ALGORITHMS AND DATA STRUCTURE LIBRARIES FOR JAVA

Patrik PERHÁČ, Slavomír ŠIMOŇÁK
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, E-mail: patrik.perhac@student.tuke.sk, slavomir.simonak@tuke.sk

ABSTRACT

This paper is dedicated to the comparison of different libraries of algorithms and data structures for the Java language. Within the paper currently available and relevant libraries are analyzed and compared. Selected libraries are compared in terms of provided implementations of the particular data structures and algorithms, but also based on the performance of these implementations. Then a summary of the results and recommendations regarding the practical uses of each library are provided. Performance is measured using the Java Microbenchmark Harness. According to the results of this paper, implementations in Java Collections Framework are suitable for most use cases, when more complex data structures are not needed. When more specific data structures or algorithms are needed, that are not covered in Java Collections, GUAVA is a good alternative. For graph-like data structures the use of libraries JUNG and JGraphT is recommended.

Keywords: *data structures, algorithms, libraries, Java*

1. INTRODUCTION

Libraries are used frequently by programmers and make their work easier. These libraries are written by professionals, who optimize the solutions to have the best possible performance. Libraries of algorithms and data structures provide implementations of common data structures which are the fundamental building blocks of every program [1]. These implementations often differ from each other in their internal structure, but also in performance. For a programmer to be able to decide which library or implementation to use in a given project, he must have at least basic knowledge of these libraries and their possible uses. Although this paper focuses on the **Java** language, libraries providing implementations for data structures and algorithms exist for other languages too. To mention only a few of them: *System.Collections.Generic* [2] and *C5 Generic Collection Library for C# and CLI* [3] for **C#**, *STL*[1] for **C++** and the *Python Standard Library*[2] for **Python**. Libraries for these languages or others could also be compared in a further work. The main goal of this paper is to present and summarize the results of our work, which provided a detailed comparison of the contents and performance of the built-in Java Collection Framework and several other libraries of algorithms and data structures. These comparisons should provide a basis on which programmers can decide which library is suitable for use in their projects.

2. RELATED WORK

A basic comparison was done, when the Data Structures Library for Java was written [4]. That paper provides comparison of Java Collections, the Generic Library for Java, the Graph Foundation Classes for Java, and the Data Structures Library in Java. However, some of these libraries are outdated or not open source. Also, the performance of these libraries is not compared. In that paper libraries were mostly compared based on the implementations they provide for the data structures which are also mentioned in this paper. That paper is not actual and does not provide detailed comparisons and descriptions of the selected libraries.

A study on usage patterns of collections implementations from six popular alternative collection libraries, and their evaluation and performance comparisons in different scenarios can be found in the paper *Empirical Study of Usage and Performance of Java Collections* [5]. 10,986 Java projects were analyzed in that paper.

Other useful information can be found on the site *algorist* [3]. This site offers basic information about a great number of libraries for a wide variety of languages. Detailed comparisons are not provided on this website.

In an article listing useful libraries for Java, among others, libraries for collections are also listed, though not compared with each other [6]. An overview of the provided data structures in the built in Java Collections Framework is provided by John Zukowski in his book [7].

In contrast to the mentioned articles and collections of libraries, in our work the selected libraries were compared in detail based on their contents, the implementations they provide for the selected data structures and algorithms, and also the performance of these implementations. The results of the comparison are also summarized and recommendations regarding the practical use of each library are provided in the work related to this paper [8].

3. METHODS

In this paper selected libraries were compared based on their contents and performance. Their contents were evaluated based on how many different implementations they provide for abstract data types and algorithms, and what these implementations provide (methods, capabilities and other attributes) [9].

Their performance was measured using the Java Microbenchmark Harness (JMH) [10] [11]. Google Caliper was also considered for this task, but JMH offers more flex-

[1] Standard Template Library `http://www.cplusplus.com/reference/stl/`
[2] Python Standard Library `https://docs.python.org/3/library/`
[3] `http://algorist.com/algorist.html`

ible and customizable measurements for Java code fragments [12] [13]. Benchmarks were run in average time mode (measures the average time it takes for the benchmark method to execute). For the measurements in this paper 10 warm up iterations were used to make sure the results are not completely random. More warm up iterations make the measurements more accurate. After warm up, 15 measurement iterations were run [14]. While the benchmarks were running, the computer it was running on was not used, and all unnecessary applications were closed to make the results as clean as possible. The results are also influenced by the hardware used, but since all benchmarks were running on the same system under the same circumstances, the values of the benchmark can be different on different systems or hardware, while the relationship between the values remain the same.

Another possibility of measuring performance is the use of Savina benchmark suite for actor oriented programs [15]. However this possibility was not explored further in the work related to this paper.

4. LIBRARIES

Libraries, which implement the basic or more advanced data structures or algorithms can be found for every higher level language. Most languages even have built in support for the most frequently used ones. In this work we focus on open source libraries that are dedicated to the Java language.

4.1. Java Collections Framework

Java Collections (JC) is the built in library of basic data structures of the Java language. It contains implementations for the ADT List, ADT Set, ADT Priority queue, ADT Dictionary and also for sorting and searching algorithms. JC is frequently updated to use the most recent technologies provided by Java. The `Collection` interface is on the top of the class hierarchy in JC, from which all other interfaces and classes are derived, the only exception is the `Map` interface. The algorithms that can be used on these data structures are implemented in the static methods of the `Collections` class.

4.2. Data Structures Library for Java

JDSL is an older library, developed at Brown University. Its newest version was released in 2005 [4]. According to the authors, this library is not supposed to replace Java Collections, but extend it. JDSL provides implementations for all the mentioned data structures except ADT Set, and covers most of the mentioned algorithms as well. The interfaces in JDSL are organised to two categories: one for positional containers and the second for key-based containers. Positional containers capture the topological relations between elements, while key-based containers are used to store key-value pairs. JDSL can be obtained from GitHub[4] or from the Maven Repository[5].

4.3. Java Universal Network/Graph Framework

JUNG is an open source library for modeling, analysis and visualization of graphs [16]. This library focuses mainly on the ADT Graph, but also provides implementations for the ADT Tree. According to the authors, the target audience is Java programmers with interest in graphs, and it is suited for building applications related to network exploration and data mining [17]. JUNG covers a large portion of the graph algorithms. All algorithms can be found in the package *edu.uci.ics.jung.algorithms*, divided into subpackages by category. Tools for graph visualization are provided in the package *edu.uci.ics.jung.visualization*, but the library Vl-jung[6] can also be used for this purpose. JUNG can be downloaded from its official site[7], or from the Maven Repository.

4.4. Google Core Libraries for Java

GUAVA is also an open source set of libraries developed by Google, which is used in almost all Google projects written in Java. It provides new types of collections, immutable collections, graph implementations, hashing and much more. GUAVA contains implementations or some kind of extension for an existing implementation for every mentioned data structure except the ADT Tree, but tree-like structures can also be represented as directed acyclic graphs. It provides tree traversal and graph algorithms. In terms of algorithms and data structures, the packages *com.google.common.collect* and *com.google.common.graph* are of the most interest to this work. The latest version of GUAVA can be imported to any project from the Maven Repository.

4.5. Apache Commons Collections

Commons Collections expands Java Collections with new interfaces and implementations. CC is an open source library which is still being developed. It decorates already existing implementations with new behaviours, such as support for use in multiple threads, or the ability to access elements of a collection using a key. It also provides its own implementations for certain data structures like lists, sets or maps. All classes and interfaces in Commons Collections are derived from existing ones in Java Collections. Apart from data structures, CC also provides different implementations of iterators and comparators. Like all previous libraries, CC can also be obtained from the Maven Repository.

4.6. JGraphT

Like JUNG, JGraphT is also specialised for the ADT graph. It covers most types of graphs from graph the-

[4]JDSL on GitHub `https://github.com/lewischeng-ms/jdsl`
[5]JDLS on Maven `https://mvnrepository.com/artifact/jdsl/jdsl`
[6]Vl-jung GitGub homepage `https://github.com/timboudreau/vl-jung`
[7]Official site of JUNG: `http://jung.sourceforge.net/download.html`

ory, and also most of the graph algorithms. JGraphT provides adapters to convert graphs from other libraries, such as GUAVA. All graph implementations in JGraphT implement the `Graph` interface and are grouped in the *org.jgrapht.graph* package. The algorithms are in the package *org.jgrapht.alg* divided into sub-packages by category. Traversal algorithms are also provided in the package *org.jgrapht.traverse*. JGrapht supports graph visualization, however an external library must be used. The authors recommend the JGraphX library for the visualization of graphs created in JGraphT [18].

4.7. Other libraries

Apart from the previously mentioned libraries, others are also available. When JDSL was created, it was compared to the libraries that were available at the time. They were the Generic Library for Java (JGL) and Graph Foundation Classes for Java (GFC). JGL was based on the design of the STL C++ library. It is not open source so it was not included in this comparison. GFC is an outdated library, last updated in 1999. It uses old technologies which is the reason it's not included in this comparison.

Other libraries, that are dedicated to a specific data structure or type of algorithm may also be available and compared with each other in the future. This work is dedicated to the libraries that cover a larger number of data structures or algorithms.

5. COMPARISON

As previously mentioned, the selected libraries will be compared based on their contents and their performance. The data structures and algorithms for which the selected libraries provide implementations are marked with a black circle in Table 1. Performance was measured for data structures and algorithm which had multiple implementations.

5.1. ADT List

The built in Java Collections Framework offers two implementations for the ADT List. The first is `ArrayList`, which internally uses an array, the size of which dynamically changes. It has an other variant, `CopyOnWriteArrayList`, which is thread safe. The other implementation is the `LinkedList`, which internally uses a doubly linked list.

JDSL provides two implementations of the ADT List: `ArraySequence` and `NodeSequence`. They are analogical to the `ArrayList` and `LinkedList` from JC. In JDSL methods for inserting to the beginning and the end of the list are also provided.

GUAVA offers a number of static utility methods for working with lists. These methods include constructing the cartesian product of multiple lists, partitioning a list to sublists, or reversing the order of elements in a list.

Commons Collections offers some thread safe implementations for the ADT List, and another implementation: `TreeList`. The `TreeList` is optimized for fast insertions and deletions based on indexes. Apart from these implementations, CC also provides multiple classes using the decorator design pattern to add new properties to existing implementations (for example `GrowthList` or `LazyList`).

Table 1 Data structure and algorithm coverage of the libraries. JC = Java Collections Framework, DSL = Data Structures Library for Java, JNG = Java Universal Network/Graph Framework, GUA = Google Core Libraries for Java, CC = Apache Commons Collections, JGT = JGraphT, PQ = Priority queue, Dict = Dictionary, BFS = Breadth First Search, DFS = Depth First Search, MST = Minimum Spanning Tree

	JC	DSL	JNG	GUA	CC	JGT
List	•	•		•	•	
Set	•			•	•	
PQ	•	•		•		
Dict	•	•		•	•	
Tree		•	•			
Graph		•	•	•		•
BFS				•		•
DFS		•		•		•
MST		•	•			•
Inorder		•				
Preorder		•		•		
Postorder		•		•		
Merge sort	•	•				
Heap sort		•				
Quick sort	•	•				
Radix sort						•
Bin. search	•					

From the mentioned implementations `ArrayList` had the best performance in each operation except inserting to the beginning of the list. `TreeList` was also fast when idexes were used. Implementations from JDSL achieved comparable results for most operations. The results of the benchmark can be seen in Table 2.

5.2. ADT Set

There are multiple implementations for the ADT Set provided by Java Collections. Thread safe implementations, like `CopyOnWriteArraySet`, and specific sets for storing enumeration types like `EnumSet` are also provided. For the performance comparison the implementations `HashSet`, `LinkedHashSet` and `TreeSet` were selected.

The GUAVA library offers multiple implementations of the Multiset data type, which is similar to ADT Set, but allows duplicit elements. There are three main implementations of this data structures in GUAVA: `HashMultiset`,

`LinkedHashMultiset` and `TreeMultiset` which are structurally similar to the implementations in JC.

In Commons Collections there are also only implementations of the ADT Multiset. The main implementation of this data structure is `HashMultiSet` which internally uses a `HashMap`. It also has a method called *setCount*, by which the number of occurrences of an element can be set. Other classes only decorate the base implementation with new features, like synchronisation or only accepting elements that match a specified predicate.

The results of the performance tests (as seen in Table 3) were done separately for the ADT Set and Multiset. The Set implementations achieved similar results in all operations except *remove* and *contains*, where `TreeSet` was slower. The same could be said about the Multiset implementations, where `TreeMultiset` had a worse performance.

Table 2 Results of the ADT List Benchmark. Unit of measurement: μs / operation

	JC		JDSL		CC
	ArrayList	LinkedList	ArraySequence	NodeSequence	TreeList
clear()	0.002	0.003	0.002	0.001	0.002
insert()	0.034	0.122	0.143	0.229	0.299
insert(first)	538.889	0.154	13380.409	0.205	0.357
insert(middle)	115.268	3050.458	7752.18	3122.877	0.372
insert(last)	9.486	28.579	45.001	32.702	0.354
remove(element)	4452.429	9080.809	n/a	n/a	23886.322
contains(element)	1488.72	2529.767	0.005	0.005	5157.715
size()	0.003	0.003	0.003	0.003	0.003
isEmpty()	0.004	0.004	0.004	0.004	0.003
insertFirst()	n/a	n/a	12894.091	0.165	n/a
insertLast()	n/a	n/a	0.192	0.15	n/a

Table 3 Results of the ADT Set Benchmark. Unit of measurement: μs / operation

	JC			GUAVA			CC
	HashSet	LinkedHashSet	TreeSet	HashMultiset	LinkedHashMultiset	TreeMultiset	HashMultiSet
clear()	0.003	0.003	0.002	0.006	0.007	0.005	0.003
insert()	0.011	0.012	0.01	0.005	0.005	0.023	0.005
remove(el.)	0.003	0.003	0.052	0.002	0.002	0.131	0.002
contains(el.)	0.006	0.006	0.047	0.007	0.008	0.055	0.006
size()	0.004	0.004	0.004	0.004	0.004	0.014	0.003
isEmpty()	0.004	0.004	0.004	0.005	0.006	0.013	0.004

5.3. ADT Priority queue

The class `PriorityQueue` represents the ADT Priority queue in Java Collections. The elements in the priority queue are ordered based on their natural ordering, or by a Comparator provided to the constructor.

In JDSL the ADT Priority queue also has one implementation: `ArrayHeap`, implemented using a binary heap.

The library GUAVA provides the class `MinMaxPriorityQueue` for representing priority queues. It is a double-ended queue, which provides access for the least and also the greatest element in the queue.

The performance of these implementations was comparable in all operations except *insert* and *poll*, where `ArrayHeap` had the worst performance and `PriorityQueue` was slightly faster than `MinMaxPriorityQueue`. The results can be seen in Fig. 1.

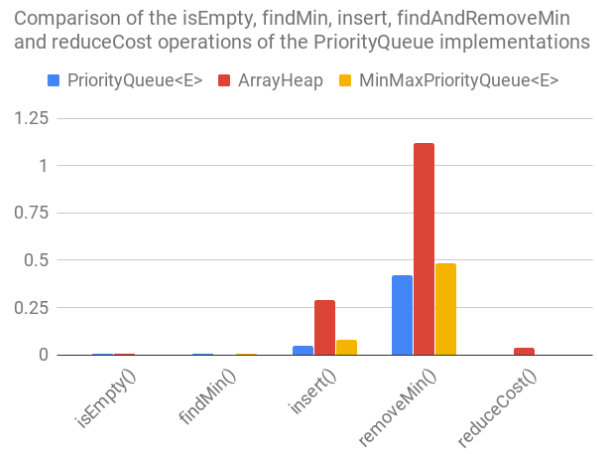

Fig. 1 Results of the ADT Priority queue Benchmark. Unit of measurement: μs / operation

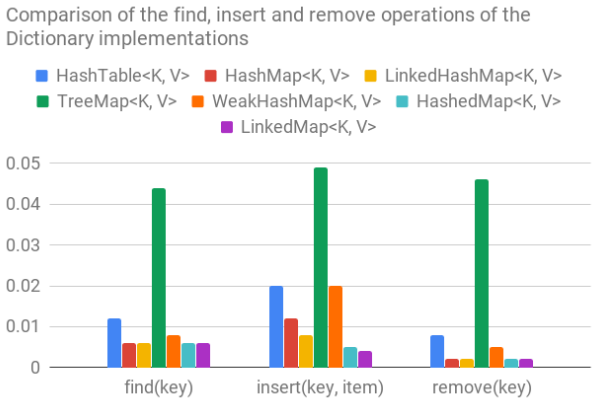

Fig. 2 Results of the ADT Dictionary Benchmark pt.1. Unit of measurement: μs / operation

5.4. ADT Dictionary

JC offers multiple implementations for the Dictionary ADT: *HashTable* (obsolete since Java 11), `HashMap` (similar to `HashTable`), `LinkedHashMap` (uses a hash table and a doubly linked list), `TreeMap` (based on a Red-Black tree), `WeakHashMap` (based on a hash table, but values are automatically removed when the key gets deleted by the garbage collector).

In JDSL the `Dictionary` interface has two implementations: `RedBlackTree` (built on a restructurable binary tree) and `HashtableDictionary` (uses a chaining hashtable).

In the GUAVA library, bidirectional maps (`HashBiMap`) and multimaps can be found. There are multiple approaches to implement a multimap based on the type of collection in which the elements are stored. The following classes were included in the performance comparison: `ArrayListMultimap`, `TreeMultimap` and `HashMultimap`.

Commons Collections has implementations for both the Map (`HashedMap`, `LinkedMap`) and Multimap data structures (`ArrayListValuedHashMap`, `HashSetValuedHashMap`). CC also provides implementations for the bidirectional map data structure, which allows bidirectional lookup between key and values.

The performance was measured for the mentioned implementations of the ADT Dictionary or Map and Multimap. Due to the number of compared implementations, the results are displayed via charts in figures Fig. 2 and Fig. 3. `TreeMap` and `TreeMultimap` had the worst performance, and also `HastableDictionary` and `RedBlackTree` from JDSL. However, the implementations from JDSL have both *find* and *findAll* operations defined, so they can be used as a map just as well as a multimap.

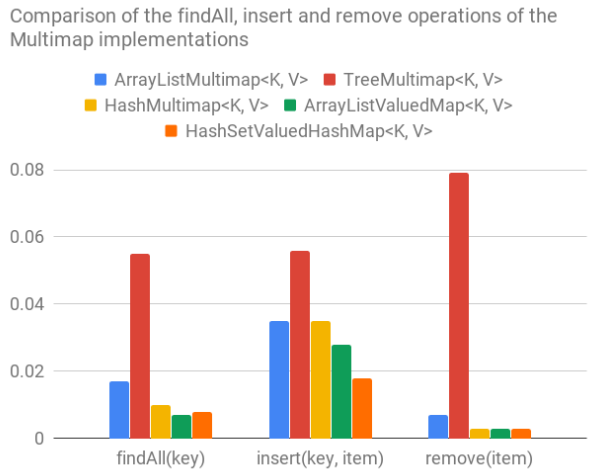

Fig. 3 Results of the ADT Dictionary Benchmark pt.2. Unit of measurement: μs / operation

5.5. ADT Tree

In JDSL `NodeTree` represents a node-based multipurpose tree, and `NodeBinaryTree` can be used as a binary tree.

JUNG also has two implementations: `DelegateTree` (delegates to a specified instance of `DirectedGraph`) and `OrderedKAryTree` (a tree where each vertex has $\leq k$ children).

The performance was measured for all mentioned implementations except `OrderedKAryTree`, due to inconsistencies with the documentation of the class. All three classes achieved similar results in all operations except *parent*, *children* and *isRoot*, where `DelegateTree` from JUNG had far worse performance than the implementations from JDSL. The results of the benchmark can be seen in Fig. 4.

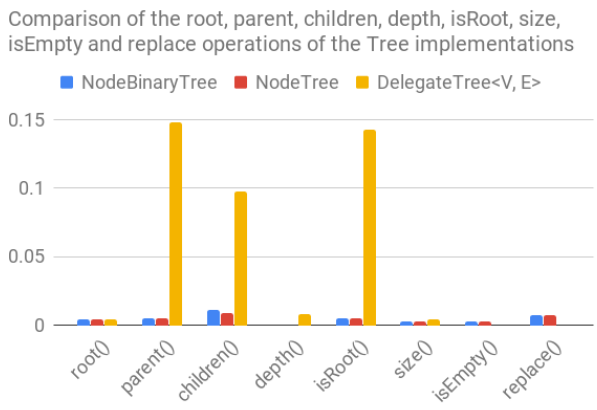

Fig. 4 Results of the ADT Tree Benchmark. Unit of measurement: μs / operation

5.6. ADT Graph

One implementation of the ADT Graph is provided in JDSL: `IncidenceListGraph`. It allows self-loops, parallel edges, mixed directed and undirected edges. It is implemented via a list of vertices and a list of edges. The neighbouring vertices are stored for each vertex.

JUNG is a library specifically for the ADT Graph. It contains implementations for almost every kind of graph. For the performance comparison the classes `SparseGraph` and `SparseMultigraph` were selected, which represent the base implementations of the graph and multigraph data structure.

In GUAVA two types of graphs can be found: `Graph` (values are only associated with the vertices) and `ValueGraph` (vertices and also edges are associated with values). In GUAVA graphs can be created with the help of the `GraphBuilder` and `ValueGraphBuilder` classes. Static utility methods are available in the `Graphs` class for determining if a graph has cycles, creating a subgraph of a graph induced by nodes, reversing the edge directions and a lot more.

JGraphT is also a library specialized for the ADT Graph. It offers implementations for every kind of graph and most graph algorithms. For the performance comparison the `SimpleGraph` and `Multigraph` classes were selected, which are the base implementations for graphs and multigraphs.

The performance of these implementations was measured on graphs with random structure. As seen in Table 4, performances are similar in most operations. `SparseMultigraph` had worse performance in the operations *addEdge* and *adjacent*. The *removeEdge* operation was the slowest for the `IncidenceListGraph`.

Further information about graphs and graph algorithms in Java can be found in the paper by Marije de Heus [19].

Table 4 Results of the ADT Graph Benchmark. Unit of measurement: μs / operation. v = vertex, w = weight

	JDSL	JUNG		GUAVA	JGraphT	
	IncidenceListGraph	SparseGraph	SparseMultigraph	Mutable-ValueGraph	SimpleGraph	Multigraph
addVertex(vertex)	0.399	0.793	1.026	0.717	0.736	0.747
addEdge(v1, v2)	0.605	0.407	4.141	0.189	0.128	1.259
addEdge(v1, v2, w)	n/a	n/a	n/a	n/a	0.067	0.067
adjacent(v1, v2)	0.209	0.681	448.264	n/a	n/a	n/a
removeEdge(v1, v2)	74.137	0.048	1.721	0.114	0.033	0.032
getVertices()	0.011	0.008	0.015	0.006	0.004	0.004

5.7. Sorting algorithms

In this paper the following sorting algorithms were taken into consideration:

Merge sort Sorting algorithm with divide and conquer approach. Merge sort divides the array or list to halves and combines them in a sorted manner [20].

Heap sort Heap sort is a comparison based sorting algorithm based on a binary heap structure. It creates a heap and then sorts the data in reverse by repeatedly placing the largest unsorted element into its correct place [20].

Quick sort Quick sort is also a divide and conquer algorithm, its approach is to separate bigger and smaller elements repeatedly. It uses a chosen pivot value that is used to divide bigger and smaller elements [20].

Radix sort Radix sort sorts strings or numbers by comparing their digits or characters. It groups the values by individual digits or characters that share the same significant position and value, then joins these groups [20].

From the selected libraries only a few provide implementations of sorting algorithms. In Java Collections the method *sort* in the `Collections` and `Arrays` classes uses

Timsort to sort Lists of objects or arrays of objects. An implementation of Dual-Pivot Quicksort can also be found in JC, however, it's limited to sorting arrays of primitive types.

JDSL provides implementations for multiple sorting algorithms: `ArrayMergeSort` (optimised for sorting `ArraySequence`) and `ListMergeSort` (optimised for sorting `NodeSequence`). JDSL is the only library from the selected ones, that provides an implementation of the Heap sort algorithm in the `HeapSort` class. An implementation of Quick sort is also provided by the `ArrayQuickSort` class.

An implementation of the Radix sort algorithm can be found in JGraphT in the `RadixSort` class, which has one method: *sort*, for sorting a list of integers. This implementation is limited to sorting lists of integers. If the number of elements in the list is less than `RadixSort.CUT_OFF`, then the standard Java sorting algorithm is used.

In the performance measurement `Arrays.sort()` achieved the best results, but it is limited to sorting arrays of primitive types. The Timsort implementation in both `Collections.sort()` and `Arrays.sort()` also were fairly fast. The Radix sort implementation from JGraphT was slower compared to the implementations in JC. The results of the implementations from JDSL were slower by multiple orders of magnitude. The results of the benchmark can be seen in Table 5.

5.8. Searching algorithms

Binary search Binary search searches for a value in an ordered collection. It operates with a left an right index, compares the middle value and if the condition is unsatisfied, the half not containing the value is eliminated and the search continues on the remaining half until it finds the target value [21].

An implementation of searching algorithms, specifically binary search, was only found in Java Collections. The *binarySearch* method can be found in both the `Collections` and `Arrays` classes. The method returns the index of the search key, if it is contained in the list, otherwise, `(-(insertion point) - 1)`[8]. In the `Arrays` class overloaded methods are also provided to search only in a specific part of an array [22] [23] [24].

Table 5 Sorting algorithms Benchmark results. Unit of measurement: μs / operation

JC			JGraphT
Collections.sort(List)	Arrays.sort(Object[])	Arrays.sort(int)	RadixSort.sort(List)
3.418	3.103	0.41	26.336
JDSL			
ArrayMergeSort	ListMergeSort	ArrayQuickSort	HeapSort
385.625	393.144	3103.646	188.939

5.9. Tree traversals

Tree data structure can be traversed in multiple ways. In this paper Inorder (visits left child first, then the root and the right child is last), Preorder (root, left child, right child) and Postorder (left child, right child, root) traversals were examined.

In JDSL traversing trees can be done using iterators. It offers three iterators: `PreOrderIterator` (can be used with binary trees and multi purpose trees), `PostOrderIterator` (also works with both types of trees) and `InOrderIterator` (only works with binary trees). In addition, JDSL also offers an implementation of the Euler tour algorithm [25] in the `EulerTour` class.

Although GUAVA does not have its own implementation of the ADT Tree, trees can be represented in GUAVA using directed acyclic graphs. A `Traverser` object can be obtained using the *Traverser.forTree()* method. After obtaining the `Traverser` object for a specific tree, Postorder traversal can be achieved by using the *depthFirstPostOrder* method and Preorder traversal by using the *depthFirstPreOrder* method on the `Traverser` object. Both methods return an iterator with the specified order of elements.

The performance was measured on trees with the same structure. As seen in Table 6, implementations from JDSL achieved better results, that could be caused by the fact that GUAVA does not have a specific tree implementation, but it uses directed acyclic graphs.

Table 6 Tree traversal algorithms Benchmark results. Unit of measurement: μs / operation

	JDSL		GUAVA
	Tree	BinaryTree	MutableGraph
PreOrder	2.186	1.936	7.513
PostOrder	2.19	1.981	8.392
InOrder	n/a	2.188	n/a

5.10. Graph algorithms

This work focuses on three graph algorithms:

[8] Collections class documentation `https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`

Breadth-first search Starts at the root node, and explores all of the neighbour nodes at the present depth, before moving on to the nodes at the next level [26].

Depth-first search Starts at the root node, and explores as far as possible along each branch before backtracking [27].

Minimum-spanning-tree MST is a subset of the edges of a graph that connects all vertices without any cycles and with the minimum possible total edge weight [28].

JDSL provides implementation for DFS in the `DFS` class. Other variants of this algorithm are also available for directed graphs and for finding cycles in graphs. An implementation of minimum-spanning-tree is also provided in the `IntegerPrimTemplate` abstract class. When extending this class, the method *weight* must be overridden. JDSL offers other graph algorithms as well, for example topological sorting, an implementation of Dijkstra's algorithm or pathfinding algorithms.

Although JUNG is a library specifically for the ADT Graph, it does not contain implementations of the DFS and BFS algorithms. It does contain a minimum-spanning-tree implementation in the *PrimMinimumSpanningTree* class. It also provides a number of different algorithms related to clustering, filtering, graph flows, scoring, shortest path and graph transformation.

GUAVA provides an implementation for both DFS and BFS algorithms, but it doesn't contain any minimum-spanning-tree implementations. Traversing graphs is similar to traversing trees in GUAVA. First, an instance of the *Traverser* object must be obtained using the *Traverser.forGraph()* method. After this, the *breadthFirst*, *depthFirstPreOrder* and *depthFirstPostOrder* methods can be called, each of which returns an *Itarable* object with the specified order of vertices.

JGraphT is also a library specifically for the ADT Graph. It contains implementations for all three previously mentioned algorithms. It also provides implementations for a large number of algorithms from graph theory. Depth-first and Breadth-first traversals can be achieved by using the *DepthFirstIterator* and *BreadthFirstIterator* classes. JGraphT contains implementations of multiple minimum-spanning-tree algorithms, such as *BoruvkaMinimumSpanningTree*, *KruskalMinimumSpanningTree* and *PrimMinimumSpanningTree*.

Performance was measured on graphs with the same structure. The results for the BFS and DFS algorithms were similar. From the implementations of the minimum-spanning-tree algorithm, *KruskalMinimumSpanningTree* was the fastest, and *PrimMinimumSpanningTree* the slowest (three times slower than the implementation of Kruskal's algorithm). The benchmark results can be seen in Table 7.

Table 7 Graph algorithms Benchmark results. Unit of measurement: $\mu s/operation$

	JDSL	GUAVA	JUNG	JGraphT			
					Boruvka	Kruskal	Prim
BFS	n/a	4.631	n/a	6.471	n/a	n/a	n/a
DFS	7.388	6.259	n/a	7.53	n/a	n/a	n/a
SpanningTree	18.48	n/a	34.696	n/a	22.675	10.047	12.751

6. CONCLUSIONS AND RECOMMENDATIONS

In this section the results of the comparison of the libraries will be summarized. Also recommendations on the possible cases in which the individual libraries can be used will be provided for each library.

6.1. Java Collections Framework

One advantage of using JC is, that it is built into Java, no additional library is needed. It provides implementations for the basic data structures, such as lists, sets, priority queues and dictionaries. These implementations are frequently updated to use the newest technologies. JC has multiple search algorithms included and is the only library from the selected libraries, that has an implementation of a search algorithm. It lacks the more complicated data structures like the ADT Tree or Graph.

Java Collection can be used when there is no need for more complex data structures and the basic implementations provided satisfy the requirements. Implementations from JC are easy to use and offer a decent performance in comparison to the other libraries.

6.2. Data Structures Library for Java

JDSL is the oldest from the selected libraries. Due to its age, it does not support generic programming which may cause issues in the form of type incompatibility. The method names often differ from the ones in other libraries. It covers all mentioned data structures except the ADT Set and supports a wider variety of operations for some data structures. It also has a built in support for converting data structures between JDSL and JC.

The use of JDSL should be only considered if an implementation is needed which is not provided by any other, more up to date library. Despite its age, JDSL has a lot of interesting implementations for most data structures.

6.3. Java Universal Network/Graph Framework

JUNG contains only implementations of the ADT Graph and Tree, but it covers a relatively large variety of graphs. It also contains tree implementations, but no tree traversal algorithms. A big advantage of JUNG is its built-in support of graph visualization in the *edu.uci.ics.jung.visualization* package.

This library can be used for modeling, analysis and visualization of graphs and networks. All implementations can be modified to meet the needs of the user. It can also be used for visualization (either using the built in tools or an external library like Vl-jung)

6.4. Google Core Libraries for Java

GUAVA is an up to date library, which provides implementations or new features for every mentioned data structure, except the ADT Tree, but trees can also be represented in GUAVA by directed acyclic graphs. It also provides tree traversal algorithms and graph algorithms. Static utility methods can be found in the `Lists`, `Sets`, `Queues`, `Maps` and `Graphs` classes, that can make the programmer's work easier.

GUAVA is the most frequently used from the mentioned libraries according to the Maven Repository. It can be used for a wide variety of projects because of its flexibility. It's a great extension for JC.

6.5. Apache Commons Collections

Commons Collections provides a number of implementations for the basic data structures but their performance is not overwhelmingly good. Some implementations can be useful in specific cases, for example `TreeList` when a lot of insertions and deletions with indexes are executed, or when a specific behaviour provided by CC is needed. In other cases the use of more universal, flexible library like GUAVA is recommended.

6.6. JGraphT

JGraphT is also a library dedicated specifically for the ADT Graph. It covers most types of graphs and graph algorithms. From the selected libraries only JGraphT provides an implementation of the Radix sort algorithm, although it is limited to sorting lists of integers. JGraphT also provides adapters for converting graphs from the GUAVA library. Graphs can also be visualized, but an external library must be used. The authors recommend using the JGraphX library for visualization.

6.7. Future work

This work can be further extended for example by widening the selection of Java libraries. Libraries for other languages and platforms could also be considered (libraries for C#, C++, Python and others) in comparison. The scope of data structures could be extended to include data structures that are not covered in this paper. Furthermore a comparison between the individual implementations could also be done based on their internal structure.

REFERENCES

[1] SHAFFER, C.A.: Data Structures and Algorithm Analysis. Edition 3.2 (Java Version). Dover Publications. Mar. 2013. url: http://people.cs.vt.edu/~shaffer/Book/JAVA3elatest.pdf

[2] Microsoft: Microsoft Docs, .NET Framework API Reference, System.Collections.Generic Namespace. url: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netframework-4.7.2.

[3] KOKHOLM, N. – SESTOFT, P.: The C5 Generic Collection Library for C# and CLI, IT University of Copenhagen. Nov. 2016. url: https://www.itu.dk/research/c5/.

[4] TAMASSIA, R. et al.: An Overview of JDSL 2.0, the Data Structures Library in Java. Aug. 2005. url: https://cs.brown.edu/cgc/jdsl/other_modules/overview/overview.pdf.

[5] COSTA, D. – ANDRZEJAK, A. – SEBOEK, J. – LO, D.: Empirical Study of Usage and Performance of Java Collections. ICPE '17: Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering. April 2017. url: https://dl.acm.org/doi/10.1145/3030207.3030221.

[6] JAVIN, P.: Top 20 Libraries and APIs Java Developer should know. Javarevisited blog, September 2019. url: https://javarevisited.blogspot.com/2018/01/top-20-libraries-and-apis-for-java-programmers.html.

[7] ZUKOWSKI, J.: Java Collections. Apress. Jan. 2008. 420p. isbn: 1430208546, 9781430208549.

[8] PERHÁČ, P.: Algorithms and Data Structures Libraries for Java, Bachelor's thesis, May 2019.

[9] GOODRICH, M.T. – TAMASSIA, R. – GOLDWASSER, M.H.: Data Structures and Algorithms in Java, 6th Edition. Wiley, 2014. ISBN: 978-1-118-77133-4.

[10] EGOROV, D.: JMH - Great Java Benchmarking. dzone.com, Performance Zone, Tutorial. Okt. 2017. url: https://dzone.com/articles/jmh-great-java-benchmarking.

[11] COSTA, D.: Benchmark-driven Software Performance Optimization. PhD. Thesis. July 2019, url: https://www.researchgate.net/publication/335014121_Benchmark-driven_Software_Performance_Optimization.

[12] SESTOFT, P.: Microbenchmarks in Java and C#. IT University of Copenhagen, Denmark. Sep. 2015. url: https://www.itu.dk/people/sestoft/papers/benchmarking.pdf.

[13] SHIPILEV, A.: (The Art of) (Java) Benchmarking. Talk given by Aleksey Shipilev for JavaOne SF 2011. Okt. 2011. url: https://shipilev.net/talks/j1-Oct2011-21682-benchmarking.pdf

[14] COSTA, D.– BEZEMER, C.P. – LEITNER, P. – ANDRZEJAK, A.: What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks, IEEE Transactions on Software Engineering. doi: 10.1109/TSE.2019.2925345. June 2019. url: https://ieeexplore.ieee.org/document/8747433.

[15] IMAM, S.M. – SARKAR, V.: Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control (AGERE! '14). ACM, New York, USA, pp. 67-80, October 2014. url: http://soft.vub.ac.be/AGERE14/papers/ageresplash2014_submission_19.pdf.

[16] O'MADADHAIN, J. – FISHER, D. – NELSON, T.: JUNG Java Universal Network/Graph Framework, JUNG Frequently Asked Questions. Jan. 2010. url: http://jung.sourceforge.net/faq.html.

[17] FISHER, D.: The Java Universal Network/Graph Framework (JUNG): A Brief Tour. A talk given by Danyel Fisher for the UC Irvine KDD Project Apr. 2004.

[18] MICHAIL, D. – KINABLE, J. – NAVEH, B. – SICHI, J.V.: JGraphT - A Java library for graph data structures and algorithms. Apr. 2019. url: https://www.researchgate.net/publication/332494171_JGraphT_--_A_Java_library_for_graph_data_structures_and_algorithms.

[19] DE HEUS, M.: Towards a Library of Parallel Graph Algorithms in Java. 14th Twente Student Conference on IT, Enschede, The Netherlands, 2011. url: https://fmt.ewi.utwente.nl/media/49.pdf

[20] MANOOCHEHR, A.: Abstract Data Types and Algorithms. Macmillan Computer Science Series. Palgrave Macmillan UK, 1990. ISBN: 978-0-333-51210-4, 978-1-349-21151-7.

[21] ZAVERI, M.: An intro to Algorithms: Searching and Sorting algorithms. codeburst.io. Mar. 2018. url: https://codeburst.io/algorithms-i-searching-and-sorting-algorithms-56497dbaef20

[22] SEDGEWICK, R.: Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, 3rd Edition. Addison Wesley, 1998. ISBN 9780201350883, 0201350882.

[23] SULTANA, N. et al.: A Brief Study and Analysis of Different Searching Algorithms. 2017 IEEE International Conference on Electrical, Computer and Communication Technologies, IEEE ICECCT 2017, COIMBATORE, Volume: 4. url: https://www.researchgate.net/publication/314175061_A_Brief_Study_and_Analysis_of_Different_Searching_Algorithms.

[24] SINGH, CH.: Search Algorithms in Java. StackAbuse.com. Mar. 2019. url: https://stackabuse.com/search-algorithms-in-java/

[25] FOURNIER, J.C.: Graph Theory and Applications: With Exercises and Problems, Chapter 9. ISTE Ltd., Jan. 2009. ISBN: 9781848210707, 9780470611548.

[26] CORMEN, T.H. et al.: Introduction to Algorithms, Third Edition. The MIT Press. July 2009. ISBN: 9780262033848.

[27] THULASIRAMAN, K.: Handbook of Graph Theory, Combinatorial Optimization, and Algorithms. Chapman and Hall/CRC. Dec. 2015. ISBN: 9781584885955.

[28] ANTOŠ, K.: Minimum spanning tree problem. 14th Conference on Applied Mathematics, APLIMAT 2015. Slovak University of Technology in Bratislava, 2015.

Received July 29, 2019, accepted April 6, 2020

BIOGRAPHIES

Patrik Perháč was born on 9. 7. 1997. In 2019 he graduated (B.Sc.) at the department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University in Košice. His thesis was titled Algorithms and data structures libraries for Java.

Slavomír Šimoňák received his M.Sc. degree in computer science in 1998 and his Ph. D. degree in computer tools and systems in 2004, both from the Technical University of Košice, Slovakia. He is currently Associate Professor at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at the Technical University of Košice, Slovakia. His research interests include formal methods integration and application, communication protocols, algorithms, and data structures.