FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Implementing multi-threaded algorithms to the JGraphT library
Student:	Mgr. Barbora Kolomazníková
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the thesis is to analyze and implement support for parallel, multi-threaded algorithms to the JGraphT library.
- Get familiar with the JGraphT library.
- Analyze possibilities of multithreaded processing in the library.
- Get familiar with parallel graph algorithms and corresponding theory.
- Propose multithreaded implementation of the following algorithms to the JGraphT library: breadth-first search, depth-first search, Dijkstra shortest path algorithm, Bellman-Ford algorithm.
- Use existing structures provided by the library appropriately.

Electronically approved by Ing. Michal Valenta, Ph.D. on 30 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementing multi-threaded algorithms to the JGraphT library

Mgr. Barbora Kolomazníková

Department of software engineering
Supervisor: Ing. Ondřej Guth, Ph.D.

May 11, 2023

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Guth, Ph.D. for his valuable insights and recommendations. In addition, I would like to thank my family for their continuous support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.
I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2023 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kolomazníková, Barbora. *Implementing multi-threaded algorithms to the JGraphT library.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

The aim of the thesis is to implement four parallel, multi-threaded algorithms to the JGraphT – a Java library providing graph data structures and corresponding algorithms. The selected multi-threaded algorithms to be implemented are BFS, DFS, Dijkstra, and Bellman-Ford. First part of the thesis provides an overview of the JGraphT and its main components. Second part presents theoretical description of the selected algorithms and their parallel versions. Third part introduces implementation requirements and describe the implementation of the algorithms. Last part of the thesis focuses on presenting the testing results, evaluating performance of the implementations, and assessing the fulfillment of the outlined implementation requirements. The main outcome of the thesis is the implementation of the selected multi-threaded algorithms which uses existing components of the JGraphT appropriately, is fully tested and documented.

Keywords graph algorithms, BFS, DFS, Dijkstra, Bellman-Ford, multi-threading, parallelism, implementing multi-threaded algorithms, JGraphT, Java

Abstrakt

Cílem této práce je implementaci čtyř vybraných vícevláknových algoritmů do knihovny JGraphT. Jedná se o Java knihovnu poskytující grafové datové struktury a související algoritmy. Čtyři vybrané vícevláknové algoritmy jsou BFS, DFS, Dijkstra a Bellman-Ford. První část práce se zabývá samotnou knihovnou a popisem jejích hlavních částí. Druhá část se soustředí na samotné algoritmy, přesněji jejich vícevláknové verze a jejich teoretický popis. Třetí část obsahuje implementační požadavky a technické detaily samotné implementace. Poslední část práce potom prezentuje výsledky testů, hodnotí efektivitu jednotlivých implmementovaných algoritmů a zkoumá naplnění implementačních požadavků. Hlavním výstupem této práce je implementace daných algoritmů, která vhodně využívá prvky knihovny JGraphT, je řádně otestována a zdokumentována.

Klíčová slova grafové algoritmy, BFS, DFS, Dijkstra, Bellman-Ford, vícevláknové zpracování, paralelismus, implementace vícevláknových algoritmů, JGraphT, Java

Contents

List of Algorithms

List of Source Code Examples

List of Figures

Introduction

It is not so long ago when hardware capabilities were fairly limited and its prices were rather high, as mentioned in [1]. However, as the author in [1] points out these days we see the opposite to be true – there are systems offering terabytes of memory with tens or hundreds of CPUs/GPUs with affordable prices. For instance, according to [1] the hardware storage sizes have risen 40,000-times over last 40 years and their prices have decreased to one hundredth of the prices of their 1980s counterparts. Consequently, the need to properly utilize those possibilities arises. Moreover, with the rise of the online world when companies are offering their services to billions of people, there is increasing demand for processing of large amounts of data. These two trends – accessibility of powerful hardware and the increasing amounts of data – moved the attention of the engineering community to creating fast, parallel, and possibly multi-threaded algorithms that would enable scalable processing of big data with proper utilization of the available hardware.

The aim of the thesis is to provide analysis of the JGraphT library and to extend the library by implementing four multi-threaded graph algorithms, namely breadth-first search algorithm, depth-first search, Dijkstra, and Bellman-Ford algorithm. Since sequential versions of BFS, Dijkstra and Bellman-Ford algorithms are already implemented in JGraphT, the goal of the new implementations is to provide the same interface and functionalities as the existing ones. Additional aim is to verify the correctness and efficiency of the implementations. This is done by using unit tests and comparison tests that test the outputs of parallel algorithms and compare them to the outputs of sequential algorithms. Moreover, performance is assessed by using performance tests in order to determine the efficiency of the parallel algorithms. Since the library contains single-threaded versions of the aforementioned algorithms, those will be tested as well to assess the efficiency improvement. The implementation will utilize existing library components appropriately and adhere to the library's conventions. In addition, all new code will be documented.

The first chapter focuses on the analysis of the JGraphT library and its individual modules. In particular, it targets the three main modules – `jgrapht-core`, `jgrapht-guava` and `jgrapht-io`. Most detailed analysis is provided for the `jgrapht-core` module as it is the one that will be extended as part of this thesis. Chapter 2 then provides an overview of the four algorithms that will be later implemented to the JGraphT library. It describes sequential versions of the algorithms as well as their parallel counterparts. Main focus is put on the parallel implementations that are later used in this work. However, different approaches to parallelizing these algorithms are mentioned as well. Chapter 3 presents the implementation requirements (both functional and non-functional), and introduces the implementation of the algorithms described in chapter 2. It describes the Java concepts related to multi-threading used for the implementation and provides a discussion of possible alternatives. Moreover, it introduces one of the new features in Java – virtual threads, and depicts their usage for implementing the underlying algorithms. Chapter 4 discusses outputs of unit and comparison tests, and presents the results of performance testing where the performance of the implemented multi-threaded algorithms are compared to the performance of their sequential counterparts.

Chapter 1

Analysis of JGraphT library

In this chapter we describe and analyze the main components of the JGraphT library.

JGraphT is an open-source library that provides data structures representing various graph types (such as directed and undirected graphs) and implementation of corresponding graph algorithms, as mentioned in [2]. Latest versions run on Java 11. Apart from graph representation the library also contains listeners that enable to react to graph modifications. In addition, the library has basic support for concurrency, which will be the main focus of this thesis. In particular, it provides a thread-safe wrapper over standard graph implementations.

Following sections target the main components of the library – `jgrapht-core`, `jgrapht-guava` and `jgrapht-io`.

1.1 jgrapht-core

`jgrapht-core` is the main part of the whole library. It contains all graph representations as well as implemented algorithms. According to [2] the module comprises of six packages – `org.jgrapht.alg`, `org.jgrapht.event`, `org.jgrapht.generate`, `org.jgrapht.graph`, `org.jgrapht.traverse` and `org.jgrapht.util`. Packages `org.jgrapht.alg` and `org.jgrapht.graph` consist of graph and algorithm implementations and will be described in greater detail in the following subsections. `org.jgrapht.util` provides utility and helper classes/methods. `org.jgrapht.traverse` contains graph iterators, while `org.jgrapht.generate` involves various graph generators – e.g. random graph generators. Finally, events reflecting graph changes and their corresponding listeners are part of `org.jgrapht.event`.

The root interface, as mentioned in [2], of the whole library is `Graph`. It takes two parameterized types representing types of edges and vertices. Baseline graph operations, such as adding and removing edges/vertices and corresponding getters, are declared by the interface. Other graph classes either directly implements this interface or extends other classes that implement `Graph`.

1.1.1 org.jgrapht.graph

The package holds classes representing different types of graphs. As mentioned above, the graph implementations are based on `Graph` interface.

`AbstractGraph` is an abstract class providing baseline implementation of the `Graph` interface. Specifically, as mentioned in [2] the class involves methods for removing edges/vertices, checking edge existence, and conversion methods to `String`. Moreover, the class overrides `hashCode` and `equals` methods in order to check for graph equivalence. In particular, two graphs are considered equal, if they are of the same type and if they contain the same sets of vertices and edges (also with the same weights).

Abstract class `AbstractBaseGraph` builds on `AbstractGraph` and provides further implementation of the `Graph` interface. Its subclasses form various types of graphs, e.g. directed vs. undirected. They place restrictions on the graph properties through `AbstractBaseGraph`'s `GraphSpecificsStrategy` property. In the documentation it is noted that this class is not concurrent safe and therefore cannot be used for simultaneous reads and writes.

If we need a graph implementation that can be used in concurrent setting, the library provides `AsSynchronizedGraph` implementation of the `Graph` interface. The class uses an instance of `ReentrantReadWriteLock` in order to maintain thread-safety of read/write operations. The implementation also provides a copyless mode which prevents the caller from creating collection copies. The purpose is to avoid situations when a new copy of `AsSynchronizedGraph` is made while the source instance is being modified by other threads. As explained in [2] the copyless mode forces the caller to explicitly synchronize the iteration instead.

According to [2] the library allows creating graphs of various types simply by creating instances of classes implementing `Graph` interface. According to [2] the following graph properties are supported: directed/undirected edges, self-loops, multiple edges, and weighted edges. The first property either determines for every edge its source and target vertex (directed edges) or does not impose any restriction on the vertices treating the edge as undirected. Self-loops, according to [2], allows for existence of edges that connect a vertex to itself. Multiple edges implies the possibility of having more than one edge

between particular two vertices, as stated in [2]. Last but not least, weighted edges property let edges be assigned a particular weight represented as `double`. As [2] points out, unweighted graphs are simply treated as weighted ones with all edges having the same weight of 1.0.

Regarding graph modifications, all `Graph` implementations provide methods for add new elements to graphs (vertices, edges) as well as methods for removing those elements. As mentioned in [2], since these methods are built on collections from `java.util`, modification operations possess the following properties. Firstly, duplicates are discarded for unique collections. That is not the case for non-unique collections, such as graphs that allows for multiple edges. In this case, we can have more than one edge between a pair of vertices. Secondly, removing a non-existing object from a graph does not result in an error, as stated in [2]. Lastly, accessing a non-existing element throws `IllegalArgumentException`. For example, as mentioned in [2], an error is thrown whenever we try to get a list of edges of some vertex which is not part of the underlying graph.

The `builder` package of `org.jgrapht.graph` provides `GraphBuilder`, another implementation of `Graph` interface. According to [2], the `GraphBuilder` allows for more convenient creation of graph objects by providing a possibility of method chaining. In addition, the package includes `GraphTypeBuilder` class which is a builder for various graph types with the properties described in the previous paragraph. In particular, the `GraphTypeBuilder` provides methods for instantiating graphs with directed/undirected edges, self-loops, weighted edges, and multiple edges. It also allows for creation of mixed graphs, i.e. graphs that have both directed and undirected edges.

1.1.2 org.jgrapht.alg

This module provides implementations of various graph algorithms as well as corresponding data structures used to represent the results of the algorithms. In this analysis we focus on two parts of the module that are of our interest – `shortestpath` and `spanning`. These two packages will be extended in the following chapters by adding parallel implementations of Breadth-first search, Dijkstra algorithm and Bellman-Ford algorithm to the `shortestpath` and parallel Depth-first search to the `spanning`.

The `shortestpath` package comprises of implementations of several algorithms for finding shortest paths in various types of graphs. Most importantly it contains sequential versions of BFS, Dijkstra and Bellman-Ford which will be later used as benchmarks for corresponding parallel implementations.

Similarly, the `spanning` package includes algorithms for finding minimum spanning trees, such as Kruskal or Boruvka algorithms. We lates extend this package with a parallel implementation of DFS returning spanning tree for a given source vertex. However, the found spanning tree will not be minimal, contrary to the existing algorithms already implemented in the package.

According to [2], it is common to use an interface when there are several algorithms solving the same problem. Such interfaces are stored in `org.jgrapht.alg.interfaces`. There can be found interfaces `ShortestPathAlgorithm` and `SpanningTreeAlgorithm` that will be used later for implementation of the parallel algorithms. The `ShortestPathInterface` requires implementation of the `getPaths` method which provides shortest paths to all reachable vertices from some provide source vertex. On the other hand, `SpanningTreeAlgorithm` interface contains class `SpanningTreeImpl` which is a representation of a spanning tree that will be used as a return value from corresponding algorithms.

1.2 jgrapht-guava

Based on [2], `jgrapht-guava` provides various adapters for `guava` graphs in order to make them compatible with `JGraphT`.

Base implementation is defined by `BaseGraphAdapter` abstract class. The class takes a type of one of `guava`'s graphs as its parameter. This `guava`'s graph type will then be *adapted*, i.e. provided with the necessary `Graph` methods to make it compatible with `JGraphT`. In addition, there are two other similar abstract classes – `BaseNetworkAdapter` and `BaseValueGraphAdapter` – providing adapters for `guava`'s `Network` and `ValueGraph`, respectively. The package then contains implementations of these abstract classes in both mutable and immutable versions.

Apart from adapters the package contains two helper classes – `ElementOrder` and `ElementOrderMethod`. The first one provides means for order elements in a set. The ordering is done either by using a provided comparator or by mapping elements to `Long` instances that act as ordering indices. Regarding `ElementOrderMethod`, this class exists in order to ensure the existence of element ordering. In particular, any set of elements should be subjected to ordering either by using a provided comparator, by using a natural comparisons (i.e. the provided elements should be comparable), by using a guava comparator, or by using an internal ordering, i.e. ordering based on `Long` indices mentioned above. As is noted in the documentation, the last ordering method, i.e. ordering by indices, requires extra memory for storing the indices, and

imposes extra time costs due to searching the map holding the indices.

1.3 jgrapht-io

This module provides means for exporting and importing graphs in various formats. In particular, it contains `GraphExporter` and `GraphImporter` interfaces that hold methods for respective input-output operations.

There are various implementations of the above-mentioned interfaces. For example, the module contains exporters/importers in formats such as CSV, matrix (text-based representations used by software such as Matlab), JSON, DOT or GEXF.

Apart from the above-mentioned interfaces the module also provides `BaseEventDrivenImporter` abstract class. This class holds various consumers of import events and notify them in case of event occurrence. As noted in the documentation, the event-driven importers take care of the notification only and the actual work is done by the provided consumers.

Chapter 2

Description of the selected multi-threaded algorithms

This chapter provides description of the targeted algorithms: breadth-first search, depth-first search, Dijkstra, and Bellman-Ford algorithm. Focus is put on their multi-threaded versions and ways of efficient implementation.

2.1 Breadth-first search

The breadth-first search algorithm is one of the graph traversal algorithms used to find the shortest path. In addition, the algorithm can also be used to find connected components of a graph or determine bipartite graphs as stated in [3]. As the authors mention, BFS is used in wide range of areas, e.g. image processing, exploring social and semantic graphs, and machine learning.

The algorithm is described in 1. It start with an initialization of the distance array *Dist* on lines 1-3. At the beginning the distance from source vertex s to all other vertices is infinity – the algorithm will then update these values accordingly. Line 4 sets the distance from source to itself to zero. The algorithm then start with examining the source vertex and its neighbors, which is depicted on line 5 by adding the source vertex to the BFS queue. The loop on lines 6-14 is then the core of the algorithm. For every vertex u in the queue, we start by removing it from the queue (line 7) and then continue with examining all its neighbors (lines 8-12). If the distance from u to its neighbor is infinity, i.e. the distance has not been updated yet, we set the correct value and put the neighbor to the BFS queue (in order to examine its neighbors in the next round). As mentioned in [4], such algorithm has time efficiency equal to $\mathcal{O}(n+m)$ where n is the number of vertices in graph G and m is the number of edges.

As the authors in [3] point out, there are several approaches to parallel optimization of the BFS algorithm. One approach is the container-centric one. As stated in [3] this optimization takes advantage of using multiple container, or data structures, for storing intermediate results. For example, two separate lists can be used and their roles will be switched after every iteration (as mentioned in [3]). The lists can be either local to every thread or it is also possible to use global lists. In this case, it is necessary to handle concurrent writes, as noted by the authors in [3]. Apart from lists it is possible to use also other data structures, such as the bag structure. In short, the bag structure is a dynamic unordered set with the following methods: create (creates new bag), insert (inserts new objects to the bag), union (merges two bags into one), split (splits one bag into halves). This structure and its usage for BFS is described in detail in [4]. Another approach for optimizing BFS is the vertex-centric approach. According to [3], in this case the parallelism is implemented in a way that each thread is assigned to a particular vertex. The process of examining vertices in one iteration can then be run in parallel. As the authors in [3] point out, the processing needs to be synchronized at the end of each vertex examination by using a barrier. This approach, as mentioned by the authors, does not need any additional data structures. Apart from the two mentioned approaches, there are other ways of parallelizing BFS. For example, [5] presents an extension of BFS using GPU optimizations. Similarly, [6] provides a BFS algorithm suited for CPU/GPU hybrid computing environment with the usage of task-based parallelization.

The parallel implementation presented later in this thesis uses the vertex-centric approach described above. The algorithm is shown in 2. It is mostly the same as the sequential version in 1, the main differences being the parallelization of the loop on line 8 and barrier synchronization at the end of each while-iteration. Synchronization is needed when a thread updates the $Dist$ array and the FIFO queue Q. The barrier synchronization then ensures that the next while-iteration will not begin until all threads finish the current one.

2.2 Depth-first search

Depth-first search is a graph traversal algorithm similar to BFS. Unlike BFS, it does not find the shortest path in a graph, since it explores graph vertices in a different manner. However, DFS is suitable for finding spanning trees in graphs. It can also detect cycles in graph paths.

The sequential DFS is depicted by algorithm 3 (as described in [7]). The algorithm starts with marking all vertices as not visited (lines 2-4). Then it iterates through the vertices and for all unvisited ones generates a new label

```
   input  : Graph G = (V, E), set of vertices V, set of edges E, source
            vertex s, empty FIFO queue Q, empty distance array Dist
   output: Array Dist with shortest-paths from s to all other vertices
 1 foreach u ∈ V \ {s} do
 2 |  Dist[u] = ∞
 3 end
 4 Dist[s] = 0
 5 Q = {s}
 6 while Q is not empty do
 7 |  u = Q.pop()
 8 |  foreach v ∈ V such that (u, v) ∈ E do
 9 |  |  if Dist[v] == ∞ then
10 |  |  |  Dist[v] = Dist[u] + 1
11 |  |  |  Q.put(v)
12 |  |  end
13 |  end
14 end
```

Algorithm 1: Breadth-first search algorithm (source [4])

(line 7) and runs the $EXPLORE$ procedure (line 8). This procedure is described on lines 11-18. It begins with marking vertex v as visited (line 12). Then it loops over all neighbors of v and for the not visited ones assigns the same label (line 15) and then explores such a neighbor by recursively calling $EXPLORE$ on it. Assigning the same label to a neighbor u as to the original vertex v essentially puts those two vertices to the same connected component of the graph. At the end of the algorithm array *label* holds the information of spanning trees present in the graph. However, note that the found spanning trees are not minimal.

As mentioned in [8] there are two main approaches to parallelizing DFS. The first one, as stated by the authors, is the stack-splitting approach. As described in [8] the approach starts with one thread processing the root vertex of the underlying graph. Once enough sub-tasks are generated, the thread splits its stack and assign the sub-tasks to other waiting threads. The waiting threads then start processing the assigned sub-tasks and do similar splitting once they generate enough work-load. This approach is used for example in [9]. As noted by the authors in [9] the efficiency is strongly impacted by the way the work is distributed among threads. Another approach to paralellizing DFS mentioned in [8] is the fixed-packet approach. Unlike the previous approach, fixed-packet DFS distributes work among threads in packets of fixed size (i.e. there is no dynamic splitting as in the stack-splitting approach). When a thread finishes its work on the assigned packet, it requests a new packet for DFS processing.

```
   input  : Graph G = (V, E), set of vertices V, set of edges E, source
            vertex s, empty FIFO queue Q, empty distance array Dist
   output: Array Dist with shortest-paths from s to all other vertices
 1 foreach u ∈ V \ {s} do
 2 |  Dist[u] = ∞
 3 end
 4 Dist[s] = 0
 5 Q = {s}
 6 while Q is not empty do
 7 |  u = Q.pop()
 8 |  foreach v ∈ V such that (u, v) ∈ E in parallel do
   |  |  // synchronization here
 9 |  |  if Dist[v] == ∞ then
10 |  |  |  Dist[v] = Dist[u] + 1
11 |  |  |  Q.put(v)
12 |  |  end
13 |  end
   |  // barrier synchronization
14 end
```

Algorithm 2: Parallel breadth-first search algorithm (based on description in [4])

The parallel DFS algorithm implemented in this thesis is taken from [7] and belongs to the fixed-packet class of parallel DFS algorithms. In other words, the threads are assigned fixed sets of vertices for processing. The results of each computation are then shared via global data structures (with synchronized access). The implemented algorithm is described by algorithm 5. The main thread starts with computing clustered labeling and with computing load-balanced indices. Then it proceeds to starting individual worker threads and assigning each the computed indices (i.e. vertices to process). After starting the worker threads the main thread waits for all of them to finish. Lastly, it merges the computed labels which produces the final spanning trees. Next subsections describe each of these phases in detail.

2.2.1 Compute clustered labeling

In order to be able to divide work among worker threads, all vertices are labelled in a way that they form groups of neighboring vertices. In particular, the labeling process is described by algorithm 4 (based on [7]). The algorithm starts with initializing a counter and a *labelMap* that will store vertices and their assigned labels. It proceeds with iterating over all vertices and assigning

a label to the unlabelled ones. Moreover, for every newly labelled vertex the algorithm explores its neighbors and labels the ones not labelled yet. As noted by the authors in [7] this way if vertex v_i has a label k then its m neighbors, that have not been labelled yet, will be assigned labels from $k + 1$ to $k + m$. This will allow us to assign vertices to worker threads in a load-balanced manner such that each thread will process vertices close to each other.

2.2.2 Compute load-balanced indices

In the previous step we computed $labelMap$ containing clustered indices for all vertices. Now we need to divide these labelled indices to groups such that each group will be processed by one worker thread. Since there are n vertices in total and T worker threads, we create the groups such that each group has n/T labels. The labels will be distributed in ascending manner, i.e. we sort the labels in ascending order and then first n/T labels will be assigned to the first worker thread, second n/T labels to the second thread etc. Due to the clustered labeling threads will mostly process vertices that are close to each other. According to [7] this will reduce the amount of labels merging at the end of the parallel DFS processing. In the end we create an array $loadbalanced$ of size T where each element $loadbalanced[i]$ stores the work-load for the particular worker thread i.

2.2.3 Worker thread processing

After computing clustered labels and load-balanced indices the algorithm proceeds with starting individual worker threads. Each thread i is assigned vertices to process via $loadbalanced[i]$. The thread then executes procedure $ThreadRun$ described in algorithm 5 on lines 5-13. It goes through its assigned vertices and for each of them creates a new label (line 11) and then starts DFS from that vertex (line 12). The DFS used here is iterative as opposed to the recursive one presented in algorithm 3. As the authors in [7] mention the choice of an iterative algorithm is made with respect to memory efficiency. The algorithm is a standard one except for lines 28-30. There the information about two vertices belonging to the same connected component is stored to a map $labelEquivMap$. Later this map will be used as an input for labels merging in order to produce the final spanning trees of the underlying graph.

2.2.4 Merging labels

After all thread finish processing their assigned vertices we obtain the $labelEquivMap$ storing information about vertices belonging to the same connected compo-

```
   input  : Graph G = (V, E), set of vertices V, set of edges E, array
            visited of visited vertices, array label holding labels of
            vertices
   output: Array label with labelled vertices
 1 Algorithm DFS()
 2 |  foreach v ∈ V do
 3 |  |  visited[v] = false
 4 |  end
 5 |  foreach v ∈ V do
 6 |  |  if !visited[v] then
 7 |  |  |  label[v] = newlabel ;              // generate new label
 8 |  |  |  EXPLORE(v)
 9 |  |  end
10 |  end
11 Procedure EXPLORE(v)
12 |  visited[v] = true
13 |  foreach (u, v) ∈ E do
14 |  |  if !visited[u] then
15 |  |  |  label[u] = label[v]
16 |  |  |  EXPLORE(u)
17 |  |  end
18 |  end
```

Algorithm 3: Depth-first search algorithm (source [7])

nents. Last step is then to merge these labels to obtain unique label sets or in other words spanning trees. The procedure $MergeLabels$ used for merging is described in algorithm 5 on lines 34-40. As the authors in [7] point out, for the merging procedure it is suitable to use the union-find data structure which provides efficient *union* operation. When the merging operations finish we obtain a union-find data structure (denoted uf in the algorithm 5) that holds information about individual spanning trees contained in the graph.

2.3 Dijkstra

Dijkstra algorithm is used to solve the single source shortest path problem. As mentioned in [10] the algorithm works for positive edge weights but does not take into account negative ones. The idea of the algorithm is described in 6. The algorithm maintains three partitions of vertices – set S contains *settled* vertices, *fringe* vertices are in F, and U consists of *unexplored* vertices. At the beginning all vertices are unexplored except for the source vertex, which is fringe. Array *tent* holds tentative distances from a source vertex s to any vertex u. For unexplored vertices this tentative distance is infinity. For the

```
   input  : Graph G = (V, E), set of vertices V, set of edges E
   output: Map labelMap with labelled vertices
 1 Algorithm Labeling()
 2     counter = 0
 3     Map labelMap
 4     foreach v ∈ V do
 5         if v not labelled then
 6             labelMap.put(v, counter++)
 7             foreach neigbour u of v do
 8                 if u not labelled then
 9                     labelMap.put(u, counter++)
10                 end
11             end
12         end
13     end
```

Algorithm 4: Labeling algorithm (based on description in [7])

source vertex $tent(s) = 0$. Throughout the algorithm this tentative distance gets updated as we explore each vertex and its neighbors. In particular, while set F is not empty, we take vertex u which has the minimum tentative distance (line 7). At the same time, we remove this vertex u from F (line 8). Then we iterate through all neighbors of u and for each of them check the condition $tent(v) > tent(u) + Cost[u, v]$. If the conditions holds, we update the tentative distance of v: $tent(v) = tent(u) + Cost[u, v]$. This basically means that we found a path to v through u that is shorter then the previously found path. This is called the relaxation step. Moreover, if the relaxed vertex v is unexplored (line 12), we move it to fringe set F (line 13). Once we are done iterating through all neighbors of u, we move u to the set of settled vertices S. Important observation, as noted in [10], is that for vertices in S their tentative distance is equal to the shortest path.

Authors in [11] propose parallelization of the algorithm by assigning each worker thread a subset of vertices, then finding in F a vertex with minimum tentative distance in parallel (i.e. each thread finds local minimum and then the global minimum is found using reduce operations) and finally relaxing corresponding neighboring vertices also in parallel. That way more than one vertex can be settled at the same time. This approach is further improved in [10] where the authors introduce stronger criteria for settling vertices. Their results show that by using those enhanced criteria lower amount of phases is achieved over the approach taken in [11]. In other words, the algorithm presented in [10] manages to identify more *correct* vertices in one iteration that can be settled and their corresponding neighbors relaxed.

In this thesis the algorithm introduced in [11] will be implemented. The algorithm can be seen in 7. The authors in [11] define the OUT-criterion which states that a vertex v can be settled if $tent(v) \leq L$ where $L = \min\{tent(u) + Cost[u, z] : u \in F \text{ and } (u, z) \in E\}$ with $tent(u)$ being the tentative distance from the source vertex to u and $Cost[u, z]$ being the cost of edge (u, z). The value $\min\{Cost[u, z] : (u, z) \in E\}$ can be computed only once at the beginning, as pointed out by the authors in [11]. This computation can also be done in parallel, where each worker thread computes the minimum cost for its assigned vertices. In every iteration, each thread first computes local minimum L_{local}, which is then used to determine global minimum L. Based on that, the worker threads determine which vertices from their assigned ones can be settled (those are the vertices with $tent(v) \leq L$). For each of these vertices, all of its neighbors are explored and relaxed if needed. Then the iteration repeats. The algorithm ends when either all vertices are settled, or there are unreachable vertices that remain unexplored (i.e. they remain in set U). Array *tent* then stores the shortest paths from the source vertex s to all settled vertices.

2.4 Bellman-Ford

Bellman-Ford algorithm is similar to Dijkstra with one significant benefit – it accounts for negative edge weights and is able to detect negative cycles. This property makes it more universal than the Dijkstra's algorithm.

Consider graph $G = (V, E)$ where V is the set of vertices with size n, and E is the set of edges. Denote $Cost$ a cost matrix containing weights of corresponding edges between any two vertices. If two vertices u and v do not share an edge, the cost is $Cost[u, v] = \infty$. The cost between u and u is zero, i.e. $Cost[u, u] = 0$. Given this setup the sequential Bellman-Ford algorithm, as described in [12], is given by algorithm 8. The procedure on lines 6-8 is called *relaxation*. As the authors in [12] point out, the existence of negative weights is accounted for, since every edge can be relaxed more than once. The time complexity of the underlying algorithm is $\mathcal{O}(n^3)$ as stated in [12].

There are several works proposing ways of parallelizing Bellman-Ford algorithm. Authors in [13] introduce a parallel implementation using NVIDIA's CUDA architecture. The main efficiency enhancement of their algorithm lies in minimizing unnecessary computations by reducing the amount of relax operations. According to [13] parallelism is also more effective due to mapping of threads to edges instead of vertices. Similar approach is taken in [14] where the authors present two kind of optimizations to the standard Bellman-Ford. Firstly, they introduce a mechanism which aims to reduce the number of relaxations by using edge classification. Secondly, the authors propose architecture-

based optimizations, mainly targeting memory efficiency adjustments (such as caching). The architecture used for experiments is Kepler GPU, as opposed to [13] where CUDA architecture was used.

Parallel implementation of Bellman-Ford in this thesis follows the one presented in [12]. The authors use OpenCL for the implementation and also provide analysis of the efficiency of the implementation when using OpenCL's implicit synchronization versus using explicit synchronization mechanisms. The implemented algorithm is described in 9. The inicialization phase (lines 1-3) is the same as for the sequential version. Lines 6-13 then describe work done by individual threads in parallel. In particular, every thread gets an edge to process, let us denote it (u, v). The thread then needs to obtain the indices of vertices u and v and allows it to access correct fields in $Dist$ and $Cost$ matrices. These indices are obtained on lines 6-7. Synchronization is not needed here, because the indices are being only read and not written. Then each thread calculates and updates the respective distances. In the first iteration ($k = 1$), the first row of the $Dist$ matrix contains the distances from the source vertex to its immediate neighbors. Vertices not reachable from the source have distance equal to infinity. First iteration then updates $Dist$'s second row using the initial values in the first row. In the second iteration ($k = 2$) distances are again updates, but now they are stored in the first row using the second row to update them. This process continues until k reaches $n - 1$. The results are then stored in either first or second row of $Dist$ depending on which one was updated the last. The update process of the $Dist$ matrix requires synchronization as parallel writes may happen at the same time. Note that the main loop (i.e. the one iteration k from 1 to $n-1$) is conducted in the main thread, i.e. if for particular iteration k some thread finishes its work and there are no more unprocessed edges in E, the thread needs to wait for the other threads to finish their work before it can proceed to phase $k + 1$.

```
input : Graph G = (V, E), set of vertices V, set of edges E, array
        visited of visited vertices, array label holding labels of
        vertices, number of available threads T
output: Array label with labelled vertices
1  Algorithm ParallelDFS()
       // Compute clustered labeling
       // Compute load-balanced indices
2      for i ← 1 to T do
           // Start a new thread with ThreadRun(loadbalanced[i])
              task
3      end
       // Wait for all threads to finish
4      MergeLabels()
5  Procedure ThreadRun(unvisitedVertices)
       // This method is invoked by each thread
6      start = unvisitedVertices.pop()
7      label[start] = newlabel                    // assigns new label
8      DFS(start)
9      while unvisitedVertices is not empty do
10         v = unvisitedVertices.pop()
11         label[v] = newlabel
12         DFS(v)
13     end
14 Procedure DFS(v)
15     Set visited
16     Stack stack
17     stack.push(v)
18     while stack is not empty do
19         curr = stack.pop()
20         visited.add(curr)
21         foreach neighbor w of curr do
22             if w is not visited then
23                 visited.add(w)
24                 label[w] = label[curr]
25                 stack.push(w)
26             end
27             if w is visited then
28                 L1 = label[w]
29                 L2 = label[curr]
30                 labelEquivMap[L1].add(L2)
31             end
32         end
33     end
34 Procedure MergeLabels()
35     UnionFind uf
36     foreach label in labelEquivMap do
37         foreach otherLabel in labelEquivMap[label] do
38             uf.union(label, otherLabel)
39         end
40     end
```

Algorithm 5: Parallel depth-first search algorithm (source [7])

input : Graph $G = (V, E)$, set of vertices V, set of edges E, number of vertices n, source vertex s, cost matrix $Cost$ containing weights for all pairs of vertices (with $Cost[u, v] = \infty$ meaning v is unreachable from u, and $Cost[u, u] = 0$)
output: Array *tent* with shortest-paths from s to all other vertices

```
1  foreach v ∈ V do
2  |  tent(v) = ∞
3  end
4  S = ∅, F = {s}, U = V \ F
5  tent(s) = 0
6  while F not empty do
7  |  u = vertex in F with min tent(u)
8  |  remove u from F
9  |  foreach neighbor v of u do
10 |  |  if tent(v) > tent(u) + Cost[u, v] then
11 |  |  |  tent(v) = tent(u) + Cost[u, v] // relaxation
12 |  |  |  if v ∈ U then
13 |  |  |  |  move v from U to F
14 |  |  |  end
15 |  |  end
16 |  end
17 |  S.add(u)
18 end
```

Algorithm 6: Dijkstra algorithm (based on [10])

input : Graph $G = (V, E)$, set of vertices V, set of edges E, number of vertices n, source vertex s, cost matrix $Cost$ containing weights for all pairs of vertices (with $Cost[u, v] = \infty$ meaning v is unreachable from u, and $Cost[u, u] = 0$)
output: Array *tent* with shortest-paths from s to all other vertices

```
1  foreach v ∈ V do
2  |  tent(v) = ∞
3  end
4  S = ∅, F = {s}, U = V \ F
5  tent(s) = 0
6  while F not empty do
7  |  L = min{tent(w) + Cost[w, z] : w ∈ F and (w, z) ∈ E}
8  |  foreach u ∈ F with tent(u) ≤ L in parallel do
9  |  |  remove u from F
10 |  |  foreach neighbor v of u do
   |  |  |  // synchronization here
11 |  |  |  if tent(v) > tent(u) + Cost[u, v] then
12 |  |  |  |  tent(v) = tent(u) + Cost[u, v] // relaxation
13 |  |  |  |  if v ∈ U then
14 |  |  |  |  |  move v from U to F
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  end
19 |  S.add(u)
20 end
```

Algorithm 7: Parallel Dijkstra algorithm (based on [10])

input : Graph $G = (V, E)$, set of vertices V, set of edges E, number of vertices n, source vertex s, cost matrix $Cost$ containing weights for all pairs of vertices (with $Cost[u, v] = \infty$ meaning v is unreachable from u, and $Cost[u, u] = 0$)
output: Array $Dist$ with shortest-paths from s to all other vertices

```
1  for i ← 1 to n do
2  |  Dist[i] = Cost[s,i]
3  end
4  for k ← 1 to n − 1 do
5  |  foreach (u,v) in E do
6  |  |  if Dist[v] > Dist[u] + Cost[u,v] then
7  |  |  |  Dist[v] = Dist[u] + Cost[u,v]
8  |  |  end
9  |  end
10 end
```

Algorithm 8: Bellman-Ford algorithm (source [12])

input : Graph $G = (V, E)$, set of vertices V, set of edges E, number of vertices n, source vertex s, cost matrix $Cost$ containing weights for all pairs of vertices (with $Cost[u, v] = \infty$ meaning v is unreachable from u, and $Cost[u, u] = 0$)
output: Array $Dist$ with shortest-paths from s to all other vertices

```
1  for i ← 1 to n do
2  |  Dist[0][i] = Cost[s,i]
3  end
4  for k ← 1 to n − 1 do
5  |  foreach v ∈ V such that (u, v) belongs to E in parallel do
6  |  |  u = get_global_id(0)
7  |  |  v = get_global_id(1)
8  |  |  if k is odd then
   |  |  |  // synchronization here
9  |  |  |  Dist[1][v] = min(Dist[0][v], (Dist[0][u] + Cost[u][v]))
10 |  |  end
11 |  |  if k is even then
   |  |  |  // synchronization here
12 |  |  |  Dist[0][v] = min(Dist[1][v], (Dist[1][u] + Cost[u][v]))
13 |  |  end
14 |  end
15 end
```

Algorithm 9: Parallel Bellman-Ford algorithm (source [12])

Chapter 3

Implementation

This chapter describes the implementation of the following multi-threaded algorithms – BFS, DFS, Dijkstra, and Bellman-Ford. The implementation uses Java Thread API, in particular `ThreadPoolExecutors` for starting the threads, `synchronized` blocks, barriers and locks for synchronization, and thread-safe, non-blocking data structures such as `ConcurrentMap` or `ConcurrentLinkedQueue`. The chapter also describes virtual threads, which represents new concurrency model introduced in Java 19. Virtual threads are used in the implementation as an alternative to standard platform threads.

First section presents implementation requirements. Seconds section then provides implementation details regarding concurrency. Third and fourth sections contain details regarding installation and code documentation, respectively.

3.1 Implementation requirements

The main purpose of this work is to implement selected multi-threaded algorithm to the JGraphT library. The selected algorithms are BFS, DFS, Dijkstra, and Bellman-Ford. Since three of the algorithms (BFS, Dijkstra, Bellman-Ford) already have their sequential counterparts implemented in the library, it is also important to adhere to their existing interface and to provide the same functionality. Moreover, all algorithm implementations should be tested in order to verify their correctness. Last but not least, all new code should be documented.

3.1.1 Functional requirements

Following is the list of functional requirements that this work aims on fulfilling.

F1 Implementation of parallel, multi-threaded BFS algorithm

F2 Implementation of parallel, multi-threaded DFS algorithm

F3 Implementation of parallel, multi-threaded Dijkstra algorithm

F4 Implementation of parallel, multi-threaded Bellman-Ford algorithm

F5 Testing of the implemented algorithms

3.1.2 Non-functional requirements

The following list provides an overview of non-functional requirements that we pose on the implementation.

N1 Documentation of new code

N2 Adhere to the conventions and style of the JgraphT library

3.2 Concurrency implementation details

This section provides on overview of Java concurrency tools used for the implementation of the algorithms.

3.2.1 Thread API

Java allows for starting parallel tasks in kernel threads via its `Thread` class. The class is described in Oracle Java Documentation [15]. Each `Thread` instance corresponds to one kernel thread. This thread is therefore scheduled and managed by the OS and its scheduler. Every thread has assigned task that it executes. Since starting a new kernel thread requires performing system calls, which are in general expensive (specifically in terms of memory usage), we try to avoid doing these calls unnecessarily. The system calls optimization is done mostly by utilizing Java's `ThreadPoolExecutor` [16]. As mentioned in the documentation in [16] using `ThreadPoolExecutor` avoids performance overhead by reducing the number of system calls when starting new threads. In particular, the `ThreadPoolExecutor` defines number of threads that it manages. Once one of the threads finishes its work, it is not terminated but instead immediately assigned another task. Therefore, we avoid a system call that would occur if the thread finished and would have to be started again. Another advantage of using `ThreadPoolExecutor` according to [16] is that it maintains some basic statistics of the execution, e.g. how many tasks completed successfully.

Code 1 shows the usage of thread pool executors in the implementations. In the code example, `ParallelTask` is a class implementing the `Runnable` interface. Instance of this class is then submitted for execution by individual worker threads. Calling `executor.submit(...)` where `executor` is an instance of `ThreadPoolExecutor` returns a `Future`. `Future` represents a

```
1  Collection<Future<?>> tasks = new LinkedList<>();
2
3  for (int j = 0; j < numberOfThreads; j++) {
4      tasks.add(executor.submit(new ParallelTask()));
5  }
6
7  for (Future<?> task : tasks) {
8      try {
9          task.get(EXECUTOR_MAX_TIMEOUT, TimeUnit.SECONDS);
10     } catch (ExecutionException | InterruptedException | TimeoutException
       ↪  e) {
11         // process exception
12     }
13 }
```

Code 1: Example usage of thread pool executors

task that will finish in the future. On line 9 the main thread waits for individual futures to finish, with a timeout threshold specified by a constant `EXECUTOR_MAX_TIMEOUT`.

3.2.2 Synchronization

Java offers several synchronization mechanisms. One option is to use locks, such as `ReentrantLock` [17]. Locks, in general, work as follows. The first thread acquiring the lock can proceed and execute the code between lock acquisition and its release. Other threads then have to wait until the lock is released and can be acquired again. As mentioned in [17], the lock release should by done in a try-finally block in order to avoid deadlock. Deadlock occurs when a thread is waiting for a lock that is acquired by another thread but that is never going to be released. This can happen when, for example, a thread encounters some exception when holding a lock. If the corresponding code is not in try-finally block, then the lock will never be released. In general, locks offer a considerable amount of flexibility when writing concurrent code. On the other hand, locks require caution when releasing (i.e. try-finally block has to be used, as explained above). Moreover, as mentioned in [18] locks perform a bit worse than another synchronization mechanism – `synchronized` blocks.

`synchronized` is one of Java's keywords and provides an alternative to locks. According to [19], whole methods can be marker as `synchronized` – then those methods are thread-safe, i.e. only one thread at the time can execute the method. One immediate advantage over locks is that we do not have to take care of any lock releases. When a method is marked as `synchronized`, all synchronization is taken care of and no further synchronization mechanisms are needed. Apart from methods, `synchronized` can also be used for synchro-

```
ReentrantLock lock = new ReentrantLock();

lock.lock();
try {
    // code that needs to be synchronized
} finally {
    lock.unlock();
}
```

Code 2: Example usage of locks

```
synchronized (object) {
    // code that needs to be synchronized
}
```

Code 3: Example usage of synchronized blocks

nizing blocks of code, as described in [19]. In this case, `synchronized` needs an object for which the lock will be acquired. Handling of the lock logic will then be taken care of by the `synchronized` mechanism.

Using `synchronized` is in general preferred over locks due to the advantages described above. However, when using virtual threads, it is necessary to use explicit locks as `synchronized` is not implemented to work with virtual threads. This mechanism will be described in detail in section 3.2.4.

Both `synchronized` blocks and locks are used in the implementations. Locks are mainly used with virtual threads, as these do not perform well with `synchronized` blocks. Code 2 displays example usage of locks as they are used in the implementations. Similarly, code 3 show example usage of `synchronized` block. By calling `synchronized` the thread acquires intrinsic lock of `object`. The lock is then automatically released when the execution reaches the end of the `synchronized` block.

3.2.3 Thread-safe data structures

Previous section describes how to achieve thread-safety through the usage of synchronization. In addition, Java also offers thread-safe data structures that handle concurrent reads and writes and therefore prevent race conditions from happening. There are two kinds of such data structures – blocking and non-blocking.

Blocking data structures blocks the thread if the structure is empty. Such structures are used mostly in consumer-producer use-cases. In the consumer-producer scenario a subset of threads act as consumers, i.e. they take elements

from some common storage and process them. Another subset of threads are producers, i.e. they prepare elements to be processed by consumer threads and fill those elements to the common storage. Such storage is usually a shared queue or dequeue, priority queue or stack. `BlockingQueue` is a Java's interface that provide a blocking queue functionality [20]. In particular, when a thread tries to retrieve and elements from the queue and there is none, the thread gets blocked until new elements arrive to the queue. `BlockingQueue` extends the `Queue` interface [21] and therefore can be used as a normal queue. Its implementations involve `LinkedBlockingQueue` [22] and `PriorityBlockingQueue` [23].

Non-blocking data structures do not block the threads. They provide a concurrent, lock-free access to the stored data based on CAS atomic operations. One of the interfaces providing this behavior is, for example, `ConcurrentMap` [24]. Its implementation `ConcurrentHashMap` [25] provides concurrency for both reads and updates. It is also fully exchange with `Hashtable` as it implements the same interface. However, unlike `Hashtable` it does not use locks for synchronization. Therefore, as pointed in [25] it cannot replace `Hashtable` in applications which rely on its synchronization mechanics. Similarly to `ConcurrentMap`, `ConcurrentLinkedQueue` [26] is another non-blocking data structure providing thread-safe access operations. Putting into or removing elements from the queue are atomic methods and therefore do not block the threads. These methods also do not require any explicit synchronization. However, as mentioned in [26] bulk operations such as `addAll` or `forEach` are not guaranteed to be performed atomically. Other non-blocking, thread-safe data structure is `ConcurrentSkipListSet` [27]. It implements the `Set` interface and provides means for storing unique data. Similarly to the other before-mentioned structures, operations for adding and removal elements are atomic. The only exception are again bulk operations, where atomicity is not guaranteed.

The implementations of the selected algorithms make use of the above-mentioned concurrent, non-blocking data structures. In particular, BFS and DFS build their efficiency mostly on using atomic operations. This allows these implementations to avoid explicit synchronization mechanisms and therefore gain performance benefits. While Dijkstra and Bellman-Ford implementations make use of the non-blocking data structures as well, they still need to utilize explicit synchronization mechanisms.

3.2.4 Virtual threads

Standard platform threads have a significant drawback – creating them and blocking (i.e. switching context) are expensive (in terms of memory and execution time) operations requiring making system calls from the Java runtime

environment. Costly thread creation is partly solved using thread pools – when one thread finishes its work, it is immediately assigned new task without termination. Therefore, there is no need to make a system call to create the thread again. However, the issue with costly blocking remained. Reactive programming targets this drawback by providing a way of writing concurrent programs without the need of thread blocking. In particular, whenever a thread is about to be blocked, it is instead provided with a callback function and assigned a new task in the meantime. The callback function is then executed once the blocking operation finishes. Although reactive programming leads to the reduced occurrence of thread blocking, the author in [28] maintains that it is not easy to learn as it requires programmers to write their concurrent programs in a completely different way than with standard multi-threading model. This can result in higher probability of the occurrence of coding errors and also can hinder the overall readability of the final program.

Java 19 has introduced new concurrency model as part of project Loom. The project aims on addressing the above-mentioned concurrency problems by implementing virtual threads (also called fibers or lightweight threads). Virtual threads unlike platform threads live only in the JVM runtime and do not have any direct connection to the OS, as is mentioned in [29]. Even the scheduler is implemented in JVM and therefore JVM is completely in charge of virtual threads management. Even though virtual threads are not connected to the OS directly, they still need the platform threads in order to execute their code. Therefore, as discussed in [29] each virtual thread needs a *carrier* thread, which is a standard platform thread, for its execution. The advantage of this approach is that whenever a virtual thread blocks, the platform thread does not need to block too. It can instead execute another virtual thread. This way the costly platform thread blocking is avoided. On the other hand, virtual thread blocking is not an issue, since the thread and its context lives only in the JVM and therefore does not require making any expensive system calls.

Since virtual threads implements the `Thread` interface (the same as platform threads), they can replace platform threads in the code quite seamlessly. However, there are some differences between virtual and platform threads that need to be reflected in the code appropriately. Firstly, virtual threads are not meant for pooling. Since pooling is used in order to avoid costly and unnecessary thread creations, it is not needed when virtual threads are in place as these are not created via system calls. Secondly, regarding synchronization mechanisms virtual threads do not work well with `synchronized` blocks, as mentioned in [30]. It is preferable to use locking instead (such as the `ReentrantLock`), as this is optimized for use with virtual threads. `synchronized` blocks, on the other hand, would perform worse in terms of efficiency (but they would still provide the necessary synchronization).

```
1  Collection<Thread> tasks = new LinkedList<>();
2
3  for (int j = 0; j < numberOfThreads; j++) {
4      tasks.add(Thread.ofVirtual().start(new ParallelTask(dist,
       ↪  currVertices, nextVertices, barrier, shouldFinish, lock)));
5  }
6
7  for (Thread task : tasks) {
8      try {
9          task.join(Duration.ofSeconds(EXECUTOR_MAX_TIMEOUT));
10     } catch (InterruptedException ex) {
11         // process exception
12     }
13 }
```

Code 4: Example usage of virtual threads

Considering how virtual threads function, they are best suited for concurrent applications with many blocking operations. On the other hand, there are not appropriate for use-cases when CPU-heavy, non-blocking computations need to be done. In this case, using virtual threads would be just an overhead which would not bring any additional benefits, since blocking would not be an issue. Therefore, the usage of virtual threads should considered with respect to the nature of concurrent operations. Moreover, the above-mentioned limitations of virtual threads (such as their incompatibility with `synchronized` blocks) should be taken into account as well when assessing their suitability for a particular use-case.

As part of the implementation, Dijkstra and Bellman-Ford algorithms have been selected to be implemented with virtual threads as well as platform ones. These two algorithms are considered best suitable for using virtual threads as they contain explicit synchronizations. On the other hand, BFS and DFS implementations do not require any explicit synchronization mechanisms and therefore they would probably not benefit from utilizing virtual threads. Parallel implementations with virtual threads of Dijkstra and Bellman-Ford are in package `org.jgrapht.alg.shortestpath` in classes `ParallelVirtualDijkstraShortestPath` and `ParallelVirtualBellmanFordShortestPath`, respectively. Code 4 shows how the implementations incorporate virtual threads. Each thread is started with `Thread.ofVirtual().start(...)`. The thread is then added to the collection of threads (line 4) so that the `join` method can then be called on them in order for the main thread to wait until the worker threads finish.

3.3 Installation

The library uses Maven for build. Hence, the build processes is defined by corresponding `pom.xml` files. The whole library can be built using command `mvn clean install`. Java 19 is needed for the build.

3.4 Code documentation

Newly added code contains documentation comments. Maven build then uses `javadoc` to generate HTML documentation from the code comments. The documentation is generated during build (after running `mvn clean install`) and it is created in a jar file `jgrapht-core-1.5.2-SNAPSHOT-javadoc.jar`. In the root directory there is a file `index-all.html` which contains the generated documentation.

Chapter 4

Testing and performance analysis

The chapter describes conducted unit and performance tests and provides assessment of their results. The first section provides an overview of conducted unit tests. Following section discusses the conducted performance tests and their results. Last part of the chapter evaluates the fulfillment of the implementation requirements.

4.1 Unit tests

All implemented algorithms have corresponding unit tests. The tests for BFS, Dijkstra and Bellman-Ford are in `org.jgrapht.alg.shortestpath` package in test classes `ParallelBFSShortestPathTest`, `ParallelDijkstraShortestPathTest`, and `ParallelBellmanFordShortestPathTest`, respectively. The tests are of two types. First, all parallel algorithms are required to pass the same unit tests as their sequential counterparts. Second, random tests are conducted that generate random data (i.e. random graphs) and compare results from sequential and parallel version of an algorithm. Tests for DFS are in package `org.jgrapht.alg.spanning` in class `ParallelDFSSpanningTreeTest`. Since there is no direct sequential counterpart to the parallel DFS, the tests class provide only standard unit tests.

4.2 Performance analysis

Results of the performance analysis can be seen in figures 4.1, 4.2, 4.3 and 4.4. The performance analysis is conducted using the script in

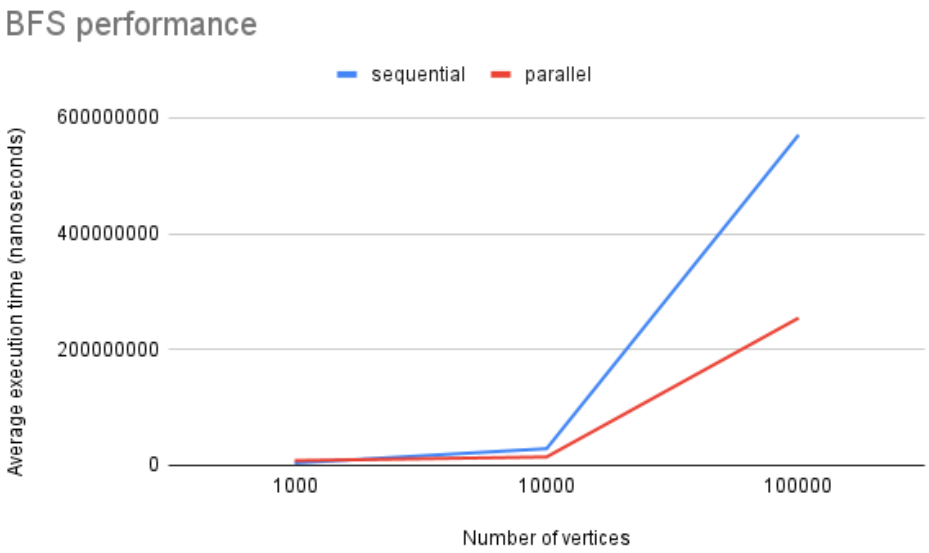

Figure 4.1: BFS performance

`org.jgrapht.demo.ParallelPerformanceDemo`. The class contains a main class, i.e. is executable, and by running it generates a CSV file with test times. The tests are conducted several times (number of iteration is specified in field `numOfIterations`. Moreover, graphs for tests are generated randomly, with increasing number of vertices. Hence, it can be compared how algorithms performance differ with respect to graph size. Next sections further describe the obtained results. The tests were performed on Intel Core i5-8350U CPU, 1.70GHz x 4 and 16GB RAM. For the testing the configuration of 5 platform threads and 20 virtual threads has been used. Following subsection describe performance results for particular algorithms.

4.2.1 Breadth-first search

Best performance results are achieved by parallel BFS, as compared to its sequential counterpart. The output of the performance analysis can be seen in figure 4.1. The reason for parallel version outperforming the sequential one is that BFS is suitable for parallelization, since the BFS processing for each vertex can be conducted in parallel. The thread code can be seen in listing 6. As can be seen, there are no synchronized blocks or usage of locks. `predecessorMap` is an instance of `ConcurrentHashMap` and therefore can be modified atomically. Updating of `predecessorMap` is the only place with

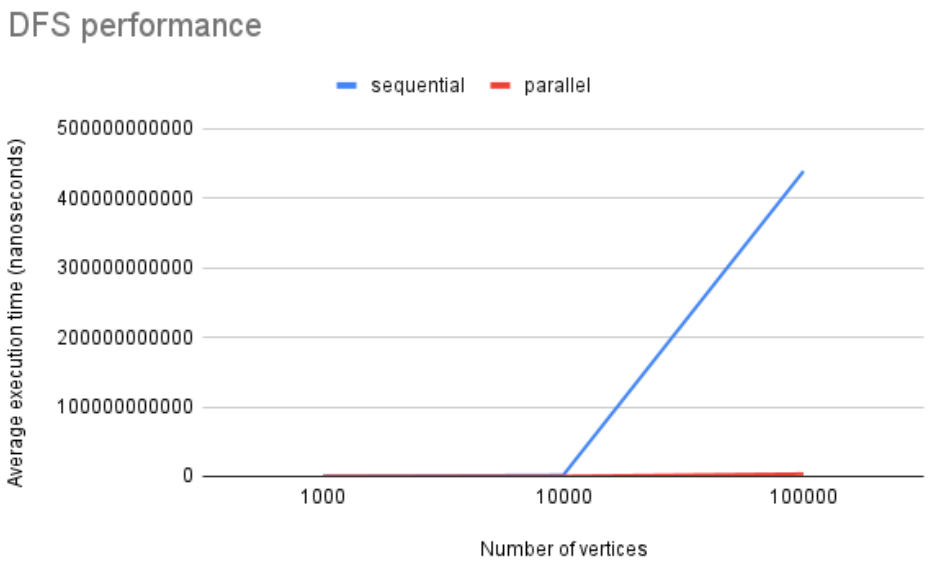

Figure 4.2: DFS performance

concurrent operations, the rest of the code is thread-safe by itself. Hence, by using a concurrent data structure the whole thread code is thread-safe without explicit usage of synchronization mechanisms.

4.2.2 Depth-first search

Similarly to BFS, DFS also performs better in parallelized version. Results can be seen in figure 4.2. The implemented algorithm allows threads to process separate parts of the graph. The individual results are then combined in the main thread to form the final output. During thread processing, which can be seen in listing 6, synchronization is needed when accessing shared data structures, in particular `visited`, which holds information about already visited vertices by all threads. However, explicit synchronization is not needed as concurrent data structures with atomic operations can be used. For example, `visited` can be implemented as `ConcurrentHashMap`. When the `visited` structure has to be updated on line 33, it is possible to use the atomic `computeIfAbsent` method instead of using `synchronized` blocks or locks.

4.2.3 Dijkstra shortest path

Unlike BFS and DFS, based on the performance analysis Dijkstra's parallel version seems to perform worse than the sequential one, as can be seen in

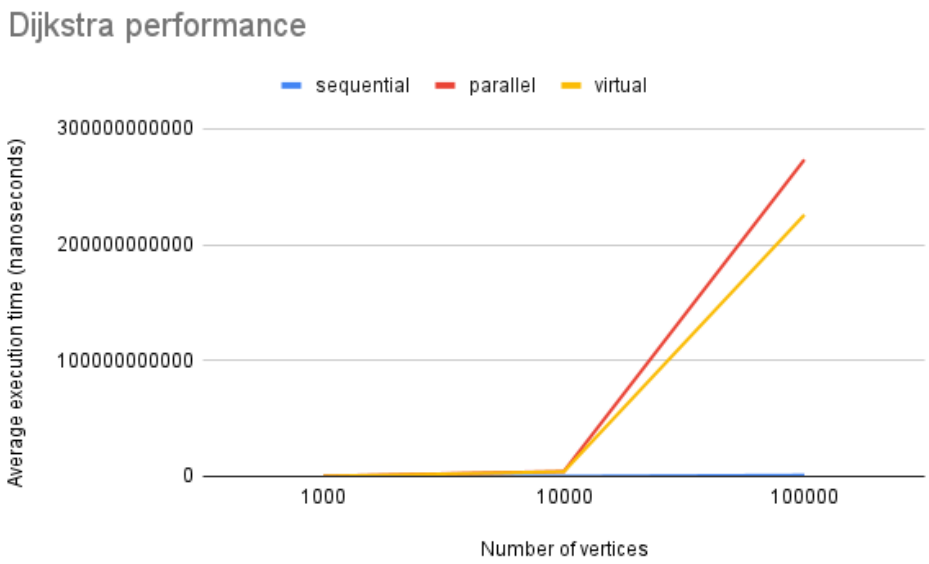

Figure 4.3: Dijkstra performance

figure 4.3. The reason is that the algorithm is not suitable for parallelization. Both BFS and DFS could make use of concurrent data structures and their atomic operations. However, this is not possible with Dijkstra's algorithm. On the contrary, as can be seen in listing 7, explicit synchronization needs to be used in order to maintain thread-safety. This results in smaller space for parallel execution and therefore in a slowdown of the whole process.

The synchronization used in of two types – `ReentrantLock` is used in order to synchronize reads/writes to shared data structures, while `CyclicBarrier` is utilized for synchronizing the whole thread processing. When all threads reach the barrier, another task will be conducted that will prepare the data for the worker threads. In particular, the barrier task fills the `neighbors` container that holds data for thread processing.

Another considerable aspect of the parallel Dijkstra performance is the nature of the implemented algorithm itself. The implemented algorithm aims on determining the OUT-criterion, which then allows to detect more then one vertex per iteration for processing. However, this approach has extra performance cost in the form of the OUT-criterion computation. This cost will pay off only when dealing with large graphs. In this case, the benefit of processing multiple vertices per iteration will outweigh the cost of the criterion computation.

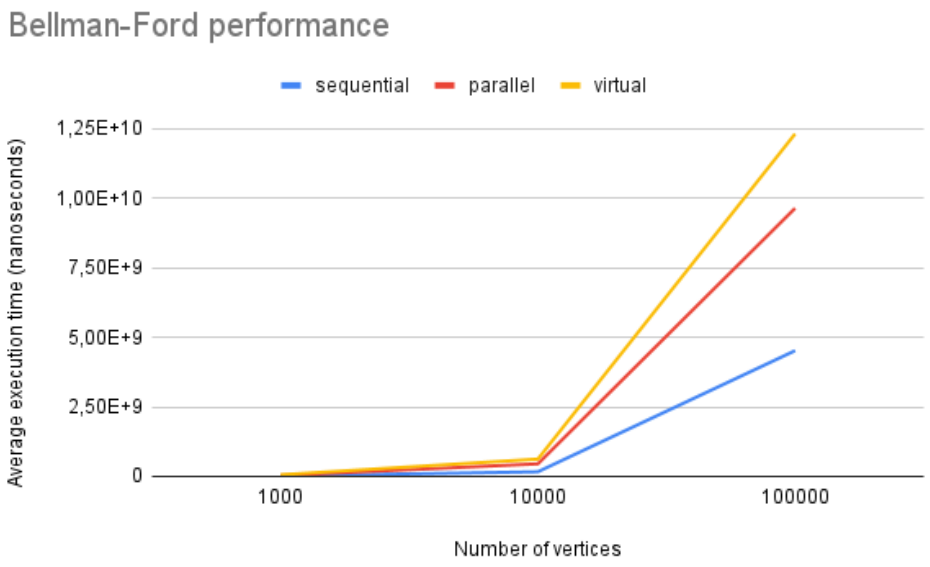

Figure 4.4: Bellman-Ford performance

Performance improvement is achieved when virtual thread are incorporated, as can be seen in figure 4.3. Since the parallel implementation contains several blocking occurrences, virtual threads can provide sufficient speed-up.

4.2.4 Bellman-Ford

Similarly to Dijkstra, Bellman-Ford performs in the performance tests worse in parallel version than in sequential one, as is shown in figure 4.4. This result is expected, since Bellman-Ford is not very suitable for parallelization. As can be noted in the thread code in code 8, distance relaxation can be dome in parallel. However, this operation cannot be done atomically. Therefore, explicit synchronization is needed. As a result, there is not much space for parallel execution, as most operations in code 8 need to be synchronized. Moreover, the algorithm requires a barrier synchronization (line 26 in code 8), in order to update the queue of vertices to process and to check whether further processing is even needed. Parallel version of Bellman-Ford might be more efficient than the sequential version for very large graphs where the synchronization cost becomes insignificant compared to the benefits of parallelization.

Unlike Dijkstra algorithm, incorporating virtual threads leads to worse algorithm performance. The reason is that blocking occurrence is not very frequent

```java
@Override
public void run() {
    while (true) {
        V vertex;

        while ((vertex = frontierQueue.poll()) != null) {

            for (E edge : graph.outgoingEdgesOf(vertex)) {
                V neighbor = Graphs.getOppositeVertex(graph, edge,
                ↪  vertex);

                predecessorMap.computeIfAbsent(neighbor, (key) -> {
                    nextFrontierQueue.add(neighbor);
                    return new Pair<>(bfsLevel.doubleValue(), edge);
                });
            }
        }

        try {
            cyclicBarrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            throw new RuntimeException(e);
        }

        if (shouldFinish.get()) {
            return;
        }
    }
}
```

Code 5: Parallel BFS thread code

and therefore the benefits of virtual threads cannot manifest. In particular, the cost of running virtual threads on top of platform ones outweighs the benefits of more effective context switching during blocking.

4.3 Evaluation of implementation requirements

Section 3.1 introduces implementation requirements, both functional and non-functional, that we pose on the implemented algorithms. Based on the previous findings, the requirements have been fulfilled as follows.

F1 Parallel, multi-threaded BFS algorithm has been implemented

F2 Parallel, multi-threaded DFS algorithm has been implemented

F3 Parallel, multi-threaded Dijkstra algorithm has been implemented

F4 Parallel, multi-threaded Bellman-Ford algorithm has been implemented

F5 Unit and performance tests have been conducted

N1 New code has been documented using javadoc

N2 New code follows the conventions of the JGraphT library

```java
@Override
public void run() {
    V startVertex = unvisitedVertices.poll();
    clusteredLabels.put(startVertex, labelCounter.incrementAndGet());
    Map<Long, List<Long>> localLabelEquivMap = new HashMap<>();
    // assure the vertex hasn't been already processed by some other thread
    ...

    V nextVertex;
    while ((nextVertex = unvisitedVertices.poll()) != null) {
        // assure the vertex hasn't been already processed by some other thread
        ...

        clusteredLabels.put(nextVertex, labelCounter.incrementAndGet());
        Map<Long, List<Long>> lle = DFS(nextVertex);

        for (Map.Entry<Long, List<Long>> entry : lle.entrySet()) {
            localLabelEquivMap.computeIfAbsent(entry.getKey(), k -> new ArrayList<>()).addAll(entry.getValue());
        }
    }

    labelEquivMap.put(threadIndex, localLabelEquivMap);
}

private Map<Long, List<Long>> DFS(V start) {
    Stack<V> stack = new Stack<>();
    Map<Long, List<Long>> llabelEquivMap = new HashMap<>();
    stack.push(start);

    while (!stack.empty()) {
        V curr = stack.pop();
        visited.put(curr, true);

        List<E> outgoingEdges = new ArrayList<>(graph.outgoingEdgesOf(curr));
        for (E neighborEdge : outgoingEdges) {
            V neighbor = Graphs.getOppositeVertex(graph, neighborEdge, curr);
            AtomicBoolean addedNewVisited = new AtomicBoolean(false);
            visited.computeIfAbsent(neighbor, key -> {
                addedNewVisited.set(true);
                return true;
            });

            if (addedNewVisited.get()) {
                finalEdges.add(neighborEdge);
                clusteredLabels.put(neighbor, clusteredLabels.get(curr));
                stack.push(neighbor);
            } else {
                Long label1 = clusteredLabels.get(neighbor);
                Long label2 = clusteredLabels.get(curr);
                llabelEquivMap.computeIfAbsent(label1, l -> new ArrayList<>()).add(label2);
            }
        }
    }

    return llabelEquivMap;
}
```

Code 6: Parallel DFS thread code

```java
@Override
public void run() {
    while (true) {
        if (shouldFinish.get()) {
            return;
        }

        Pair<E, AddressableHeap.Handle<Double, Pair<V, E>>> neighbor =
        ↪  neighbors.poll();
        if (neighbor == null) {
            try {
                barrier.await();
            } catch (BrokenBarrierException | InterruptedException e) {
                throw new RuntimeException(e);
            }
            continue;
        }

        E edge = neighbor.getFirst();
        AddressableHeap.Handle<Double, Pair<V, E>> minFringe =
        ↪  neighbor.getSecond();
        V minVertex = minFringe.getValue().getFirst();
        V neighborVertex = Graphs.getOppositeVertex(graph, edge,
        ↪  minVertex);
        double weight = graph.getEdgeWeight(edge);
        if (comparator.compare(weight, 0.0) < 0) {
            throw new IllegalArgumentException("Negative edge weight not
            ↪  allowed");
        }

        Double distRelaxed = minFringe.getKey() + weight;
        if (comparator.compare(distRelaxed, radius) >= 0) {
            continue;
        }

        synchronized(seen) { {
            AddressableHeap.Handle<Double, Pair<V, E>> neighborNode =
            ↪  seen.get(neighborVertex);

            if (neighborNode == null) {
                neighborNode = fringe.insert(distRelaxed,
                ↪  Pair.of(neighborVertex, edge));
                seen.put(neighborVertex, neighborNode);
                fringeNodes.add(neighborNode);
            } else {
                Double distNeighbor = neighborNode.getKey();

                if (comparator.compare(distRelaxed, distNeighbor) < 0) {
                    neighborNode.decreaseKey(distRelaxed);
                    neighborNode.setValue(Pair.of(neighborVertex, edge));
                    fringeNodes.add(neighborNode);
                }
            }
        }
    }
}
```


Code 7: Parallel Dijkstra thread code

```java
@Override
public void run() {
    while (true) {
        V vertex = currVertices.poll();

        while (vertex != null) {

            for (E edge : graph.outgoingEdgesOf(vertex)) {
                Double edgeWeight = graph.getEdgeWeight(edge);
                V edgeTarget = Graphs.getOppositeVertex(graph, edge,
                ↪  vertex);

                synchronized (dist) {
                    Double currDistance =
                    ↪  dist.get(edgeTarget).getFirst();
                    Double newDistance = dist.get(vertex).getFirst() +
                    ↪  edgeWeight;
                    if (comparator.compare(newDistance, currDistance) <
                    ↪  0) {
                        dist.put(edgeTarget, Pair.of(newDistance, edge));
                        nextVertices.add(edgeTarget);
                    }
                }
            }

            vertex = currVertices.poll();
        }

        try {
            barrier.await();
        } catch (BrokenBarrierException | InterruptedException e) {
            throw new RuntimeException(e);
        }

        if (shouldFinish.get()) {
            if (maxHops >= graph.vertexSet().size()) {
                detectNegativeCycle();
            }
            return;
        }
    }
}
```

Code 8: Parallel Bellman-Ford thread code

Conclusion

The main purpose of this thesis is to implement parallel, multi-threaded versions of BFS, DFS, Dijkstra, and Bellman-Ford algorithms to the JGraphT library. Since BFS, Dijkstra and Bellman-Ford are already implemented in JGraphT in sequential manner, the new implementations aim on providing the same interface and functionalities as the existing ones. Furthermore, performance tests are conducted where the performance of the multi-threaded algorithms is compared to the performance of their sequential counterparts. Last but not least, the thesis provides an overview of the two existing concepts of multi-threading in Java – platform threads and virtual threads. Their suitability for the implementation of the underlying algorithms is examined and verified experimentally as part of the performance tests.

As the results of performance analysis show, BFS and DFS parallel implementations achieve better performance than their sequential versions. The reason behind these results is that both BFS and DFS are suitable for parallelization, i.e. most parts of the algorithms can be conducted in parallel and there is not much need for synchronization. Moreover, these two implementations make use of Java thread-safe non-blocking data structures that provide atomic read-/write operations and therefore provide further performance improvement. On the other hand, parallel implementations of Dijkstra and Bellman-Ford do not show any performance benefit over the sequential implementations. The reason is two-fold. Firstly, both algorithms are difficult to parallelize as there is not much space for parallel execution. In addition, explicit synchronization is required limiting concurrent access to significant amount of code and therefore causing further slowdown. Secondly, the performance tests were conducted on graphs of maximum size of 10,000 vertices with a degree of 20. In particular, the benefits of parallel Dijkstra and Bellman-Ford implementations might manifest on larger graphs and using better hardware allowing for the usage of more threads. However, when incorporation virtual threads, Dijkstra algorithm achieve performance improvement over standard platform

threads. The reason is that the implementations involves significant amount of blocking which cost virtual threads aim on mitigating. On the other hand, Bellman-Ford does not achieve any speed-up with virtual threads, as there are not so many blocking parts that would make use of the benefits of virtual threads.

To sum up, four new parallel, multi-threaded algorithms have been implemented to the JGraphT library. All implementations make use of the existing interfaces and data structures in the library and follow the JGraphT's code conventions. In addition, all new code is documented and tested using unit tests of the sequential algorithms, comparison tests comparing results of sequential algorithms to their parallel counterparts, and performance tests verifying efficiency of the implementations. Last but not least, Dijkstra and Bellman-Ford algorithms are implemented using both platform threads and new virtual threads.

Further work should aim at implementations of other graph multi-threaded algorithms to the JGraphT library. Moreover, additional examination of the suitability of Java's virtual threads should be conducted. Virtual threads should then be used wherever possible in order to leverage the additional efficiency that they provide over standard platform threads. Another direction would be to analyze the possibility of algorithm implementations supporting GPU architectures (such as CUDA).

Bibliography

1. SEGAN, Sascha. *1982 vs. 2022: Has Technology Really Become More Affordable?* [online]. pcmag.com, 2022-06-15 [visited on 2023-05-11]. Available from: `https://www.pcmag.com/news/1982-vs-2022-has-technology-really-become-more-affordable`.
2. *Overview for Application Developers* [online]. [N.d.]. [visited on 2023-04-19]. Available from: `https://jgrapht.org/guide/UserOverview`.
3. BERRENDORF, Rudolf; MAKULLA, Mathias. Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems. In: *Nygard, Tamir (Eds.): Future Computing 2014, The Sixth International Conference on Future Computational Technologies and Applications. Venice, Italy, May 25-29, 2014* [online]. ThinkMind, 2014, pp. 26–31 [visited on 2023-04-23]. ISBN 978-1-61208-339-1. Available from: `https://www.thinkmind.org/index.php?view=article&articleid=future_computing_2014_2_20_30037`.
4. LEISERSON, Charles E.; SCHARDL, Tao B. A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* [online]. Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 303–314 [visited on 2023-04-23]. SPAA '10. ISBN 9781450300797. Available from DOI: `10.1145/1810479.1810534`.
5. MERRILL, Duane; GARLAND, Michael; GRIMSHAW, Andrew. Scalable GPU Graph Traversal. *SIGPLAN Not.* [Online]. 2012, vol. 47, no. 8, pp. 117–128 [visited on 2023-04-23]. ISSN 0362-1340. Available from DOI: `10.1145/2370036.2145832`.
6. MUNGUIA, Lluis-Miquel; BADER, David A; AYGUADE, Eduard. Task-based parallel breadth-first search in heterogeneous environments. In: *19th International Conference on High Performance Computing* [online].

IEEE, 2012, pp. 1–10 [visited on 2023-04-23]. Available from DOI: 10.1109/HiPC.2012.6507474.

7. RAYASAM, Harish; NASRE, Rupesh. Parallelization of Depth-First Traversal using Efficient Load Balancing. In: *22nd International Conference on High Performance Computing (Student Research Symposium)* [online]. 2015 [visited on 2023-04-23]. Available from: https://hipc.org/hipc2015/documents/HiPC-SRS-Paper/1570220263.pdf.

8. REINEFELD, A.; SCHNECKE, Volker. Work-load balancing in highly parallel depth-first search. In: *Proceedings of the Scalable High-Performance Computing Conference* [online]. 1994, pp. 773–780 [visited on 2023-04-23]. ISBN 0-8186-5680-8. Available from DOI: 10.1109/SHPCC.1994.296719.

9. KUMAR, Vipin; RAO, V Nageshwara. Scalable parallel formulations of depth-first search. *Parallel algorithms for machine intelligence and vision* [online]. 1990, pp. 1–41 [visited on 2023-04-23]. ISBN 978-1-4612-3390-9. Available from DOI: 10.1007/978-1-4612-3390-9_1.

10. KAINER, Michael; TRÄFF, Jesper Larsson. More parallelism in Dijkstra's single-source shortest path algorithm. *arXiv preprint* [online]. 2019 [visited on 2023-04-30]. Available from DOI: 10.48550/arXiv.1903.12085.

11. CRAUSER, Andreas; MEHLHORN, Kurt; MEYER, Ulrich; SANDERS, Peter. A parallelization of Dijkstra's shortest path algorithm. In: *Mathematical Foundations of Computer Science 1998: 23rd International Symposium, MFCS'98 Brno, Czech Republic, August 24–28, 1998 Proceedings 23* [online]. Springer, 1998, pp. 722–731 [visited on 2023-04-30]. Available from DOI: 10.1007/BFb0055823.

12. HAJELA, Gaurav; PANDEY, Manish. Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford. *International Journal of Computer Applications* [online]. 2014, vol. 95, no. 15 [visited on 2023-04-14]. ISSN 0975-8887. Available from DOI: 10.5120/16667-6659.

13. SURVE, Ganesh G; SHAH, Medha A. Parallel implementation of Bellman-ford algorithm using CUDA architecture. In: *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)* [online]. 2017, vol. 2, pp. 16–22 [visited on 2023-04-19]. Available from DOI: 10.1109/ICECA.2017.8212794.

14. BUSATO, Federico; BOMBIERI, Nicola. An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* [online]. 2016, vol. 27, no. 8, pp. 2222–2233 [visited on 2023-04-19]. Available from DOI: 10.1109/TPDS.2015.2485994.

15. *Oracle Java Documentation - Thread* [online]. oracle.com, 2023 [visited on 2023-04-30]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html`.

16. *Oracle Java Documentation - ThreadPoolExecutor* [online]. oracle.com, 2023 [visited on 2023-04-30]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html`.

17. *Oracle Java Documentation - ReentrantLock* [online]. oracle.com, 2023 [visited on 2023-05-01]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html`.

18. LIVERAMP. *Java Performance: synchronized() vs Lock* [online]. medium.com, 2011-06-17 [visited on 2023-05-01]. Available from: `https://medium.com/liveramp-engineering/java-performance-synchronized-vs-lock-301130e62f47`.

19. HORSTMANN, Cay. Synchronization in Java, Part 2: The synchronized keyword. *Java Magazine* [online]. 2022 [visited on 2023-05-01]. Available from: `https://blogs.oracle.com/javamagazine/post/java-thread-synchronization-synchronized-blocks-adhoc-locks`.

20. *Oracle Java Documentation - BlockingQueue* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/BlockingQueue.html`.

21. *Oracle Java Documentation - Queue* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html`.

22. *Oracle Java Documentation - LinkedBlockingQueue* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/LinkedBlockingQueue.html`.

23. *Oracle Java Documentation - PriorityBlockingQueue* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/PriorityBlockingQueue.html`.

24. *Oracle Java Documentation - ConcurrentMap* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ConcurrentMap.html`.

25. *Oracle Java Documentation - ConcurrentHashMap* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html.

26. *Oracle Java Documentation - ConcurrentLinkedQueue* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ConcurrentLinkedQueue.html.

27. *Oracle Java Documentation - ConcurrentSkipListSet* [online]. oracle.com, 2023 [visited on 2023-05-07]. Available from: https://docs.oracle.com/en/java/javase/17/docs/api//java.base/java/util/concurrent/ConcurrentSkipListSet.html.

28. SHIRIZADA, Rashad. *Reactive Programming — What is it and why should you care?* [online]. medium.com, 2022-06-09 [visited on 2023-05-11]. Available from: https://medium.com/@rashadsh/reactive-programming-what-is-it-and-why-should-you-care-12ad44928d0.

29. LAKSHMANAN, Ram. *Java Virtual Threads — Easy introduction* [online]. medium.com, 2023-02-24 [visited on 2023-05-11]. Available from: https://medium.com/@RamLakshmanan/java-virtual-threads-easy-introduction-44d96b8270f8.

30. JENKOV, Jakob. *Java Virtual Threads* [online]. jenkov.com, 2023-02-04 [visited on 2023-05-11]. Available from: https://jenkov.com/tutorials/java-concurrency/java-virtual-threads.html.

Appendix A

Acronyms

BFS Breadth-first search

CAS Compare-and-swap

CPU Central processing unit

CSV Comma-separated values

DFS Depth-first search

FIFO First-in, first-out

GEXF Graph exchange XML format

GPU Graphics processing unit

HTML Hypertext markup language

JSON JavaScript object notation

JVM Java virtual machine

OS Operating system

XML Extensible markup language

Appendix B

Contents of electronic attachment

- `readme.md` ..... description of the contents of the electronic attachment
- `jgrapht` ... the directory with the source codes of the extended JGraphT library
- `data` ..... the directory with results of performance analysis
- `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `kolombar-bachelor-thesis.pdf` ..... the thesis text in PDF format