock Essentials [7] is an open-source graph processing library written in CUDA and developed by the University of California. Gunrock follows *data-centric*, *bulk-synchronous* programming model which focus on graph operations expressed as iterative convergent processes.

In *bulk-synchronous* models each algorithm is defined as series of parallel operations separated by global barriers which ensure synchronization. Gunrock operations are defined through a series of operators designed for different computations. Each operator works on a frontier, which is a set of vertices and edges that are actively participating on the computation.

The parallel operators include:

- *Advance operator*, which generates a new frontier from the current frontier by visiting the neighbors of the current frontier. Each input item maps to multiple output items from the input item's neighbor list.
- *Filter operator*, which generates a new frontier from the current frontier by choosing a subset of the current frontier based on criteria specified by the programmer. Each input item maps to zero or one output item.
- *Intersection operator*, which takes either two input node frontiers with the same length or two edge frontier and returns both the number of intersection as well as the intersected id.
- *Compute operator*, which execute an operation on all the elements of the input frontier. This can be used together with all the previously mentioned operators.

Gunrock graph primitives are written by using composition of these four operators which are executed sequentially one after the other and that stops when the current frontier reach convergence, which can be an empty frontier, a maximum number of iterations or other Boolean conditions.

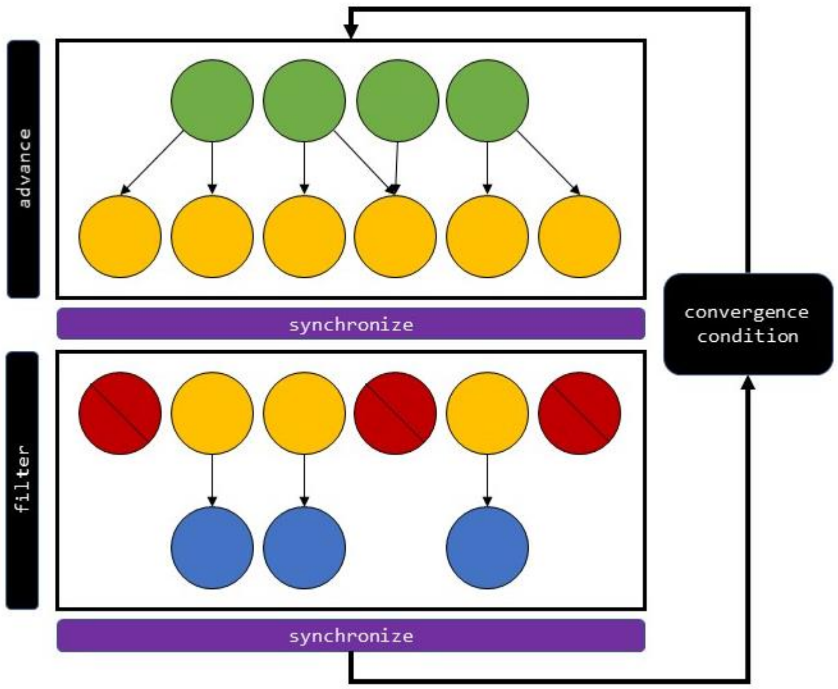


Figure 4.2: Example of an algorithm written with the use of two Gunrock operators: advance and filter

4.4.2 Implemented graph primitives

Using these operators Gunrock implements a parallel Bellman-Ford algorithm for the single-source shortest path problem, where the frontier is represented by the nodes that have been modified in the previous iteration. This process is similar to the Workfront sweep proposed by Davidson et al. [2], that keep track of the modified nodes with the use of a queue.

The algorithm is implemented with the use of two graph operators, an advance operator that explore the neighbors of each vertex within frontier and a filter operator, which trim the output of the advance operator removing duplicates and vertices without edges.

ALGORITHM 7: GUNROCK SINGLE-SOURCE SHORTEST PATH

```
advance_operator = [distances, single_source] __host__ __device__(...){
     weight_t new_dist = distances[source] + weight;
     weight_t old_dist = atomic::min(distances + neighbor, new_dist);
     return new_dist < old_dist;
};

filter_operator = [G, visited, iteration] __host__ __device__(...){
     if (G.get_number_of_neighbors(vertex) == 0) return false;
     if (visited[vertex] == iteration) return false;
     visited[vertex] = iteration;
     return true;
};
```

The library also implements a wide range of graph primitives: A* Search, Betweenness Centrality, Breadth-First Search, Connected Components, Graph Coloring, Geolocation, RMAT Graph Generator, Graph Trend Filtering, Graph Projections, Random Walk, Hyperlink-Induced Topic Search, K-Nearest Neighbors, Louvain Modularity, Label Propagation, Max Flow, Minimum Spanning Tree, PageRank, Local Graph Clustering, Graph SAGE, Stochastic Approach for Link-Structure, Analysis, Subgraph Matching, Shared Nearest Neighbors, Scan Statistics, Triangle Counting, Top K, Vertex Nomination, Who To Follow.

Chapter 5 Preparing the environment

5.1 Installing CUDA Toolkit

5.1.1 Checking basic system requirements

To install the NVIDIA CUDA Toolkit, the system requires:

- A CUDA-capable device
- A supported version of Linux with a gcc compiler and toolchain

To verify the availability of a CUDA-capable GPU device and the current system distribution we are running, we can either check under the System Properties or run the following commands:

```
lspci | grep -i nvidia
uname -m && cat /etc/*release
```

The gcc compiler is required for development using the CUDA Toolkit. It is usually installed as part of the Linux distribution. To check the current version installed we use the following command:

```
gcc --version
```

Lastly, before installing the CUDA Drivers we need to verify that the system has the correct Kernel Headers and Development Packages installed. The kernel headers and development packages must resemble the current kernel version of the system. For example, if the system is running kernel version 3.17.4-301, the 3.17.4-301 kernel headers and development packages must also be installed.

The current version of the kernel that the system is running can be found with:

```
uname -r
```

While the correct kernel headers and development packages can be installed using:

```
sudo apt-get install linux-headers-$(uname -r)
```

5.1.2 Package manager installation

The first thing to do is remove the outdated signing key:

```
sudo apt-key del 7fa2af80
```

Then we can either choose to proceed using a local repository or a network repository. The following commands need to be modified accordingly to the distribution, version and architecture we are using.

If we are using a local repo, we can install it on the file system using:

```
sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
```

Then to enroll the public GPG key we need to run:

```
sudo cp /var/cuda-repo-<distro>-X-Y-local/cuda-*-keyring.gpg
/usr/share/keyrings/
```

With a network repo instead, we just need to download the deb file with:

```
wget
https://developer.download.nvidia.com/compute/cuda/repos/$distr
o/$arch/cuda-keyring_1.0-1_all.deb
```

And then install the new cuda-keyring package with:

```
sudo dpkg -i cuda-keyring_1.0-1_all.deb
```

At this point independently by the repository that we used in the previous step, we can install the CUDA SDK and GDS packages using:

```
sudo apt-get install cuda
sudo apt-get install nvidia-gds
```

A reboot is necessary to complete the installation.

5.1.3 Updating the environment variables

Lastly, we need to update and create some environment variables. The PATH environment variable need to include the cuda bin folder. Also, we need to set a new environment variable called LD_LIBRARY_PATH with the cuda lib folder location, using lib64 for 64-bit operating systems and lib for 32-bit operating systems.

This can be done with the following commands (that can be also added to the .bashrc file):

```
export PATH=/usr/local/cuda-12.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-12.0/lib64\
     ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

5.2 Conda

Conda is an open-source package management system and environment management system for any programming language: Python, R, Ruby, Lua, Scala, Java, JavaScript, C/C++, Fortran. Conda as a package manager helps you find and install and updates packages and their dependencies. It was initially created for Python programs, but it can package and distribute software for any language.
As an environment manager conda helps creating, managing, and switching between environments easily. For example, if a different version of Python is required for a project, we can set up a totally different environment with that version while also keeping the usual version on another environment.

The conda package/environment manager is bundled with *Anaconda*, an open source optimized Python and R distribution, which is why conda is usually only associated with the Python programming language. However, it also possible to install conda with Miniconda, which is a small bootstrap version of Anaconda that includes only conda and its dependencies.

5.2.1 Installation on Linux

For the installation it is sufficient to download the bash installer for either Anaconda or Miniconda, then run it with the command:

```
bash Anaconda-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

5.3 RAPIDS

5.3.1 System requirements

To install the RAPIDS suite of libraries, the system requires:

- A CUDA Capable GPU, with compute capability 6.0+ (Pascal series)
- A supported OS, such as: Ubuntu 18.04/20.04, CentOS 7 /Rocky Linux 8
- A gcc/g++ compiler with version 9.0+
- One of the following supported versions of CUDA and NVIDIA Drivers:
    - CUDA 11.2 with Driver 460.27.03 or newer
    - CUDA 11.4 with Driver 470.42.01 or newer
    - CUDA 11.5 with Driver 495.29.05 or newer

5.3.2 Installation using Conda

To create an environment in which using the RAPIDS suite we can run the following command:

```
conda create -n rapids-22.10 -c rapidsai -c nvidia -c conda-
forge rapids=22.10 python=3.9 cudatoolkit=11.4
```

This will create an environment provided with all the rapids libraries available (cuDF, cuML, cuGraph, cuSpatial, cuXFilter, cuSignal, cuCIM), however the installation can be customized, by specifying each library independently.

5.3.3 C/C++ Development with Conda

As mentioned before conda can also be used as package manager for other programming languages other than Python. In this case the RAPIDS suite also provides the compiled libraries and headers to use the related C++ GPU libraries (libcugraph, libcudf, etc). When using conda this way there are a lot of useful packages, such as build tools (like CMake, Make, Ninja, and Bazel), compilers (C, C++, CUDA, and Fortran) and header only libraries (OpenMP, GMP, Zlib, Boost, and Fmtlib).

One thing to note is that the cudatoolkit provided by conda doesn't contain a working CUDA compiler driver (NVCC). There however a couple of ways to get the NVCC compiler installed:

- The first solution includes the `cudatoolkit-dev` package, available from the `conda-forge` channel. The package consists of a post-install script that downloads and installs the full CUDA toolkit (NVCC compiler included) in the conda environment.
- The other approach consists of installing the NVIDIA CUDA Toolkit on the system and then install the package `nvcc_linux-64` from `conda-forge`, a wrapper that configures the conda environment to use the NVCC compiler installed inside the system combined with the other components of the conda environment.

5.4 Gunrock Essentials

5.4.1 System requirements

To compile the Gunrock Essentials library, the system requires:

- A supported OS, such as: Ubuntu 18.04/20.04
- A gcc/g++ compiler with version 9.0+
- CUDA 11.2+, latest CUDA version is recommended.
- CMake version 3.20.1+

All the other external dependencies can be fetched during building process by cmake.

5.4.2 Building from source

We can start by cloning the repository on our local storage using either:

```
git clone https://github.com/gunrock/gunrock.git
```

Or:

```
wget --no-check-certificate
https://github.com/gunrock/gunrock/archive/refs/heads/master.zi
p
unzip master.zip
```

Then we can proceed by changing directory and creating a build folder.

```
cd gunrock
mkdir build && cd build
```

Before proceeding with the building of source code we need to make sure that the CMakeLists.txt file is targeting the right CUDA architectures. This can be done by opening the previously mentioned file and checking the current targeted CUDA architectures under the SET TARGET PROPERTIES section (line 76), changing the value if needed.

```
set_target_properties(essentials
  PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED ON
    CXX_EXTENSIONS OFF
    CUDA_STANDARD 17
    CUDA_STANDARD_REQUIRED ON
    CUDA_EXTENSIONS OFF
    CUDA_RESOLVE_DEVICE_SYMBOLS ON
    CUDA_SEPARABLE_COMPILATION ON
    CUDA_ARCHITECTURES 75 # Set required architecture.
    # CUDA_PTX_COMPILATION ON
)
```

Then we can build the source code with the following command:

```
cmake ..
```

At this point, we can build a specific application by specifying its name after `make`:

```
make sssp
```

All the binaries can be found under the `bin` folder inside the `build` directory.

5.4.3 Implementing new algorithms

New algorithm needs to be included under the `algoritms` folder, with a `.hxx` file extension. Application on Gunrock Essential are written using four struct: `param`, `result`, `problem`, and `enactor`. The `param` and the `result` structs don't have any functions and are only used as data containers. The `param` struct holds all the relevant user-defined parameters for the application, where the `result` struct instead holds all the data structures will be returned by the application.

The `problem` struct holds data structures that are used internally by the application, as well as the graph we're running our application on. This structure should define two methods: `init` and `reset`. The `init` method should be called the first time the problem is instantiated with a dataset, allocating memory for the internal data structures, and computing any necessary internal parameters. The `reset` method instead should be called before running the same application multiple times on the same dataset, recomputing any necessary internal parameters.

Lastly, we have the `enactor` struct, where the computation of the application happens. This struct need to implement three methods: `prepare_frontier`, `loop`, and `is_converged`. The first method, `prepare_frontier` should initialize the frontier for the first iteration. The `loop` method contains the core computational logic of the application. The `is_converged` method checks whether the algorithm has converged. The default convergence check returns true whenever the current frontier becomes empty. The last thing needed is wrapper function `run` that will be used to call the application.

To compile and test the algorithm we need to build a new example, the first thing to do is to create a new folder under the `examples` path. Then we need to tell the build system about new addition, adding the following line inside the `examples/CMakeLists.txt` file:

```
add_subdirectory(<name>)
```

Then we need to specify the building procedure by adding a CMakeLists.txt file inside the new folder. To simplify this step, we can simply copy the CMakeLists.txt file present inside another folder, making sure to replace the APPLICATION_NAME variable with the name of the new algorithm.

```
cp examples/bfs/CMakeLists.txt examples/<name>/CMakeLists.txt
sed -i "s/set(APPLICATION_NAME bfs)/set(APPLICATION_NAME
<name>)/" examples/<name>/CMakeLists.txt
```

Lastly, we only need to create the example file `<name>.cu`, putting all the function calls needed inside it, and then compile it using the instruction provided in the previous section.

Chapter 6 Scripting & Code

6.1 Python scripts

6.1.1 add_random_weights.py

Arguments: `input_dataset [output_dataset]`

This script can add random weights between 1 and 64 to each edge of an input graph dataset. Input dataset must be a graph file with format Matrix Market. If the ouput dataset path is not defined then the weighted graph will have the same name of the input file, with "w_" added at the start.

Note: The Matrix Market header will be changed from pattern to real.

6.1.2 fix_dataset.py

Arguments: `dataset_list`

This script checks the status of all the graph datasets passed as argument. For each graph dataset with format Matrix Market the script will check and fix the following things:

- The first line of the file should contain a valid Matrix Market header.
- Vertices must have value between 1 and n where n is the number of rows and columns indicated in the first non-comment line.

6.1.3 graph_connected.py

Arguments: `input_dataset`

This script checks if the graph dataset provided as argument is connected, meaning that each vertex is at least reachable by one other vertex. If not, the script will return a list of all the vertices that are not reachable.

6.1.4 generate_random_network.py

Arguments: `format vertices num`

This script generates random graphs with the settings provided as arguments:

- `format` should either be `mtx` or `csr`, for either Matrix Market files or binary compressed sparse row files.
- `vertices` specifies the number of vertices that each graph will have.
- `num` specifies the number of graphs that should be generated.

Note: To get more information on the structure of these random generated graphs check the **Section 7.3**.

6.1.5 generate_random_network_variance.py

Arguments: `format vertices mu sigma1 [sigma2] [...]`

The script generates random graphs with the settings provided as arguments:

- `format` should either be `mtx` or `csr`, for either Matrix Market files or binary compressed sparse row files.
- `vertices` specify the number of vertices that each graph will have.
- `mu` specifies the mean (centre) that should be used in the gaussian distribution.
- `sigma` specifies the standard deviation (spread or “width”) of the distribution. Must be non-negative.

The script generates a graph for each sigma argument provided, one minimum.

Note: To get more information on the structure of these random generated graphs check the **Section 7.5**

6.1.6 mtx_to_csr.py

Arguments: `dataset_list`

The script takes a list of graph datasets as argument, for each graph dataset with Matrix Market format the script creates a copy with a binary CSR (compressed sparse row) format.

6.1.7 generate_network_info.py

Arguments: `dataset_list`

The script generates a recap file that contains a list of metrics for each graph dataset provided as arguments. For each graph dataset a csv file is created, which includes information regarding:

- Number of vertices
- Number of edges
- Graph density
- Maximum degree within vertices of the graph
- Minimum degree within vertices of the graph
- Average degree of the vertices of the graph
- Variance of the average degree
- Global clustering coefficient of the graph
- Average clustering coefficient of the graph

6.1.8 execution_times.py

Arguments: `dataset_list number_of_execution`

The script takes the execution time of the shortest path algorithms for all the graph datasets provided as arguments. For each graph dataset, with either Matrix Market or binary CSR format, the script execute the shortest path algorithm a given number of times using different random vertices as sources, then it generates a csv file containing the details of each execution.

6.1.9 stats_datasets.py

Arguments: `dataset_list output_file`

The script generates a csv recap file that contains various information for all the graph datasets provided as arguments. For each graph dataset, with either Matrix Market or binary CSR format, the output csv file will contain: the metrics of the graphs, the shortest path algorithm executions, the speedup of the parallel algorithm, and the number of edges visited by each algorithm.

6.1.10 frontier_data.py

Arguments: `dataset_list number_of_execution`

The script takes data regarding the number of edges explored by each shortest path algorithms for all the graph datasets provided as arguments. For each graph dataset, with either Matrix Market or binary CSR format, the script generates an output csv file containing the details of each execution.

6.1.11 frontier_stats.py

Arguments: `dataset_list output_file`

The script generates a csv recap file that contains various information regarding the number of edges explored by the parallel algorithm at each iteration. For each graph dataset, with either Matrix Market or binary CSR format, the output csv file will contain: the average number of edges explored, the average elapsed time taken to explore the frontier of vertices.

6.2 Gunrock testing code

During the testing of the Gunrock Essential library to retrieve the execution time spent by the algorithms, we used a similar structure to the one provided in the Gunrock example repository.

Here we provide an outline of the most important parts of the code.

6.2.1 Types definition

In this first part we define what are the types that we want to use to for the indexes of vertices and edges and what type to use for the weights associated with the edges. Also, we define the type used for compressed sparse row (CSR) struct, while also providing what location to use either: host (CPU) or device (GPU).

```
using vertex_t = int;
using edge_t = int;
using weight_t = float;
using csr_t = format::csr_t<memory_space_t::device, vertex_t,
edge_t, weight_t>;
```

6.2.2 Input graph file

Here we load the dataset from memory, using the file path passed as argument. If the file is using a Matrix Market format, then there is an intermediate step, where the graph is initially stored with a coordinate list (COO) struct and later converted to the compressed sparse row (CSR) struct. Otherwise, with binary CSR files the content is immediately parsed into the compressed sparse row (CSR) struct.

```
csr_t csr;
std::string filename = argument_array[1];
if (util::is_market(filename)) {
    io::matrix_market_t<vertex_t, edge_t, weight_t> mm;
    csr.from_coo(mm.load(filename));
} else if (util::is_binary_csr(filename)) {
    csr.read_binary(filename);
}
```

6.2.3 Graph object construction

In this section we convert the previously loaded dataset into a Graph object, which is the object that is used inside the algorithms. The difference between the Graph object and the previous structs is that: a single Graph object can maintain different graph representation inside it, where instead the previous structs where used to store these graph representations.

```
auto G = graph::build::from_csr<memory_space_t::device,
graph::view_t::csr>(
     csr.number_of_rows,
     csr.number_of_columns,
     csr.number_of_nonzeros,
     csr.row_offsets.data().get(),
     csr.column_indices.data().get(),
     csr.nonzero_values.data().get()
);
```

6.2.4 Output vectors

Here, we define the vectors that will be used inside the algorithms to store the output results: distances and predecessors list.

```
// GPU Output vectors
thrust::device_vector<weight_t> distances(n_vertices);
thrust::device_vector<vertex_t> predecessors(n_vertices);
// CPU Output vectors
thrust::host_vector<weight_t> h_distances(n_vertices);
thrust::host_vector<vertex_t> h_predecessors(n_vertices);
```

6.2.5 Running the algorithms

Then here we call each algorithm inside a for loop, using the same random single source vertex for each algorithm. Each function returns the elapsed time in milliseconds, which is calculated using the CUDA directives for the parallel gpu algorithm and using the chrono library for the sequential cpu algorithms.

```
srand(time(NULL));
std::cout << "Gunrock,Dijkstra,SLF-LLL" << std::endl;
for (auto i = 0; i < num_runs; i++){
     single_source = rand() % n_vertices;
     std::cout << gunrock::sssp::run(single_source, … ) << ",";
     std::cout << sssp_cpu::dijkstra(single_source, … ) << ",";
     std::cout << sssp_cpu::slflll(single_source, … ) <<
     std::endl;
}
```

Chapter 7 Experiments

7.1 Architecture used

During our experiments for the parallel executions, we used a NVIDIA GeForce RTX 3090 (CUDA Capability 8.6) with 24GB of dedicated memory, 6MB of L2 cache and 10,496 CUDA cores. For the sequential execution instead, we used a 3.00GHz Intel Core i9-10980XE CPU with 125 GB of memory.

7.2 Notes on the experiments

Before proceeding in detail with the experiments we add some clarification on the precautions taken to have consistent results. Each graph has *weighted* edges, for *unweighted* graphs we added a random floating-point weight $w \in [1,64]$, this was done to avoid turning the solution of the shortest path problem into the one of shortest segment problem.

All the vertices of a graph need to be reachable by at least one other vertex, for graphs with vertices that are not reachable we only consider the connected part of the graph. The average execution time and average visited neighbours are taken by averaging the results of 100 executions. During the testing of the single-source shortest path algorithms the source vertices were selected at random, but in order to stay consistent we used the same list for all the algorithms of a graph.

7.3 Testing the libraries

After an initial look at the available libraries, we decided to set up a first experiment using a subset of graph from a known graph collection, called DIMACS10. These first tests were meant to get an initial look at the execution time of each parallel implementation of the shortest path algorithm and compare them with a sequential implementation.

DATASET	VERTICES	EDGES	AVERAGE DIJKSTRA SSSP EXECUTION TIME (MS)	AVERAGE GUNROCK SSSP EXECUTION TIME (MS)	AVERAGE CUGRAPH SSSP EXECUTION TIME (MS)
AK2010	45,292	217,098	**6.71**	11.27	160.48
COAUTHORSDBLP	299,067	1,955,352	83.38	**2.37**	32.70
DELAUNAY_N17	131,072	786,352	23.038	**12.33**	142.60
DELAUNAY_N21	2,097,152	12,582,816	460.53	**256.00**	548.76
DELAUNAY_N23	8,388,608	50,331,568	2,127.051	**1,088.52**	1,205.78
GERMANY_OSM	11,548,845	24,738,362	1,679.61	**1,032.26**	2,891.86
ASIA_OSM	11,950,757	25,423,206	**1,186.34**	7,140.67	20,310.82
ROAD_USA	23,947,347	57,708,624	3,567.77	**3,110.21**	4,688.72
EUROPE_OSM	50,912,018	108,109,320	8,293.36	**5,113.50**	10,504.08

Table 7.a: Results for the execution of the shortest path algorithms on some of the DIMACS10 graphs

Even though we were working with a small subset of graphs, these initial results lead us to discard CuGraph from the future testing, since the average execution time of its implementation for the shortest path problem was higher than the average execution time of the sequential implementation for almost all the datasets tested. And even with the datasets where the average execution time was better than the sequential implantation, the CuGraph parallel algorithm was still lacking in comparison with the Gunrock algorithm.

The reason for this is probably related with the data structure used by CuGraph to store the graph, which is a coordinate list (COO). As we said earlier in Chapter 2.1, these kinds of structures offer better performances during the construction of the graph, but worse performances during the iteration of the elements.

7.4 Testing with random generated graphs

After removing CuGraph from the testing we decided to proceed by testing with random generated graphs, which can give us a wide range of datasets to use all with similar characteristics. The aim of these tests was to identify the metric that could influence the most the growth of the speedup in this category of graphs.

7.4.1 Definition

Generally random graphs are graphs where some specific sets of parameters are fixed, and others are randomly generated. Here we provide the definition that we used to define our random generated graphs:

Given a number vertices $n \in \mathbb{N}$ and a probability $p \in [0,1]$ we construct a directed graph $G_{n,p} = (V_n, E_{n,p})$ where $V_n = (v_1, v_2, \dots, v_n)$ and such that for each pair of vertices (v_i, v_j) with $i \neq j, i \in [1, n]$ and $j \in [1, n]$ there is a probability p that $(v_i, v_j) \in E_{n,p}$.

In other words, a graph with n vertices where for each pair (i, j) of distinct vertices there is a probability p that those vertices are connected by an edge.

7.4.2 Selected graphs

Starting from these criteria, after fixing a value for n, we created different graph datasets with the same number of vertices and different number of edges. For the probability p we decided to start from a probability of 1, which gave us a fully connected graphs, and then proceed by halving the probability at each new dataset created. We repeated this for different values of n, starting from graphs with one hundred vertices and reaching graphs with forty thousand vertices.

DATASET GROUPS	VERTICES	DATASETS GENERATED	FILE FORMAT
RAND1000	1,000	11	Matrix Market
RAND2000	2,000	11	Matrix Market
RAND3000	3,000	11	Matrix Market
RAND4000	4,000	11	Matrix Market
RAND5000	5,000	11	Matrix Market
RAND10000	10,000	11	Matrix Market
RAND20000	20,000	11	Binary CSR
RAND30000	30,000	11	Binary CSR
RAND40000	40,000	10	Binary CSR

Table 7.b: The collection of random generated graphs used for the testing

For the graphs with 40,000 vertices we couldn't past the graph with probability $p = 50\%$, since its fully connected graph doesn't fit into the memory of the GPU.

7.4.3 Selected metrics

Being graphs highly characterized by the number of vertices and edges, and not by the edges itself, there are some considerations we can make about what could be the most interesting metrics to explore. The simplest one is the *density* of the graph, which as we can see in the Table 7.c tends to resemble the probability p of generating an edge between a pair of vertices (i, j).

Other interesting metrics could be the ones that measure the "*transitivity*" of the graph, which are the *global clustering coefficient* and the *average clustering coefficient*. Both of this metric should increase as the number of edges increases, reaching the *perfect transitivity* with fully connected graphs. However, as we can see in the Table 7.c, both of those metrics tends to follow the same values represented by the density, without adding much information.

DATASET	EDGES	PROBABILITY OF EDGE (i, j)	DENSITY	GLOBAL CLUSTERING COEFFICIENT	AVERAGE CLUSTERING COEFFICIENT
RAND10000_010	97,558	0.098%	0.000976	0.000949	0.001598
RAND10000_020	196,141	0.195%	0.001962	0.001945	0.002430
RAND10000_039	391,043	0.391%	0.003911	0.003931	0.004383
RAND10000_078	781,532	0.781%	0.007816	0.007816	0.008236
RAND10000_1	1,561,630	1.562%	0.015618	0.015610	0.016014
RAND10000_3	3,125,201	3.125%	0.031255	0.031258	0.031649
RAND10000_6	6,247,503	6.25%	0.062481	0.062459	0.062838
RAND10000_12	12,491,232	12.5%	0.124925	0.124925	0.125272
RAND10000_25	25,002,882	25%	0.250054	0.250004	0.250317
RAND10000_50	50,000,150	50%	0.500052	0.500066	0.500266
RAND10000_100	99,990,000	100%	1.000000	1.000000	1.000000

Table 7.c: Metrics for some of the random generated graphs

7.4.4 Average execution times

Here we present the average execution of each algorithm as the density of the graph increase. Each line represents a group of random generated graphs all with the same number of vertices but different density, and each dot represent one of those graphs. On the x-axis we have the graph density, where on the y-axis we have the average execution time expressed on milliseconds. Both the axis uses a logarithmic scale with base 10.

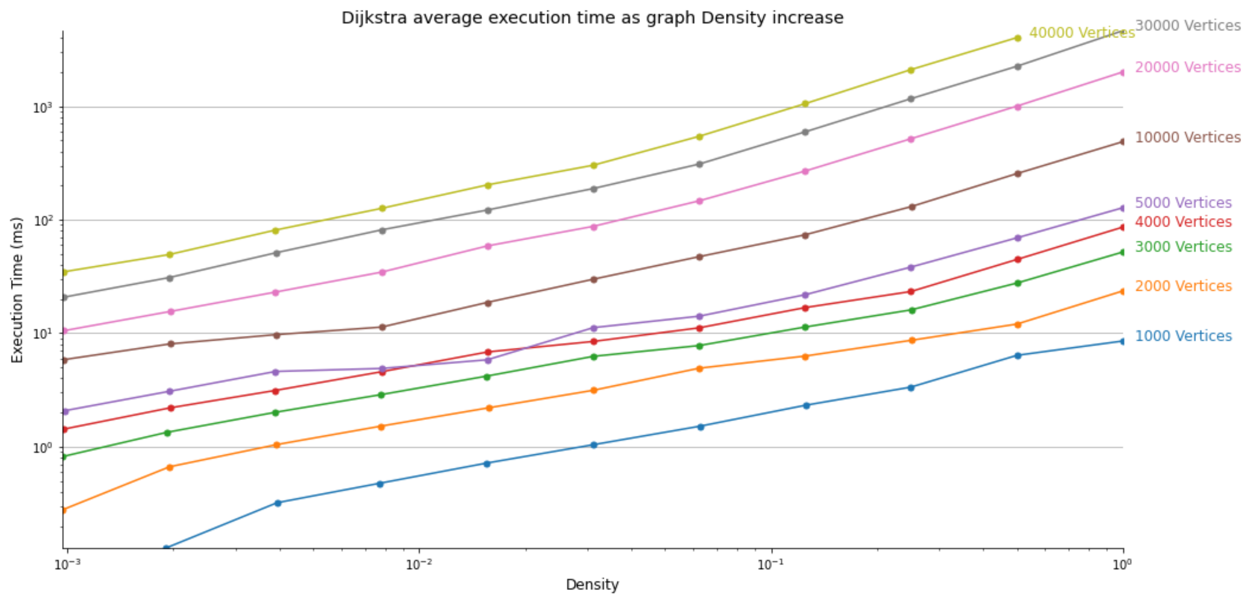


Figure 7.1: How the average execution time of Dijkstra changes when the Density of the graph increase

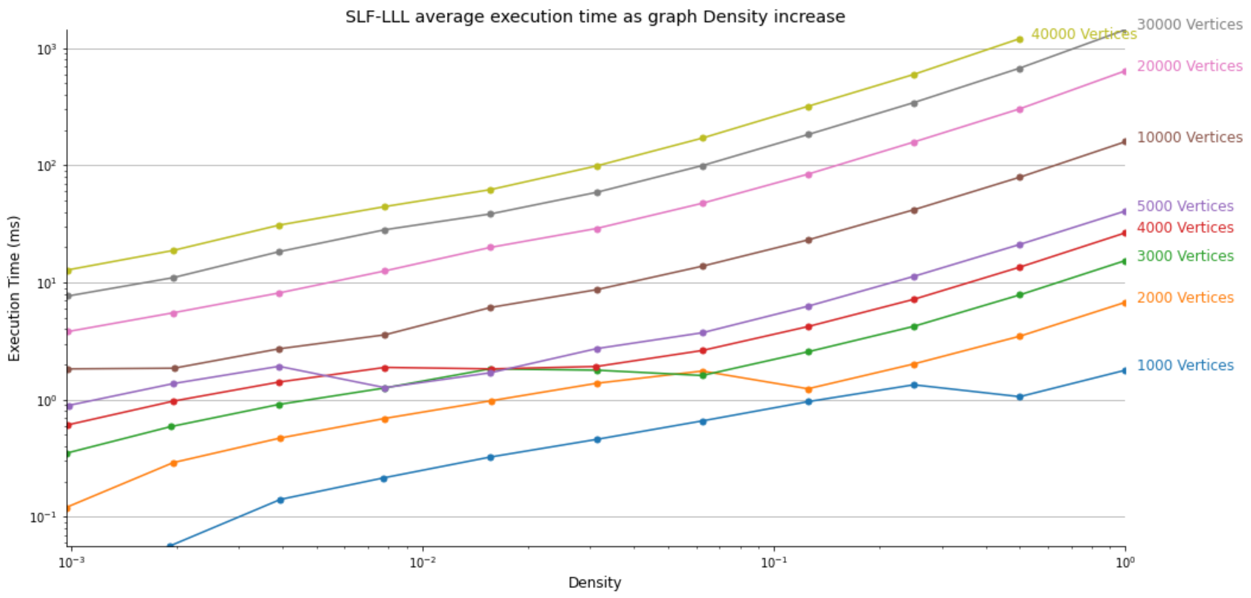


Figure 7.2: How the average execution time of SLF-LLL changes when the Density of the graph increase

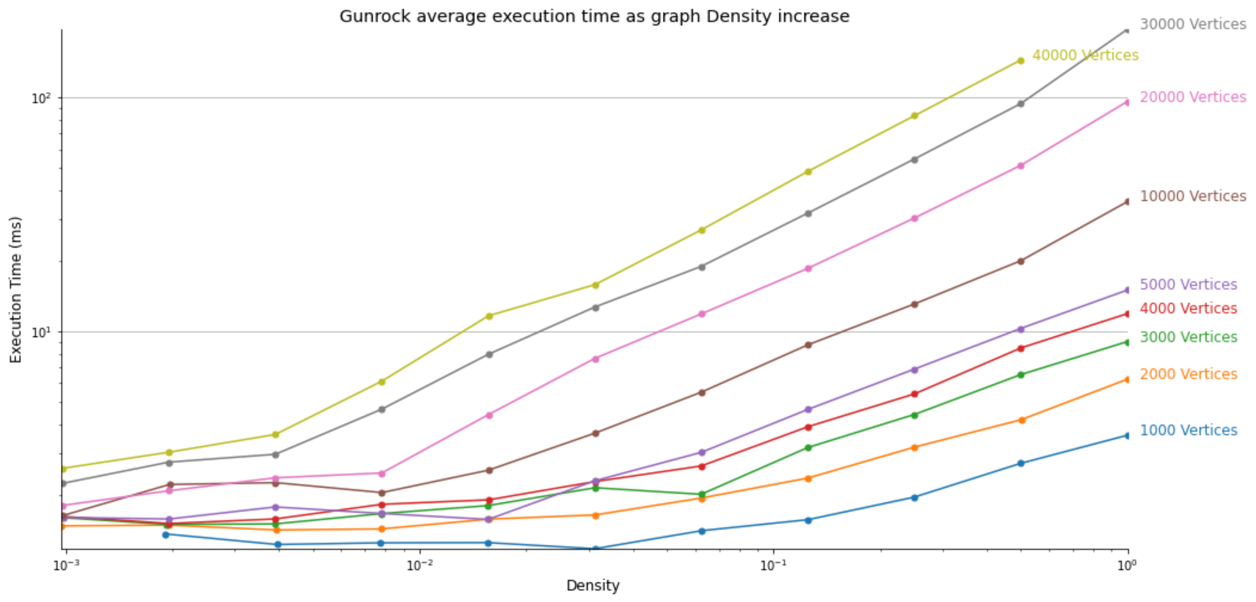


Figure 7.3: How the average execution time of Gunrock changes when the density of the graph increase

Looking both Figure 7.1 and Figure 7.2, which express the average execution time for the two sequential shortest path algorithms, we can see that the elapsed time tend to grow at a mostly stable pace as the density of the graphs increase. With few exceptions for the graphs with a lower number of vertices. Also, comparing the two sequential algorithms we can notice that the average execution time of the Dijkstra's algorithm is always higher than the average execution time of the SLF-LLL heuristics. For example, comparing the execution time for the fully connected graph with 30,000 vertices we have $4{,}640.41$ ms for the Dijkstra algorithm against the $1{,}440.30$ ms of the SLF-LLL heuristic.

Instead, if we look at Figure 7.3, which express the average execution time for the parallel shortest path algorithm, we can see that it grows at two different paces: a slower one at first and a faster one later. The density at which the execution time start to increase faster it's not fixed, but it shifts based on the number of vertices of the graph. For example, if we look at the graphs with 40,000 and 30,000 vertices, we can see that they change pace at a lower density compared to the graphs with 20,000 and 10,000 vertices.

7.4.5 Speedups

Moving to the speedups here we present how the speedup of the parallel algorithm changes when the density of the graph increase. As before each line represents a group of random generated graphs all with the same number of vertices but different density, and each dot represent one of those graphs.

On the x-axis we have the graph density, and on the y-axis we have the speedup of the parallel algorithm over the sequential ones. The red dotted line at $y = 1$ represents the threshold at which the parallel algorithm starts to perform better than the sequential algorithm.

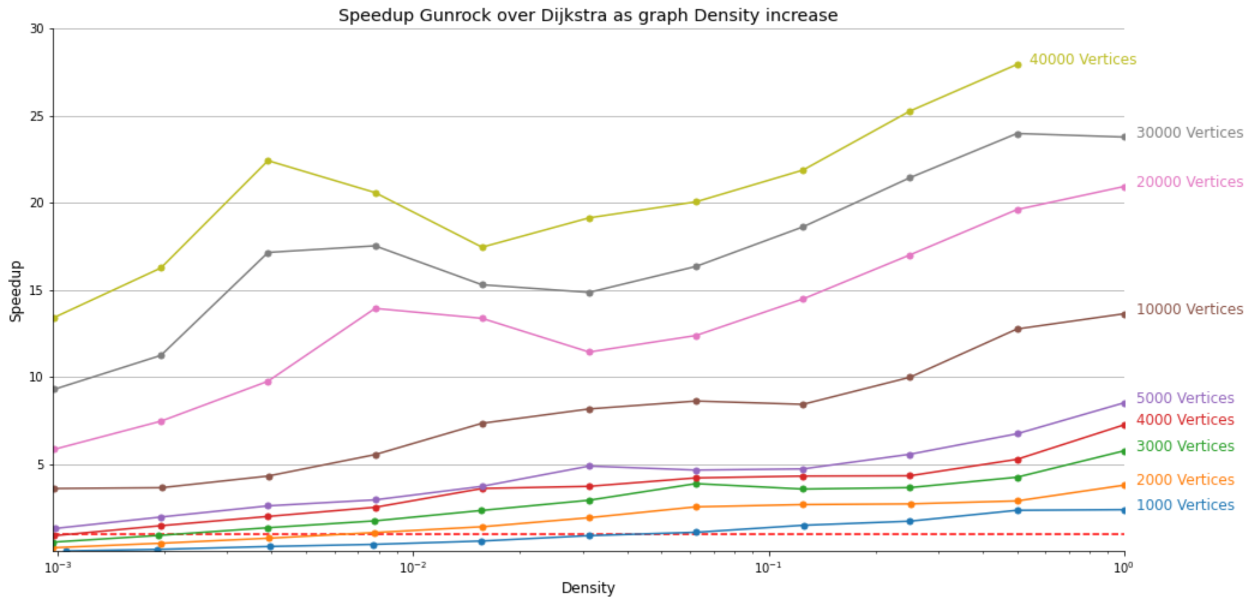


Figure 7.4: How the speedup of Gunrock over Dijkstra changes when the Density of the graph increase

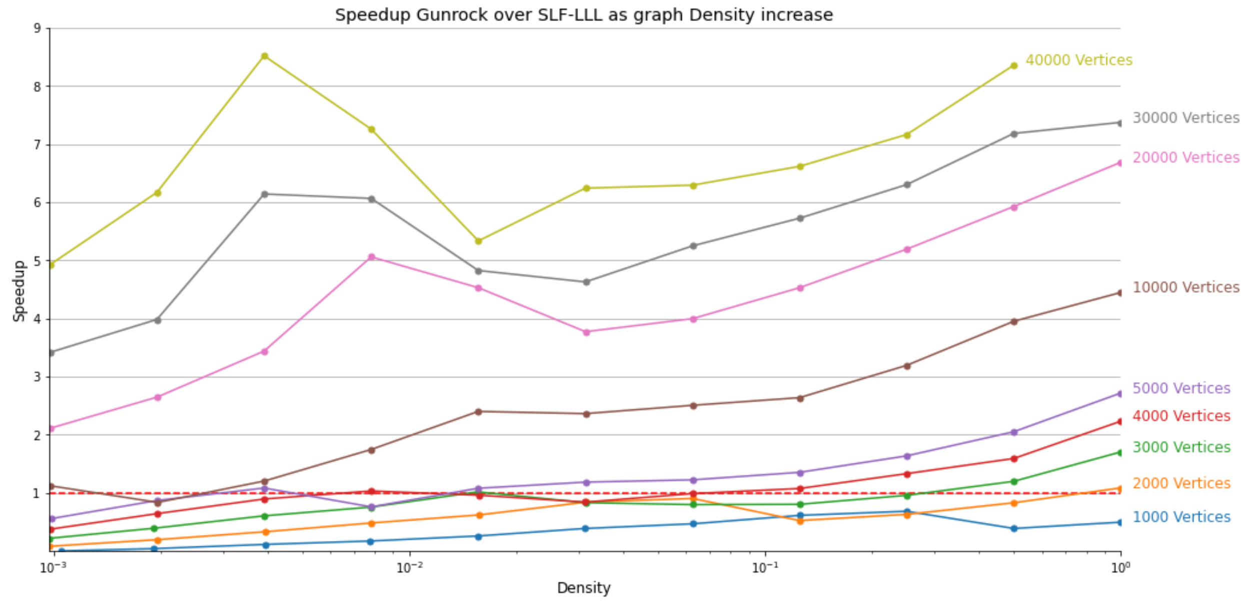


Figure 7.5: How the speedup of Gunrock over SLF-LLL changes when the Density of the graph increase

Looking both at Figure 7.4 and Figure 7.5 we can see that the speedup over the sequential algorithms tends to grow as the density of the graph increase. Also, the initial speedup over graphs with low density is always higher than the initial speedup over graphs with the same density but a smaller number of vertices.

Going more on details, we can see at Figure 7.4 that the Gunrock parallel implementation starts to compete with the Dijkstra algorithm almost immediately, even with graphs with a small number of vertices and low density, while also reaching some impressive speedup values of 15 and 20 for the graphs with more vertices, considering also the sparse matrix representation used for the graphs.

Where instead looking at Figure 7.5, the Gunrock implementation barely shows any improvement compared to the SLF-LLL heuristic for graphs with a small number of vertices, only reaching a speedup higher than 2 on the fully connected graphs with 4,000 and 5,000 vertices. Starting from the graphs with 20,000 vertices, and partially the 10,000 vertices one, the parallel algorithm starts to get us more valuable results, quickly scaling from a speedup of 2 to a speedup of 5 even at low density.

Continuing this topic, another thing that we can notice on both Figures 7.4 and 7.5 is the pattern followed by the datasets with a larger number of vertices, where it reaches a momentary peak followed by a drop-down and a then by a recovery. Also, as we can see this pattern seems to shift depending on the number of vertices of the graphs, where with the graph with 40,000 vertices it appears at a lower density compared to the graph with 20,000 vertices.

In order to explain the presence of these patterns we firstly need to take a deeper look back at the average execution times, focusing on the graphs with 20,000 and 40,000 vertices.

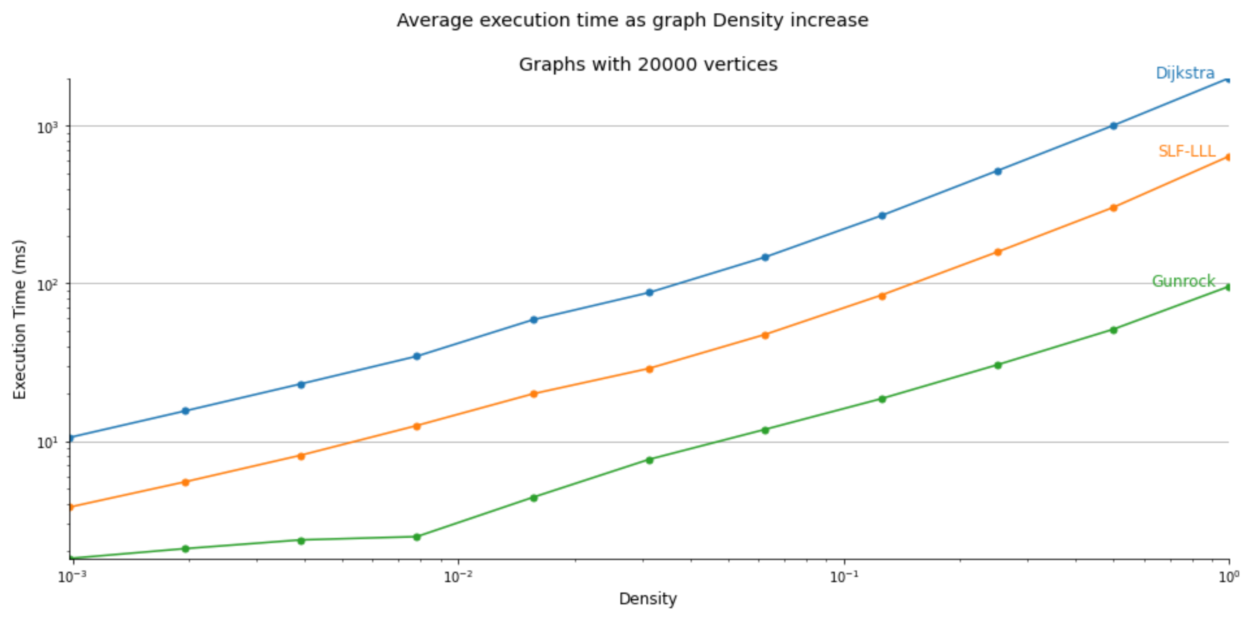


Figure 7.6: How the average execution time of the algorithms changes when the density of the graphs with 20,000 vertices increase

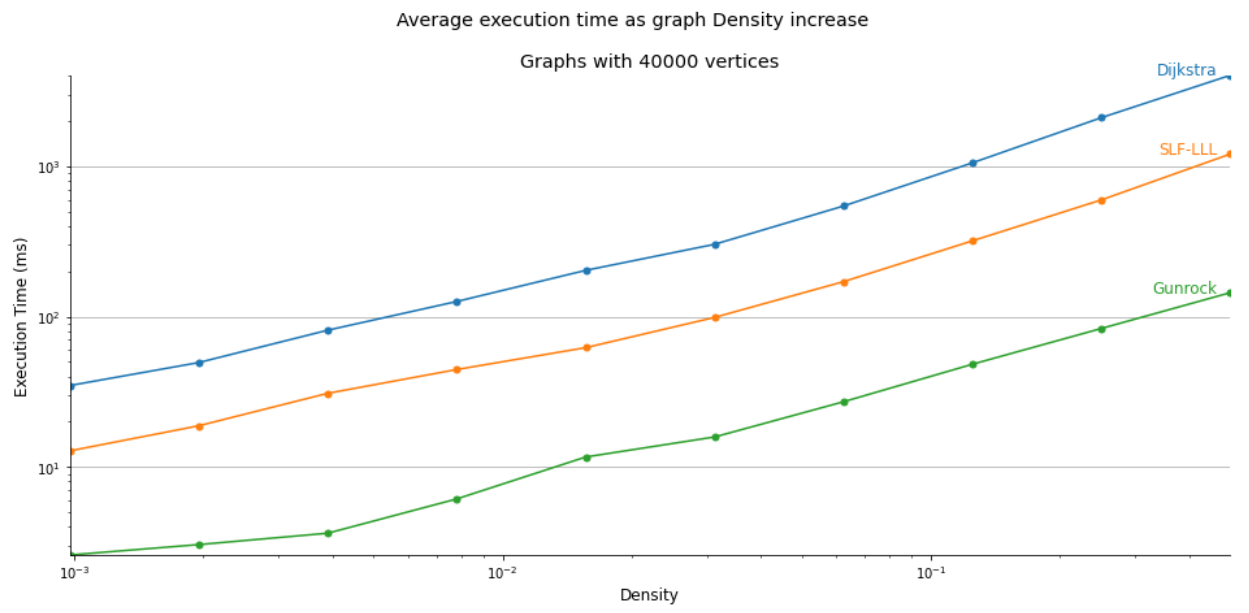


Figure 7.7: How the average execution time of the algorithms changes when the density of the graphs with 40,000 vertices increase

If we compare the speedup result for the graphs with 20,000 vertices from Figure 7.5 with the average execution times shown on Figure 7.6, we can see that density of the initial highest peak in the speedup corresponds with the density at which the pace of the execution time of the parallel algorithm changes. The same goes for the speedup results for the graphs with 40,000 vertices from Figure 7.5 and their average execution times on Figure 7.7.

For both cases the speedup increases at first because the execution time of the parallel algorithm increases at a really slow pace, where the sequential ones instead increase at stable pace throughout the whole chart. After reaching the point where we have a change in pace for the parallel algorithm, its execution time starts to increase at a faster pace, meaning that it is slower than before, and there we have a drop down, since the speedup need to adjust to the new speed at which the execution time is growing.

This give us an idea of the behaviour of the speedup, but it doesn’t provide us an explanation on why the execution time of the parallel algorithm increases at a slower pace with low density. To do that we need to take a deeper look at the parallelization done by the algorithm. Note that since we are working with an external library with a high-level abstraction, where we don’t have full control over all the aspects of the parallelization process, we can’t provide a specific in-depth analysis of the code. However, we can still proceed to give an explanation of the phenomenon by going over the functioning of the parallel algorithm and the data that we collected about it.

As we said before in chapter 4, the Gunrock library implements a version of the Bellman-Ford algorithm with the use of two of its operators. The algorithm proceeds in an iterative way, exploring frontiers of vertices by going over their edges on a parallel way. The number of iterations done by the algorithm, and the number of edges explored parallelly at each iteration can gives us an idea of what happens during the executions.

Here we present, the speedup of the parallel algorithm over one of the sequential algorithms (in this case SLF-LLL) for the graphs with 20,000 and 40,000 vertices. We annotated the most relevant point of interest in the speedup with data coming from the executions: regarding the number of iterations done on average by the algorithm, the number of edges explored on average at each iteration and the time requested in milliseconds to complete the iteration.

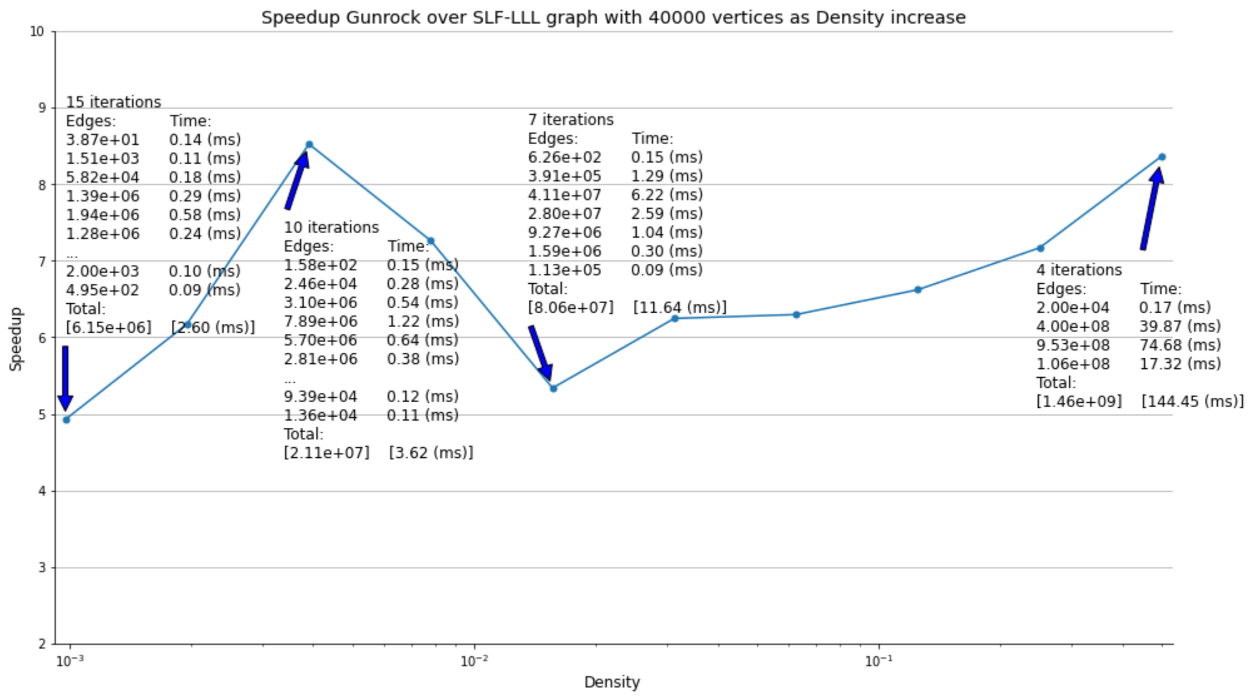


Figure 7.8: How the number of iterations and the number of edges explored by the Gunrock algorithm changes in relation to the speedup for the graphs with 40,000 vertices

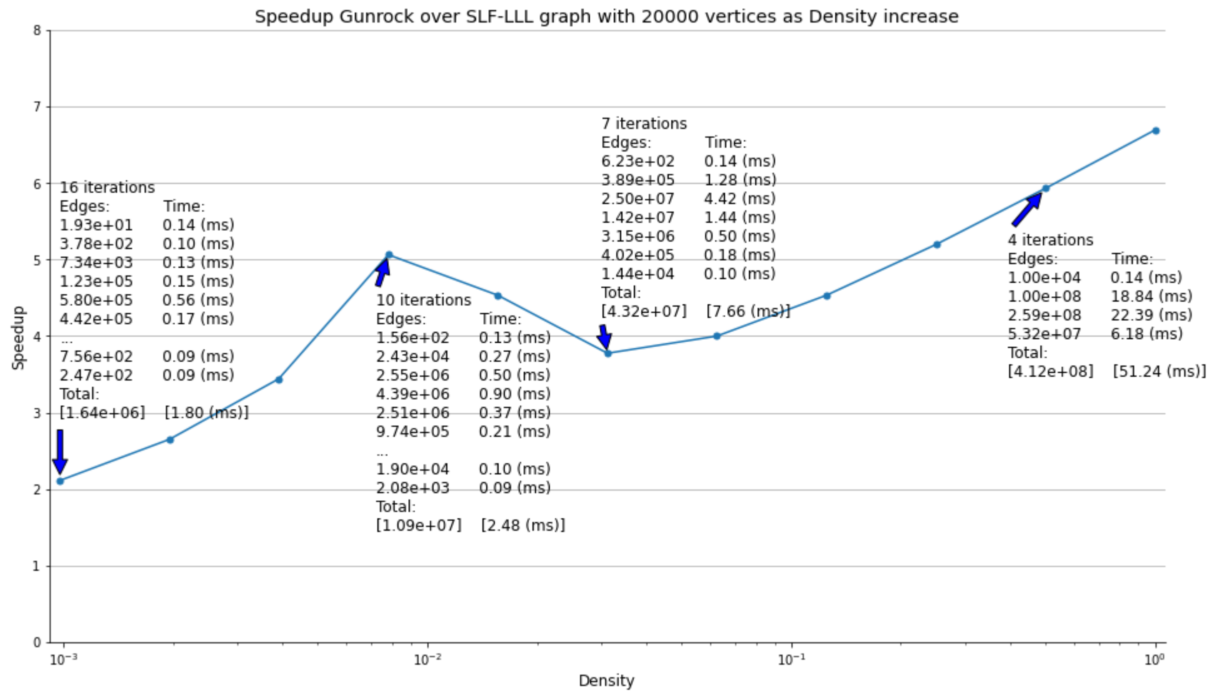


Figure 7.9: How the number of iterations and the number of edges explored by the Gunrock algorithm changes in relation to the speedup for the graphs with 20,000 vertices

Looking at both Figure 7.8 and Figure 7.9, we can notice that the number of edges visited by the algorithm tends to follow the same pattern each time where it starts from a low number of edges and progressively increase until it reaches a cap, then it starts decreasing again. This is reasonable, as the parallel algorithm initially starts from a frontier composed only by the source vertex. Expanding itself more in the beginning where most vertices are labelled with a distance equal to ∞.

Other things we can notice are: that the number of iterations done by the algorithm tends to decrease as the density of the graph increases, where instead the number of edges visited by the algorithm at each iteration tends to increase. Meaning that despite the fact that we are doing less iterations in each execution the total amount of edges visited is still higher than before.

The time spent at each iteration depends on many factors, and not only the number of edges processed. In particular there are many overheads to be considered. For example the need to use atomic operations, because some CUDA threads can try to update at a time distances and predecessors for the same vertex. Then the possible low use of coalesced I/O operations, because the considered edges are normally spread in the compressed sparse row structure. All of which are particularly difficult to track with high-level libraries. However, we can still provide an explanation to the behaviour of the execution time of the parallel algorithm at low density by comparing the number of edges explored and the time spent.

We can see at both Figure 7.8 and Figure 7.9 that the parallel visit of a number of edges $e < 10^6$ usually takes less than ~ 0.5 ms, this because our GPU has ~10,000 CUDA streaming processors and is able to compute most of them in parallel exactly at the same time. With a number of edges e between $10^6 \leq e \leq 10^7$ the time slightly increase to $\sim 0.6/1.0$ ms. The algorithm seems to spike however when the number of edges $e > 10^7$, which takes between 2 ms and 10 ms depending on the size, here the computation must be distributed among different warps that are executed sequentially and divergency problems can slow down the overall execution time.

With this in mind we can see that most of iterations with the datasets at low density explore up to 10^6 edges, which keeps the overall execution time low, where instead at a higher density upon reaching iterations that explore more than 10^7 edges the execution time start to grow faster with the consequent drop in the speedup.

7.5 Testing with mesh graphs

7.5.1 Selected graphs

Starting from 7 meshes with a different number of triangles, we calculated for each of them 5 different graphs. Each graph has a number of vertices equal to number of vertices of the mesh but a different number of edges.

For each mesh the edges are defined as follows:

- **Primal**: The edges are the same of the mesh.
- **Extended Opposite**: Each vertex is connected with its direct neighbours and with the vertices opposite to it in the triangles adjacent to its 1-ring.
- **Extended K3**: Each vertex is connected with all the vertices up to its 3-ring.
- **Extended K4**: Each vertex is connected with all the vertices up to its 4-ring.
- **Extended K5**: Each vertex is connected with all the vertices up to its 5-ring.

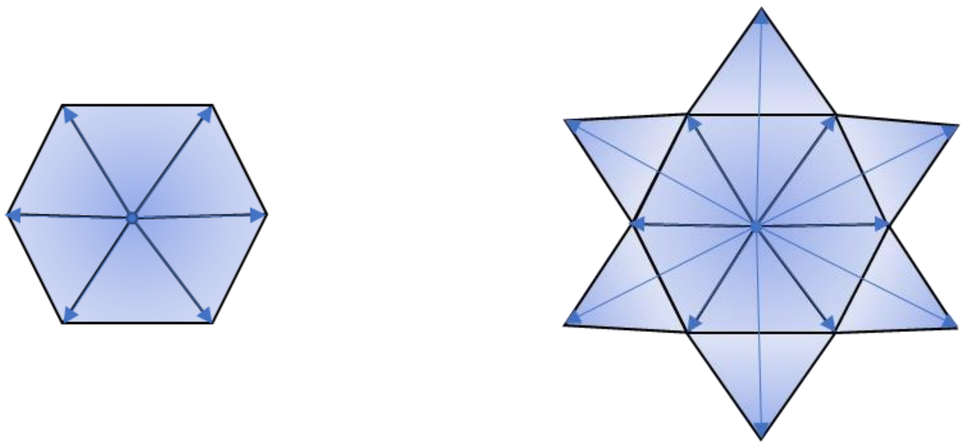


Figure 7.10: On the left an example of primal graph, where the central vertex is connected to its direct neighbours. On the right an example of extended opposite, where the central vertex is also connected to vertices opposite to its adjacent triangles.

7.5.2 Selected metrics

Graphs generated by a mesh structure are particularly interesting because of their degree distribution. In fact, most of the vertices of these type of graphs tends to have the same degree. With this in mind, we can try to establish what the most interesting metrics could be with these types of datasets.

The simplest ones, and the most obvious ones, are the degree metrics: the *average degree* can give us a good estimate of the degree distribution, where instead the *maximum degree* and *minimum degree* are mostly irrelevant since the degree distribution has low variance. Due to the high regularity of the graph other metrics like the *global clustering coefficient* and the *average clustering coefficient* are ineffective and don't show us much difference from a graph to another. The *density* offers us a wide range of values; however, these values are too scattered to provide any good metric of comparison between graphs with different vertices in this case.

DATASET	EDGES	AVERAGE DEGREE	AVERAGE DEGREE VARIANCE	GLOBAL CLUSTERING COEFFICIENT	AVERAGE CLUSTERING COEFFICIENT
PRIMAL_5000	29,988	5.99760	0.182794	0.265045	0.267130
EXTENDED_OPP_5000	59,976	11.99520	0.731177	0.247934	0.247618
EXTENDED_K3_5000	182,962	36.59240	3.634260	0.293703	0.293667
EXTENDED_K4_5000	306,922	61.38440	12.770200	0.277129	0.276815
EXTENDED_K5_5000	463,182	92.63640	40.303800	0.264424	0.263763
PRIMAL_50000	299,988	5.99976	0.180400	0.265016	0.267044
EXTENDED_OPP_50000	599,976	11.99950	0.721600	0.250360	0.250019
EXTENDED_K3_50000	1,832,482	36.64960	2.387970	0.300297	0.300329
EXTENDED_K4_50000	3,076,290	61.52580	5.244210	0.284939	0.284886
EXTENDED_K5_50000	4,645,520	92.91040	10.120600	0.272835	0.272724

Table 7.d: Metrics for some of the mesh graphs

7.5.3 Average execution times

Here we present the average execution of each algorithm as the average degree of vertices in the graph increase. Each line represents a group of mesh generated graphs all with the same number of vertices but different average degree, and each dot represent one of those graphs. On the x-axis we have the average degree, where on the y-axis we have the average execution time expressed on milliseconds. For the y-axis we use a logarithmic scale of base 10.

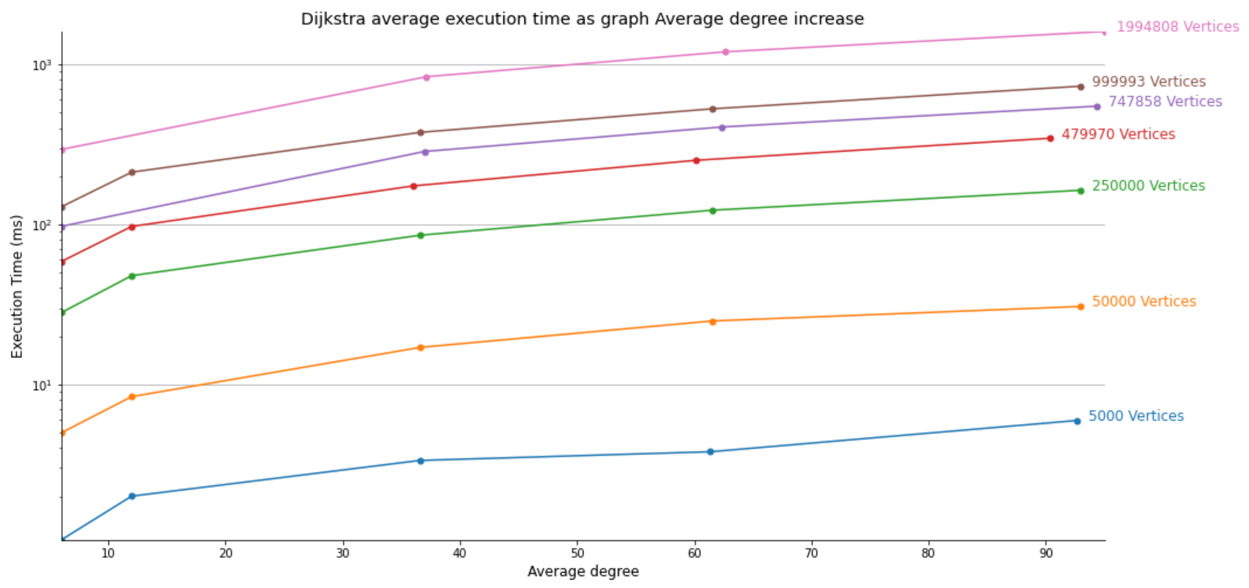


Figure 7.11: How the average execution time of Dijkstra changes when the average degree of the vertices in the graph increase

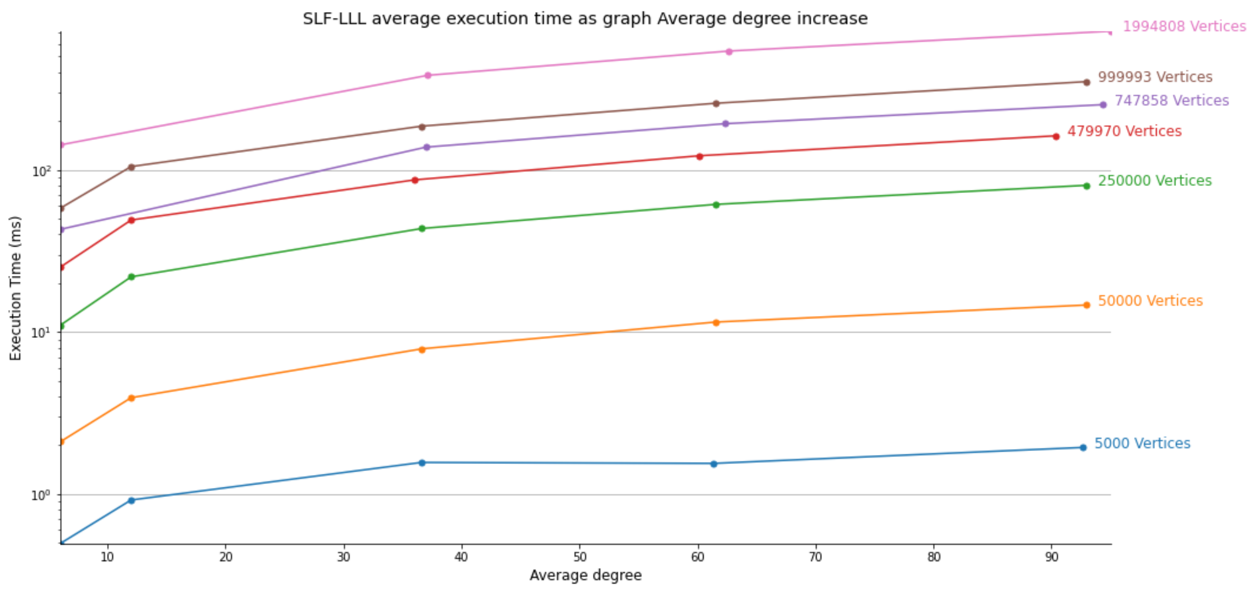


Figure 7.12: How the average execution time of SLF-LLL changes when the average degree of the vertices in the graph increase

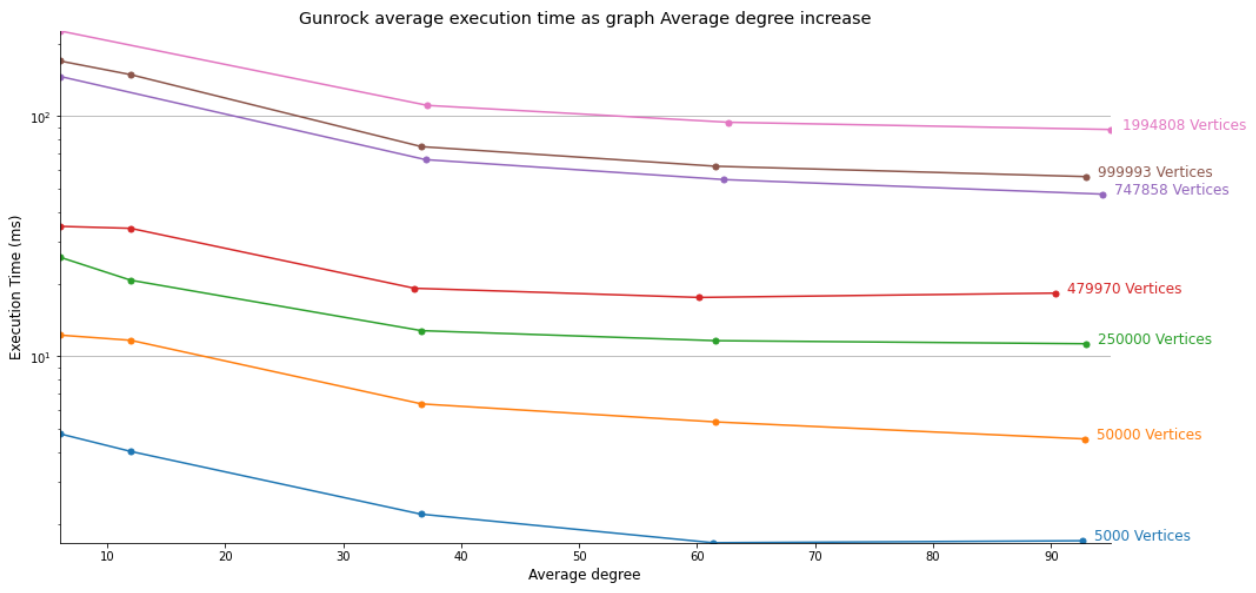


Figure 7.13: How the average execution time of Gunrock changes when the average degree of the vertices in the graph increase

Looking at both Figure 7.11 and Figure 7.12, which express the average execution time for the two sequential shortest path algorithms, we can see that the elapsed time tends to grow as the average degree of the graphs increase. As before, if we compare the average execution times for the sequential algorithms, we can see that that the values for the Dijkstra's algorithm are always higher than the ones of the SLF-LLL heuristic. For example, if we look at the graph with ~2,000,000 vertices we have $1{,}607.03$ ms for the Dijkstra algorithm against the 717.25 ms of the SLF-LLL heuristic.

Where instead, if we look at Figure 7.13, which presents the average execution time for the parallel shortest path algorithm, we can see the opposite trend. We have higher elapsed times with graphs where the average degree is low, and lower elapsed times with graphs where the average degree is high. This is related to the number of iterations done by the parallel algorithm, since highly regular graph with a low average degree requires many iterations in order to expand to the farthest vertices of the graph, but this will be covered more on details later.

7.5.4 Speedups

Continuing with the speedups, here we present how the speedup of the parallel algorithm changes when the average degree of vertices in the graph increase. As before each line represents a group of mesh generated graphs all with the same number of vertices but different average degree, and each dot represent one of those graphs.

On the x-axis we have the average, and on the y-axis we have the speedup of the parallel algorithm over the sequential ones. The red dotted line at $y = 1$ represents the threshold at which the parallel algorithm starts to perform better than the sequential algorithm.

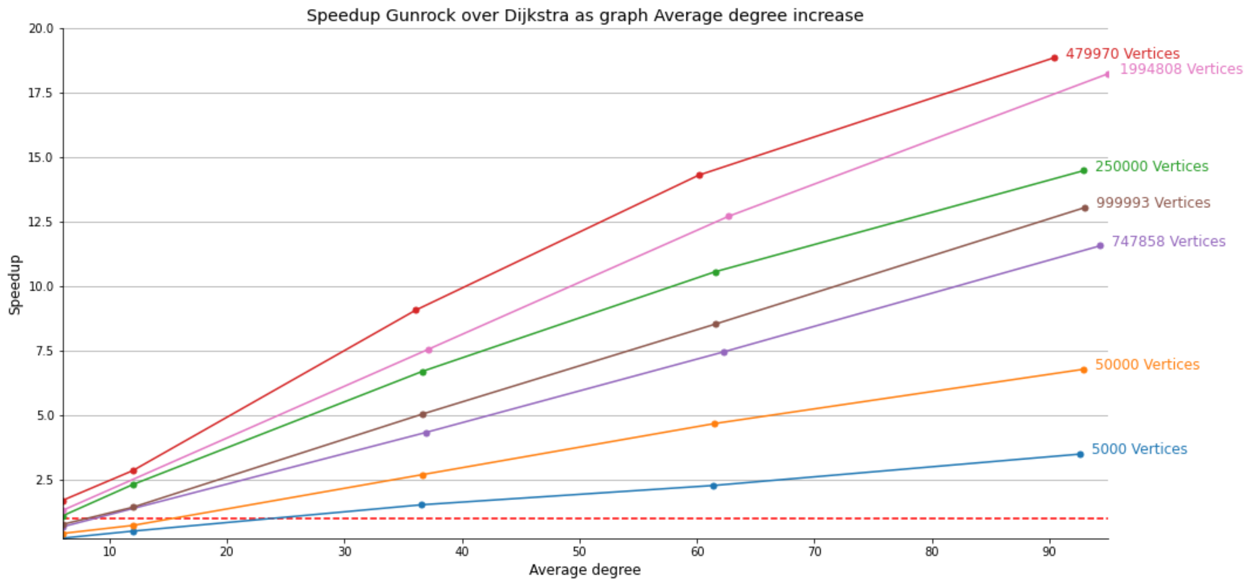


Figure 7.14: How the speedup of Gunrock over Dijkstra changes when the average degree of the vertices in the graph increase

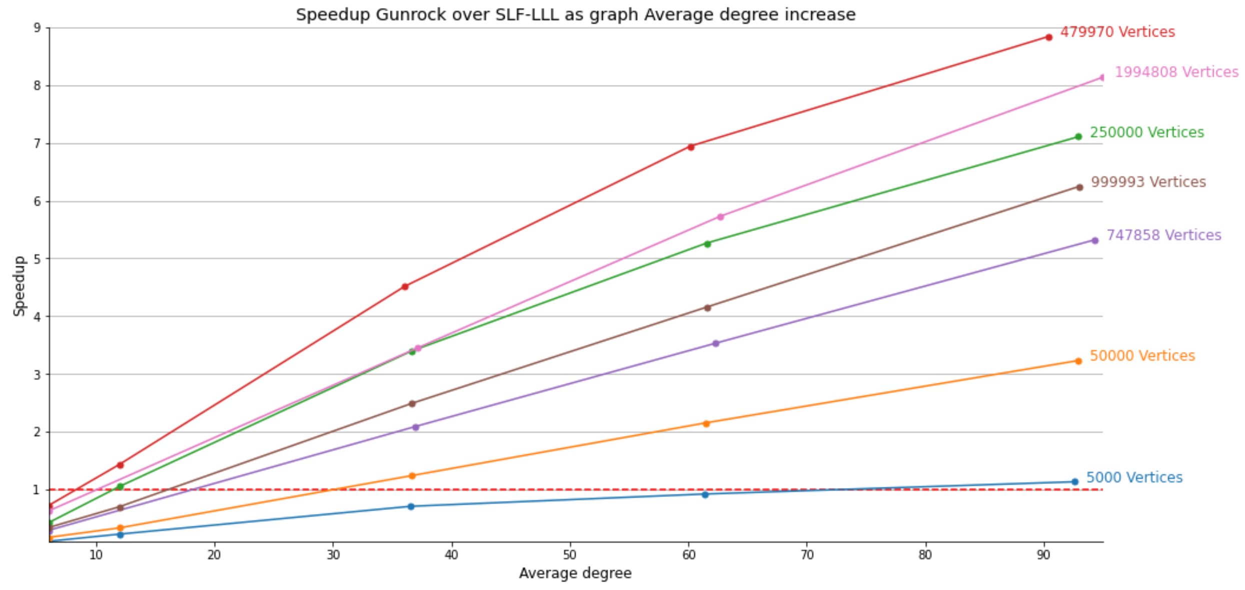


Figure 7.15: How the speedup of Gunrock over SLF-LLL changes when the average degree of the vertices in the graph increase

Looking at both Figure 7.14 and Figure 7.15, we can see that the speedup of the algorithm tends to increase as the average degree of vertices in the graphs increase.

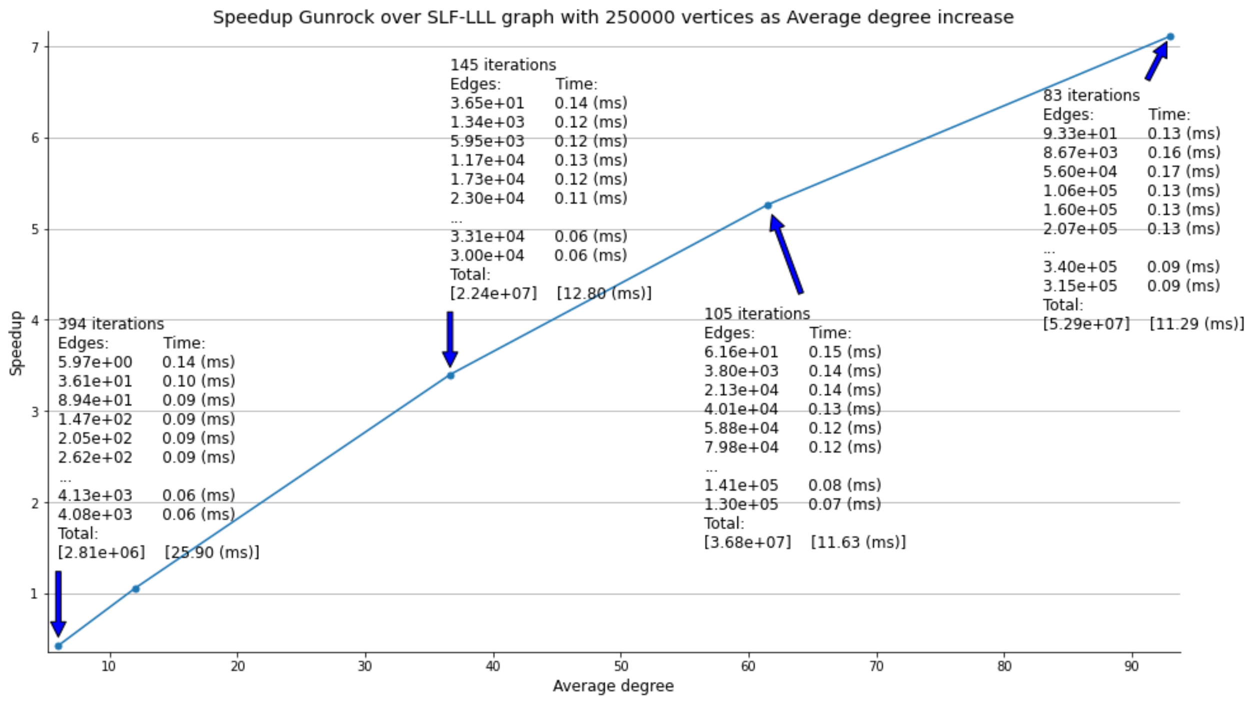
Going more into the details, we see at Figure 7.14 that the Gunrock algorithm for graphs with a small number of vertices (50,000 and 5,000) starts to perform better that Dijkstra only when the average degree of vertices graph reach values higher than 30. Where instead for the other graphs it starts to compete with Dijkstra already at an average degree of 12.

Looking at the speedup values at Figure 7.15 instead, we see that the graph with 5,000 vertices barely reach the threshold of 1 when the average degree is at its maximum, and the graph with 50,000 vertices start to gets us a reasonable speedup of ~ 2 only past an average degree of 60. The speedup for the other graphs instead grows faster and reaches reasonable values between ~ 2 and ~ 4.5 already at an average degree of 12.

Another interesting thing we can see on both graphs is related to the speedup obtained by the graphs with an high number of vertices. Up to graphs with ~500,000 vertices the speedup obtained at each different average degree is always higher than the speedup of graphs with a lower number of vertices at the same average degree. However, starting from graphs with ~750,000 vertices the speedup obtained drops, with values that goes below the speedup of the graph with 250,000 vertices for the same average degree. The same goes for the graphs with ~1,000,000 and ~2,000,000 vertices which obtains lower speedups than the graphs with 250,000 and ~500,000 vertices respectively.

In order to understand this behaviour, we proceed with an analysis similar to the one done during the previous testing (Section 7.4.6). Comparing the number of iterations done by the parallel algorithm, the number of edges explored parallelly and the elapsed time of each iteration.

Here we present, the speedup of the parallel algorithm over one of the sequential algorithms (in this case SLF-LLL) for the graphs with ~750,000 and 250,000 vertices. As before, we annotated the most relevant point of interest in the speedup with data coming from the executions: regarding the number of iterations done on average by the algorithm, the number of edges explored on average at each iteration and the time requested in milliseconds to complete the iteration.



394 iterations

Edges:	Time:
5.97e+00	0.14 (ms)
3.61e+01	0.10 (ms)
8.94e+01	0.09 (ms)
1.47e+02	0.09 (ms)
2.05e+02	0.09 (ms)
2.62e+02	0.09 (ms)
...	
4.13e+03	0.06 (ms)
4.08e+03	0.06 (ms)
Total:	
[2.81e+06]	[25.90 (ms)]

145 iterations

Edges:	Time:
3.65e+01	0.14 (ms)
1.34e+03	0.12 (ms)
5.95e+03	0.12 (ms)
1.17e+04	0.13 (ms)
1.73e+04	0.12 (ms)
2.30e+04	0.11 (ms)
...	
3.31e+04	0.06 (ms)
3.00e+04	0.06 (ms)
Total:	
[2.24e+07]	[12.80 (ms)]

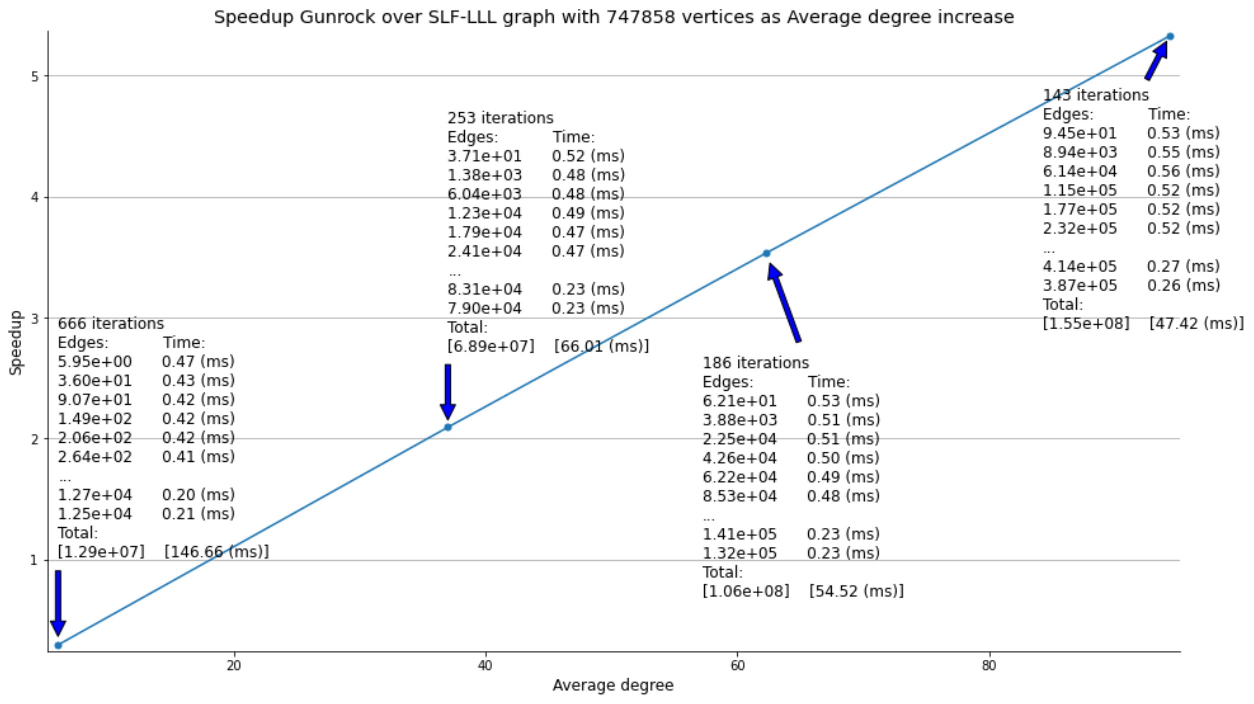
105 iterations

Edges:	Time:
6.16e+01	0.15 (ms)
3.80e+03	0.14 (ms)
2.13e+04	0.14 (ms)
4.01e+04	0.13 (ms)
5.88e+04	0.12 (ms)
7.98e+04	0.12 (ms)
...	
1.41e+05	0.08 (ms)
1.30e+05	0.07 (ms)
Total:	
[3.68e+07]	[11.63 (ms)]

83 iterations

Edges:	Time:
9.33e+01	0.13 (ms)
8.67e+03	0.16 (ms)
5.60e+04	0.17 (ms)
1.06e+05	0.13 (ms)
1.60e+05	0.13 (ms)
2.07e+05	0.13 (ms)
...	
3.40e+05	0.09 (ms)
3.15e+05	0.09 (ms)
Total:	
[5.29e+07]	[11.29 (ms)]

Figure 7.16: How the number of iterations and the number of edges explored by the Gunrock algorithm changes in relation to the speedup for the graphs with 250,000 vertices



666 iterations

Edges:	Time:
5.95e+00	0.47 (ms)
3.60e+01	0.43 (ms)
9.07e+01	0.42 (ms)
1.49e+02	0.42 (ms)
2.06e+02	0.42 (ms)
2.64e+02	0.41 (ms)
...	
1.27e+04	0.20 (ms)
1.25e+04	0.21 (ms)
Total:	
[1.29e+07]	[146.66 (ms)]

253 iterations

Edges:	Time:
3.71e+01	0.52 (ms)
1.38e+03	0.48 (ms)
6.04e+03	0.48 (ms)
1.23e+04	0.49 (ms)
1.79e+04	0.47 (ms)
2.41e+04	0.47 (ms)
...	
8.31e+04	0.23 (ms)
7.90e+04	0.23 (ms)
Total:	
[6.89e+07]	[66.01 (ms)]

186 iterations

Edges:	Time:
6.21e+01	0.53 (ms)
3.88e+03	0.51 (ms)
2.25e+04	0.51 (ms)
4.26e+04	0.50 (ms)
6.22e+04	0.49 (ms)
8.53e+04	0.48 (ms)
...	
1.41e+05	0.23 (ms)
1.32e+05	0.23 (ms)
Total:	
[1.06e+08]	[54.52 (ms)]

143 iterations

Edges:	Time:
9.45e+01	0.53 (ms)
8.94e+03	0.55 (ms)
6.14e+04	0.56 (ms)
1.15e+05	0.52 (ms)
1.77e+05	0.52 (ms)
2.32e+05	0.52 (ms)
...	
4.14e+05	0.27 (ms)
3.87e+05	0.26 (ms)
Total:	
[1.55e+08]	[47.42 (ms)]

Figure 7.17: How the number of iterations and the number of edges explored by the Gunrock algorithm changes in relation to the speedup for the graphs with 747,858 vertices

Looking at Both Figure 7.16 and Figure 7.17, we can see that the number of edges visited by the algorithm tends to follow the same pattern at each average degree. Where it starts from a small number of edges and progressively grows, exploring more edges at each iteration. However, unlike in previous testing with random generated graphs here we have a smaller number of edges explored at each iteration and significantly higher number of iterations. This can be attributed to the highly regular degree distribution of the graphs, that requires the algorithm more iterations to expand to the farthest vertices especially when the average degree is small.

Another thing that we can notice is that most of the iterations in each algorithm execution seems to take the same amount of time, meaning that the total number of iterations done by the algorithm is more relevant than before. As we have seen in Section 4.4, each iteration doesn't come for free, since at the end of each parallel operator we have global barriers that need to ensure the synchronization of the data.

Going over the number of iterations at each average degree we can see that the graph at Figure 7.17 with ~750,000 vertices take around 1.7 times more iterations than the graph at Figure 7.16 with 250,000 vertices in order to reach the results. Which due to overheads that comes with the synchronization at each iteration can explain why the speedup of these graphs is lower than the ones with a smaller number of vertices.

7.6 Testing with graph with high degree variance

Lastly, since we worked with graph generated from meshes, which as we said earlier are highly regular graphs, during our testing we thought it would be interesting also to explore some random generated graph with the same average degree but different degree distribution, in order to see how it affects the speedup over the graphs.

7.6.1 Definition

Given a normal distribution $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$, where μ is the mean of the distribution and σ is its standard deviation, we construct an undirected graph $G = (V_n, E)$ with $n \in \mathbb{N}$ vertices, where $V_n = (v_1, v_2, \dots, v_n)$ and such that the degree k_i of each vertex v_i is randomly selected from the distribution $f(x)$.

Note on the $\boldsymbol{\sigma}$ value: Following 3-sigma rule, about 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95% of the values lie within two standard deviations; and about 99.7% are within three standard deviations. Exploiting this, since the degree $k_i \in \mathbb{N}$, we need to select a σ parameter such that the values drawn within three standard deviations are strictly positive.

7.6.2 Selected graphs

Starting from these criteria we generated different classes of graph datasets, all with the same average degree μ, but exploiting different combination of number of vertices n and σ values. Here there are some of the generated graph datasets with $\mu = 92$:

DATASET	VERTICES ($\boldsymbol{n}$)	STANDARD DEVIATION ($\boldsymbol{\sigma}$)	AVERAGE DEGREE ($\boldsymbol{\mu}$)	FILE FORMAT
RAND50000_92	50,000	[0,30]	92	Matrix Market
RAND100000_92	100,000	[0,30]	92	Matrix Market
RAND250000_92	250,000	[0,30]	92	Matrix Market
RAND500000_92	500,000	[0,30]	92	Matrix Market
RAND50000_184	50,000	[0,60]	184	Matrix Market
RAND100000_184	100,000	[0,60]	184	Matrix Market
RAND250000_184	250,000	[0,60]	184	Matrix Market
RAND500000_184	500,000	[0,60]	184	Matrix Market

7.6.3 Selected metrics

Given the definition used to generate the graphs, most of the metrics used previously are not useful in this case. And the only one worth exploring is the *average degree variance*, which can give us an idea of how scattered the degree of the vertices inside the graph are.

DATASET	EDGES	DENSITY	AVERAGE DEGREE	AVERAGE DEGREE VARIANCE
RAND50000_184_0	9,200,000	0.003680	184.000	0.0000
RAND50000_184_5	9,225,436	0.003690	184.509	25.2011
RAND50000_184_10	9,225,706	0.003690	184.514	100.6000
RAND50000_184_15	9,225,300	0.003690	184.506	225.1240
RAND50000_184_20	9,229,578	0.003692	184.592	403.5160
RAND50000_184_25	9,220,976	0.003688	184.420	622.8670
RAND50000_184_30	9,216,680	0.003687	184.334	894.3810
RAND50000_184_35	9,239,566	0.003696	184.791	1,229.0100
RAND50000_184_40	9,234,960	0.003694	184.699	1,591.8300
RAND50000_184_45	9,240,824	0.003696	184.816	2,025.3200
RAND50000_184_50	9,205,128	0.003682	184.103	2,484.6400
RAND50000_184_55	9,215,482	0.003686	184.310	2,993.9800
RAND50000_184_60	9,233,696	0.003694	184.674	3,633.9200

Table 7.e: Metrics for some of the random generated graphs

7.6.4 Speedups

Here we present how the speedup of the parallel algorithm changes when the average degree variance of the graph increase. As before each line represents a group of random generated graphs all with the same number of vertices but different average degree variance, and each dot represent one of those graphs.

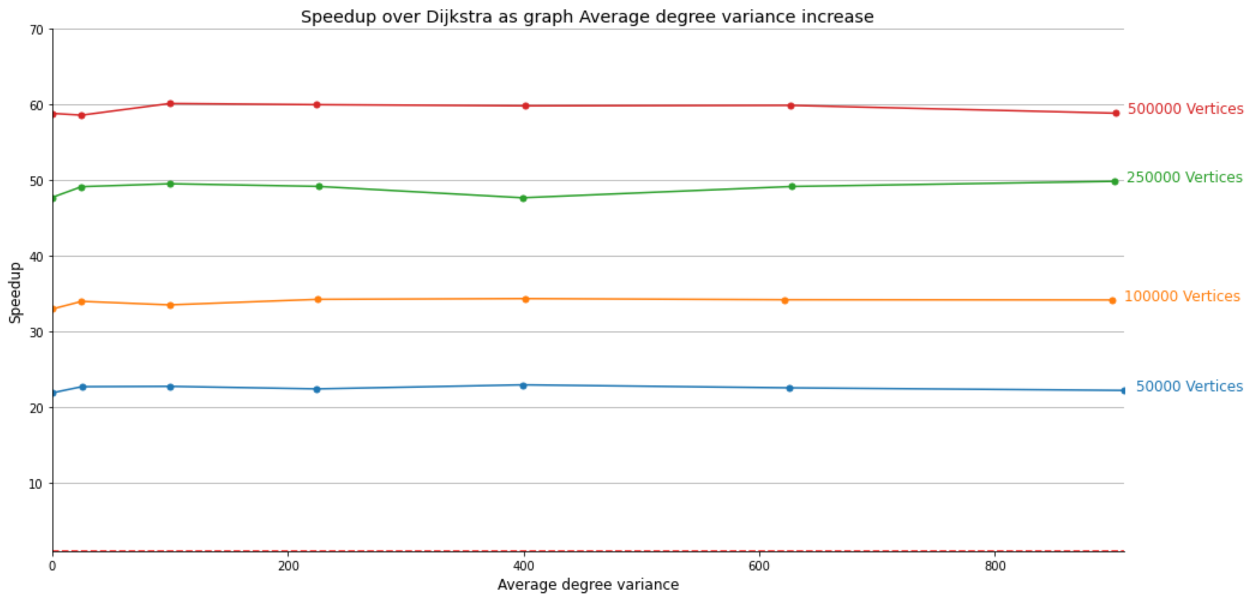


Figure 7.18: How the speedup of Gunrock over Dijkstra changes when the average degree variance of the vertices in the graph increase

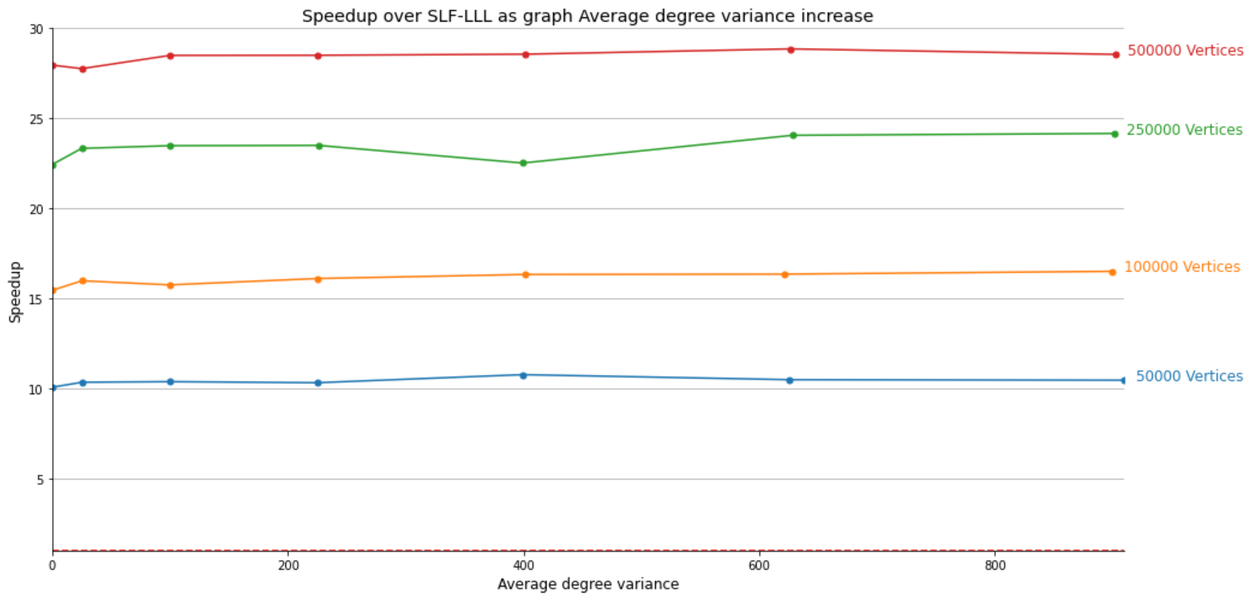


Figure 7.19: How the speedup of Gunrock over SLF-LLL changes when the average degree variance of the vertices in the graph increase

Looking both Figure 7.18 and Figure 7.19 we can see that the speedup obtained over either of the sequential algorithms doesn't seems to be affected by the increase of the average degree variance.

Chapter 8 Conclusion and Future works

The GPU parallel solution of the shortest path problem provided by the Gunrock library has shown notably improvements only for the larger graph datasets (with a high number of vertices and/or edges). Also, the speedups obtained are different based on the sequential algorithm that we consider, the Dijkstra's algorithm known for its result on smaller graphs and the SLF-LLL heuristic which as shown better result on our datasets. All this considered, even though the improvements are notable I wouldn't advise to buy a GPU only for the parallel shortest paths computation.

Looking at our future work on the topic we would like, to develop a CPU alternative using a similar multithread approach but trying to distribute the workload on the available x86 cores of a workstation, which at the state-of-the-art are 40 cores for Intel architecture and 96 cores for AMD architecture.

To mitigate the constraints of the amount of memory available, we could try more powerful graphics cards. Two of the most interesting GPU available are the NVIDIA A100 Tensor Core with 80 GB of RAM and the AMD Instinct MI250 with 128 GB of RAM. However, due to the high price a more in-depth analysis on the performances is required. Alternatively, exploiting multiple GPUs at the same time can also mitigate the constraints related to the single GPU memory. But on the other side it introduces new challenges related to the synchronization and distribution of data on multiple devices. Nonetheless, parallel graph libraries like CuGraph and Gunrock have already implemented multi-GPU versions of their graph primitives, and it would be interesting to evaluate what are the performances and thresholds of these methods.

Lastly, since the SLF-LLL heuristic has proven to perform greatly for most of the graph datasets that we used, and that unlike the Dijkstra's algorithm the SLF-LLL heuristic has room for parallelization, we could implement a parallel version of the algorithm. Using an approach similar to the Near and Far pile proposed by Davidson et al., which we discussed in Section 3.1.2, the algorithm could exploit the large label last (LLL) method to prioritize part of the vertices labelled with values lower than average vertex label on queue. The algorithm could be easily expressed with the use of the Gunrock Essential library, given that some efficient operators to join and split the frontiers are implemented.

Bibliography

[1] Newman, Mark, Networks: An Introduction, 1st edn (Oxford,2010; online edn, Oxford Academic, 1 Sept. 2010)

[2] Davidson, Andrew, et al. "Work-efficient parallel GPU methods for single-source shortest paths." *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014.

[3] Meyer, Ulrich, and Peter Sanders. "Δ-stepping: a parallelizable shortest path algorithm." *Journal of Algorithms* 49.1 (2003): 114-152.

[4] Bertsekas, Dimitri. *Network optimization: continuous and discrete models*. Vol. 8. Athena Scientific, 1998.

[5] Boisvert, Ronald F., Ronald F. Boisvert, and Karin A. Remington. *The matrix market exchange formats: Initial design*. Vol. 5935. US Department of Commerce, National Institute of Standards and Technology, 1996.

[6] Nazzaro, Giacomo, Enrico Puppo, and Fabio Pellacini. "geoTangle: interactive design of geodesic tangle patterns on surfaces." *ACM Transactions on Graphics (TOG)* 41.2 (2021): 1-17.

[7] Wang, Yangzihao, et al. "Gunrock: GPU graph analytics." *ACM Transactions on Parallel Computing (TOPC)* 4.1 (2017): 1-49.

[8] Osama, Muhammad, Serban D. Porumbescu, and John D. Owens. "Essentials of Parallel Graph Analytics." *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022.

[9] Busato, Federico, and Nicola Bombieri. "An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures." IEEE Transactions on Parallel and Distributed Systems 27.8 (2015): 2222-2233.