Bitmap Compression Techniques for Large Graph Treewidth Computation

Jasper Munnichs

Supervisors: Hisao Tamaki and Erik Jan van Leeuwen

Second Examiner: Hans Bodlaender

August 2021

Abstract

In this thesis we investigate whether bitmap compression techniques could be useful data structures to represent vertex sets, for treewidth algorithms on large graphs. We consider bitmap compression techniques Roaring Bitmap and EWAH, and compare them with two more commonly used graph data structures: the bitmap and the array of integers. We investigate the behaviour of the data structures in two experiments. In the first, we investigate their computation time and memory consumption of typical vertex sets in the computation of separated components. In the second, we compare their performance on the Minimal Minimum Degree algorithm. We find that the array of integers representation performs best. However, as typical vertex sets that the data structures have to deal with get denser and more diverse, we observe that Roaring Bitmap starts to perform better.

Contents

1 Introduction

Developments in technology lead to rapid growth in the aggregation and application of data. This gives rise to the need for efficient algorithms that are able to handle large data sets. A fruitful approach to model large amounts of data makes use of a graph (a.k.a. "network") representation. Graphs are useful tools to model all kinds relational data. They find applications from biology (protein interactions) [37] and social sciences (relations between people) [21] to linguistics (co-occurence of words) [12].

Many computational problems on graphs have been shown to be NP-hard. This means that it is not possible to find algorithms that run in polynomial time with respect to the size of the problem, unless $P = NP$. Instead, we often have to rely on exponential time algorithms, which generally do not scale well to large problem instances. A lot a research is dedicated to making these problems tractable. One such approach makes use the treewidth of a graph. Intuitively, treewidth says something about how closely a graph resembles a tree. Formally there are multiple ways to characterize treewidth [8, 9], the most commonly used being the tree decomposition. Having a tree decomposition of a graph of small width can lead to efficient algorithms that make use of bottom-up dynamic programming. They run exponentially with respect to treewidth, while running polynomially or often even linearly with respect to the problem size. It has for example been shown that for a graph G of treewidth k, it can be determined in $O(n^{k+1})$ time whether a graph H of size n has a subgraph that is isomorphic to G [31]. This means that for graphs of small treewidth this problem becomes efficiently solvable. Some other famous problems that become efficiently solvable for graphs of small treewidth are Hamiltonian Circuit, Independent Set, and Vertex Cover.

Many different algorithms that compute the treewidth of a graph have been proposed, see 2.1 for a concise overview. The difficulty of computing treewidth is clearly shown by the fact that it is rare for state of the art exact algorithms to solve problem instances larger than 1000 vertices [19]. This has led to the search for heuristic algorithms that find good upper bounds on the treewidth. This approach is based on the idea that the time saved on constructing an optimal tree decomposition, could be better spent on other parts of the problem.

Many such algorithms need to represent partial graphs, which is usually done by representing the corresponding vertices. Especially algorithms based on dynamic programming on separators and connected components can make heavy use of vertex sets [11, 36]. Efficient ways to represent and perform operations on them could be a way to improve those algorithms in both memory and time consumption.

One commonly used approach of storing vertex sets is to make use of bitmaps, which store information as a sequence of bits. Bitmaps also happen to find employment in storing large amounts of data in databases, and a lot of research has been done to find efficient bitmap compression techniques [14, 38]. In a database, data structures have to consume little memory, while still allowing for fast query operations. These operations often require simple logical oper-

ations, such as intersections and unions between sets. Since these operations also find frequent use in treewidth algorithms, bitmap compression techniques might also prove useful for treewidth algorithms.

The main goal of the thesis will be to investigate bitmap compression techniques for treewidth algorithms, in their performance for large graphs. Hopefully this can improve implementations of algorithms such that they are able to handle larger and more diverse graph instances, making them more widely usable.

The bitmap compression techniques we consider are EWAH and Roaring Bitmap. We use these techniques because Wang et al. [38] studied both the memory consumption and time efficiency for a large number of bitmap compression techniques. They compared them with inverted list compression techniques, with respect to their performance on space overhead, decompression time, intersection time, and union time. For the bitmap compression techniques they found that Roaring Bitmaps performed best, followed by EWAH. Since graph algorithms often make use of simple set operations, we expect that their results give an indication of what works well in our experiments. We compare Roaring Bitmaps and EWAH with the data structures that are commonly used in context of graphs, namely the array of integers and the regular bitmap. Note that the array of integers is closely related to inverted list compression techniques, since the latter is basically a compression of the first.

One of the reasons for this research is to further improve the HBT algorithm by prof. Tamaki [35], which is a heuristic treewidth algorithm that makes use of dynamic programming. We base most of the research on this algorithm, with the aim that the results might provide improvements on this particular algorithm. Still, it is important to note that the result are also applicable and can possibly be extended to other algorithms.

Since HBT is quite a complicated algorithm, implementing all the data structures is costly. For this reason we develop the so-called Benchmark Experiment, that simulates the most time-consuming operation in HBT, as well as the typical vertex sets that it has to store. In this experiment we investigate the computation of the components separated by the set of separators. To make sure that we get a broader insight in how the data structures deal with such a task, we compare them in two different algorithms in the Benchmark Experiment. The first makes use of a regular Breadth First Search algorithm, the second one computes the separated components by making use of many logical operations. We compare the data structure/algorithm pairs on four measures: the time it takes to compute the separated components, the memory required to store the graph, the set of separators, and the set of components that the separator separate. This way we not only get a sense of which data structures are efficient time wise, we also get a sense of which data structures are efficient in storing typical vertex sets.

In a second experiment we compare the data structures when they are implemented in the Minimal Minimum Degree algorithm (MMD Algorithm) [5]. This algorithm computes a minimal triangulation of a graph in a greedy fashion, by making use of the computation of separated components. Note that obtain-

ing a tree decomposition from a triangulated graph is a trivial operation. We compare the data structures on the time it takes to complete the algorithm, the memory required to store the resulting triangulated graph, and an indication of the maximum memory required. We use this second experiment to observe how the data structures operate in a serious treewidth problem. Furthermore, we expect that it also tells us something about the predictive behavior of the Benchmark Experiment. The idea is that if the data structure performs well in the Benchmark Experiment, it is also likely to perform well in the MMD Experiment. More generally, such data structures are likely to perform well in algorithms that make use of component computations or have to store large amounts of vertex sets, such as the HBT algorithm.

Treewidth algorithms often operate on graphs of small treewidth. Since they run exponentially in treewidth, they would otherwise not lead to efficient algorithms. Graph of small treewidth are usually sparse and have a large clustering coefficient [18]. For this reason we will construct a benchmark for experimenting with treewidth algorithms. This benchmark consists of random partial 40-trees, which are graphs with treewidth at most 40.

This thesis starts with a literature study, which explains the context of our experiments. Then we describe the data structures, graphs, and experiments in more detail in the methodology. We continue with describing the outcome of the experiments and finally get to our conclusions and the discussion of our results.

2 Literature study

2.1 Treewidth

Treewidth is one of the most well studied graph parameters. The term was first introduced by Robertson and Seymour in their study on graph minors [32]. Though a number of equivalent definitions had already been shown to exist [8]. The most common and useful definition of treewidth is by a tree decomposition [32].

Definition 2.1 (Tree decomposition)**.** Let $G = (V, E)$ be a graph. A tree decomposition of G is a tree T, such that each node in T has a 'bag' associated with it containing one or more vertices in G, such that:

- Each $v \in V(G)$ is contained in at least one bag.
- For each edge $(v, w) \in E(G)$ there is at least one bag containing both v and w.
- All the bags containing a vertex form a connected subtree in T.

The width of a tree decomposition is the size of its largest bag minus one. The treewidth of a graph is the minimal width over all tree decompositions. For an example of a graph and a tree decomposition of width two, see Figure 1.

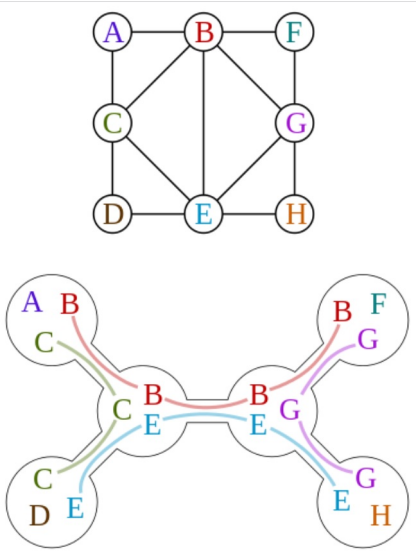

Figure 1: A graph (top) and an optimal tree decomposition (bottom) [1]

Other ways to define treewidth are based on graph triangulations, perfect elimination orderings, among many others. For details see [9] and [8].

Treewidth is such an effective graph parameter for FPT algorithms because it often enables efficient dynamic programming algorithms. These algorithms are of the following form: first compute a tree decomposition of the graph, then solve the problem using dynamic programming on the tree decomposition [4]. The reason this approach is successful comes from the fact that for each bag in the tree decomposition, the nodes corresponding to that bag form a separator in the original graph G.

Definition 2.2 (Separator)**.** Let G be a graph. Then a *separator* is a vertex set S, such that $G \setminus S$ is disconnected.

Definition 2.3 (Minimal separator)**.** Let G be a graph and S be a separator. We say that S is an *a-b separator* if vertices a and b are *separated* by S. This means that a and b are connected in G, but not in $G \setminus S$. A separator is *minimal* if and only if $\exists a, b \in G$ such that S is an a-b separator and $\forall s \in S$ we have that $S \setminus \{s\}$ does not separate a and b.

So if we remove all vertices in a bag B from G, denoted as $G \setminus B$, we get two separated subgraphs, also know as *connected components* and *separated components*. It is important to note that the open neighborhood of such a separated component is a minimal separator, as we will be computing these minimal separators in the MMD Algorithm. For problems which are FPT in treewidth, we can usually first solve the problem on these separated subgraphs and then combine the solutions with a function that only depends on the vertices in B.

To get a bit more familiar with treewidth, let's consider a tree, which actually is the same as a graph with treewidth 1. Many tree algorithms use the fact that a single node separates the graph in multiple subgraphs. Recursively combining the results of these subgraphs in the separating nodes, lead to bottom up dynamic programming algorithms.

Generally, when considering algorithms that are FPT in treewidth and we have a tree decomposition of treewidth k, this leads to algorithms that runs exponentially in the size of the bag and polynomially, and often even linearly, in the size of the graph. This leads to algorithms with worst case running time of $O^*(f(k))$, where $f(k)$ is some exponential function of treewidth k.

2.2 Treewidth algorithms

Computing treewidth has been shown to be a NP-complete problem [2]. So there is little hope of finding algorithms that run in polynomial time. Theoretically, the fastest algorithm found thus far run in time $O^*(1.7347^n)$ [20], and time linear in graph size and $k^{O(k^3)}$ in treewidth [7]. Experimentally, state of the art exact treewidth algorithms are able to solve problem instances up to size 1000 within about 100 seconds, as seen in the 2017 PACE challenge for exact treewidth [19].

Another field is the finding of good heuristic treewidth algorithms. These algorithms do not find optimal solutions, but instead look for a decent approximation and require lower running time. This approach is based on the idea that the time saved on constructing an optimal tree decomposition, could be spent more effectively on different parts of the problem. Broadly speaking, there are two main branches of heuristic treewidth algorithms [9]. The first approach constructs a perfect elimination ordering, often leading to greedy algorithms, while the second approach makes use of separators.

2.2.1 Elimination ordering

It is possible to triangulate a graph based on any ordering of the vertices [9]. The idea is that if we form each vertex into a clique with all its neighbors that are after it in the ordering, we get a triangulation of the graph. It has been shown that a triangulated - or chordal - graph has a treewidth of the size of the largest clique minus one [9]. This gives rise to algorithms that compute treewidth by constructing the so-called *elimination ordering* in some smart way, such that the corresponding triangulated graph has a small largest clique and thus a small treewidth.

Definition 2.4 (Chordal graph)**.** A graph G is chordal, or triangulated, if and only if every cycle in G of size ≥ 4 has a chord, connecting two non adjacent vertices of the cycle.

An effective approach is to make use of a greedy heuristic. This goes as follows. Recursively select a vertex from the graph, based on some criteria, then fill its neighborhood into a clique and remove the vertex from the graph.

Different vertex selection criteria have been proposed. Two examples are the Minimum Degree heuristic (MD) [30], which chooses a vertex of lowest degree, and the Minimum Fill In heuristic [24], which chooses a vertex such that a minimum amount of edges have to be added.

Other approaches make use of multiple variants of local search; see [15] for an example that uses Tabu search. The QuickBB algorithm searches the space of *perfect elimination orderings* by making use of a branch and bound scheme [22]. Note that a perfect elimination ordering indicates that it is the case that each vertex forms a clique with all its neighbors that are after it in the elimination ordering.

2.2.2 Separators

The second key approach makes use of separators. One approach finds minimal separators and forms them into a clique, this way building a triangulation of the graph [10]. The Minimum Separator Vertex Sets heuristic [25] uses a top down decomposition approach. It starts with a simple tree decomposition and recursively replaces the largest bag with smaller ones by making use of minimum separators in that bag. Ben Strasser's second place submission to the PACE 2017 heuristic treewidth competition is a recent example of such a top down approach [19]. It decomposes the graph with a graph bisection algorithm that makes use of maximum flow [33].

Another promising recent approach uses dynamic programming to recursively split the graph in components, separated by some separator, after which the tree decompositions of the components are glued together. This approach won the first place prize in the PACE 2017 competition [19]. An example of this approach is the heuristic version of [34], which iteratively improves an upper bound by considering a subset of the minimal separators. The HBT algorithm is another example [35], which heuristically uses the Bouchitté and Todinca algorithm [11].

Especially the algorithms that make use of dynamic programming have to store large amounts of vertex sets. These vertex sets range from very sparse, when representing separators, to very dense, when representing the separated components. When dealing with large graphs this gives rise to the need for efficient data structures to store them. As mentioned, this is one of the main reasons that we look into bitmap compression techniques.

2.2.3 Elimination ordering and separators

There are even algorithms that combine the two approaches. We will discuss one such approach we will be using in this thesis. The algorithm is called the Minimal Minimum Degree algorithm (MMD) [5]. It is based on the MD heuristic, described above, and gives a triangulation of the graph. The idea of the MMD algorithm is that we run the MD heuristic. But when we eliminate a vertex v, so when we fill the neighborhood of v into a clique and remove it from the graph, we do not fill the neighborhood into a clique. Instead we consider

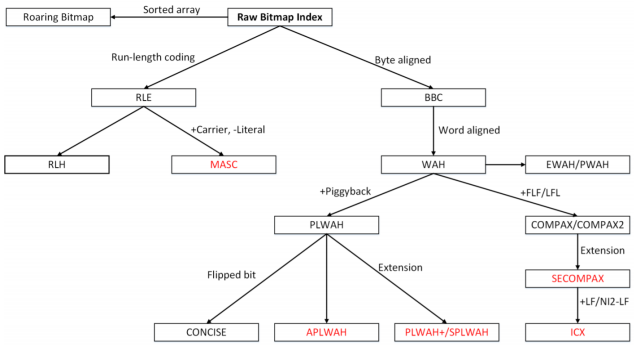

Figure 2: Bitmap compression techniques [14]

minimal separators of the graph inside the neighborhood of v and form only form them into cliques, before eliminating the vertex. It turns out that MMD adds fewer edges than MD to get to a triangulation.

2.3 Bitmap compression

Bitmaps, or bit sets, are a way to store information as a sequence of bits. They can be used to implement simple set data structures, for example sets of numbers, character traits, or in our case, vertex sets. The i-th bit of the bitmap is 'true' if the set contains the item or value corresponding to that bit. Otherwise the i-th bit is 'false'. Bitmap operations can be very efficient, since implementations can make use of parallel computing in the processor: 64-bit computers are able to perform 64 bitwise operations simultaneously.

Because bitmaps are an efficient way to store large amounts of information they have attracted attention in the field of databases. Databases need to store large amounts of information and support fast query operations. To improve memory consumption and speed of operations, numerous bitmap compression methods have been proposed to improve bitmaps. See Figure 2 for overview of different compression techniques and [38] and [14] for two surveys of the field. In what follows we briefly describe the main characteristics of bitmap compression techniques.

The approach first proposed is called run length encoding (RLE). RLE looks for runs of consecutive bits containing only ones or only zeros, and replaces them with a marker that indicates which value is being repeated, and how many repetitions there are. For example the sequence 1111000000, which contains four ones and then six zeros, can become 4160. See [3] for an example.

Most methods try to make use of the efficient parallel computing offered by bitmaps. The first to propose this technique used chunks of bytes for encoding, leading to Byte-aligned Bitmap Compression (BBC) [23]. Extending the chunks to 32 or 64 bits, however, quickly led to faster operations. This gave rise to Word-aligned bitmap compression techniques, such as WAH, EWAH, and

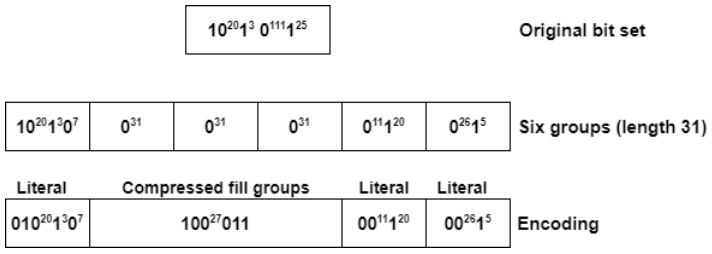

Figure 3: Example of WAH encoding. Top: the original bit set. Middle: the division into groups. Bottom: the encoding (three literals, one fill group compression).

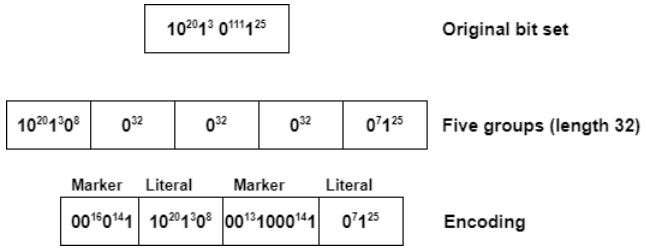

Figure 4: Example of EWAH encoding. Top: the original bit set. Middle: the division into groups. Bottom: the encoding (two marker words and two literal words).

CONCISE [40, 27, 16].

WAH stands for Word Aligned Hybrid [40]. It encodes sequences of 31 bits (63 bits for WAH 64) into groups. There are two types of groups. The first bit of a group indicates of which type it is. 0 indicates a 'literal group', meaning that the 31 bits are a mix of ones and zeros. Literal groups are not encoded. 1 indicates a 'fill group', meaning that the 31 bits are all ones or all zeros. The number of repeating fill groups is encoded as follows. The first bit is 1, indicating the fill group. The second bit indicates whether the bits are all one (1) or all zero (0). The next 30 bits indicate how many repeating fill groups we have. See Figure 3 for an example of WAH encoding.

Enhanced WAH [27], or EWAH, works in similar fashion. It divides an uncompressed bitmap into 32-bit (or 64-bit) groups. It encodes a sequence of p fill groups and q literal groups into one marker word followed by q literal words. The literal words are again stored in the original form. The marker word stores the following information. The first bit indicates the type of fill group (ones or zeros), bits 2-17 represent p, bits 18-32 represent q. An example of EWAH encoding is shown in Figure 4.

In 2016 a different approach was introduced, called Roaring Bitmap (Roaring) [13]. Roaring uses a hybrid data structure. It divides the original bitmap into chunks of size 2^{16} and distinguishes between dense and sparse chunks. The

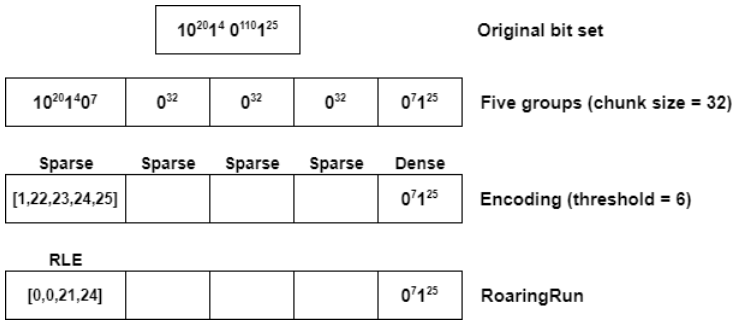

Figure 5: Example of Roaring encoding. Top: original bit set. Second row: division into chunks. Third row: Roaring. Bottom: RoaringRun.

threshold between dense and sparse lies at 2^{12} 1-bits. Dense chunks are encoded as the original bitmap. Sparse chunks indicate the location of the 1-bits as an array of Short numbers, by storing integer i for the i-th bit.

To reduce memory usage, Roaring has an option to include a third kind of chunk. These chunks are represented by RLE. If Roaring makes use of this option, called *run optimization*, chunks are transformed to RLE chunks if this reduces memory usage. Figure 5 shows an example with adjusted chunk size and threshold value. Note that RLE is only performed in the case that memory is reduced, in this case the amount of Short numbers that has to be stored is reduced by one.

3 Methodology

In this section we describe the methods we use. We start with describing the data structures in more detail. Secondly we explain how we construct the graphs we will perform the experiments on. Then we discuss the setup of the benchmark experiment and the MMD experiment. Finally we devote a couple of words to how we perform the measurements.

3.1 Data structures

The goal of our research is to look into data structures that can be used to represent vertex sets in graphs. As mentioned earlier, Wang et al. [38] found that Roaring Bitmaps performed best, followed by EWAH when they investigated the memory consumption and time efficiency for basic logical operations for a large number of bitmap compression techniques. Since we believe that these results are likely to translate to graph algorithms, we use Roaring Bitmaps and EWAH for our experiments. We compared them with two more commonly used vertex set representations, arrays of integers and a regular bitmap implementation.

In this section we discuss the implementation details of the data structures and say something about their memory consumption. At the end of this section

we say something about the functions used to optimize performance of the data structures and how we made sure that these data are compared in fair fashion.

3.1.1 Sorted arrays of Integer numbers

Sorted arrays of Integer numbers, from here on indicated with 'Ints', store a vertex set simply as a sorted set of the vertex IDs in an array of Integer numbers. Storing the third, fifth, and eighth vertex would be represented by the array $[3, 5, 8]$. Note that this means that the array has length three, so there are no unassigned array elements. In Java this is instantiated by 'new int[3]'.

The memory required to store Ints consists of some memory for the object, an array in this case, combined with 32 bits for each vertex in the set. Since the memory required to store a single vertex is relatively large, Ints are inefficient for storing large vertex sets. For example, when we want to store the full vertex set of a graph with size a million, Ints would have to store 32 million bits. As a comparison, a bitmap would only need a million bits, one bit for each vertex.

On the other hand, for small vertex sets Ints have the advantage over (compressed) bitmaps that it only has to store the vertices and does not have to allocate any memory for vertices not present in the vertex set. This makes Ints efficient in storing small vertex sets. For storing the empty vertex set, Ints would only have to allocate memory for the empty array. While a bitmap would still have to store a bit for each vertex in the graph, as well as the bitmap object. For a graph with size a million and leaving object overhead out of account, this would result in a difference of a million bits for a bitmap and 0 bits for Ints.

3.1.2 Uncompressed bitmaps

Uncompressed bitmaps - from here on simple denoted as bitmaps - store a vertex set as set of bits. Each vertex is allocated to a single bit. So a bitmap contains a number of bits equal to the number of vertices in the graph. Bitmaps excel at storing random data of medium density. However, for storing highly structured data, bitmaps are not an efficient data structure. For example, when considering a vertex set that consists of a single vertex, it is not efficient to store all vertices that are not present for a graph of size a million. In such cases Ints or bitmap compression techniques are expected to be more efficient.

The implementation for uncompressed bitmaps that we use is called XBitSet. This implementation is developed by Prof. Tamaki and extends Java's Bitset implementation. The reason that we use this implementation is due to the fact that a couple of algorithms that we use are based on algorithms provided by Prof. Tamaki which were implemented in XBitSet.

An XBitSet is stored as an array of Long numbers (64 bits). The memory used by this implementation is composed of the overhead to store the array and 64 bits for each long number.

3.1.3 EWAH

Of all the data structures under consideration, EWAH benefits the most from structure in the data. For data that does not contain stretches of ones or zeros that that are at least size 32 (or 64), it performs strictly worse than a regular bitmap. However, for data that contains large stretches of ones and zeros, EWAH can lead to significant improvements over bitmap.

We use the JavaEWAH implementation by Lemire et al. [29]. In this implementation EWAH is also stored as an array of Long numbers. So the blocks are of size 64. Note that there is also a version available for blocks of size 32. The authors note that this implementation is more efficient memory wise, but less efficient time wise. For this reason, we chose to work with the version in which blocks have a size of 64.

Similarly to bitmap the memory consumption is built up from the array and the Long numbers. An important thing to note is that EWAH uses dynamic memory allocation. It does not store the zeros that appear after the vertex with the largest ID. If a vertex with an even larger ID is added, the range of the EWAH data structure is extended to deal with this. All methods on EWAH run efficiently with this.

3.1.4 Roaring Bitmap

Roaring seems to be the data structure that is most widely applicable. Because it handles chunks of different density differently, it is able to handle both very sparse and dense graphs, as well as graphs of medium density. Roaring stores a vertex set as an array of objects. Each of these objects corresponds to a chunk. Depending on the density of the chunk, it is either a bitmap or an array of Short numbers. When we use RoaringRun, a chunk can also be represented as a run length encoding. Because Roaring stores the vertex set as an array of objects - each chunk is an object, it requires relatively a lot of memory for object overhead.

We use the implementation by Lemire et al. [28], which uses chunks of size 2^{16}. So each each chunk is able to store up to $2^{16} = 65.536$ vertices. If the chunk is stored as an array of Shorts or as RLE, the chunk is stored as an array of Short numbers. If the chunk is stored as a bitmap, it is stored as an array of Long numbers.

It is important to note that each chunk only stores values 0 to $2^{16} - 1$. The other information is deduced from the chunk the value is in. This means that if we would want to store the vertex with ID 100.000 in a chunk that is represented as an array of Shorts, the number 34.464 would be stored in the second chunk. The fact that this number resides in the second chunk provides the information that we have to add 2^{16} to 34.464 to get the vertex ID.

We also consider a number of special cases of Roaring. First of all, we considered Roaring both with and without run optimization. We indicate Roaring without run optimization with 'Roaring' and Roaring with run optimization

with 'RoaringRun'. We also experiment with Roaring using different chunk sizes. We use chunk sizes of 2^{10}, 2^{12}, 2^{14}, and 2^{16}. The corresponding implementations of Roaring are indicated with Roaring10, ..., Roaring16, and RoaringRun10, ..., RoaringRun16.

Note that in certain situations, regular Roaring can also make use of a chunk that is represented with RLE. This only happens when a chunk is completely full, so when all vertices are present in the set.

It was not straightforward to adapt Roaring to handle different chunk sizes. The code was hardwired to contain the numbers 16 and 2^{16}. We had to figure out which of these had to be adjusted and which had to be kept as they were. In total, adjustments were made to about six different classes.

After spending about four weeks on getting Roaring fully adjusted, we still did not manage to get all teething problems fixed. One large problem that still remains is that for RoaringX - where X is the variable that indicates the chunk length - we are not able to handle graphs that are larger than 2^{2X}. For example for Roaring10 $2^{20} = 1.048.576$. This is due to the fact that original Roaring is only meant to deal with Java unsigned integers, so its maximal reach is 2^{32}. This should be able to be improved up to at least graph sizes of 2^{16+X}, for RoaringX, since the amount of chunks that can be represented by RoaringX should not be influenced by X. Each type of Roaring should be able to contain 2^{16} chunks. However, we changed this value 16 to be equal to X and with the limited amount of time in mind we decided not to fully optimize this and continue with the experiments. Because of this Roaring10 is not able to the handle the largest graphs under consideration.

Another thing to note is that for Roaring16 the chunks that are represented by an array of Shorts or RLE are fully optimized. This is because the numbers are represented by Shorts and the maximum value that a number in such a chunk can represent corresponds to the maximum value of a Short (2^{16}). However, for Roaring10 the maximum number that has to be represented is 2^{10}, which is way less than the maximum number of a Short. So it structurally deals with quite a lot of unused stored memory. Perhaps a custom data type, which is able to represent numbers of up to 2^{10}, could result in a further improvement of Roaring with a small chunk size. Note that chunks that were represented with a bitmap did not see this problem, as the bitmaps were easily adopted to have the proper length (2^X for RoaringX). In any case it is important to keep in mind that in some respects Roaring16 is more optimized than the version with smaller chunk size.

3.1.5 Checks on data structures

Before we were able to start the experiments, we had to make sure that the data structures function properly and at their best performance. So we checked whether each of the data structures gave the same outcome for the experiments and we checked that binary operations between two sets gave the same outcomes. We also made sure that we use the most efficient way to iterate through vertex sets, since that is the action that the data structures have to perform frequently.

We verified that the data structures gave similar outcomes in the Benchmark Experiment by checking that the computed separated components coincided. For the MMD Experiment we checked that the number of edges that we added to construct the triangulated graph coincided.

For the second and third of these checks we constructed lists of random numbers of size 100 to 1.000.000. We tested the binary operations that we use in the experiments, which are AND, ANDNOT, and OR. We concluded that all data structures gave the same results after the logical operations of two sets.

For iterating through a set we did the following tests. For Ints we used a simple for loop. For XBitSet we tested the functions *for (i in set)* and a for loop in combination with *nextSetBit()*. These gave similar results, we used the second option. For EWAH we tested the for loop in combination with *getFirstSetBit()* and the iterators *IntIteratorImpl()*, *intIterator()*. Since the iterator *intIterator()* performed best, we used it. For Roaring we performed these test for Roaring16. We tested the for loop with the function *nextValue()*, the iterator *getIntIterator()*, and the function *forEach(new IntConsumer() public void accept(int v) ;)*. Since the iterator *getIntIterator()* performed best, we use it.

It is important to note that the above experiment was not meant to give decisive evidence to use one type of sets of iterator over the other. It was meant as a quick check of what performed best in this situation. Since the sets we are dealing with in the experiments are unordered as well, we expect them to give a decent indication of what works well. But if you plan on using the data structure yourself, we recommend performing your own experiments for deciding on a set iterator.

Finally we performed a preliminary check for Roaring, in which we made sure that the amount of chunks was correct. For example, for a graph of size 40,000, Roaring16 would have a single chunk, while Roaring14 would have three chunks. These checks succeeded.

3.2 Graphs

In this section we discuss the graphs that we use for our experiments. The most important trait of the graphs is that treewidth could be a useful tool on them. For this reason we chose sparse graphs. Initially, we wanted to use three different sets of graphs and perform the experiments on all three of them. These were the instances of the 2017 PACE heuristic treewidth track, random Erdős–Rényi graphs, and random partial k-trees. In the end we decided only to use random partial k-trees, which we will describe in section 3.2.1.

The Erdős–Rényi random graphs turned out not to be useful for our purposes, since they have a huge treewidth. When using the Minimum Degree algorithm the width of the computed tree decomposition turned out to be about a third of the graph size. This made it not only not representative for our intents and purposes, it also resulted in large computation times. For graphs of size 64,000, the Minimum Degree algorithm took about a day to compute a tree decomposition.

The instances of the 2017 PACE heuristic treewidth track are very relevant for comparing the performance of different treewidth algorithms. They consist of graphs from a range of problems where treewidth algorithms can be useful, such as SAT problems and road networks. All PACE 2017 graphs have been cleaned in the following way. Vertices of degree one are removed and for vertices of degree two its neighbors are connected and then it is removed. This is done because these subgraphs result in trivial partial tree decompositions. Finally only the largest connected component is kept. The graphs range in cardinality between 82 and 4.732.056 and in number of edges between 146 and 18.105.276.

Despite the fact that it would definitely be worthwhile to investigate the performance of the data structures on these graphs, we decided to go for partial k-trees, or PKTs. The reason for this is that PKTs are very consistent in the way they behave. This makes it easier to see trends in the behaviour of the data structures. It would be interesting to compare the results on the PKTs with the results on the PACE instances. However, computation time was an important factor. So we chose to value the fact that we could include larger graphs over the fact that we could compare the different graphs instances, in line with the goal of the project to consider huge graphs.

3.2.1 Partial k-trees

A k-tree is a graph that is constructed as follows. It starts as a complete graph of size $k+1$ and each additional vertex v that is added gets connected to each member of a clique in the graph of size k. A partial k-tree is a subgraph of a k-tree. K-trees have a treewidth of k. This can easily be shown, because the tree decomposition that consists of bags which correspond to the maximal cliques has a width of k. Partial k-trees have a treewidth less than or equal to k. For a treewidth of width k the same tree decomposition can be considered.

To construct the PKTs we first construct the backbone of the k-tree, which corresponds to the structure of how the maximal cliques are connected. This way, each vertex in the backbone represents a $k+1$ clique in the k-tree. To construct this backbone we create a spanning tree of a complete graph of size $n-k$ using Wilson's algorithm. Wilson's algorithm [39] recursively does the following: start at a random vertex that has not been visited yet and perform a random walk until it reaches a vertex that has been visited. Connecting these random walks results in a spanning tree of the graph. Remember that in our case we compute the random spanning tree for a complete graph.

We then basically connect the cliques according to the spanning tree, as if we create a k-tree, but we do this partially. This means that we only add a fraction of the edges. We start at the root of the tree with a clique of size $k+1$. To add a neighboring partial clique in the spanning tree we do the following. We add a new vertex to the graph and for each of the vertices in the neighboring partial clique we create an edge between that vertex and the new vertex with probability p.

We construct PKTs for the following parameters $k = 40$, $p = 0.075$, $n \in \{96, \lfloor 96 * 1.1 \rfloor, \lfloor \lfloor 96 * 1.1 \rfloor * 1.1 \rfloor, ..., 4.817.330\}$. We chose this value for k because

the treewidth is limited this way, but the graphs are still quite complex. This way they are still able to represent large variety of problems. However, for many treewidth computations, a treewidth of 40 can already be too large. So when using these benchmark graphs, it could be worthwhile to consider a lower value for k.

We then chose the value of p such that the graph turned out to be relatively sparse. We cleaned the PKTs similarly to the PACE instances, such that the graphs did not contain vertices of degree one or two and consisted of a single connected component. For large values of p we found that, after cleaning the graph, the amount of vertices that remained did not differ much. For this value of p we found a significant drop in the number of vertices that remained, indicating that many edges are crucial to sustain the cleaned graph. So for this value of p, the graph was in some way close to sparse as can be.

This resulted in graphs of size 77 up to 3.298.946, with the amount of edges being about a factor six larger than the number of vertices. Note that these graphs are smaller than the parameter n used for their construction, due to the cleaning of the graph. Also note that for both experiments we use a subset of the constructed partial k-trees.

3.3 Benchmark Experiment

In the Benchmark Experiment we try to mimic the most important operations in both HBT and MMD. In both algorithms the most heavy computations are to compute minimal separators in a separator. To do this they basically compute the components that the separator separates, these components are called separated components. The open neighborhood of such a separated component is the minimal separator that the algorithms use.

To mimic this operation we do the following. We start with a set of separators, which we obtain from the Minimum Degree heuristic. For these separators we compute the separated components. We can then measure the amount of time these computations take and the amount of memory is needed to store the graphs, separators, and separated components. This way we get a sense of which data structures are quick for this operation and which data structures are efficient for storing different types of vertex sets.

3.3.1 Separators

In this section we will describe how we obtain the set of separators to perform the Benchmark experiment with. To compute the set of separators we make use of the Minimum Degree heuristic (MD) [30], as introduced in section 2.1. We compute a tree decomposition T of the graph with MD. Then we know that the vertex set that corresponds to each of the bags in T is a separator of the graph, a general property of tree decompositions. We use these vertex sets as separators for our experiment.

As a quick recap, Minimum Degree heuristic recursively selects the vertex from the graph that has the lowest degree. The neighborhood of the selected

vertex is filled into a clique and then the vertex is removed from the graph. See Algorithm 1. Constructing the tree decomposition is straightforward: each eliminated vertex with its neighborhood at the time of deletion forms a bag.

In 2019 Cummings et al. [17] showed that an $O(nm)$ algorithm exists for the Minimum Degree heuristic. Our implementation, however, makes use of a more straightforward approach and runs in $O(n^3)$. It is based on an implementation of a different treewidth heuristic, Minimum Average Fill, provided by Prof. Tamaki. We changed the heuristic and did several adjustments to improve the running time and memory efficiency, such that it is able to effectively deal with large graphs. Note that we clearly specify which vertex we pick at each step of the algorithm, namely the one with least amount of neighbors and lowest ID. This allows for reproducibility of the experiment.

Algorithm 1: Minimum Degree heuristic

```
Result: An elimination ordering for graph G
G' ← G;
order ← order vertices by degree and ID;
while order is not empty do
    v ← the first vertex in order;
    remove v from order;
    add v to elimination ordering;
    fill N(v) into clique in G';
    remove v from G';
    foreach neighbor n ∈ N(v) do
        update order;
    end
end
```

The first adjustment that we made to improve performance was with respect to the data structure used to represent (partial) graphs. In the original implementation XBitSet was used as data structure. This led to problems with having the graph in memory. An adjacency array of bitmaps for a graph with a million vertices already needs to store $10^{2*6} = 10^{12}$ bits, or about 125 GB. So for our purposes we changed it to an Ints representation, in which each vertex has an array of Integers representing its neighbors.

The algorithms keeps track of the ordering, the *order* in which the vertices are to be eliminated. The data structure of the ordering needs to be efficient with respect to the fact that after each elimination the ordering of the neighboring vertices needs to be updated and the next vertex has to be selected. In the original implementation the ordering was represented by an unordered ArrayList of sets of vertices, in which the vertices were grouped by degree. This turned out to be quite slow for large graphs, because each time that a vertex has to be selected for elimination, the algorithm had to search for the lowest vertex ID in these unordered sets.

For the first attempt to improve the ordering we used an array of XBitSets. For each of the degrees that vertices had during run time we created an XBit-

Set, in which value '1' at position x of the y^{th} XBitSet indicated that vertex v_x had degree y. This implementation was quite fast. It did not have to search in unordered arrays anymore; instead, it could efficiently find the first bit with value '1', using the function 'nextSetBit()'. Updating the ordering was efficient as well, with the constant time 'get' and 'set' functions of the Java Bitset implementation.

When Erdős–Rényi random graphs were still part the plan, however, this approach led to memory problems. Because for these graphs the size of the largest bag of the tree decomposition typically turned out to be about a third of the graph size. So a graph of size $|V(G)| = 10^6$ would need an ordering consisting of, say, $3 * 10^5$ bitmaps of size 10^6 resulting in a need to store $3 * 10^{11}$ bits, or about 37 GB. We tried improving the memory requirements, by deleting the unused bitmaps every now and then, trading some speed for more efficient memory. This showed an improvement, but this was not enough to make this approach feasible for large Erdős–Rényi random graphs. It is interesting to note that for the PKTs, this would not have been a problem. For the largest PKTs the size of the largest bags turned out to be around 0.1% to 1% of the graph size, resulting in an ordering of a size of at most about 13 GB.

Because at the time we still wanted to use random graphs as well, we implemented a third version of the ordering, trading in some speed for better memory efficiency. We used a sorted array of arrays of Integers, where the i^{th} array contains all vertex IDs with degree i. Obtaining the first ranked vertex of a specific degree becomes a simple lookup, since it's the first element of the array.

In the experiment we take a selection of the separators that we obtained. We perform 15 runs and in each run we take 20 different separators, randomly selected without repetition. Note that this means that for the smallest graphs it turns out to be the case that we use some separators multiple times, thus these results are less accurate compared to the results for the larger graphs. However, since the focus of the thesis is on huge graphs this would not pose a large problem.

3.3.2 Algorithms

After having obtained the separators, we can move on to describing the algorithms that we use in the Benchmark Experiment. The algorithms compute the separated components that are the result of removing the separator from the graph. Or in other words: suppose we remove the vertices in the separator from the graph, which mutually disconnected subgraphs do we end up with? This means that we have to search the entire graph to determine which vertex belongs to which separated components.

We want to compare the data structures at their best performance. To do this we search the space using two different algorithms. The first algorithm that we use is a simple Breadth First Search algorithm (BFS), which keeps track of which vertices have been visited and to which separated component they belong. The second algorithm does something similar, but instead of keeping track of the vertices that have been visited, it constructs the separated components

with logical operations. It basically takes the union of the partial separated component with the neighborhoods of the vertices it encounters. This algorithm is denoted as the Parallel Algorithm. First we discuss BFS in a bit more detail, then the Parallel Algorithm.

BFS keeps track of which vertices have been visited and from which vertex they were reached. It does this by labeling the vertices: vertices that belong to the same separated components get the same label. This labeling algorithm is shown in Algorithm 2. Note that BFS has a worst case running time of $O(m)$, since each edge has to be visited by the algorithm. We implemented the queue as a Java LinkedList of Integers. Once we have this labeling, constructing the separated components is straightforward. We construct the vertex sets, consisting of all the vertices with the same labeling.

Algorithm 2: Breadth First Search Algorithm

```
Result: A labeling of the vertices, such that the vertices that belong to
        the same separated component have the same label.
mark all vertices with 0;
mark all vertices in the separator with −1;
m ← 1;
foreach vertex v that has not been marked do
    mark v with m;
    add v to queue;
    while queue is not empty do
        w ← queue.poll();
        foreach Vertex x in N(w) do
            if x is marked with 0 then
                mark x with m;
                add x to queue;
            end
        end
    end
    m ← m + 1;
end
```

The Parallel Algorithm does not take the intermediate step of computing a labeling for each vertex. Instead it computes the components directly. The algorithm is shown in Algorithm 3. Remember that $N[v]$ indicates the closed the neighborhood of v, meaning that v is included in the set. $N(v)$ indicates the open neighborhood of v, meaning that v is not included in the set.

Note that the Parallel Algorithm is built upon logical operations, such as the union between sets and the subtraction of one set from another. Since this is different compared to BFS, which mainly relies on iterating through the vertex sets, we expect different results.

The worst case running time of the Parallel Algorithm turns out to depend on the data structure. Each vertex has to be 'scanned' and for each vertex the union has to be computed with 'comp'. This operation can be performed in

$O(n_{small})$ time for XBitSet, where n_{small} is the cardinality of the smallest of the two sets, since setting each bit is a constant time operation. So for XBitSet we have that for each vertex we have to perform an amount of operations of at most the amount its neighbors, so the algorithms runs in $O(m)$.

For Ints taking the union between two sets is an $O(n_1+n_2)$ operation, where n_1 and n_2 are the lengths of the sets. So for Ints the algorithm runs in $O(n^2)$.

For Roaring the time complexity is a bit more complicated, since different chunks are represented differently. The union of two bitmap chunks is similar to the XBitSet case. The union of a bitmap chunk and an array of Integers chunk is also fast, since we can iterate over the array and simply set the values we encounter to true in the bitmap chunk. The union of two array of Integers chunks is the same as for Ints, except that the maximum cardinality of the outcome is limited. For RoaringX, where X is variable, the outcome cardinality can be no larger than 2^{X-4}, so for Roaring16 this would be 2^{12}. Because of this last term we have a running time of $O(n^2)$ for Roaring. This results also holds when we include chunks that are represented with RLE, since they can be transposed to either a bitmap or array of Integers chunk in $O(n)$ time.

For EWAH the union operations are not straightforward as well. The compression has to be interpreted by reading the marker words. Some parts can be easily skipped over, for example when one of the sets contains stretches of fill words (long stretches of ones or zeros). Other parts require computation similar to that of regular bitmaps, when two literal words (which contain both ones and zeros) have to be combined. At the end the new set has to be compressed. Since the compression has a worst case scenario of $O(n)$ the worst case running time for EWAH is $O(n^2)$ as well.

Algorithm 3: Parallel Algorithm

```
Result: the separated components (comps), which are separated by
        separator S.
foreach vertex v that has not been visited do
    comp ← N[v];
    toScan ← N(v) \ S;
    while toScan is not empty do
        compOld ← comp;
        foreach Vertex w in toScan do
            comp ← comp ⋃ N(w);
        end
        toScan ← comp \ compOld \ S;
    end
    comp ← comp \ S;
    mark vertices in comp as visited;
end
```

We made sure that both algorithms gave the same results for each data structure. We did this by running the algorithms for each data structure, casting the outcomes to Strings, and verifying equality of the Strings.

Initially, we wanted to include a third algorithm to the experiment, namely the Depth First Search algorithm (DFS). However, we decided not to do this for two reasons. First of all, we found that the running time of DFS was roughly equal to the running time of BFS for each of the data structures. This makes sense because both algorithms basically iterate through the neighborhood of each vertex, so the basic operations are similar. Secondly, we ran into a problem for using Roaring on large graphs. The need to keep all the iterators in memory turned out to be too much, resulting in stack overflow errors. The problem was that for some separators one of the separated components comprises almost the entire graph. Thus keeping an iterator for each vertex in the separated component in memory is about equal to keeping the entire graph in memory. And keeping the entire graph in memory twice turned out to be too much when running the experiments on my laptop with 8 GB RAM. Because we wanted to be able to perform preliminary experiments on my laptop, combined with the fact that using both BFS and DFS did not contribute a lot to the research, we decided to only use BFS.

Note that there are many options for different algorithms to consider. One could for example investigate the behavior of an algorithm that combines elements of BFS and the Parallel Algorithm, that uses BFS but does not label the vertices but instead performs the union operation to construct the components. And many other algorithms can be considered. However, we decided to go with these two algorithms. BFS is widely used and the Parallel Algorithm turned out to be most efficient for the operation for small to medium-sized graphs using XBitSet.

3.4 MMD Experiment

The goal of the experiments on the benchmark is to be able to quantitatively compare performance of different data structures. To investigate whether these results translate to real world algorithms we also implement the data structures in the Minimal Minimum Degree algorithm [5].

Initially we did not want to perform the experiments on MMD, but instead we wanted to work with the HBT algorithm. There are two reasons we chose not to pursue this path. The first is that HBT has difficulty scaling to large graphs. Graphs of size 100.000 already seem to lie out of reach for HBT. Because we want to investigate the behavior of data structures for huge graphs HBT did not seem suited. We could try to improve HBT up to a point where it would be able to handle huge graphs. However, we would have no guarantee that this would succeed. The second reason was that implementing the data structures in HBT would be quite time consuming.

Both these problems seemed to be more manageable for MMD. At the same time, the core operations of MMD and HBT resemble each other closely, since the bottleneck for both algorithms is that they have to computes the open neighborhoods of separated components. Because of this MMD seemed like a decent replacement algorithm for HBT, so we decided to run the second experiment with MMD.

3.4.1 MMD

The MMD algorithm is closely linked to the Minimum Degree Algorithm, which is shown in Algorithm 1. MMD gives a triangulation of the graph in question. At each iteration it eliminates a vertex and forms its so-called *substars* into cliques, see Definition 3.2. A key difference with MD is that MMD produces a *minimal triangulation*. Also, it introduces at most the same amount of fill edges, because it computes a minimal triangulation which is a subgraph of the triangulation computed by MD [5].

Definition 3.1 (Minimal triangulation)**.** Let G be a graph. A *triangulation* G' of G is a graph, such that G' includes all vertices and edges in G and has enough added edges such that G' is chordal. We say that a graph G' is a *minimal triangulation* if and only if removing any of the added edges from G' leads to the fact that G' is no longer chordal.

MMD is shown in Algorithm 4. Note that we keep track of two different fill graphs. In the *elimination graph* G' the neighborhood of vertices are filled into a clique when they are eliminated. This is basically the same graph that MD works with. Note that we select the vertex to eliminate next from G'. In *fill graph* H we construct the triangulation. In H we do not form the entire neighborhoods in cliques, but instead the computed substars. It is important to note that the substars that are formed into cliques in H are computed in G'.

Definition 3.2 (Substars)**.** Suppose we are eliminating vertex v in fill graph G'. For each separated component C of $G' \setminus N[v]$, we have that the open neighborhood of C is a substar of v.

If it is the case that for a vertex in H, each of its substars is a clique, then vertex is called *LB-Simplicial*. If all the vertices in a graph are LB-Simplicial, the graph is triangulated, or chordal [26]. Remember that a chordal graph has treewidth of width equal to its largest clique minus one. Constructing a tree decomposition of G from this chordal graph H turns out to be trivial, if we simply take all maximal cliques in H as bags.

Algorithm 4: Minimal Minimum Degree Algorithm (MMD)

Result: a minimal triangulation H of graph G.
$G' \leftarrow G$;
run MD on G' - whenever a vertex v is eliminated, compute the substars of v in G' and form them into cliques in H;
while *H is not chordal* **do**
  $G' \leftarrow H$;
  remove all LB-Simplicial vertices from G';
  run MD on G' - whenever a vertex v is eliminated, compute the substars of v in G' and form them into cliques in H;
end

Note that in some cases a second round of the algorithm is needed to obtain a minimal triangulation, in our experiments we observed that this is the case for

large graphs. If this is the case, we let fill graph G' be equal to H and remove all the LB-Simplicial vertices from G'. This means that in the second round the algorithm deals with a graph that is much smaller than the original graph. Also note that in practice the steps of checking whether H is chordal and removing the LB-Simplicial vertices of G' are not performed consecutively, as it might seem. Instead these steps are performed simultaneously. This way we have to check whether the vertices are LB-Simplicial only once.

MMD differs from MD in three main aspects. Firstly, in what happens when a vertex is eliminated. Secondly, in the fact that for large graphs it often needs more than one iteration of the algorithm to terminate. Finally, in the fact that MMD is guaranteed to give a minimal triangulation, while MD does not.

The authors of MMD have shown that MMD adds at most as many fill edges as MD and they observed that the relative improvement with respect to MD is small. In our experiments we found similar results. MMD never added more fill edges than MD. We also found that for small graphs MMD does not lead to an improvement. For large graphs it does lead to an improvement of up to 0.3%, where larger graphs see a larger improvement. Since MMD is way more time consuming than MD and the improvement in the number of fill edges is small, we believe that MMD is unlikely to find application in practice. However, the authors of MMD note that it can be useful in helping to further understand why MD gives triangulations, which are so close to minimal.

At the latest hour, we unfortunately found a bug in our implementation of MMD: MMD does not always terminate, while it is supposed to. We were not able to correct this mistake, because we discovered it too late, due to the fact that earlier we wrongly believed that MMD is not guaranteed to terminate. While running the experiments, we found that the algorithm did not terminate twice: once for the largest graph size and once for the second largest graph size. We dealt with this by not including these problem instances to our results. So in the end we only used graphs for which our implementation did terminate.

Fortunately, the bug is small and does not influence the results. It turned out that when the algorithm did not terminate, it was stuck in an infinite loop with a couple of vertices that had no neighbors in elimination graph G'. The problem is that our implementation handles these vertices incorrectly: it considers them to be not LB-Simplicial, while they clearly are. This means that this bug can easily be fixed and that for the cases that the algorithm did terminate, the bug does not seem to have an impact on the correctness of MMD. So we do not have any indication that our results are in some way incorrect.

Note that we expect the first experiment to provide good predictions for the second experiment. We believe this because the separators that we use in the first experiment are quite similar to the separators in the second experiment, since both have a strong connection to the MD algorithm. In the part of MD in which we eliminate vertices, the separators are exactly the same. Only in the part in which we decide whether the graph is chordal we find that the separators are somewhat different, though we still believe that they are quite similar.

In the following section we will suggest a couple of improvements to MMD. Of course, these improvements will lead to a larger difference between performance

of the data structures between the first and second experiment. However, we believe that since the basic operations are still relatively the same, this should not be too large of a bother. In the results section we will see how this plays out.

3.4.2 Improvements to MMD

A straightforward implementation of MMD turns out to be quite costly time wise. It turns out that the main time-consuming operations are the substar computations. Note that these operations are performed twice in the algorithm, so they have to be computed twice for each vertex. First, when computing the substars in elimination graph G', then when checking whether graph H is chordal.

Checking whether H is chordal turns out to be the most time consuming. This makes sense, because the separated components that need to be computed are on average larger than is the case in G'. This is due to the fact that vertices that are already eliminated in G' are no longer included in separated components of vertices that follow.

These time heavy computations run in time $O(n^2m)$, since for each vertex we have to search through each edge and in the worst case scenario we have to perform n rounds, eliminating only a single vertex in each round. We will later see that this term dominates the running time of MMD in our experiments. Note that $O(n^2m)$ is larger than the worst case running time of MD, which is $O(n^3)$.

Since the goal of this thesis is to compare the data structures on large graph instances, we made an effort to improve MMD such that it is able to handle larger graphs. In the end we made three noteworthy heuristical improvements to the substar computations, which we will describe below.

3.4.3 Shallow BFS

The first improvement tries whether it is possible not having to search the entire separated components to determine the substars. To do this it performs a BFS up to a certain depth in the neighborhood of v. We called this method Shallow BFS.

Suppose we have a vertex v and we want to compute the substars of v. Shallow BFS starts at a vertex v_0 in the second neighborhood of the vertex in question - see Definition 3.3 - and searches the component separated by the neighborhood of v, to which v_0 belongs. If all the neighbors of v are found in the BFS, we know that there is a separated component such that the neighborhood of this component contains all the neighbors of v. This means that all neighbors of v form a substar, which means that in the algorithm all neighbors of v have to be formed into a clique. If this is the case, we do not have to search the entire separated component, so we can save computation time.

Definition 3.3 (Second neighborhood)**.** Given a vertex v. The second neighborhood of v is the set of vertices, such that for each vertex w in the set, there

exists a path from w to v of length 2, while no such path exists of length smaller than 2.

Note that it is important that the search does not start in a vertex in $N[v]$. It could be the case that this vertex is part of multiple substars. If this is the case and we would start in such a vertex, we would be searching two different separated components, so we would not be able to conclude on the fact that the entire neighborhood of v is in a single substar.

When implementing this method we performed some small experiments on the best depth of the Shallow BFS and found that for our purposes the depth of five suited best. This turned out to be the best trade-off between computation time spent in the shallow BFS and success rate of method. Here the success rate is defined as the fraction of times in which we found that the neighborhood of v is in a single substar.

In practice we found significant improvements. In the part of the algorithm in which we eliminate the vertices, this method succeeds in about 96% of cases. When we check whether H is chordal, the method succeeds in about 66% of cases. We found that both of these percentages are independent of graph size.

3.4.4 Checking for Cliques

After having implemented Shallow BFS, we came to the following insight. When Shallow BFS succeeds in the section of the algorithm in which H is chordal, all neighbors of the vertex are reached and they form a clique. Every time that this is the case we know that the entire neighborhood of v is a clique. So instead of doing the Shallow BFS, we can also simply check if all neighbors of v form a clique. We call this method Checking For Cliques.

It turns out that checking whether $N(v)$ is a clique is a strict improvement over Shallow BFS. Since it captures all the cases that Shallow BFS captures and it includes cases that Shallow BFS does not capture. When, for example, Shallow BFS does not have a large enough depth or when it starts to searching in the wrong separated component, it could fail to come to the conclusion that the entire neighborhood of v is a clique. Checking For Cliques turns out to succeed in about 67% of the cases. Note that it does not only capture more cases, it is also an large improvement with respect to time consumption, since performing Shallow BFS is relatively costly.

So we implement Shallow BFS in the part of the algorithm in which we eliminate vertices, and we implement Checking for Cliques in the part of the algorithm in which we checked whether H is chordal. We see the improvements in performance with respect to the original algorithm in Table 1. MMD0 indicates MMD without improvements and MMD1 indicates MMD with adding Shallow BFS and Checking for Cliques.

There are two important things to note. First of all that these are preliminary experiments. To obtain these results we performed a single run of the algorithm. This means that these outcomes are not meant to give solid indications of how well each implementation performs at a certain graph size. More

Graph size	MMD0	MMD1	MMD2
682	278	158	157
1.003	324	181	171
1.474	633	364	433
2.133	1.509	419	422
3.183	2.337	926	923
4.563	4.330	1.525	1.597
6.721	10.529	4.533	3.304
9.886	23.115	6.935	6.361
14.426	58.755	17.274	13.370
21.087	146.815	39.164	21.832
30.857	372.037	90.518	59.606
45.365	891.912	209.955	158.249
66.137	2.238.599	526.406	294.061
97.128	NA	1.297.731	546.222
142.175	NA	3.128.211	726.473
207.543	NA	7.463.027	1.878.815
304.512	NA	14.808.751	3.991.300
445.092	NA	33.811.409	5.042.758

Table 1: The computation time (ms) of different implementations of MMD with respect to the graph size (number of vertices). MMD0 uses none of the improvements described above. MMD1 uses of Shallow BFS and Checking for Cliques. MMD2 uses Shallow BFS, Checking for Cliques, and Separating Substars. Note that NA indicates that we did not perform those experiments due to the large computation time.

experiments are needed for more definite conclusions on this. However, since the difference between the implementations are large, we can make conclusions for which implementation to use in our experiments. Secondly, we only compare MMD0 and MMD1 up to a graph size of 66.137. This gives a clear enough indication that MMD1 outperforms MMD0. So we conclude Shallow BFS and Checking for Cliques are improvements to MMD.

3.4.5 Separating Substars

After implementing Shallow BFS and Checking for Cliques, we found that the large majority of computation time - about 90% - took place in the part of the algorithm which decides whether fill graph H is chordal. More specifically, in performing the substar computation such that the entire separated components has to be searched. The last improvement we made focuses on improving this part of the algorithm.

During the algorithm we are constantly computing minimal separators, namely the substars. The main idea of the method, called Separating Substars, is that

we make use of a subset of these separators, in such a way that we do not have to search the separated components in their entirety. The set of Separating Substars is defined as follows.

Definition 3.4 (Set of Separating Substars)**.** Let H be a graph and A be the set of substars computed up to that point, in the process of determining whether H is chordal. Then, a *set of Separating Substars* is a subset of A.

We make use of these Separating Substars in the following manner. Suppose we are computing a separated component C_v for vertex v and we encounter a Separating Substar S_w, that we computed earlier for vertex w. We can use the fact that S_w is a graph separator, to deduce that we can reach all the vertices that lie behind S_w in C_v.

To make sure that this always holds, we have to pose a restriction on the Separating Substars: they cannot overlap with the neighborhood of the vertex v. In other words: S_w cannot overlap with $N[v]$. With this restriction in place, we know that S_w lies completely inside C_v. Since we are able to reach each vertex in C_v with regular component computation, we know now that we are able to reach all the vertices in S_w and all the vertices that lie behind S_w. This means that, as soon as we reach S_w, we no longer have to compute the vertices in S_w and behind S_w. So we only have to continue the computation of the separated component on the side of S_w, we reached it from.

When we implemented the method of Separating Substars, we had to make sure that we knew from which side we reach them. Again, suppose we have a vertex v and we are computing a substar S_v, with a connected component C_v associated with it. If we decide to make S_v a Separating Substar, we compute and store the two neighborhood sets of S_v: $N_1 = N(S_v) \bigcap C_v$ and $N_2 = V_H \setminus (C_v \bigcup S_v)$. Here V_H denotes the set of all vertices of graph H. At a later stage in the algorithm, when we are computing a substar for a different vertex and we encounter a vertex s that is in S_v, we figure out whether we reached s from N_1 or N_2. Suppose, without loss of generality, that we reached s from a vertex in N_1. Then we continue searching in each vertex of N_1 and do not continue searching in any of the vertices in S_v. This way, we do not reach any vertices that lie behind S_v. An important detail is that this also guaranties that we reach every vertex that lies on the side of S_v, we reach it from. So no vertices are missed which could be a part of the substar we are computing.

The idea is shown in Figure 6, where we compare MMD with and without making use of Separating Substars. Note that when using Separating Substars (bottom) we only need to search through C_1 and N_1, which is less than the entire separated component we need to search through when we do not use Separating Substars (top).

When implementing this method, there are a couple decisions one has to make. The main problem of interest was which Separating Substars to use. The most important decision we had to make is whether to consider a fixed set of Separating Substars, irrespective of the vertex under consideration, or let the

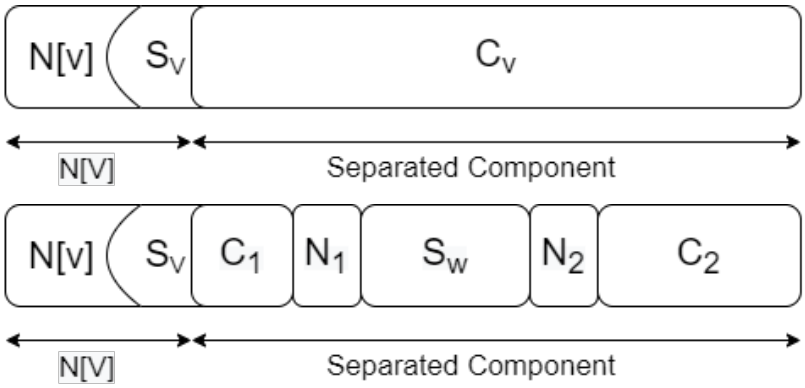

Figure 6: MMD with Separating Substars. S_v is the substar we are trying to compute. In the top figure, C_v is the corresponding separated component. In the bottom, S_w indicates a Separating Substar, with correspondingly N_1 and N_2, and C_1 and C_2 indicate regular parts of the separated component.

set of Separating Substars depend on the vertex. The first of these options came to mind first, so we went with that. However, the second option is definitely worthwhile to investigate further. We will first describe the implementation that we used for the experiments and then devote a couple of words to the second option.

When considering a fixed set of Separating Substars, an important decisions that we had to make was whether to make room for Separating Substars to overlap with each other. The upside of allowing some kind of overlap is that it could lead to a smaller portion of the graph which has to be searched, since more Separating Substars can be used simultaneously. The downside would be that we would have to decide what would happen if we would reach overlapping Separating Substars. Since this is not straightforward we decided to go for the option of not letting Separating Substars overlap. Though it could definitely be interesting further research to investigate whether it could be worthwhile to let them overlap.

So we decided to aim for a fixed set of Separating Substars, which cannot overlap. Now the question becomes which substars do we choose to become Separating Substars. It turns out that this is quite important question, since we found that in our experiments the amounts of Separating Substars were quite small and the computation time depended relatively strongly on the selection of Separating Substars. In the final configuration we found that for graphs up to size 7000 we only had a single Separating Substar, graphs of size 20,000 had about four, graphs of size a hundred thousand about nine, and graphs of size 300.000 about 28. Though note that we optimized the choices of substars for graphs of size 50.000.

The reason that we turned out to have so few Separating Substars is due to the fact that they cannot overlap. We found that for our experiments the

Separating Substars have a typical size of about 40. Since the graphs that we consider are partial 40-trees, this comes down to about the size of the treewidth of the graphs. This hypothesises a relation between the number of Separating Substars that can be used and the treewidth of the graph. Perhaps this method turns out to be more successful for graphs of even lower treewidth. It would be interesting to observe its behaviour on, for example, random partial 10-trees.

Because of the limited amount of Separating Substars that we could use, we spent quite some effort on optimizing the substars chosen to become Separating Substars. First of all we decided not to use the substars which are a clique with the entire neighborhood of the vertex in question. We did not have any information available about these substars, except that they were a clique with the neighborhood of the vertex in question. So it would require extra computation to tell whether these substars could make good Separating Substars. Also, these substars turned out to have a relatively large probability to reside at the edge of the graph, so they don't separate that much. So we decided to use substars that we encountered when the neighborhood of the vertex was not a clique; when we had to perform the full computations to determine whether the vertex was LB-Simplicial.

We found that we obtained good results if we chose the substars in such a way that they separate a decent portion of the graph. We did some preliminary experimentation for graphs of size up to 50.000 to decide what 'a decent portion' meant in this case. We found that results were best when we did not compare the amount of vertices that the substar separates with the total amount of vertices in the graph. Instead, it gave better results to compare with the amount of vertices encountered in the combined substar computations of the vertex in question. This way we try to consider vertices which lie somewhat neatly in between two other separators.

When performing experiments we found that the algorithm performed best when the substar separates between 4% and 90% of the vertices, that we encountered in the combined substar computations of that vertex. It is important to note that these results are based on preliminary research and are not conclusive. We did not fully optimize this and differences were quite small. So when implementing this approach it could be worthwhile to further optimize this. Note that we did not expect that the selection of substars to become Separating Substars would have a large influence on the relative performance of the data structures, so there was no need for far-reaching optimization.

Using Separating Substars gave a significant speed up, as can be seen in Table 1. Note, first of all, that MMD2 outperforms MMD1. Especially for large graphs the difference becomes quite large. So making use of Separating Substars seems to be an improvement for the algorithm. It is also interesting to note the manner in which the implementations scale. Both MMD0 and MMD1 seem to scale quite consistently. When comparing consecutive graph sizes the factor in which the computation time is multiplied seems relatively constant. However, for MMD2 this scaling behavior appears to be less consistent. Sometimes the computation time grows a lot and sometimes it grows a little. This has to do with the fact that the computation time depends on the Separating Substars used. This

means that we can expect a relatively large variance in the computation time when performing the experiments.

In our experiments we apply all suggested improvements. Shallow BFS, Checking for Cliques, and Separating Substars. Note however, that we only apply Separating Substars in the first round of the algorithm. Because in the second round we have to deal with a low number of vertices and for a low number of vertices applying separating substars does not lead to an improvement, as can be seen in Table 1.

3.4.6 Separating Substars that depend on the vertex

As mentioned earlier, we also discovered a different approach that might be fruitful. Instead of working with a fixed set of Separating Substars, let the set of Separating Substars depend on the vertex under consideration. There are multiple approaches to consider. The approach that we investigated was to use a tree decomposition of some kind. Note that we did not implement these methods. They are mentioned for possible future exploration.

The most obvious candidate tree decomposition is to use the tree decomposition provided by MD, in the part of the algorithm in which vertices are eliminated. The upside of using this tree decomposition is that we have already constructed it in the MMD Algorithm. Note, however, that it could also be worthwhile to investigate different tree decompositions.

In the tree decomposition that MD provides, bags consists of a vertex and its neighborhood at the time of eliminating v. It is important to understand that this tree decomposition is a tree decomposition of H. MD computes a tree decomposition for both G and the fill graph G'. Since the edges of H are a subset of the edges in G', we know that the tree decomposition is also a tree decomposition of H.

We can use the MD tree decomposition to limit the area of H that has to be searched, based on the idea that each bag in the tree decomposition is a separator of H. Similarly to in the description of the Separating Substars, we can use the fact that we encounter a separator to know that we can reach all vertices that find themselves behind it. So we can basically stop searching whenever we reach a bag that is sufficiently far away from the vertex under consideration.

In the tree decomposition, we consider the first bags that do not contain any of the vertices in $N[v]$, as the separators. Since these separators do not overlap with $N[v]$, we have a similar situation as in our implementation. Since we know that the separators lie completely inside the separated components, we know that all vertices in the separated components can be reached and that we do not miss out on any vertices in the substars.

We can use that fact that we know beforehand which separator we will encounter, to implement this technique in a smart way. Instead of considering the entire graph and stop searching further beyond the separators, as we did with the Separating Substars, we could try the following. We could create

the subgraph, induced by the bags that contain vertices in $N[v]$, together with the bags that we use as separators. Note that it is important that we form the separators into cliques. This way, when we search through the induced subgraph, we make sure that we do not miss out on any relevant vertices; all vertices in the substar can be reached by performing the component computation on the induced subgraph.

The second approach is a bit more ambitious. The idea is that it further limits the induced subgraph that has to be searched, to only include the vertices of those bags which contain vertex v. It is based on the following result.

Proposition 1. Let TD be a tree decomposition of graph H such that the intersection of every pair of adjacent bags is a clique. Let v be the vertex under consideration, B_v the set of bags in TD containing v, and U_v the union of the bags in B_v. Then: v is LB-Simplicial in H if and only if v is LB-simplicial in $H[U_v]$.

To prove this result we first suppose that v is LB-simplicial in H. We need to show that v is LB-Simplicial in $H[U_v]$. Let C be a connected component of $H[U \setminus N_H(v)]$ and C' the connected component of $H \setminus N_H(v)$ that contains C. If C is empty we are done, then there is no neighborhood that has to be a clique. So suppose C is not empty. Let w be a vertex in $N_H(C')$. We need to show that w is adjacent to a vertex in C.

Suppose w is not adjacent to a vertex in C, but instead to a vertex x in $C' \setminus C$. Let D be the connected component of $C' \setminus C$ to which x belongs. Since C' is connected, we have that $N_H(D) \cap C$ is non-empty; let y be a vertex in this intersection. Since $N_H(D)$ is a clique (see next paragraph) and contains w and y, w is adjacent to y. Therefore w belongs to $N_{H(U)}(C)$. Therefore, we have $N_H(C') = N_{H[U]}(C)$. Since v is LB-simplicial in H, $N_{H[U]}(C) = N_H(C')$ is a clique in H and hence in $H[U]$. This shows that v is LB-simplicial in $H[U]$.

Proof that $N_H(D)$ is a clique: B_v forms a subtree of TD. Let TT be the set of subtrees of TD that result from removing B_v from TD. For each component D of $G \setminus U$, there is a subtree T in TT such that no other subtree in TT contains a bag intersecting D. Let X be a bag of T and Y a bag in B_v such that X and Y are adjacent to each other in TD. Then, we have $N_G(D) \subseteq X \cap Y$. And, since $X \cap Y$ is a clique, $N_G(D)$ is a clique.

Suppose now that v is LB-simplicial in $H[U]$. We need to show that v is LB-Simplicial in H. Let C be a connected component of $H \setminus N_H(v)$ and let $C' = C \cap U$. If C' is empty, then C is a connected component of $H \setminus U$ and hence $N_H(C)$ is a clique by the property of TD. If C' is non-empty, then we have $N_H(C) = N_{H[U]}(C')$ by an argument similar to the above and hence $N_H(C)$ is a clique from the assumption that v is LB-simplicial in $H[U]$. Therefore, v is LB-simplicial in H and that concludes the proof.

It is important to note, however, that having a tree decomposition, such that the intersection of every pair of adjacent bags is a clique is not trivial. A tree decomposition probably has to be computed specifically for this purpose.

However, it could be worthwhile to investigate this option. If such a tree decomposition is computed easily, it could lead to improvements, since the amount of vertices that has to be searched to determine whether a vertex is LB-Simplicial is relatively small.

3.5 Performing the measurements

In this section we describe the way in which we perform the time and memory measurements, but first we will say something about the hardware and software used to perform the experiments.

3.5.1 Hardware and software

We perform the experiments on the Gemini server of Utrecht University, which runs Scientific Linux 7.9 on DELL PowerEdge R730 (2x) and R640 (3x). For our experiments we used 20 GB of RAM. As we will later see that this only limited XBitSet. The other data structures performed well with this amount of memory available. As noted earlier we implemented the experiments in Java. To perform the experiments the server used Java's 'openjdk version 1.8.0_292'.

3.5.2 Time measurements

For measuring time we used the Java 'system.nanoTime' function, which returns the elapsed time since some fixed moment in the past in nanoseconds. The elapsed time - or wall time - of a part of the program can be calculated by doing a time measurement before and after it and subtracting the one from the other. So this is basically the time it takes to perform the operation in real world time at the surface of the Earth. Even though 'system.nanoTime' is not accurate in its last decimals, we we still preferred this function over the alternative 'System.currentTimeMilis()'. The reason was that the second function too often gave an elapsed time of zero for the smallest graphs in the Benchmark Experiment.

An alternative that we considered was, instead of measuring real world time, to measure the CPU time. CPU time basically comes down to the amount of computations that the CPU has to perform. The upside of using CPU time over wall time is that it gets less influenced by other processes running on the computer. However, we did not manage to get this up and running on the university server on which we performed to measurements.

The server did not always perform the time measurements consistently. It could, for example, pause the computation for some time or be influenced by different processes. We made sure that we were able to overcome large measurement errors due to the server. To do this we delete the outliers from our time measurements using the Interquartile Range method. This methods considers data points as outliers if they are either smaller than $Q_1 - 3/2 * IQR$ or larger than $Q_3 + 3/2 * IQR$. Here Q_1 and Q_3 are defined as the first and third quartile, which means that a quarter of the data points is lower (or higher) than Q_1 (or

Q_3), and $IQR = Q_3 - Q_1$ is the Interquartile Range. In the results section we will elaborate further on this.

3.5.3 Memory measurements

We considered three approaches to measure memory. The most straightforward way is to make use of the built-in memory functions to calculate the *free memory*, the amount of RAM memory that is available to the program. In Java this can be done with $Runtime.freeMemory()$. The memory used to store an object can be computed by comparing the free memory before and after creating the object.

The second approach we considered makes use of profiling tools to measure the memory. Such tools have specific methods to calculate the memory of objects. However, when counting large amounts of objects these tool might give incorrect results. As objects can share a part of their memory, this approach could lead to overcounting memory usage.

The third approach computes a theoretic amount of memory that has to be used. This would mean that an Integer would be stored with 32 bits plus some overhead, and a bitmap of size 100 with 100 bits plus some overhead. Since this gives a way to control how much memory is spent on object overhead, this could lead to a fairly platform independent measure for memory usage. However, this approach would also introduce decisions we would have to make about how much memory we assign to certain objects, thus introducing arbitrariness. This approach also has the problem of dealing with memory sharing.

Since the approach of using the free memory available to the program seems to give the most stable and reproducible results, we decided to use it. However, before we could get stable measurements we had to overcome a problem. When using the free memory approach it is important to make sure that we do not measure too many objects. Think, for example, of the Integers that are created when iterating over an array. So to properly measure the memory consumption we need to run the garbage collector before performing the measurement. We had to make sure that the garbage collector behaves as expected and that it runs properly before each measurement.

At first this was clearly not the case. The garbage collector did not run consistently when called for. Because of this the outcome of the measurement would depend on whether and how the garbage collection was performed. We dealt with this by implementing a function which forces the garbage collector to execute. The function is shown in Figure 7. The idea behind the function is that we create such a large object that it can't be ignored by the garbage collector. Because we create the object with a Weak Reference (from package java.lang.ref.WeakReference), we make it ripe to be deleted. And because we check that it actually is deleted before performing the measurements, we make sure that the garbage collector has run before executing the measurement. Note that if it is not deleted the program will terminate. Since this has never happened, the method seems to function consistently.

It is important to note that, because we always forced the garbage collector

```java
public long measureMemoryForceGC() {
  WeakReference<long[]> garbage = new WeakReference <long[]> (new long[10000000] );
  garbage.get()[100] = 100;
  runtime.gc();
  if (garbage.get() != null) {
    System.out.println("GC failed...");
    System.exit(1);
  }
  long memory = runtime.totalMemory() - runtime.freeMemory();
  return memory;
}
```

Figure 7: Java function to force the garbage collector. Note that runtime is a Runtime object.

to run, we had to make sure to reference the object we wanted to measure somewhere in the program after the garbage collector had run. Otherwise it could get garbage collected itself and then we would not be able to measure it.

Another important note that we have to make is that creating a similar object in a different programming language can lead to different memory consumption, since overhead for creating objects can be different. An important assumption that we made is that these results generally translate to implementations in other languages, so that our results are at least somewhat independent of the programming language used.

4 Results

In this section we discuss the results that we obtained after running the experiments. We start with the Benchmark Experiment and then discuss the MMD Experiment. Unfortunately, we were not able to include different shapes for each data combination of data structure and algorithm, since that did not consistently leave room to show the standard deviations. For this reason the different combinations are indicated in color, so the figures have to printed in color. If no color printer is available we advice to study the figures on a computer screen.

4.1 Benchmark Experiment

We performed the Benchmark Experiment in the following configuration. Remember that the graphs that we considered were random partial 40-trees, with the probability of an edge $p = 0.075$, as described in section 3.2.1. For the Benchmark Experiment the smallest graph that we used consisted of 80 vertices and the largest graph consisted of 3.298.946 vertices. The number of edges of these graphs are about a factor six larger than the amount of vertices.

The data structures that we considered were Ints, XBitSet, EWAH, and Roaring. We considered the following implementations of Roaring: Roaring10, Roaring12, Roaring14, Roaring16, RoaringRun12, and RoaringRun16. Remember that the number of Roaring indicates the chunk size of that version: for

Roaring14 the chunk size is 2^{14}. Also remember that 'RoaringRun' indicates the use of run optimization, and that simply using 'Roaring' indicates not using it. We figured that these versions of Roaring would give a good indication of the behaviour of Roaring. Adding more versions would enlarge the computation time too much.

We perform the experiments for two different algorithms, the BFS Algorithm and the Parallel Algorithm. We implemented the BFS Algorithm for each of the data structures. We did not implement the Parallel Algorithm for Roaring10 and Roaring14. After performing preliminary experiments, we found that the Parallel Algorithm performed worse than the BFS Algorithm for Roaring on large graphs. For this reason we figured that adding Roaring10 and Roaring14 would not lead to many new insights. So to save computation time, we did not include Roaring10 and Roaring14 in combination with the Parallel Algorithm. So in the end we implemented the Parallel Algorithm for Ints, XBitSet, EWAH, Roaring12, Roaring16, RoaringRun12, and RoaringRun16. It is important to note that we performed the experiment for Ints with the Parallel Algorithm at a later moment in time then the other experiments. We will further discuss this in section 4.1.4. For each data structure '*name_data_structureP*' denotes that it makes use of the Parallel Algorithm and '*name_data_structureB*' denotes that it makes use of the BFS Algorithm. So for example we have XBitSetP and XBitSetB.

In the experiment we compared the data structures on four measures. The time it takes to compute the separated components and the memory that is needed to store the graph, the separators, and the separated components.

For each combination of graph, data structure, and algorithm we performed the following 15 sub experiments. By using 15 repetitions of the experiment we are able to say something about the measurement variance. In each of the 15 experiments we randomly selected 20 separators, without replacement, and for each of the separators we computed the separated components. Remember that these separators correspond to bags in the tree decomposition generated by the MD algorithm. Note that for the next sub experiment the same separator could be used. This means that for the smallest graphs that separators are likely to be used multiple times, so the outcomes for these experiments do not accurately reflect the actual distribution. However, since the focus of the thesis is on large graphs and the number of bags in the tree decomposition is typically around 70% of the graph size - so quite large - we figured that this would not be too large of a problem for our purposes.

An important thing to note is that the memory measurements are deterministic in nature. Every time we perform the measurements we get exactly the same results. On the other hand, time measurements are not deterministic. Other processes on the server have an influence on the time it takes to perform our computations. So the uncertainty of the time measurements is not only due to the fact that we have a different set of separators, it is also due to the behaviour of the server.

To implement the experiments, we made as much use of an abstract class as we could, to let the different data structures share as much code as possible. We

were able to perform most of the experiment and all the measurements in this class. Only a few functions had to be implemented separately for each of the data structures. These were functions as a cast from the set of integers to the data structure and performing the part of the algorithm in which we computed the separated components.

In total it took about 18 days to perform the Benchmark Experiment on the server. In the coming sections we will first describe the reasoning of why we got rid of the outliers for the time computations. Then we discuss how each of the data structures perform based on the algorithm used. Note that in these sections we only discuss the differences between the different implementations of the data structures. If we find no differences in one of the measures, for example the memory needed to store the graph, we postpone the discussion to the next section, in which we compare the best versions of the data structures with each other. Finally, we dedicate a couple of words to the time behaviour of the data structures, implemented in the Parallel Algorithm.

4.1.1 Outliers for time measurements

To get a sense of what happens with the time measurements we consider Figure 8, 9, 10, and 11. Here we see the boxplots for the time measurements for EWAWB, IntsB, Roaring16B, and XBitSetP. In these boxplots the horizontal bar at the bottom corresponds to $Q_1 - 3/2 * IQR$, the bar above that corresponds to Q_1, then Q_2, Q_3, and $Q_3 + 3/2 * IQR$. Outliers, which are outside of the region $[Q_1 - 3/2 * IQR, Q_3 + 3/2 * IQR]$, are indicated with the open circles. Note that in the boxplots the x-axis denotes the graph ID, here '1' denotes for the smallest graph, 'm' denotes for the m-smallest graph, and so on. The y-axis indicates the measured values, which are scaled to the mean of the measured values. So for each graph the average outcome equals 1.

The first thing that strikes our attention is that small graphs can have huge outliers, for each of the data structures used. For EWAH and XBitSet they get as large as respectively a factor 10 and 7 larger than the mean. These outliers probably arise when the server pauses the computation for a moment. For computations times - which are typically about 1 ms long for these graphs - these pauses can have a significant influence. We also observe that there seem to be more outliers that are too large, than too small. This could also be explained by the fact that the server does not always perform stable measurements. For these reasons we decided to eliminate the outliers.

Secondly, we observe that the interval $[Q_1, Q_3]$ seems to be fairly consistent not that large. In most cases Q_1 and Q_3 are all off from the mean by a factor five to twenty percent. This indicates that measurements are quite stable, as we will later also see, when we plot the mean values with standard deviation with respect to the graph size.

For EWAHB and XBitSetP it is a bit harder to see, but for Roaring16B and IntsB the following is more clear. There does not seem to be a clear pattern in the boxplots. Some have a median that is higher than half way up the box, some have median that is lower than half way up the box. The same holds

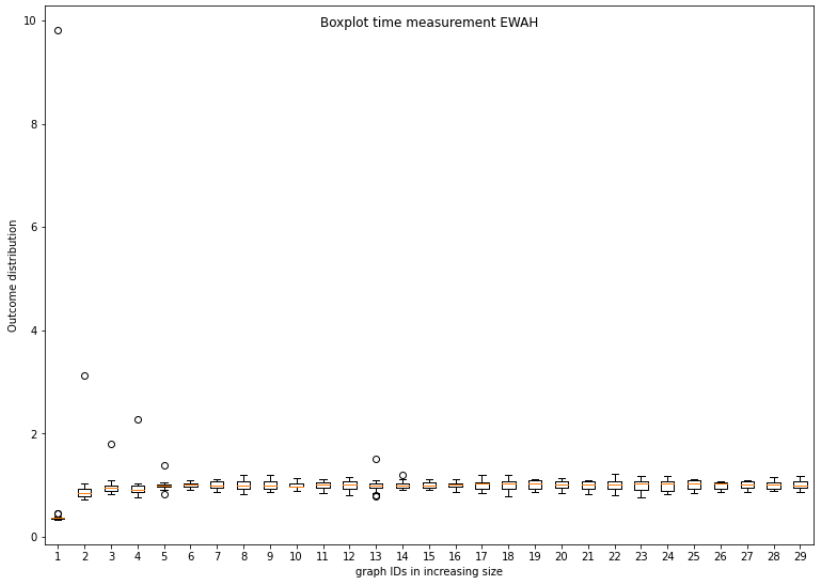

Figure 8: Boxplot for the time measurements for EWAHB, scaled to an average value of one.

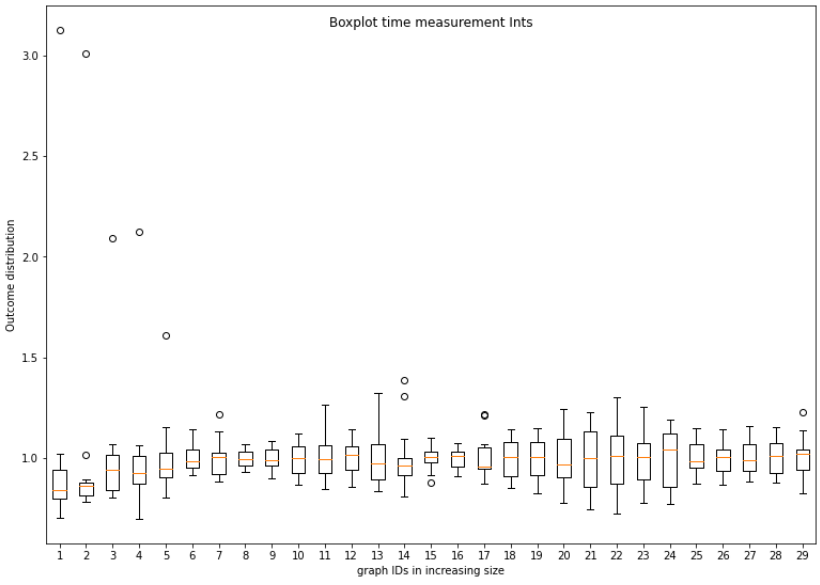

Figure 9: Boxplot for the time measurements for IntsB, scaled to an average value of one.

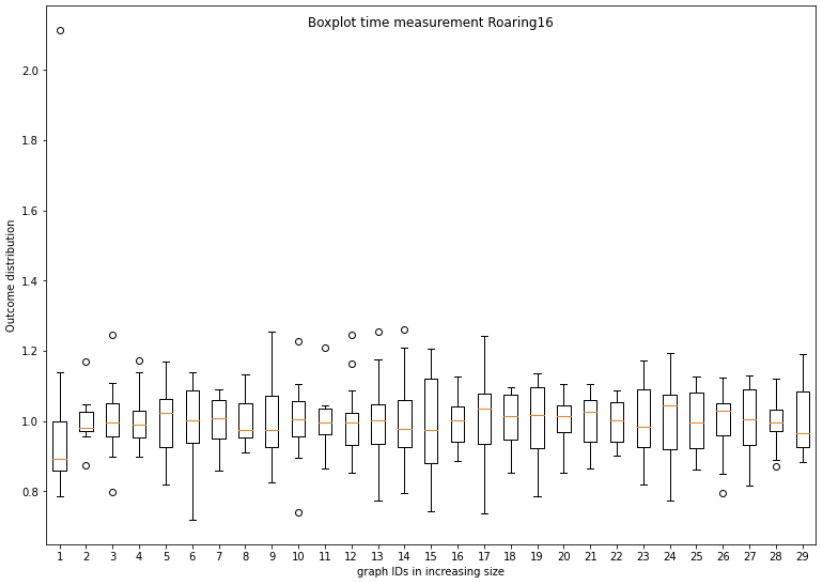

Figure 10: Boxplot for the time measurements for Roaring16B, scaled to an average value of one.

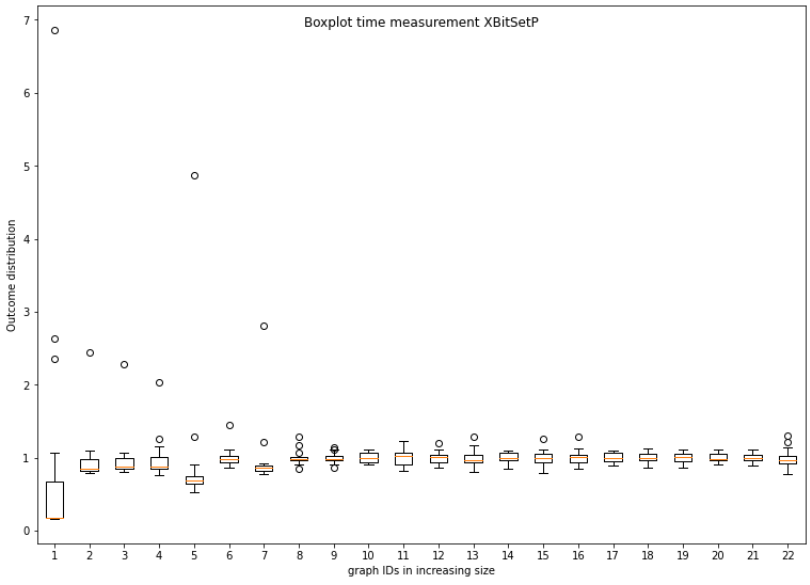

Figure 11: Boxplot for the time measurements for XBitSet, scaled to an average value of one.

for the whiskers. Sometimes they extend longer to the top sometimes to the bottom. From this information we can conclude that there does not seem to be a underlying distribution that is skewed according to which the measurements behave themselves. So this could be a signal that perhaps a normal distribution would describe the measurements quite well. However, we decided not to dive further into the statistical behaviour of the data points. To further conclude on this, more analysis would be necessary.

4.1.2 XBitSet

In this section we discuss the results for XBitSet. But before we do so, we have to note that we likely did not use the optimal implementation for our experiments. We instantiated each XBitSet with the function $XBitSet(G(N))$, creating bitmap with length equal to the graph size $G(N)$. Instead, we could have used $XBitSet()$, which uses dynamic memory allocation. This means that when the bitmap is instantiated it is as long as its largest '1'-value. If at a later moment a '1'-value is added that is larger than its largest value, the XBitSet is extended. Note that if the largest value is at some point set to '0', the bitmap retains its size. So the size of the XBitSet in this case, is the largest value that it has ever contained.

Especially for storing small vertex sets, implementing $XBitSet()$ could lead an improvement over $XBitSet(G(N))$. In many such cases a significant stretch of 0's at the end would not have to be stored, thus consuming less memory. Time wise it would probably also lead to an improvement, since iterating through a set and performing logical operations will take less time for shorter bitmaps. Though it should be noted that some operations might also take longer, because bitmaps might have to be enlarged during an operation. However, in our experiments we expect improved performance when making use of $XBitSet()$. We expect this because the most time-consuming operations take place with relatively small vertex sets, namely sets of neighbors in the graph.

To get an indication of the improvement we could expect, we measured the difference it could maximally make to implement the experiments with $XBitSet()$ instead of $XBitSet(G(N))$. To do this we compared the combined length of all the bitmaps to store the graph. We found that $XBitSet()$ needed about 81% less bitmap length. So this maximally corresponds to an improvement of a factor up to as low as 81% for both the memory consumption to store the graph and the run time. When storing the separators and the separated components we also often have to store small vertex set. So we also expect to see improvements in these measures.

However, to fairly compare the other implementation with the others data structures, we would have to perform all the experiments again. The server performance is not constant over time; it varies with possible updates performed and other processes running. So if we would only perform the experiments for XBitSet() and compare them to the other results, we would be comparing apples to oranges for the time measurements. Because the experiments took quite a long time to compute we decided not to redo the experiments. However, it is

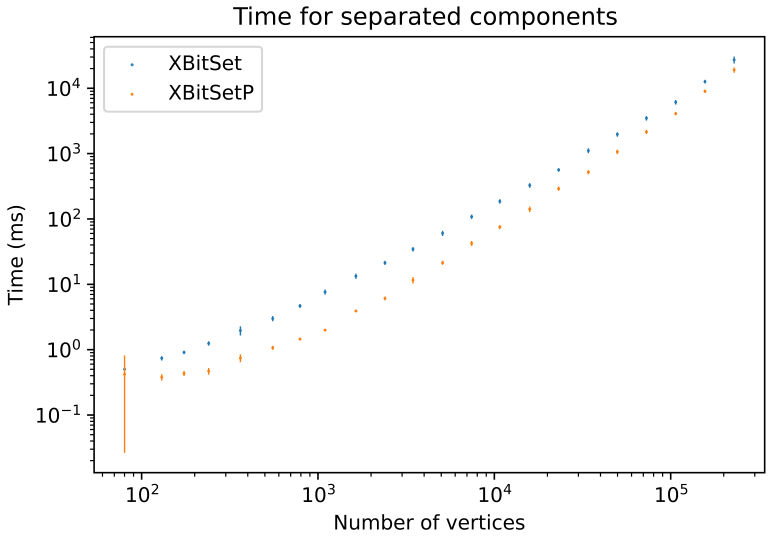

Figure 12: Time performance with respect to graph size for XBitSet. Note that XBitSet denotes the BFS algorithm and that XBitSetP denotes the Parallel Algorithm. Also note that the vertical bars show the standard deviation of the measurements.

important that we reflect on the fact that optimal results for running time and storing the graph can be improved by a factor of up to as low as 81%.

On to the results that we found while making use of $XBitSet(G(N))$. We did not find differences between the algorithms for the memory used. This makes sense because each vertex set that is stored corresponds to exactly the same amount of memory, namely a bitmap with length $G(N)$.

A large limitation of XBitSet is that it takes a lot of memory to store the graph. This is due to the fact that storing a graph with bitmaps scales quadratically in the graph size. Because when adding a vertex we do not only have to store another bitmap. It is also the case that each bitmap that we have to store becomes longer. Note that this is also true for the XBitSet() implementation. Because of this we were only able to perform the experiments for graphs up to a size of 229.060.

In Figure 12 we see the time performance of the two algorithms for XBitSet. It is easy to see that XBitSetP outperforms XBitSetB for all graph sizes, except for the smallest graph, where we are not able to tell due to the large variance. Note that the variance for the other measurements is quite low, supporting our conclusion.

However, when we consider scaling behavior, we have to conclude something different. We see that for the larger graphs the difference between the two algorithms becomes smaller. The largest difference between the two seems to be

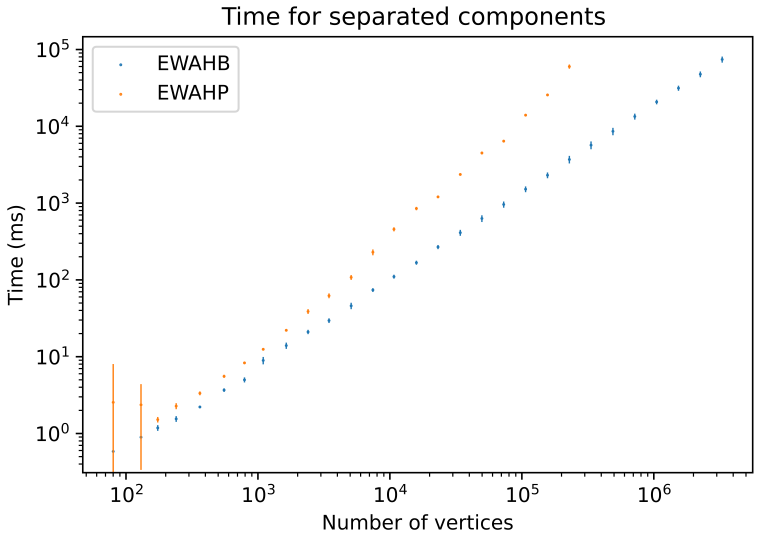

Figure 13: Time performance with respect to graph size for EWAH.

at a graph size of about 1000, when it is almost a factor 4. But for largest graph we only see a difference of a factor of about 1.4. So we have to conclude that XBitSetB scales better to large graphs than XBitSetP. If we would extrapolate these results to even larger graphs, we expect XBitSetB to outperform XBitSetP at some point. However, it seems unlikely that XBitSet is an efficient data structure for such graphs, since it is not memory efficient for large graphs.

From these results we can conclude that XBitSet is quite efficient with handling logical operations. So when using XBitSet it can be worthwhile to consider these kinds of operations, when making a decision of which algorithm to use or how to implement it. However, for larger graphs BFS seems to be performing relatively better, indicating that for huge graphs many logical operations become relatively costly.

4.1.3 EWAH

For EWAH we see the time performance of the two algorithms depicted in Figure 13. Note that, because the Parallel Algorithm turned out to be relatively slow for EWAH, we only implemented Parallel for EWAH for graphs up to a size of 229.060. We can clearly see that EWAHB outperforms EWAHP. Note that the difference becomes larger for larger graphs sizes, so EWAHB also seems to scale better than EWAHP.

Similarly to XBitSet, we found for EWAH that the memory needed to store the graphs and the separators does not depend on the algorithm used. However, for the separated components we do see a difference. In Figure 14 we see that EWAHB structurally outperforms EWAHP. Even though the variance is rela-

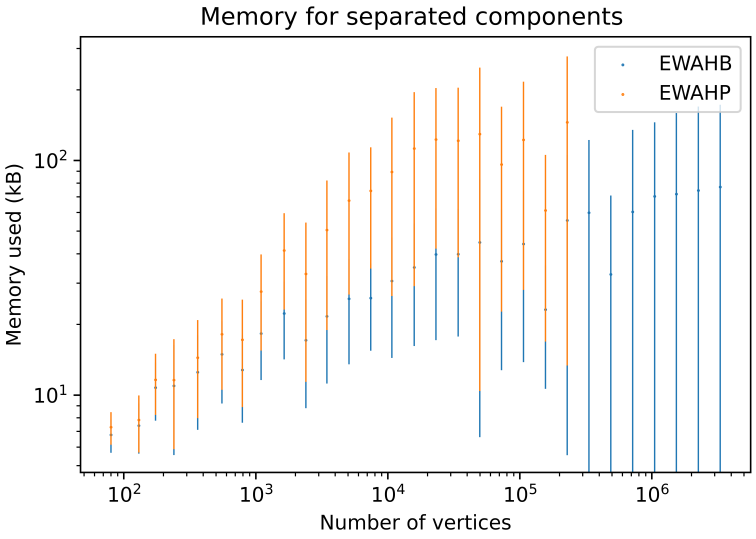

Figure 14: Memory consumption for storing the separated components with respect to graph size for EWAH.

tively large, we observe that for each measurement EWAHB performs better, as expected. The observed effect is quite large, since the difference of the outcomes can be as large as a factor 3.

This cannot be explained by dynamic memory allocation alone, since EWAH can store a stretch of zeros at the end very efficiently, with one marker word and one literal word. So with only two Long numbers - or 128 bits, way less than the observed differences. We conjecture that this has something to do with the specific EWAH implementation, which perhaps trades in some memory optimization for increased speed of logical operations. Further investigation would be needed to fully explain this.

Note that the standard deviation is quite large for storing the separated components. For a discussion about this, we refer to the end of section 4.1.7.

4.1.4 Ints

At first, we did not perform the IntsP experiment due to the fact that it performed too poorly. However, at a later stage we realized that it would be interesting to show its behaviour, because it could tell us something about the relative performance of the data structures in the Parallel Algorithm. This, in turn, provides us with new insights about the data structures. Because we performed the IntsP experiment at a later moment in time, it is important to note that the results relative to the other data structures might be somewhat off, due to variability in the server performance.

The time performance of Ints is shown in figure 15. We clearly see that

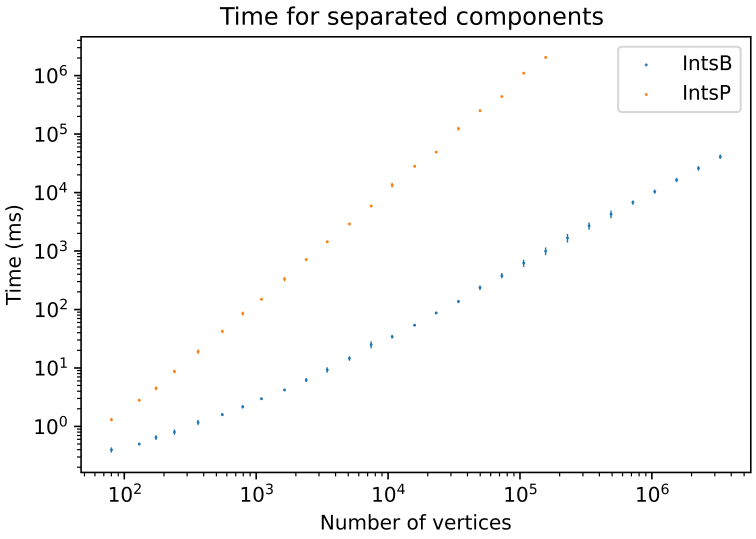

Figure 15: Time performance with respect to graph size for Ints.

IntsB outperforms IntsP for each graph size and that it scales better to large graphs. The reason that this is the case is that logical operations for Ints are quite inefficient, especially when vertex sets get dense, as is the case in the Parallel Algorithm. In that algorithm we grow the separated component, by adding the neighborhoods of vertices to it. So the sets starts out small, but for large separated components they get very large. Especially for these large components we have that the operation of adding the neighborhoods to the components is inefficient for Ints.

Consider the case that we have two vertex sets and we want to compute their union. A bitmap can simply flip all the bits from '0' to '1' for each of the vertices that need to be added, which is a constant time operation we have to perform as many times as the size of the smallest set. For Ints, however, we have to create a new array and repeatedly figure out which vertex is the smallest and then copy it to a new array. This operation has running time of $O(n_1 + n_2)$, where n_1 and n_2 are the lengths of the vertex sets. For large vertex sets this operation becomes costly.

For memory consumption, we observed that for Ints the choice of algorithm does not have an influence, since the memory only depends on the integers which are stored. These integers only depend on the set they represent, not on the path taken to obtain that set.

4.1.5 Roaring10B to Roaring16B

For Roaring we conducted a large number of experiments. We start with the comparison between Roaring10B, Roaring12B, Roaring14B, and Roaring16B.

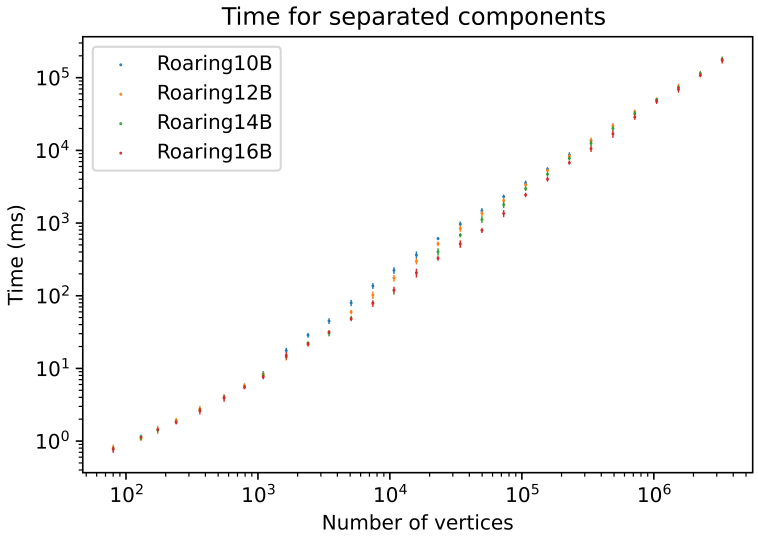

Figure 16: Time to compute the separated components with respect to graph size for Roaring10B, Roaring12B, Roaring14B, and Roaring16B.

Then we will discuss the behaviour of Roaring12B, Roaring12P, RoaringRun12B, RoaringRun12P, Roaring16B, Roaring16P, RoaringRun16B and RoaringRun16P.

The time measurements for the versions of RoaringB are shown in Figure 16. The first thing to note is that, as mentioned earlier, we do not perform the experiments for Roaring10B for the largest graphs, due to the teething problems of our implementation of the different versions of Roaring.

What stands out is that the differences between the different versions of RoaringB are quite small, especially for the small and largest graphs. For the smallest graphs this makes a lot of sense. Up to the graph size of $2^{10} = 1024$, these versions of Roaring behave relatively the same in the BFS Algorithm. Up to this graph size, each version of Roaring consists of a single chunk. Also, practically all of the vertex sets that BFS has to deal with - namely the neighborhood sets of the vertices - are quite small and thus represented as arrays of integers. This explains the fact that up to a graph size of 1024 we observe roughly the same results.

As graphs grow larger than the size of 1024, first Roaring10B gets a little bit slower than the competitors, and then, for even larger graphs, Roaring12B follows, followed by Roaring14B. Note that we start to observe this divergence behaviour at a graph size of 2^X for RoaringX. This corresponds to the moment that they start to consist of more than one chunk. At the same time we observe that as Roaring12B starts to slow down, it starts to grow closer to the time efficiency of Roaring10B. The same holds for Roaring14B and later for Roaring16B. We see that the time performances start to converge for the largest graphs. So it seems that Roaring16B is the most time efficient data structure,

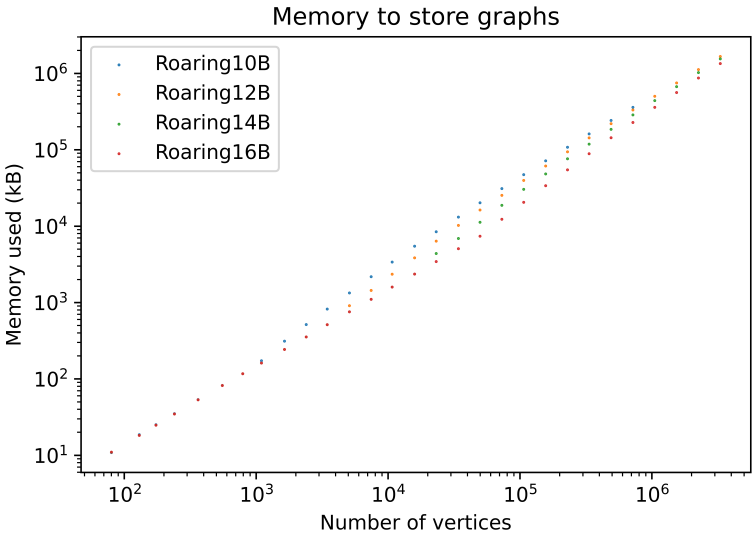

Figure 17: Memory required to store the graphs with respect to graph size for Roaring10B, Roaring12B, Roaring14B, and Roaring16B.

for the largest range of graph sizes. However, we also observe that the scaling behavior to huge graphs seems similar for the different versions of RoaringB.

To repeat, we observe that all versions of RoaringB seem to slow down, as soon as we reach the threshold at which they start consisting of more than one chunk, and that their behavior converges for large graphs. Before we explain this convergence behavior for large graphs, let us consider the memory consumption for storing the graphs and separators, shown in figures 17 and 18. In both figures we observe exactly the same behavior. Different versions of RoaringB have the same results for small graphs. Then we observe that first Roaring10B starts to perform relatively poorly, followed by Roaring12B and Roaring14B. Finally, the behavior converges for the largest graphs. So again, Roaring16B seems to perform best if we consider the entire range of graph sizes.

The reason that we observe this behaviour for large graphs has to do with the fact that we are dealing with small vertex sets. This leads to the fact that for large graphs more and more chunks are not instantiated, since chunks are only instantiated if at least one value in that chunk is present. It is important to note that this effect is larger for versions of Roaring with smaller chunk size, since more chunks will not be instantiated. This explains the convergence for large graphs. So what we basically observe is that the time it takes to iterate through a set and the memory required to store a set seems to depend on the number of instantiated chunks.

To get a better understanding of this behaviour, we explain this with an example that compares Roaring12B with Roaring16B. Suppose we either have to store a set of small vertices, such as a separator or the neighborhood of a

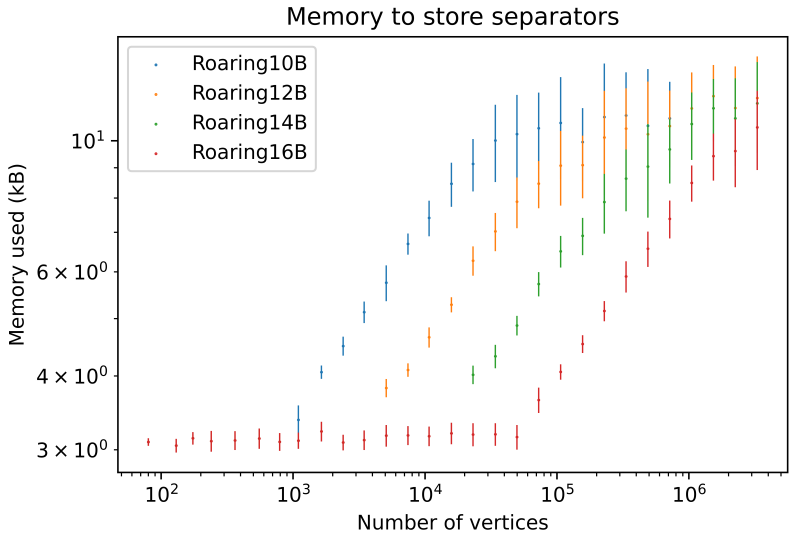

Figure 18: Memory required to store the separators with respect to graph size for Roaring10B, Roaring12B, Roaring14B, and Roaring16B.

vertex, or that we have to iterate through a small vertex set. Say that the vertex set under consideration consists of 10 vertices, with random vertex IDs. Now suppose that we have a graph size of 1000, then it is the case that both Roaring12B and Roaring16B have a single chunk containing all 10 vertices. Now suppose we have a graph of size 6000. Since this is larger than 2^{12}, Roaring12B is likely to store the vertex set over two different chunks, while Roaring16B still stores it in a single chunk. In this scenario, when we store the vertex set or iterate through the neighborhood, we have to consider two chunks for Roaring12B and only one chunk for Roaring16B. We believe that such cases lead to the better performance on medium sized graphs that we observed for Roaring16B as compared to Roaring12B. Now consider a graph of size 10.000.000. Since Roaring16B can consist of up to 153 chunks and Roaring12B can consist of up to 2442 chunks for a graph of this size, both Roaring16B and Roaring12B are likely to consist of 10 chunks, one chunk for each vertex that it has to represent. This explains that for such a graph size Roaring16B and Roaring12B are expected to behave similarly.

Let us now describe the memory requirements to store the separated components. Consider Figure 19 and 20, the latter without standard deviation shown. Again, for a discussion about the standard deviations, we refer to section 4.1.7. Interestingly, we observe the opposite happening of the measurements of the time and memory for graphs and separators. Again, we observe that for small graphs the different versions of RoaringB behave similarly. Though now we observe that as graphs get larger, when at some point Roaring10B diverges from

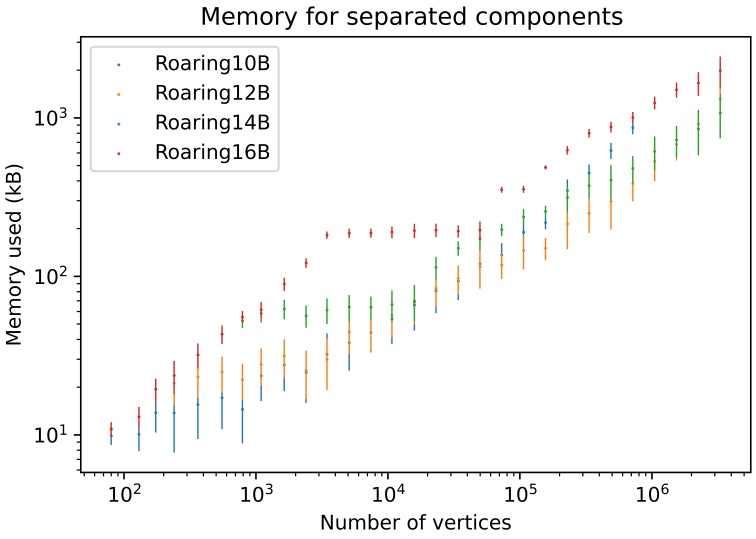

Figure 19: Memory required to store the separated components with respect to graph size for Roaring10B, Roaring12B, Roaring14B, and Roaring16B.

the others, it leads to an improvement. The same holds for Roaring12B and Roaring14B.

Instead of having 2^X as the inflection point for RoaringX, we observe that these inflection point corresponds to 2^{X-4}. This means that the inflection point indicates that besides sparse chunks, also dense chunks can from there on be present. This makes a lot of sense. Since separated components are often very dense vertex sets, representing them as arrays of integers is not efficient. We observe that in these cases having the option to represent them as bitmaps leads to improvements.

Note that the data points have a second inflection point, which appears at 2^X for RoaringX. Especially for Roaring16B we notice a clear straight line up starting form the first until the second inflection point, indicating the fact the memory consumption for storing the separated components does not grow in this interval. After the second inflection point, the memory consumed starts to grow again. The reason for the growth is that for graphs larger than the second inflection point, we have that vertex sets have to be represented by more than one chunk. We can explain the lack of growth in between the two inflection points, by the fact that the dominant factor in the memory consumption for storing the separated components is storing dense chunks. We know this because bitmaps chunks have a constant size, namely the size of an entire chunk. The fact that this constant size is observed in each data point in the interval, indicates that the memory required for the other data sets is not very relevant.

These observations make sense when taking the distribution of component sizes into account, since we have that a very large portion of the components

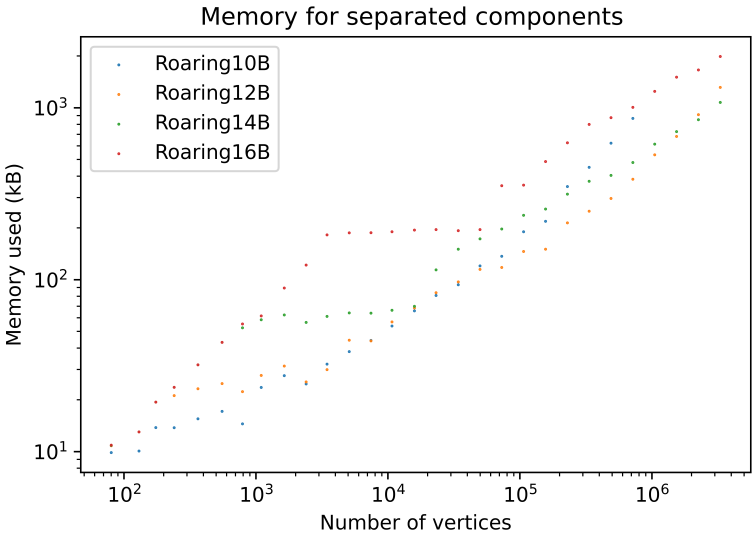

Figure 20: Memory required to store the separated components with respect to graph size for Roaring10B, Roaring12B, Roaring14B, and Roaring16B, without standard deviation.

is almost the empty set, a large portion is almost the full set, and there is not much in between. So we find that storing almost empty sets does not take a lot of memory and that storing the bitmap chunks is the dominant factor.

Another interesting thing to notice, is that when the interval between the two inflection points starts for Roaring16B, we see that Roaring16B consumes a factor 4 more memory than Roaring14B. This is due to the fact that the bitmap that Roaring16B stores is a factor four larger than the bitmap that Roaring14B stores. This is simply due to the fact that the chunk sizes differ between the two versions, so the memory required to store the bitmap difference as well. Note that the same idea holds for the different versions of RoaringB, though it is not as clearly visible.

So for small to medium-sized graphs we observe that Roaring10B performs best. For graphs that are larger than about a size of 4000, we see that Roaring12B is starting to outperform Roaring10B. Similarly, for the largest graphs we see that Roaring14B is starting to outperform Roaring12B. It seems that for relatively large graphs having a larger chunk size is advantageous. Note that this also becomes apparent when considering the slope of the data points. Following this reasoning, we can expect Roaring16B to at some point outperform Roaring14B, though this does not become clear from our experiments and it is not sure whether such large graph sizes will become tractable in the near future.

The fact that RoaringB with large chunk sizes outperforms their smaller chunk size counterparts, is interesting behavior. While RoaringB with small chunk sizes has to store more chunks and so has more overhead, it is also the case

that for relatively large graphs and dense vertex sets it has a larger probability of representing the chunk with RLE, which happens only if the chunk is completely full.

We are not sure about the following reason, but it could be the case that it does turn out to be the overhead that determines the behavior. Suppose that we have many chunks that are completely full, then each of these chunks has to be represented by RLE and importantly, each of these chunks has overhead. Then, it can become an advantage for Roaring with large chunk sizes to represent fewer such chunks. For example, for a completely full set for a graph size of a million, Roaring10B would have to store about a thousand RLE chunks, while Roaring12B we have to store about 250, and Roaring14B about 60. Again, we is not sure that this explains the behavior, but we know that this is a factor that we have to take into account for very dense vertex sets.

4.1.6 Roaring, RoaringRun, and the Parallel Algorithm

Here we discuss the comparison between Roaring12, RoaringRun12, Roaring16, and RoaringRun16, both implement with the BFS Algorithm and the Parallel Algorithm. In Figure 21 and 22 we see the time behaviour of the variants we investigated of respectively Roaring12 and Roaring16. The first thing that should be noted, is that in both figures and irrespective of the algorithm used, the data points that correspond to Roaring and RoaringRun basically overlap. So it seems to be the case that there is no difference between Roaring and RoaringRun in the Benchmark Experiment for the time to compute the separated components.

This behavior makes sense, because the vertex sets that the algorithms are working with correspond to the neighborhood of vertices. The vertices have a low number of neighbors and converting sparse and unstructured sets to RLE does not to lead to improvements in memory consumption. The fact that these vertex sets are not converted to RLE chunks basically means that we get more or less the same computations for Roaring and RoaringRun and this is what we see.

The second thing that stands out is that in both figures we see that for large graphs RoaringB outperforms RoaringP, and considering that slope we see that the difference only grows as the graphs get larger. We see that Roaring16B continuously outperforms Roaring16P. We also see that there is a slight the window in which Roaring12P seems to outperform Roaring12B, for graphs sizes between 800 and 5000. Note that for these graphs Roaring12P also outperforms Roaring16B and Roaring16P, as shown in Figure 23.

There are a couple of interesting things to note in the performance of RoaringP. First of all that while RoaringB seems to scale quite steadily, this is not the case for RoaringP. For small graphs RoaringP starts with a relatively steep slope, then at some point the computation time regresses as the graph sizes increase, and then it continues a relatively steep slope. To understand this we need to understand what happens in Roaring. When we look at that the inflection point of Roaring12P, after which the computation time will decrease as the

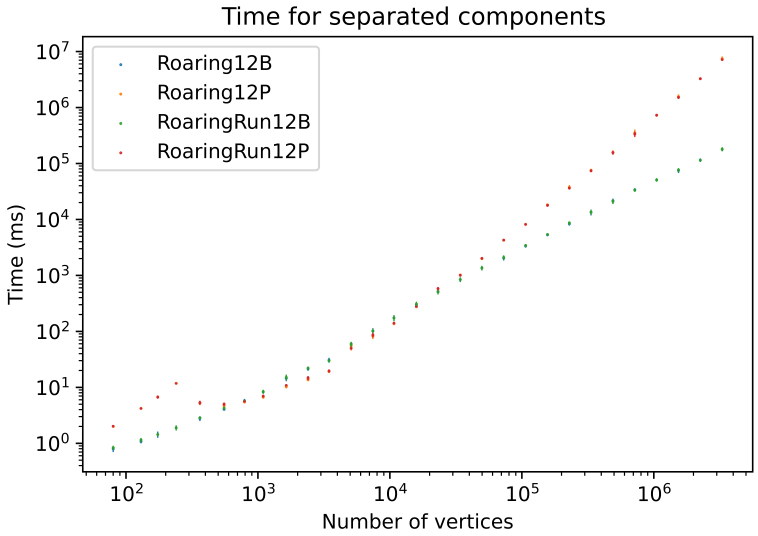

Figure 21: Time performance with respect to graph size for Roaring12B, Roaring12P, RoaringRun12B, and RoaringRun12P.

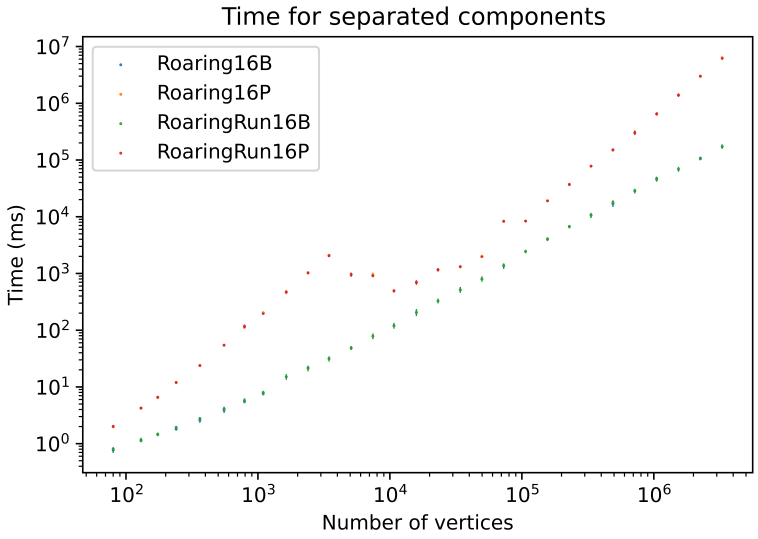

Figure 22: Time performance with respect to graph size for Roaring16B, Roaring16P, RoaringRun16B, and RoaringRun16P.

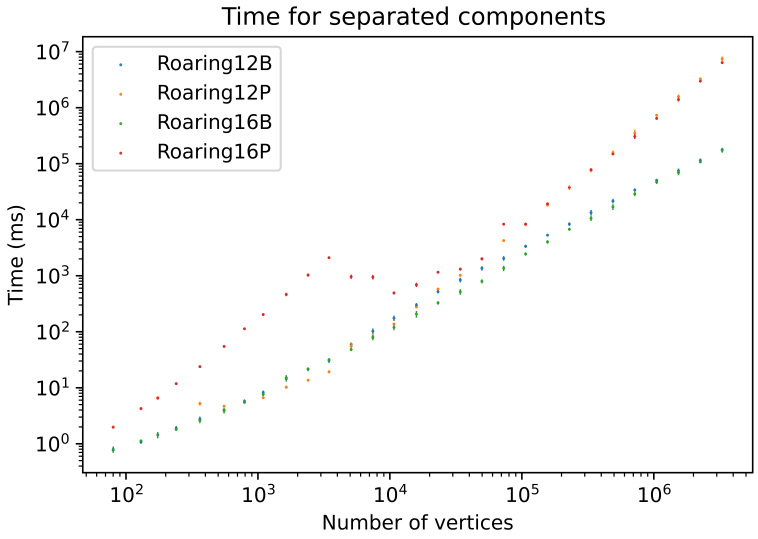

Figure 23: Time performance with respect to graph size for Roaring12B, Roaring12P, Roaring16B, and Roaring16P.

graph sizes increases, we see that this corresponds to a graph size of $2^8 = 256$. For Roaring16P the inflection point corresponds to a graph size of $2^{12} = 4096$. In both these cases this corresponds to the graph size, such that chunks can from there on out not only be sparse - and thus represented as arrays of integers, but also dense - and thus represented as bitmaps.

This leads to the situation that the separated components that we are computing, which frequently have a size in the order of the graph size, will no longer always be represented by sparse chunks. Chunks that make up the dense separated components will be represented by bitmaps, after they become larger than this inflection point (2^{X-4} for RoaringX). As we discussed earlier, performing logical operations on bitmaps is generally faster than performing logical operations on arrays of integers. As long as one of the chunks is a bitmap the operations can be very efficient. Of course combining two bitmaps is efficient as well. But note that this is also the case when we combine an array with a bitmap. Then we can iterate over the array of integers and simply set each value that we encounter to '1' in the bitmap, which is efficient. So the fact that the separated component can be represented by a bitmap for these graph sizes, leads to better performance for logical operations, which explains the regression in computation time for RoaringP. Note that this behaviour would make it interesting to further investigate the running time of Roaring10P and Roaring14P. Unfortunately, this lies outside of the scope of this project.

The reason that we see that RoaringP scales worse to larger graphs than RoaringB is similar to the behaviour that we saw for the other data structures. For huge graphs it does not seem efficient to perform many logical operations.

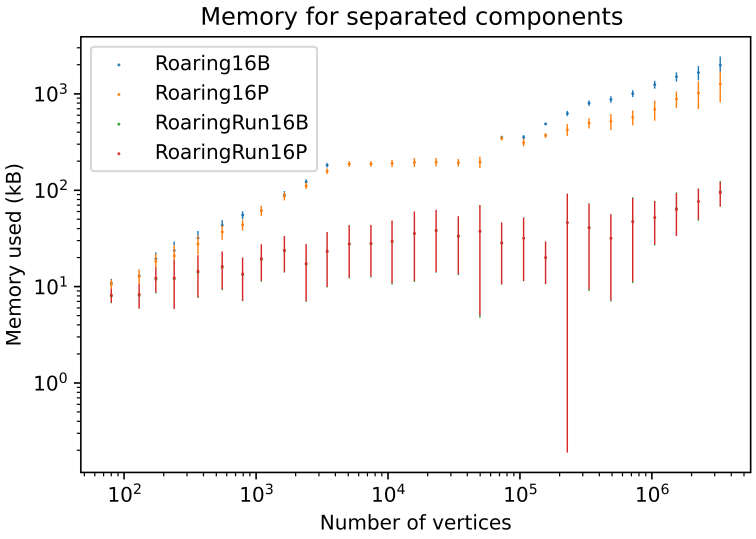

Figure 24: Memory consumption of the separated components with respect to graph size for Roaring16B, Roaring16P, RoaringRun16B, and RoaringRun16P.

For huge graphs it simply seems more competitive to use the BFS algorithm, which iterates through all the vertices.

Similarly to XBitSet and EWAH we found that the memory needed to store the graphs and the separators is the same, irrespective of the algorithm and whether we use Roaring or RoaringRun. For the memory requirements for the separated components we consider Figure 24 and 25. Because the large overlap for Roaring12 we also consider Figure 26, where we see the results for Roaring12 shown without the standard deviation.

The first thing that should be noted is that for both RoaringRun16 and RoaringRun12 the results do not depend on the algorithm used. We also observe that RoaringRun16 and RoaringRun12 seem to outperform their counterparts, which do not make use of run optimization. Finally, we observe that the algorithm used makes a difference, when we compare Roaring16B with Roaring16P and Roaring12B with Roaring12P.

The reason that we see these differences is not due to the following effect, though it is important to be noted. A chunk is only instantiated when it contains at least one vertex. If this is the case there will be overhead generated to store the chunk. And because the Parallel Algorithm adds and removes vertices from the set, more chunks will be instantiated than will be the case for BFS. This would lead to a larger memory consumption for RoaringP than for Roaring B. In fact we see this effect explains that Roaring12B outperforms Roaring12P for some graphs of small size. It also explains that RoaringRun12B seems to outperform RoaringRun12P slightly for some graphs, though the reverse seems

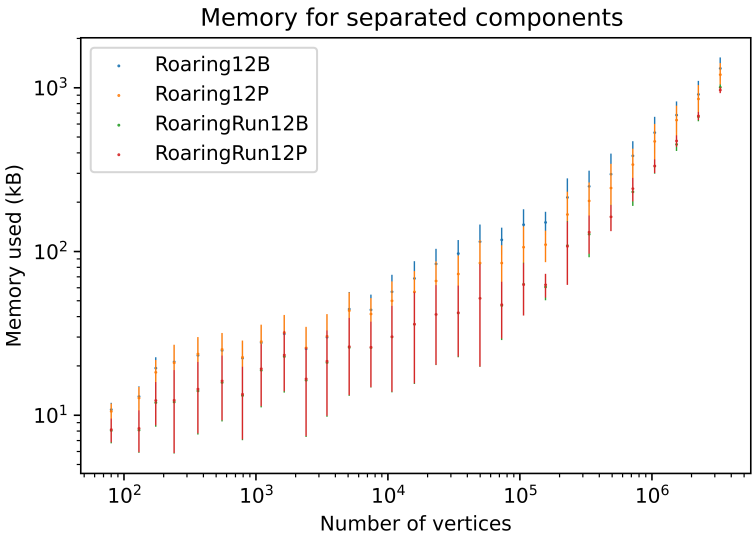

Figure 25: Memory consumption of the separated components with respect to graph size for Roaring12B, Roaring12P, RoaringRun12B, and RoaringRun12P.

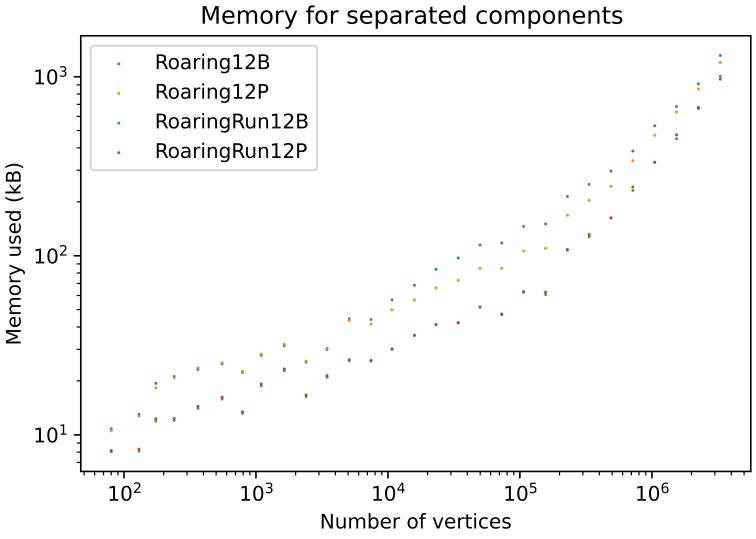

Figure 26: Memory consumption of the separated components with respect to graph size for Roaring12B, Roaring12P, RoaringRun12B, and RoaringRun12P, without standard deviation.

never to be the case.

However, especially for large graphs we see that RoaringP seems to outperform RoaringB. This is due to the fact that if a chunk is completely full it will be converted to RLE. For the same reason that the Parallel Algorithm instantiates more chunks, we see that the Parallel Algorithm also generates more RLE chunks. Note that a chunk can be converted to RLE at some point. If, at a later moment, a vertex is removed from that chunk, it will remain a RLE chunk. Since separated components can turn out to be close to the complete set, storing chunks for these vertex sets as RLE greatly outperforms storing them as bitmaps.

As mentioned, we see that RoaringRun seems to outperform regular Roaring with respect to storing the separated components. This is in the nature of RoaringRun, since it only transforms chunks if it needs to reduced memory consumption. The fact that the difference can turn out to be so large is, similar to the reasoning above, due to the fact that storing almost full bitmap chunks can easily be improved using RLE. When using RLE storing a stretch of ones or zeros is done with two Short numbers, the first indicating the start of the sequence in the second indicating that a sequence ends. Storing two Short numbers takes 32 bits, which roughly corresponds to the bitmap of length 32. So if a typical stretch of ones or zeros is larger than 32 RLE is expected to be an improvement over bitmaps. Note that this turns out to be frequently the case for separated components which almost contain the entire graph. That is why we see the relatively good performance of RoaringRun.

The final thing that strikes our attention is that the difference between RoaringRun16 and Roaring16 seems to be significantly larger than between RoaringRun12 and Roaring12, especially when considering large graphs. To dive into this we consider Figure 27, which shows Roaring12B, RoaringRun12B, Roaring16B, and RoaringRun16B. First of all we observe that RoaringRun16B performs best, then RoaringRun12B, Roaring12B, and Roaring16B.

Now let us focus our attention on the difference between RoaringRun12B and RoaringRun16B. We see that up until a graph size of about 10,000 RoaringRun12B sometimes perform slightly better, though differences are minimal to non-existent, and for graphs larger we see that RoaringRun16B starts to perform significantly better, up to a factor 10 for the largest graph. We try to explain this by the fact that most separated components are either almost completely full or completely empty. For example, there are separated components consisting of a single vertex, while others consist of all vertices except for the separator. For these separated components it is the case that almost all chunks are either (almost) completely full or (almost) completely empty. For the largest graph, RoaringRun16 has to store $3298946/2^{16} = 51$ chunks, while RoaringRun12 has to store $3298946/2^{12} = 806$ chunks. For storing an almost completely full or empty chunk, the chunk overhead counts for a large amount of the memory consumption. Since this is up to a factor 16 larger for RoaringRun12, we conjecture that it explains the difference. Though note that we are not a completely sure that this effect explains the behaviour in its entirety.

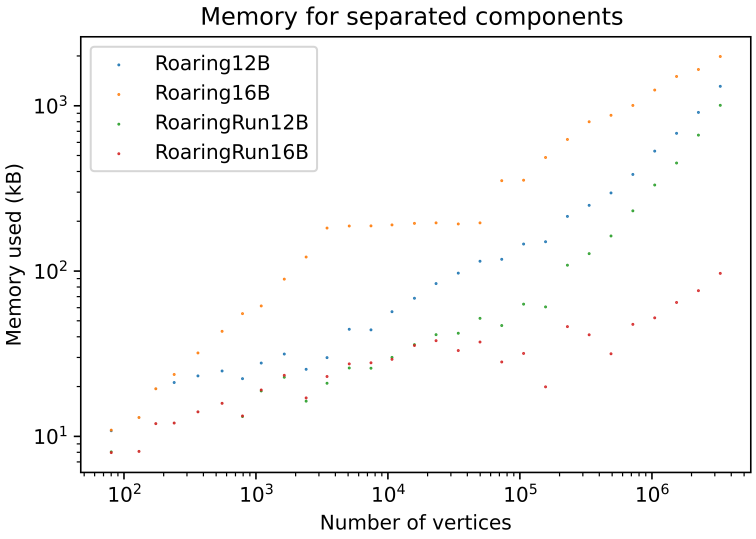

Figure 27: Memory consumption of the separated components with respect to graph size for Roaring12B, RoaringRun12B, Roaring16B, and RoaringRun16B.

4.1.7 Best implementation of each data structure

In this section we compare the data structures with each other. To do this we consider the data structures at their best performance. So we consider data structure/algorithm pairs that performed best and the best version of Roaring. This comes down to comparing XBitSetP, IntsB, EWAHB, and RoaringRun16B.

Figure 28 shows the time performance of the data structures. We observe that for large graphs we have IntsB performs best, followed by EWAHB, RoaringRun16B, and XBitSetP. Moreover, we observed that IntsB outperforms EWAHB and RoaringRun16B for each graph size. Though note, that for the largest graphs EWAHB seems to grow at a slower rate than IntsB, suggesting that at some graph size, EWAHB might become competitive with or even exceed the performance of IntsB. For small graphs we see that XBitSetP outperforms the others. Though it is also clear that XBitSetP seems to scale the worst to larger graphs. Finally, it is interesting to note that for all graph sizes EWAHB seems to outperform RoaringRun16B, except for the region of graph sizes between 2000 and 8000, in which we observe that RoaringRun16B and EWAHB seem to perform similarly.

The impressive performance of XBitSet for small graphs is very notable. Since many problems can be described as small graphs, XBitSet can be the data structure of choice for many problems. The reason that it performs so well has to do with the fact that it is very efficient in performing logical operations on sets. However, XBitSet does not seem to scale well to large graphs. In those cases having to store or iterate through all stretches of zeros becomes quite

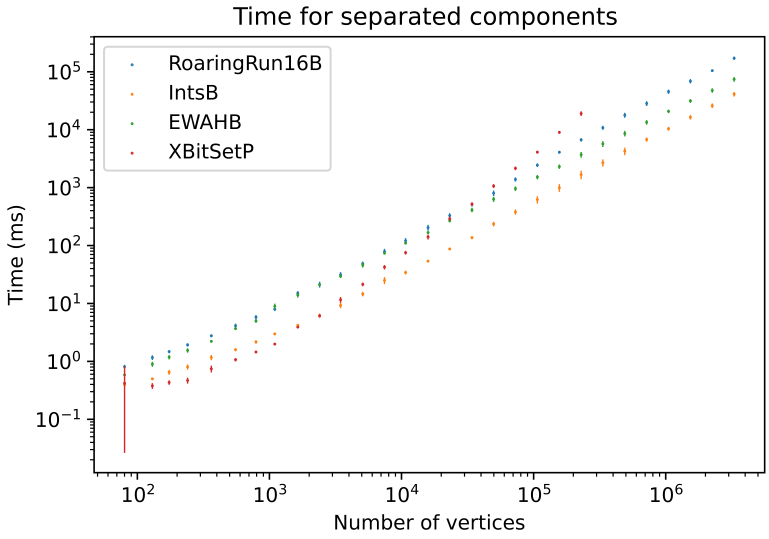

Figure 28: Time measurement for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

costly.

Note, that if we would have implemented XBitSet in the way in which it could have performed better by up to as low as 81% as its current run time, it would not significantly alter our observations. We would see that the range of graphs in which XBitSet performs best would be a little larger, though for large graphs its scaling behaviour would still be the worst.

Now let us turn to the discussion of why we observe that Ints performs so well. When we compare IntsB with EWAHB and RoaringRun16B, it is important to remember the discussion we had in section 3.1.5. There we discussed the quickest way to iterate through a set. Since the BFS algorithm practically does this for the neighborhood set of each vertex, iterating through these sets turns out to be the largest factor that determines the running time.

In section 3.1.5 we found that the quickest way to iterate through sets for EWAH and Roaring turned out to be to construct an iterator and then visit each element of that iterator. When we consider Ints, we see that Ints already practically has such an iterator, namely the set of Integer numbers itself. So it turns out that EWAH and Roaring first have to perform an intermediate step, before they can start the iteration, while Ints does not have to perform such an intermediate step. Ints can start iterating right away. We conjecture that this explains the better performance of Ints.

In this discussion, it is important to note that the vertex sets we are dealing with, namely the neighborhood sets, are relatively small. It could be the case that for iterating through larger, more dense vertex sets, Roaring and

EWAH have some tricks up their sleeves to improve performance. Additional experiments could be held to determine the performance in a wider variety of circumstances. We are safe to conclude that for large and sparse graphs, Ints seem to be a good data structure to perform the BFS algorithm.

Another way to look at the difference between Ints and Roaring, is that for the sparse vertex sets that make up the neighborhood sets, it is the case that each of the chunks in Roaring is represented as an array of integers. This means that in these cases Roaring comes down to being practically the same as Ints, except that it has extra baggage in the form of consisting of different chunks and simply being a more complicated data structure. Because Roaring does not seem to have any benefit to having this extra baggage, it makes sense that it performs less well than Ints.

We believe that we also have to see the relative performance of EWAHB and RoaringRun16B in the same light, as the comparison with Ints. Both data structures first constructed an iterator, then they iterate through it. It seems that for the vertex sets that we are dealing with, EWAHB outperforms RoaringRun16B, especially for large graphs. Since Roaring basically only has to unpack its sparse chunks, the fact that EWAH outperforms Roaring says something about how efficient EWAH is in looping through its values.

Let us now turn our attention to the memory performance of the data structures. We consider Figure 29 and Figure 30 for their performance in respectively storing the graphs and separators. First of all, note that these figures bear a great resemblance to each other. The graph sizes for which one data structure starts to outperform another are exactly the same in both figures. This makes sense because graphs and the set of separators both consist of small vertex sets. Note that a graph is stored by for each vertex storing the set of neighboring vertices. Since in sparse graphs, vertices have a small number of neighbors, these vertex sets are small.

In both figures we observe that Ints performs best for all graph sizes. We also see that, while XBitSet performs well for small graphs, it scales worst to larger graphs, somewhat similarly to what we observed for the time measurements. Interestingly, we observe that EWAH outperforms Roaring for the smallest graphs, then for medium-sized graphs Roaring seems to outperform EWAH, and for the largest graphs we see that EWAH outperforms Roaring again.

The fact that Ints performed best has to do with the size of the vertex sets. Because these vertex sets are small, we see that Ints is a very good data structure for storing such sets. It is interesting to compare this with RoaringRun16B, which stores the sets - possibly in different chunks - as an arrays of Short numbers. Suppose, however, that the graph is small enough that RoaringRun16B only needs a single chunk to represent the vertex sets. In that case, the memory consumption of RoaringRun16B consists of a Short number for each vertex plus overhead for one chunk. To compare this we Ints, we see that Ints needs a Integer number for each vertex plus overhead for the array. The fact that we observe that RoaringRun16 consumes more memory than Ints, indicates that the overhead for storing chunks for RoaringRun16 is relatively large. So from these

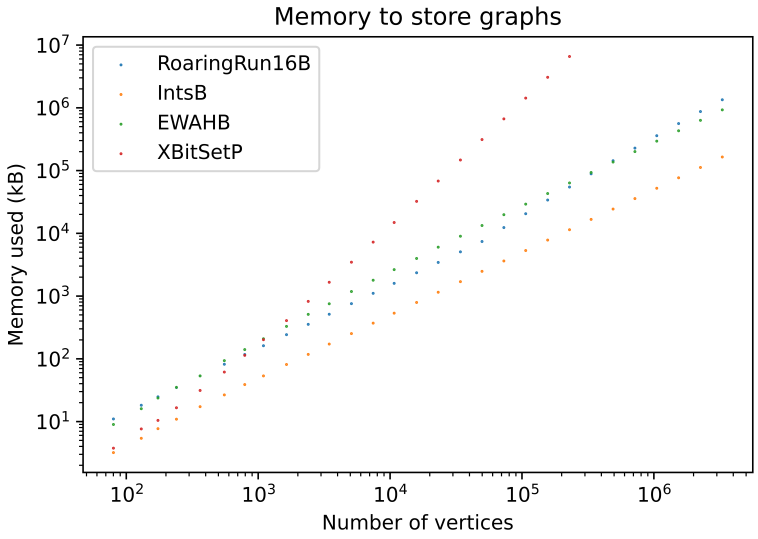

Figure 29: Memory consumption of the graphs with respect to graph size for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

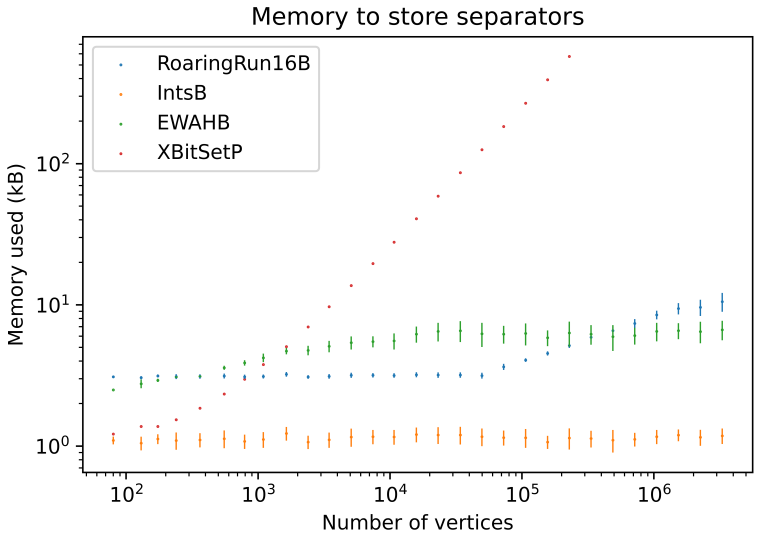

Figure 30: Memory consumption of the separators with respect to graph size for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

observations we have to conclude that Roaring is not an efficient data structure for storing very sparse sets, due to the large amount of chunk overhead. Though note, that as vertex sets get denser, RoaringRun16B will be at an advantage. Each extra vertex costs RoaringRun16B 16 bits to represent, compared to 32 bits for Ints.

The reason that Ints outperforms EWAH also has to do with the fact that the vertex sets are sparse. Suppose we have a large and sparse vertex set, then EWAH probably has to store two Long numbers for each vertex. The first Long number corresponds to the *marker word* and indicates the stretch of zeros. The second Long number stores the *literal word*, the bitmap of length 64 that contains the vertex. So for large and sparse vertex sets, we expect EWAH to need to store two Long numbers for each vertex, while Ints only has to store only a single integer number for each vertex. So we expect that EWAH consumes about a factor four more memory than Ints. When we look at the figures, this roughly corresponds to what we observe.

Note that the tipping point, after which EWAH might have to store more than one marker word for a vertex, lies at a distance between two vertices of $2^{32} = 4.294.967.296$, well beyond our reach. However, if we would have used the 32-bit version of EWAH, instead of the 64-bit version, this tipping point would have been at a distance between two vertices of $2^{16} = 65.536$. This means that in our experiments we would have encountered cases where the single vertex represented by more than two numbers. Though note, that the 32-bit version of EWAH stores the marker words and literal words as Integer numbers, instead of Long numbers. This improves its memory performance by up to a factor two. So if memory requirements are a bottleneck, it makes sense to consider 32-bit EWAH, instead of 64-bit EWAH.

The comparison between XBitSet and Ints is quite straightforward. An integer is represented by 32 bits, which would correspond to the bitmap of length 32. So if the vertex density of the set is higher than $1/32$ we have that XBitSet performs better, otherwise we have that Ints performs better. We have very sparse data sets, way sparser than a density of $1/32$, that is why we observe that Ints outperforms XBitSet. Since the amount of vertices that need to be represented grows slower than the total amount of vertices in the graph, we also observe that Ints scales better.

Comparing XBitSet with EWAH is quite interesting. For small graphs we observe that XBitSet outperform EWAH. This makes a lot of sense, because for small graphs EWAH is basically a worse version of XBitSet. In these cases EWAH typically will consist of a single marker word, followed by a set of literal words, which describes the entire vertex set. XBitSet will basically only have to store these literal words. This explains that XBitSet outperforms EWAH for small graphs. For large graphs we observe that the compression of EWAH is way more efficient than storing the entire vertex sets as regular bitmaps.

Finally, we compare EWAH with RoaringRun16B. In both figures we observe that EWAH performs best for the smallest graphs, RoaringRun16B is better for medium-sized graphs and that EWAH performs better for large graphs. The reason that EWAH starts to perform worse for medium-sized graphs, is that for

larger graphs this less likely that either two vertices are in the same literal words and two vertices are in neighboring literal words. In the first of these options two fewer Long numbers have to be stored, in the second option one fewer Long number has to be stored. This means that for larger graphs more Long numbers have to be stored, so the memory consumption grows. For the largest graphs we see that Roaring can starts to consist of more than one chunk. This simply leads to more memory consumption. Especially in Figure 30 we can clearly see that this starts to happen from a graph size larger than the chunk size of RoaringRun16B, namely $2^{16} = 65.536$. This effect leads to our observations that EWAH starts to outperform RoaringRun16 for the largest graphs.

One last thing to note in Figure 30 is the fact that the standard deviations of the data structures behave quite differently. For Ints we see that the standard deviation exists, purely due to the fact that the separators have different sizes, which is relatively independent of the graph size. We observe that XBitSet does not have any standard deviation, because it always stores the full set. For RoaringRun16B we observed that for small and medium-sized graphs the standard deviation is relatively small, while it gets larger for larger graphs. This probably has to do with the fact that for large graphs the memory consumption depends on how many chunks are instantiated, which is more subject to fluctuation for larger graphs. Finally, for EWAH we see that the standard deviation starts out small, but gets large relatively quickly. This probably has to do with the fact that the amount of Long numbers that it needs to store is subject to the distance in between those numbers, as described above.

When we compare Figure 28 with figures 29 and 30 we observe similarities. Especially for large graphs we have that the best data structures in all figures are ranked as follows: Ints, EWAH, RoaringRun16B, XBitSet. This gives an preliminary indication that there exists a relation for data structures between the memory required to store a set and the time it takes to iterate through that set. This could be interesting to investigate further.

Now let us turn to the behavior of the data structures for storing the separated components, which is shown in Figure 31, and without standard deviations in 32. We will discuss the standard deviations after the relative results. The first thing that strikes our attention is that Ints performs quite poorly. We also observe that for the smallest graphs XBitSet performs best, but that it does not scale well to large graphs. Surprisingly, we observed that EWAH and RoaringRun16B behave quite similarly. For small graphs EWAH seems to be a little bit better, for medium to large size graphs RoaringRun16B seems to be little the better, and for the largest graph EWAH seems to be a bit better. However, note that if we take variance into account the differences between EWAH and RoaringRun16B appear to be not significant.

The reason that XBitSet scales poorly is exactly the same as for the other memory measures. It needs to store the bitmap with the size of the number of vertices in the graph. The reason that Ints scales so poorly is that Ints has to store an Integer number for each vertex in the set. As separated components can contain almost the entire graph, a lot of Integer numbers need to be stored.

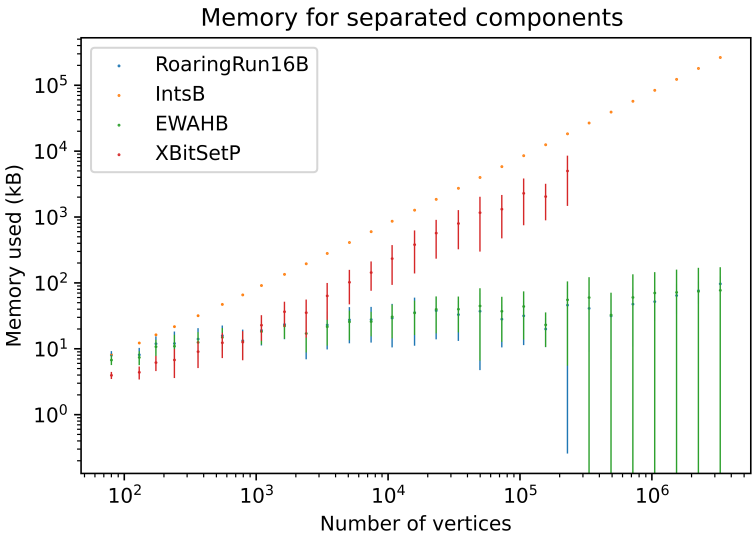

Figure 31: Memory consumption of the separated components with respect to graph size for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

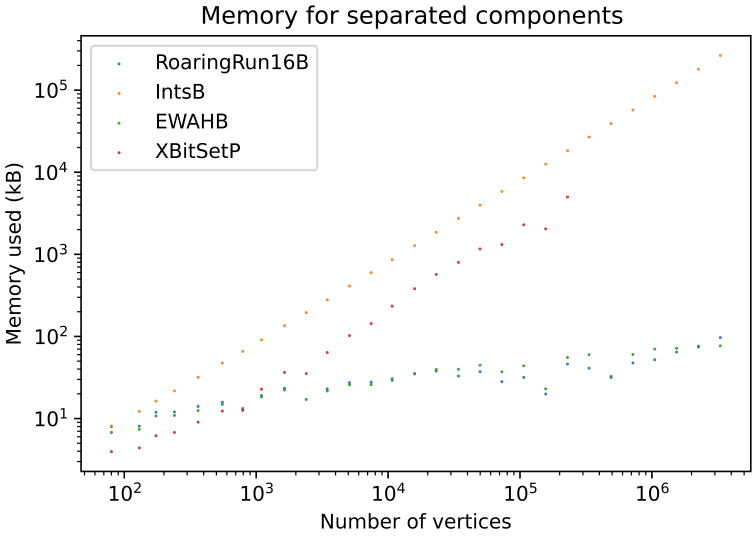

Figure 32: Memory consumption of the separated components with respect to graph size for XBitSetP, IntsB, EWAHB, and RoaringRun16B, without standard deviation shown.

The strength of bitmap compression techniques really comes into play here. When not only very sparse, but also very dense vertex sets have to be stored, which is typical for separated components, EWAHB and RoaringRun16B seem to perform relatively well. Especially when considering the scaling behavior, we observe that the amount of memory consumed for storing the largest graphs is only slightly more than the memory consumed for small graphs. This indicates that we expect EWAHB and RoaringRun16B to be able to efficiently store even larger graphs.

The fact that the results for EWAHB and RoaringRun16B behave so similarly, is somewhat surprising. As we have already compared their behavior for sparse vertex sets, we will discuss their behavior for dense vertex sets here. The result that we observe turns out to be a mix of these results.

It is insightful to start the comparison with a large vertex set that is completely full. For EWAHB this is simply stored with one marker word and one literal word, which represents the final word. So for EWAHB we have to store two Long numbers: 128 bits. RoaringRun16B stores this with RLE chunks. One such chunk consists of two shorts, indicating that the stretch of ones fills the entire chunk. So Roaring needs to store 32 bits plus overhead for each chunk.

Now let us see what happens when we remove a single vertex. For EWAHB, this generally means that it needs to store two extra Long numbers, since the stretch of ones gets interrupted somewhere. So EWAHB needs to store 128 extra bits. For RoaringRun16B, the interrupted stretch of ones gets handled more efficiently. One of the chunks gets altered, to contain two stretches of ones, instead of one, and so RoaringRun16B needs to store two extra Short numbers. So RoaringRun16B needs to store 32 extra bits.

So while EWAHB stores the full vertex set more efficiently, RoaringRun16B seems able to better handle perturbations from the full vertex set. As mentioned, the observed differences between EWAHB and RoaringRun16B do not seem significant. However, based on this reasoning we expect that as graphs get even larger - and the stretches of ones correspondingly - that EWAHB will outperform RoaringRun16B, since the chunk overhead will weigh relatively heavy on RoaringRun16B. The three largest graphs might start to capture this phenomenon, though experiments with larger graphs are needed to either accept or refute this idea.

Finally, some words about the large amount of standard deviation for these measurements. During the experiment we kept track of the amount of separated components that needed to be computed. We found that the amount of separated components between different runs of the same experiment differed by up to a factor 10. The reason for this is that when a separator resides at the edge of the graph, it typically separates one or two separators. But when it has a more central position, it can also separate a couple of separated components of very small sizes, sometimes consisting of just a single vertex.

Note that this behavior can clearly be seen in the standard deviation of XBitSetP, since XBitSetP always stores the full vertex set. The same behavior explains the large standard deviation for EWAHB and RoaringRun16B. Note that Ints does not have a standard deviation, since it basically as to store each

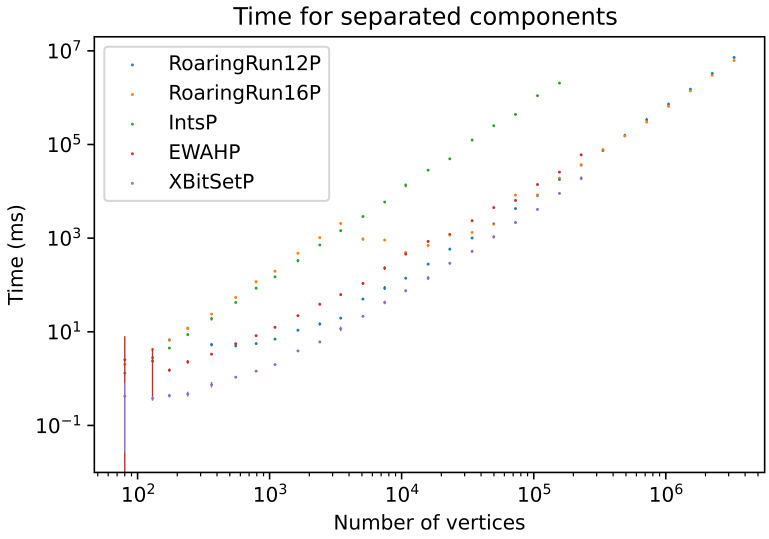

Figure 33: Time needed to compute the separated components with respect to graph size for Ints, XBitSetP, EWAHP, and RoaringRun12P, with the Parallel Algorithm.

vertex that does not reside in separator. As we can see, this turns out to be a relatively steady amount of vertices, similar to the observed standard deviation for storing the separators.

4.1.8 Time performance in the Parallel Algorithm

In this section we discuss the results of the time measurements for each of the data structures with the Parallel Algorithm. This way, we hopefully get somewhat of an indication of how well the data structures perform in computing logical operations between vertex sets. Note, that we only include the time measurements in this discussion, because they differ significantly from their BFS counterparts and add new information. In this section we compare IntsP, XBitSetP, EWAHP, RoaringRun12P, and RoaringRun16P. Note, again, that the IntsP measurements can be somewhat off, due to the fact that those measurements are performed at a later moment in time.

The results are shown in Figure 33. Note that the fact that we stopped the EWAHP computation at a graph size of about 200,000 does not seem to make sense. At the time we did this because EWAHB significantly outperformed EWAHP. Because this was less clear for Roaring, we decided to perform the experiments fully for Roaring. In hindsight, it would also have been interesting to see how EWAHP behaves for the largest graphs, especially since for the larger graphs the relative performance with respect to RoaringRun12P and RoaringRun16P seems to improve.

We observe that for large graphs XBitSetP performs best, then RoaringRun16P and RoaringRun12P, then EWAHP, and IntsP performs worst.

For small and medium size graphs we observe that RoaringRun16P performs worse than for large graphs. This can be explained by the fact that for these graph sizes - up to a graph size of 2^{12} - RoaringRun16P only makes use of sparse chunks. This means that all operations are between sets of Short numbers. This explains that for these graph sizes the performance of RoaringRun16P resembles the performance of IntsP. For graphs larger than 2^{12} we observe that RoaringRun16P also starts consisting of bitmap chunks, which are more efficient in these kinds of operations.

We conjecture that the reason that RoaringRun12P and RoaringRun16P at some point start to outperform EWAHP has to do with the fact that Roaring is more efficient in performing logical operations on vertex sets of high density than EWAH. At some point, as vertex sets get denser, EWAH will start to consists of a single marker word, followed by basically only literal words. So it will start to resemble an uncompressed bitmap. However, EWAH does have a negative trait as compared to an uncompressed bitmap. When one of the words gets to be completely full, it gets to be labeled as a fill word consisting of ones. If this happens EWAH creates a new marker word to compress that fill word. If the vertex sets get to be almost the full, as is often the case for separated components, this means that EWAH has to convert almost each literal word to a fill word. As EWAH has about $G(N)/64$ words, we believe that this process will have a significant influence on its time performance. This can explain the observed difference with RoaringP and XBitSetP.

The fact that XBitSetP performs best, is not only explained by the fact that it is efficient in performing logical operations. In the Parallel Algorithm we construct the separated components step by step, at each step adding the neighborhood of another vertex. This means that the logical operations that we perform typically happen between one vertex set of small size - the neighborhood set of a vertex, and one vertex set that ranges from very small to very large - the separated component we are constructing. If the vertex set gets large, it will go through a phase of intermediate size. The fact that XBitSet is very efficient for performing operations on random medium-sized sets, also explains the good performance of XBitSetP.

4.2 MMD Experiment

In this section we will describe the results of the experiments of the data structures on the MMD Algorithm. Similar to the Benchmark Experiment, the graphs that we considered are random partial 40-trees, with the probability of an edge $p = 0.075$, as described in section 3.2.1. For the first experiment the smallest graph that we used consisted of 79 vertices and the largest graph consisted of 304.512 vertices.

The computation time for the MMD Experiment turned out to be significantly longer than for the Benchmark Experiment. Therefore, we decided not perform the MMD Experiment for each of the data structures, for which we per-

formed the Benchmark Experiment. This way we were able to consider larger graphs. Initially, we wanted to use the data structure/algorithm combinations that performed best in the Benchmark Experiment. As described in section 4.1.7, these turned out to be IntsB, EWAHB, RoaringRun16B, and XBitSetP. Note that only for XBitSet we have that the Parallel Algorithm performed best. An important decision that we had to make, was that we decided to implement XBitSetB, instead of XBitSetP. The following paragraph describes our for reasoning for this decision. Finally, to get more insight into the behavior of Roaring, we also implemented Roaring16B and Roaring12B. Since we have that each data structure is implemented with the BFS Algorithm, we will from here on out drop the 'B' to indicate this.

Before we describe the reason that we implemented XBitSet with the BFS Algorithm, instead of the Parallel Algorithm, it is important to realize that regular MMD - without any of the improvements that we made - can straightforwardly be implemented with both the BFS and Parallel Algorithm. The computation of separated components is a key procedure in MMD, used to compute the substars. This way, MMD decides which vertex sets have to be formed into cliques and which vertices in fill graph H are LB-Simplicial. However, the computation of the separated components turned out to be the most time-consuming part of the algorithm. Therefore, we made a couple of improvements to it. Remember that these improvements were called Shallow BFS, Checking for Cliques, and Separating Substars, as described in section 3.4. These improvements made it less trivial and a larger effort to implement the Parallel Algorithm. We do believe that the Parallel Algorithm can be efficiently implemented with these improvements. However, we observed that the data structures that scale well to larger graphs seem to perform best with the BFS Algorithm and that this is the main focus of this project. So we decided not to implement MMD with the Parallel Algorithm. Note, that this probably puts XBitSet somewhat at a disadvantage. However, based on our conclusions from the Benchmark Experiment, we expect that this decision will not influence our conclusions, since the main goal is to investigate the behaviour for large graphs and XBitSet did not perform well for large graphs.

In the experiment we compared the data structures on three measures. The time it takes to finish the algorithm, the memory consumption to store fill graph H, and an indication of the maximum required memory during the algorithm. Note that it is also interesting to consider the memory requirements to store the original graph G. However, since the graphs coincide with the Benchmark Experiment, we have already discussed these results. So for this discussion we refer to section 4.1.7.

For each combination of graph and data structure we performed 15 sub experiments. In each of these experiments we shuffled the vertex IDs randomly, to include variance in the measurements. This way, the order in which the vertices get eliminated is different each time, resulting in different memory and time behaviour.

Similarly to the Benchmark Experiment, it is important to note that the memory measurements are deterministic in nature. Repeating a measurement

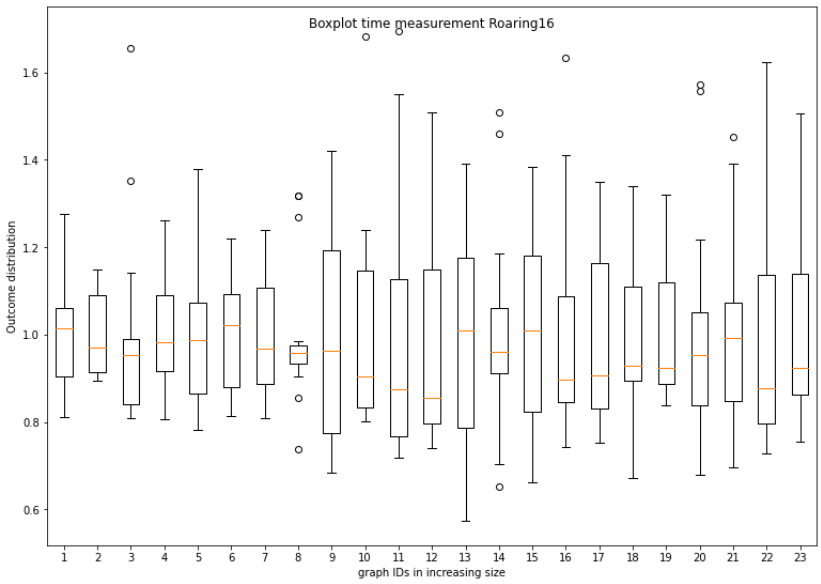

Figure 34: Boxplot for the time measurements for Roaring16, scaled to an average value of one.

gives exactly the same results. Again, the time measurements are not deterministic, due to the behaviour of the server.

In the implementation of the experiments, we made as much use as possible of an abstract class. This way we let the different data structures share as much code as possible. We were able to perform a large part of the experiment and all the measurements in this class. About 25 functions had to be implemented separately for each of the data structures.

It took about 17 days to perform the MMD Experiment. In the coming sections we will first describe the time measurements and then the memory measurements. Similarly to the Benchmark Experiment, we first compare the different versions of Roaring. Then we compare the best version of Roaring with the other data structures.

4.2.1 Time measurements

Before we discuss the results, we first take a look at the boxplots of the time measurements. For brevity we only consider Roaring16, Ints, XBitSet, and EWAH in this discussion. The boxplots are shown in Figure 34, 35, 36, and 37. Note that, similarly to the Benchmark experiment, the values in the boxplot are scaled to the mean outcome, which leads to an average boxplot value of one, and that the boxplots are ordered from left to right, such that the graph size increases.

Similarly to the benchmark experiment, we observe that the typical size of

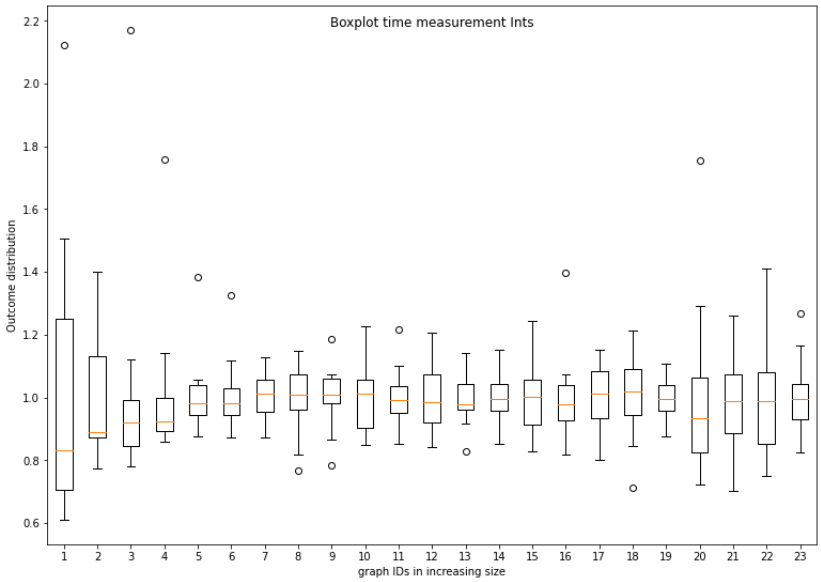

Figure 35: Boxplot for the time measurements for Ints, scaled to an average value of one.

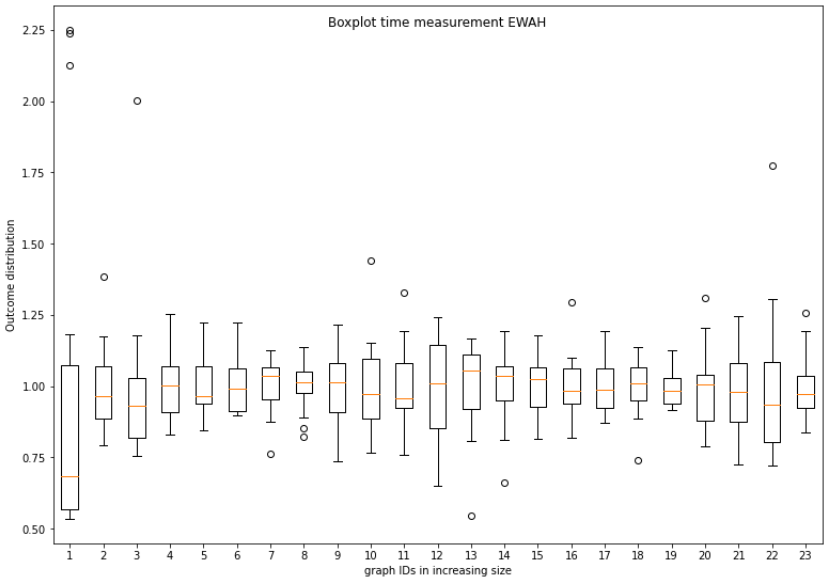

Figure 36: Boxplot for the time measurements for EWAH, scaled to an average value of one.

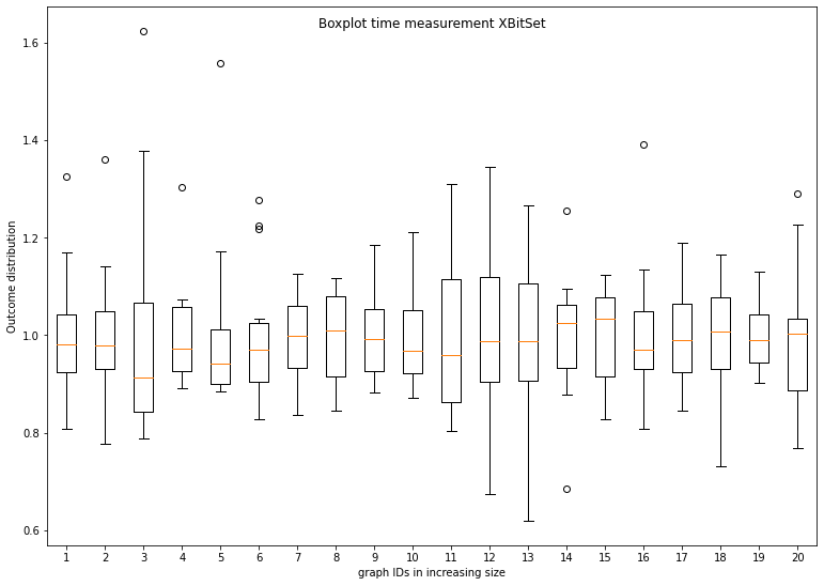

Figure 37: Boxplot for the time measurements for XBitSet, scaled to an average value of one.

Q_1 and Q_3 is off from the mean by about a factor five to twenty percent. This behaviour seems to be independent of the graph size. This indicates that the measurements are somewhat stable, as we will later see when we consider the figures in which we compare the data structures.

It is important to note that the MMD experiment considers a smaller range of graphs. This means that the smallest graphs in these boxplots correspond to a larger computation time, when compared to the smallest graphs in the boxplots of the Benchmark Experiment. This explains that we observe that the outliers for the smallest graphs are less spectacularly large. For the smallest graph, the largest outliers are for EWAH. They get as large as a factor 2.25 larger than the mean. Note that this is considerably smaller than what we observed for the Benchmark Experiment, where the outliers for the smallest graph were as much as a factor 10 larger than the mean. Since we saw that the outliers for the smallest graphs in the Benchmark Experiment were so large, we figured that they probably came into existence due to the server behaviour, so we decided to remove them. Since we do not observe such extreme outliers in the MMD Experiment, we do not have a clear indication that these outliers need to be removed.

However, we observe that the outliers seem to be independent of the graph in question, because different data structures show outliers for different graphs. For this reason, we believe that these outliers are not intrinsic to the problem instances, but come into existence due to the server. This could also explain the fact that - similarly to the Benchmark Experiment - we observe a much larger

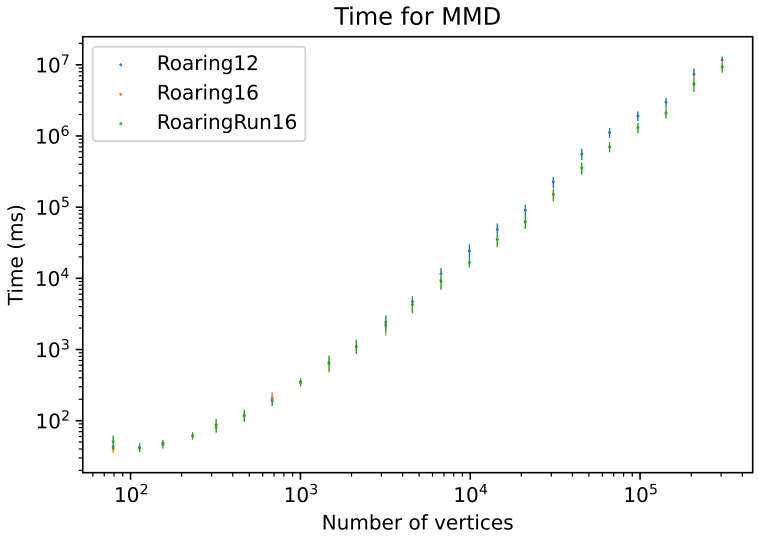

Figure 38: Time needed to perform MMD with respect to graph size for Roaring12, Roaring16, and RoaringRun16.

portion outliers being too large, then too small. For these reasons we decided to remove the outliers from the results.

The results for the different versions of Roaring are shown in Figure 38 and 39, the latter without standard deviation. The first thing that we observe is the fact that Roaring16 and RoaringRun16 give practically the same results. Note that we observed the same behaviour in the Benchmark Experiment, see Figure 22.

Secondly, we see that Roaring12 starts to perform slightly worse than Roaring16 and RoaringRun16, starting from a graph size of about 7000. Again, we observe the same behaviour for the Benchmark Experiment, as shown in Figure 16. Note that in the benchmark we started to observe a significant difference starting at a graph size of about 5000, slightly earlier. This difference can be explained by the fact that we have a larger standard deviation for Roaring in the MMD Experiment, than in the Benchmark Experiment.

Finally, remember that in the Benchmark Experiment we observed that for the largest graph sizes, the different versions of Roaring converged in their running time. In the MMD Experiment it seems that we start to see the same behaviour, as would be expected, since the same principles hold. However, we do not have enough evidence to conclude on this. Experiments with larger graph sizes are needed to accept or reject this hypothesis.

The results for Ints, EWAH, XBitSet, and RoaringRun16 are shown in Figure 40 and 41, the latter without standard deviation. Note that we have shown the

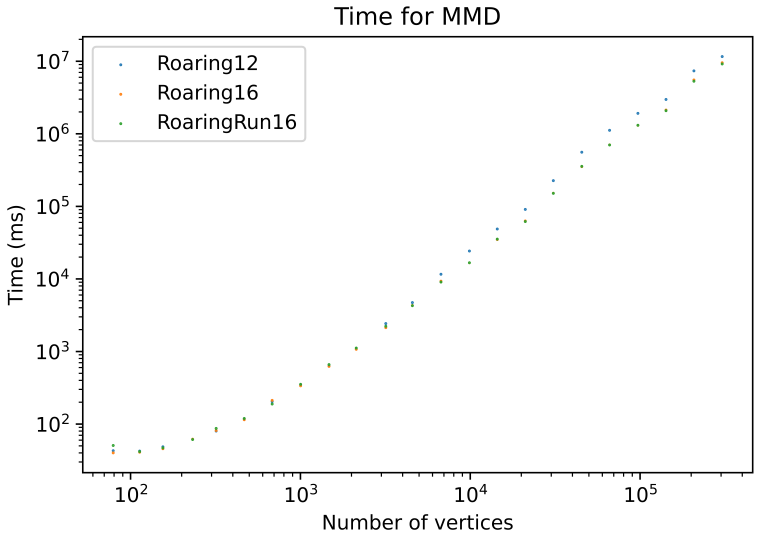

Figure 39: Time needed to perform MMD with respect to graph size for Roaring12, Roaring16, and RoaringRun16, without standard deviation.

version of Roaring that performs best. Also note, that we did not perform the experiment for XBitSet for the largest graphs. Since XBitSet performed significantly worse for large graphs than its competitors, we decided to spend more computation time on the EWAH, Ints, and Roaring. This way, we were able to perform the experiment for larger graphs.

Again, we observe a couple of the same patterns as we saw in the Benchmark Experiment. Ints seem to perform best, followed by EWAH and RoaringRun16, which perform quite similarly, and XBitSet performs the worst. For XBitSet, we see that for small graphs it does perform well, but that it scales worse than the competitors.

It is important to note that XBitSet could probably be improved in two ways. First of all, XBitSet performed better with the Parallel Algorithm. Secondly, XBitSet could be implemented to make use of dynamic memory allocation, probably saving computation time. Since most of the computation time is spent to check whether fill graph H is chordal and we know that H is denser than original graph G, we expect that the improvement for using $XBitSet()$ instead of $XBitSet(G(N))$ will be smaller, than we expected for the Benchmark Experiment. Still, these possible improvement will likely lead to better performance for XBitSet. Though, for the largest graphs we expect this improvement not to be large enough to be able to compete with the other data structures. Especially, since we observed that in the Benchmark Experiment XBitSetP outperformed XBitSetB most drastically for small graphs. For larger graphs the difference seemed to shrink. Taking this into account, however, leads us to the believe that XBitSet seems to be the fastest data structure for MMD on small graphs,

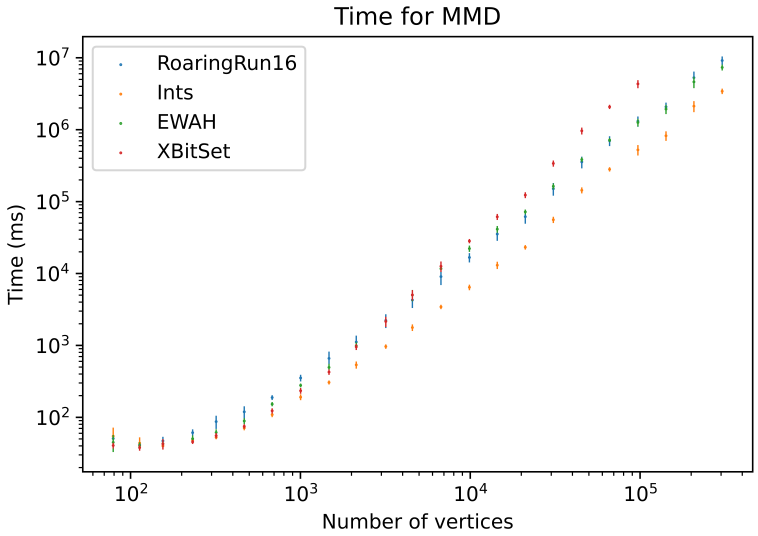

Figure 40: Time needed to perform MMD with respect to graph size for Ints, EWAH, XBitSet, and RoaringRun16.

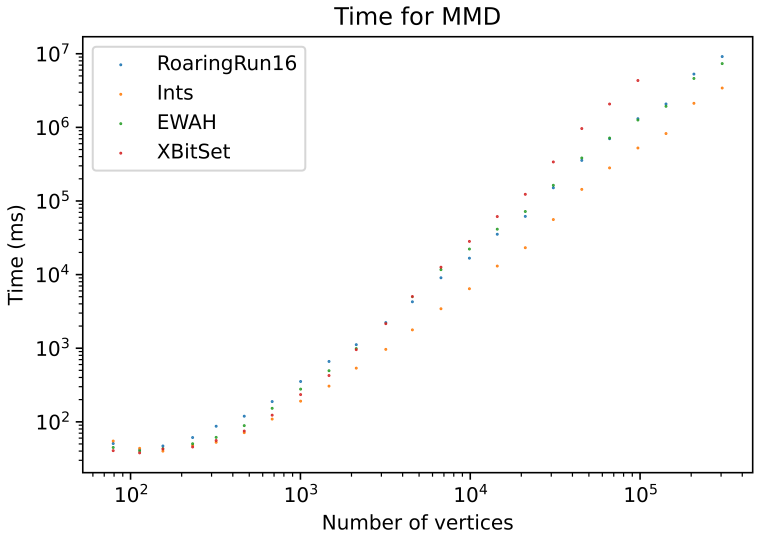

Figure 41: Time needed to perform MMD with respect to graph size for Ints, EWAH, XBitSet, and RoaringRun16, without standard deviation.

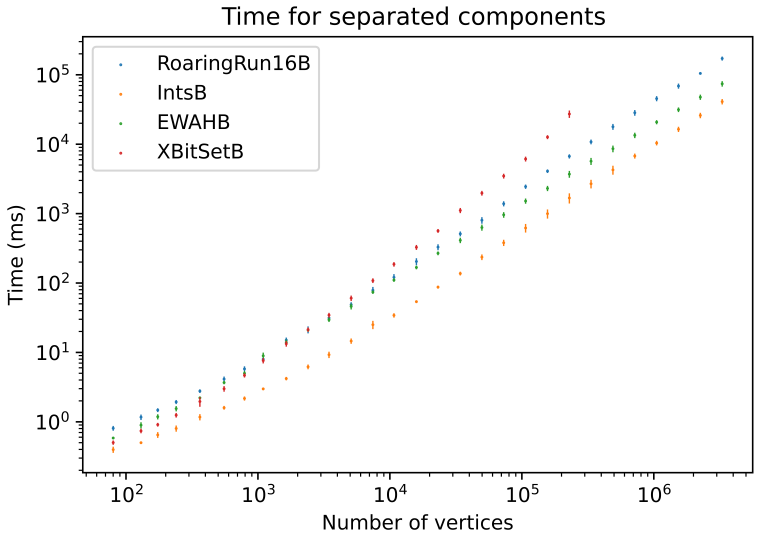

Figure 42: Time needed to compute the separated components in the Benchmark Experiment with respect to graph size, for Ints, EWAH, XBitSet, and RoaringRun16. Each of the data structures make use of the BFS Algorithm.

similarly to the Benchmark Experiment.

It is interesting to compare the results with the results for the Benchmark Experiment, in which each algorithm uses the BFS Algorithm. We show this in Figure 42. Note that, in style accordingly to the Benchmark Experiment, the data structures are denoted as IntsB, XBitSetB, EWAHB, and RoaringRun16B. We observe that the results seem to correspond quite nicely to each other. First of all, we observe that the graph size at which RoaringRun16 and EWAH start to outperform XBitSet corresponds to the MMD Experiment, namely for graphs of size about 2500 to 3000.

However, we also observe two main differences. The first is that we seem to observe a significant difference between the two experiments for the smallest graphs. In the Benchmark Experiment we observe that the data structures behave significantly different. This is not the case in the MMD Experiment. There we observe that the differences between the data structures do not seem significant. We also get an indication that Ints performs worst on small graphs in the MMD Experiment, while we observe that in the Benchmark Experiment Ints performs best for those graphs.

Before we explain this behaviour, we consider Figure 43, which indicates the relative difference in the time performance between the Benchmark Experiment with the BFS Algorithm and the MMD Algorithm. Here we observe that, as noted, the relative performance for small graphs is relatively poor, since we expect the MMD Algorithm to scale worse to larger graphs than the BFS Algorithm. So we would expect a monotonically increasing function.

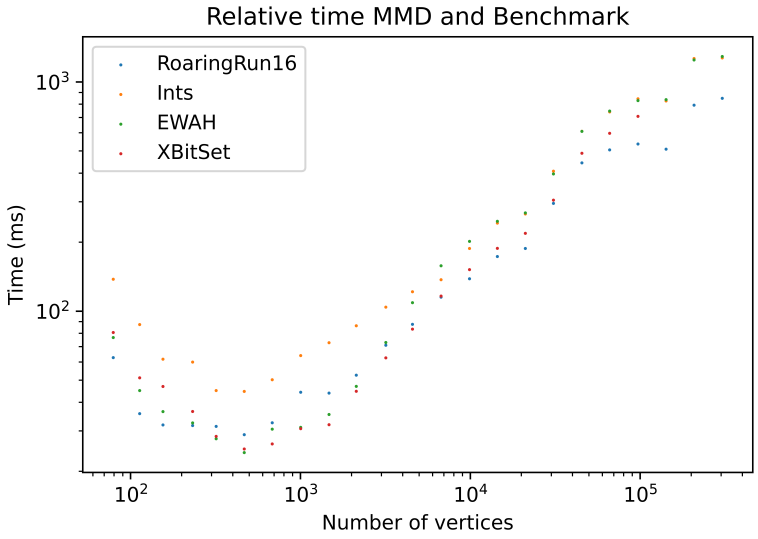

Figure 43: Relative time performance between the Benchmark Experiment with the BFS Algorithm and the MMD Experiment, with respect to the graph size, for Ints, EWAH, XBitSet, and RoaringRun16.

The reason that we observe this behavior has to do with the fact that in the MMD Algorithm, a wider range of behaviour is included in the experiment. The data structures do not only have to perform the part of the algorithm, in which they compute substars. Also, setting up the algorithm is measured, as well as the set up for round 2. In the set up, the elimination graph G' and the fill graph H have to be copied from the original graph G and cast to the data structure in question. In this set up for round 2, this basically gets repeated for those vertices that are not yet LB-Simplicial. Note that round 2 is a relatively small procedure, since only a small fraction of the vertices turns out not to be LB-Simplicial after round 1. For small graphs, the set up determines up to 85% of the running time of the algorithm, as can be shown in Figure 44. For large graphs, however, eliminating the vertices and checking for chordality seems to be dominant in determining the running time. So we have that for small graphs, the operation performed in the MMD Experiment are significantly different from the operations of the Benchmark experiment. This explains that for small graphs the two experiments give different results.

Note that Figure 44 does not give an indication of which data structure performs best in these operations, since the data points are relative to the total running time. Their absolute performance in is shown in Figure 45. Note that for small graphs the results seem indecisive. For large graphs we observe that Ints performs best, followed by EWAH, RoaringRun16, and XBitSet.

The second main difference that we observe between the MMD Experiment and the Benchmark Experiment, has to do with the relative performance of

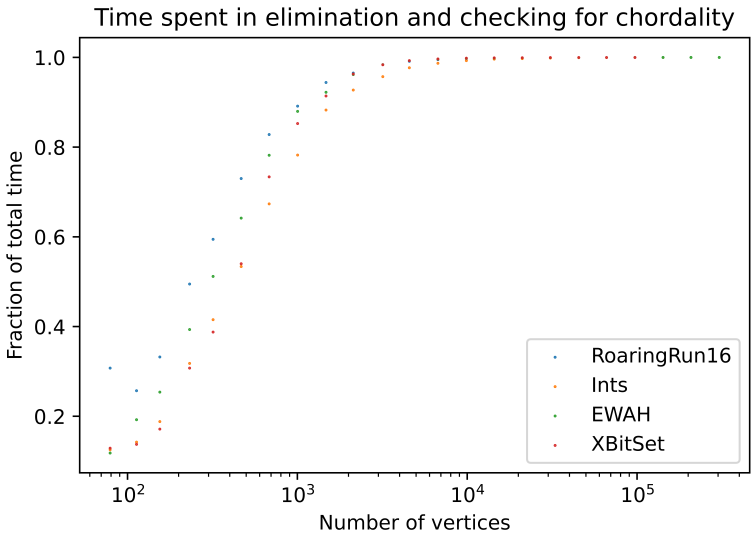

Figure 44: Fraction of the running time that is spent in the part of MMD in which we eliminate the vertices and check for chordality, for increasing graph size. We consider Ints, EWAH, XBitSet, and RoaringRun16.

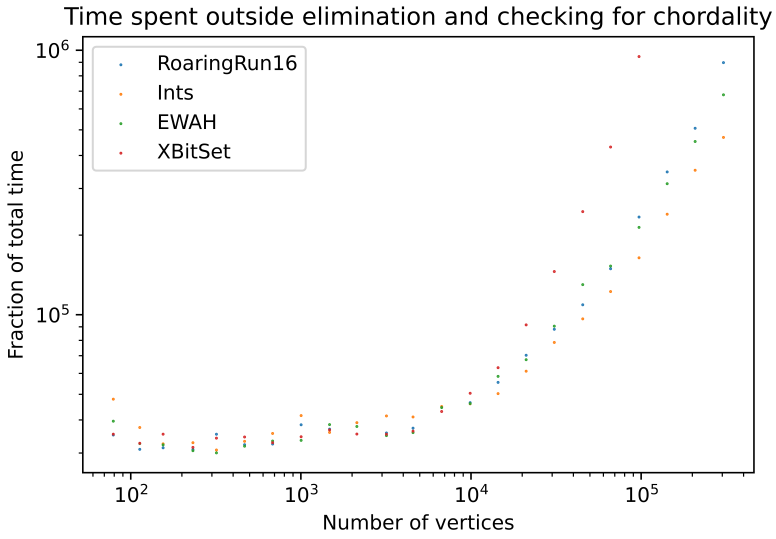

Figure 45: Running time that is spent on setting up the graphs, for increasing graph size. We consider Ints, EWAH, XBitSet, and RoaringRun16.

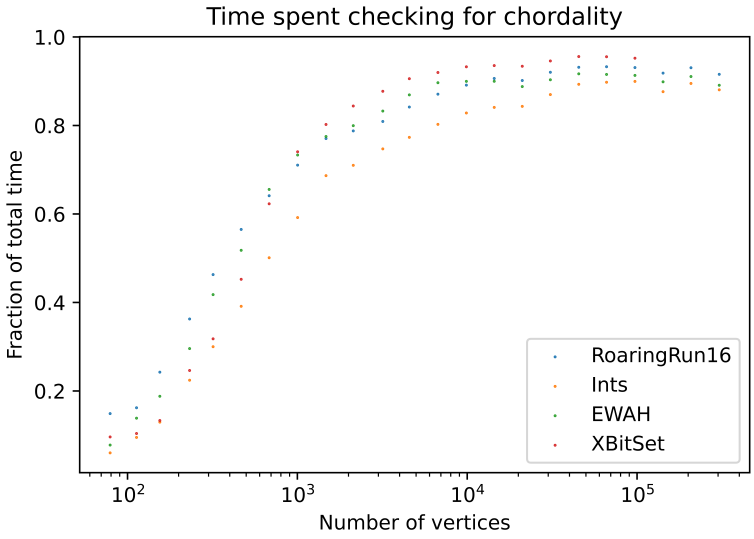

Figure 46: Fraction of time that is spent to check whether the graph is chordal, for Ints, EWAH, XBitSet, and RoaringRun16.

the data structures. In Figure 43 we observe that relatively to Benchmark Experiment Ints performs worst, followed by EWAH, XBitSet, and RoaringRun16 performs relatively best.

The fact that these difference are significant can be shown by comparing Figure 42 with Figure 40. In the Benchmark Experiment we observe that EWAH starts to outperform RoaringRun16, starting from a graph size of about 20.000. However, in the MMD Experiment we do not observe this behavior. Instead, we observed that for graphs of size of around 10,000, RoaringRun16 seems to outperform EWAH. For large graphs in the MMD experiment, we do see that the difference becomes smaller and then at some point EWAH starts to outperform RoaringRun16 slightly. However, the evidence does not seem strong enough to support the conclusion that EWAH scales better to large graphs than RoaringRun16. Further research on larger graphs would be needed to verify this behavior.

We conjecture that the difference between the data structures for the different experiments has to do with the graph density. Figure 46 shows that for large graphs, the algorithms spends about 90% of the computation time to check whether the graph is chordal. When we measured fill graph H, we found that it contains an amount of edges of about a factor 2 larger than original graph G, indicating that MMD adds that a lot of edges to triangulate the graph. This means that in 90% of the time, the data structures have to deal with a graph that is about twice as dense, than is the case in the Benchmark Algorithm. This means that each data structures typically have to deal with data sets that are about twice as dense.

XBitSet seems to perform relatively well, as we would expect. XBitSet does not need to store any extra information and it only has to iterate through extra bits that are set to '1'.

It is important to note that, even though the vertex sets are twice as dense, we are still dealing with relatively sparse vertex sets. For EWAH, when considering large graphs, this means that each vertex set contains up to twice as many marker and literal words. It has to contain twice as many vertices and the larger graphs get, the more unlikely it is that those vertices reside in the same or a neighboring literal word. This means that creating vertex set iterators and iterating through them will take longer. Especially interpreting the extra marker words can have a significant impact. Note however, that for small graphs we observe that EWAH performs relatively well. This is because in this case, it is likely that the extra vertex can be stored in an already existing literal word. EWAH handles this efficiently.

For RoaringRun16 we observe the least of an impact. As long as RoaringRun16 represents the vertex sets with a single chunk, so if the graph size is smaller than $2^{16} = 65.536$, we have that that for each extra vertex we only have to include an extra Short number in the chunk. Note that this only holds if the vertex set is sparse enough to be represented by an array of integers, which is the case for sparse graphs such as H. However, if new chunks have to be created, their creation and iterating through them will cost extra computation time. This will start happening for graphs larger than 2^{16}. This could explain the fact that in Figure 40, RoaringRun16 seems to start performing somewhat worse relatively to EWAH for the largest graphs. Note, that this effect could be a factor that starts to have a significant impact on the performance of Roaring as graphs grow even larger. We get an indication for this in the results of the Benchmark Experiment, in which we observe that for the largest graphs EWAH scales better than RoaringRun16.

We seem to observe that Ints is burdened most by this. For each extra vertex, Ints needs to iterate through one extra Integer number, which seems relatively costly. However, we believe that this is not true. We believe that the relative performance between the experiments for the others is better, not because they deal worse with more dense vertex sets, but because the others perform relatively poorly in the Benchmark Experiment.

Another way to look at this is to, once again, consider Figure 42. Note that the fact that the ratio between the number of edges and vertices is about constant for all graphs, leads to the another insightful property of this figure. Namely that the vertex sets that the data structures are dealing with, are denser for small graphs and sparser for large graphs. Since vertices have about six neighbors on average, this means that for a graph of size a hundred the typical vertex set has a density of about six in a hundred. For a graph of size a million, the typical vertex set density will be about six in a million. For Ints, this sparsity does not have any influence on its performance, since it only has to iterate through the set of integers and because we observe in Figure 42 that Ints performs well for both large and small graphs, so for both sparse and dense vertex sets. So we have to conclude that Ints performs well for relatively dense

vertex sets as well.

It is also interesting to note how the other data structures are expected to be effected by sparsity. XBitSet is bothered the most, since it has to iterate through all the extra bits which are not present. EWAH and Roaring scale better with more sparsity. EWAH simply skips the stretches of zeros, and Roaring behaves similarly to Ints, as long as vertex sets are sparse enough and it does not need to instantiate new chunks.

4.2.2 Memory measurements

In this section we describe the memory measurements. We start with comparing the data structures in how well they perform in storing fill graph H, then we will say something about the maximum amount of memory required during the algorithm. Similarly to the Benchmark Experiment, we have that the memory measurements are deterministic in nature. For this reason we decided not to consider any data points as outliers.

When comparing the different of versions of Roaring, we did not observe any significant differences to the results from the Benchmark Experiment. So we skip right away to compare the behaviour of Ints, EWAH, XBitSet, and RoaringRun16. We start with the discussion of memory required to store fill graph H, as shown in Figure 47. We compare the results with the graph measurements in the Benchmark Experiment, as shown in Figure 29. Note that these are the same graphs that the MMD Experiment is performed on. Finally, we also consider Figure 48, which shows the relatively performance of the data structure in storing H and G. Remember that the main difference between G and H is that the latter is more dense, by about a factor 2.

We observe that for large graphs Ints performs best, followed by respectively RoaringRun16, EWAH, and XBitSet. We observe two main differences in storing H, with respect to storing G.

The first thing that stands out is that XBitSet performs relatively well. For a larger number of small graphs XBitSet seems to perform best. This is due to the fact that the other data structures are negatively influenced by the fact that H is more dense. For XBitSet this is not the case, since it simply stores one bit for each vertex in the graph, irrespective of the amount of vertices in the set. This can also clearly be seen in Figure 48, where the data points of XBitSet have values close to one.

Note that it is surprising that these data points are not exactly equal to one. We conjecture that this has to do with the measurement process. To make sure that we did not measure all kinds of other objects, created during the algorithm, we measured H by making a copy of it and subtracting the free memory before from the free memory after its recreation. Perhaps this leads to a situation in which we have that G is more efficient in memory sharing between the different vertex sets than H is. However, we are not sure about this reasoning.

The second noteworthy difference with the measurement of G is that RoaringRun16 performs relatively well. When storing G, we observed that RoaringRun16 outperformed EWAH up to a graph size of about 300.000, after which

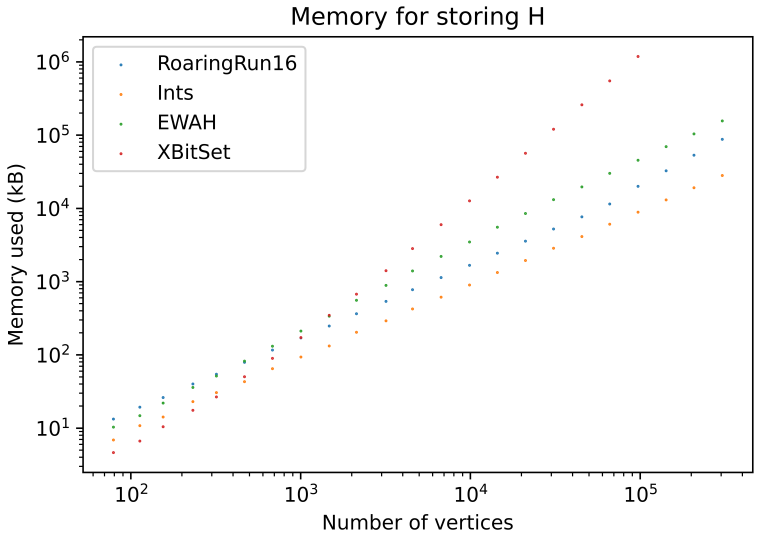

Figure 47: Memory consumption to store H with respect to graph size, for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

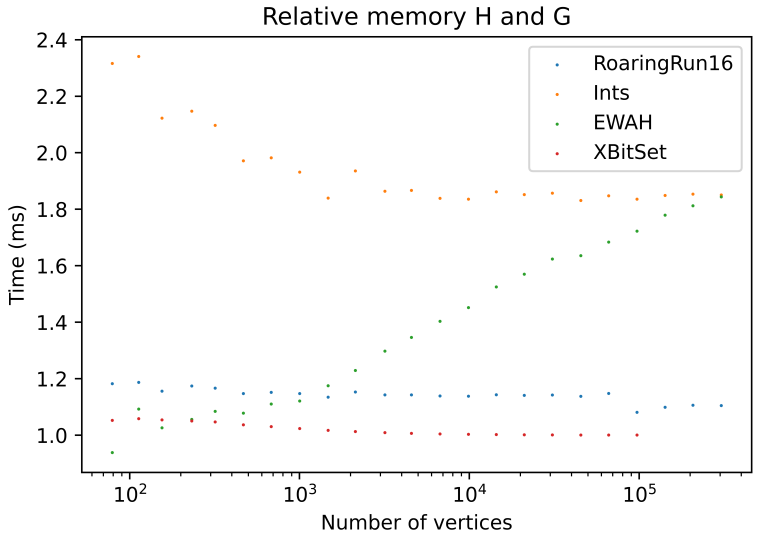

Figure 48: Relative memory required for storing H and G, with respect to the graph size, for Ints, EWAH, XBitSet, and RoaringRun16.

EWAH started to outperform RoaringRun16. In storing H we observe that even for graphs of size 300.000, RoaringRun16 still significantly outperforms EWAH.

The reasoning for this difference is the same as we saw with the relative time performance in the MMD Experiment between EWAH and RoaringRun16. There we observed that, since the vertex sets are sparse, there is a difference between them in what happens when they have to store an extra vertex. As graphs get larger, we observe that for EWAH it gets more like that it has to store another marker and literal word for each added vertex. In these cases it has to store two extra Long numbers, or 128 bits. This is also the reason that for even larger graphs EWAH will probably to start to scale worse than Ints.

RoaringRun16 typically has to store one extra Short number, as long as it is not the case that a new chunk needs to be instantiated. This means that RoaringRun16 has to store an additional 16 bits. This explains the fact that RoaringRun16 performs relatively well in the MMD Experiment. Though also note that the fact that RoaringRun16 has to store twice as many vertices, only results in a factor 1.2 more memory required. This indicates that the chunk overhead consumes about 80% of the memory when storing G, and about 67% of the memory for storing H. This leads us to believe that as graphs get larger, the relative memory performance of Roaring will further improve.

Finally, we know that Ints has to store 32 extra bits for each extra vertex, and we observe that this scales relatively poorly. This indicates that at some density, higher than we observe here, XBitSet and RoaringRun16 will start to outperform Ints. This corresponds to the observations that we made of the measurements of the separated components in the Benchmark Experiment. For these dense vertex sets, Ints performed significantly worse than the others.

To get an indication of the maximal amount of memory required, we had to resort to sub optimal measurements. The measurement of the maximum memory would need to take place deep inside the BFS part of the algorithm, since that is the moment that the most memory is needed. However, if we would perform the measurements in this part of the algorithm, we would need to perform the memory measurement quite frequently to get a good indication of the maximum memory. However, we quickly found out that this would greatly influence the time requirements of the experiment. When performing the measurements about $G(N)$ times, this would typically make the algorithm slower by a factor 10. When considering the Ints data structure, measuring the memory 10 times for a graph of size 1000, still meant that the memory measurements consumed about half of the computation time. Because we also want to be able to properly distinguish between the data structures for graphs of size 1000, we decided to go with the following approach.

First of all, we measured the maximum amount of memory only 20 times: 10 times in the part of the algorithm in which we eliminate vertices and 10 times in the part of the algorithm in which we check whether the graph is chordal. We also decided not to measure the actual maximum memory, since that would have to be performed deep inside the BFS. Instead, we performed the measurements right after the BFS. This way, we were able to measure the largest amount of

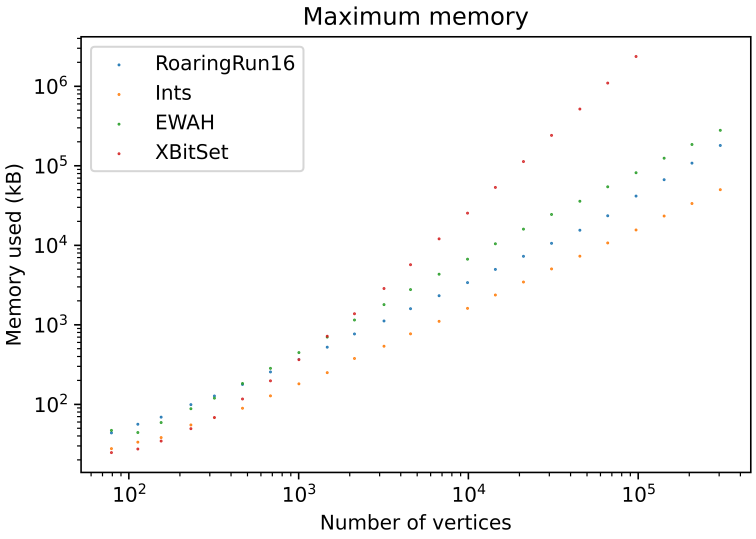

Figure 49: Maximum memory consumption with respect to graph size, for XBitSetP, IntsB, EWAHB, and RoaringRun16B.

objects, that needed to be stored anywhere in the algorithm, outside of the BFS. It also meant that the measurements did not influence the functioning of the BFS. To make sure that the memory measurements did not influence the time measurements, we subtracted the time it takes to perform the memory measurements from the elapsed time.

It is important to note that the memory measurements are not optimized. The neighborhood sets of the fill graph G' remain after elimination, while they could also be deleted. So we overestimate the maximum amount of memory required at the specified location in the algorithm. However, we expect that these measurements do give an indication of how the data structures perform during the algorithm.

The results are shown in Figure 49. Note that these results are basically the same as we observed in Figure 47, which shows the memory to store fill graph H. We explain this by the fact that the largest memory requirement of MMD is to store the graphs. MMD has to store both G' and H, which are both denser graphs than G. So in this measure we seem to capture the memory needed to store both G' and H, when they are relatively dense, so somewhere at the end of round 1 of the algorithm. The claim that we capture both G' and H at a relatively dense time is supported by the fact that Figure 49 seem to have more resemblance to Figure 47 - which shows the memory required to store the relatively dense graph H, than with Figure 29 - which shows the memory required to store the relatively sparse graph G.

5 Conclusion and Discussion

The main goal of this project has been to investigate bitmap compression techniques for treewidth algorithms on their performance on large graphs. The bitmap compression techniques we considered were EWAH and Roaring Bitmaps. We compare these bitmap compression techniques with the data structures that are commonly used in context of graphs, namely the sorted array of Integer numbers - denoted as Ints - and the regular, uncompressed bitmap. For the latter we used prof. Tamaki his XBitSet implementation. For Roaring Bitmaps we considered multiple implementations, which differed in their so-called chunk size and whether they made use of RLE to represent chunks.

Another goal of the project was to investigate whether the use of these data structures could improve the performance of prof. Tamaki his HBT algorithm. Since HBT is quite a complicated algorithm, we did not implement the data structures in HBT. Instead, we conducted two experiments, the Benchmark Experiment and the MMD Experiment. Both experiments were performed on randomly generated partial 40-trees.

In the Benchmark Experiment, we simulated the most time-consuming operation in HBT, as well as the typical vertex sets that is has to store. We investigated the computation of the components separated by a set of separators. The separators we used, corresponded to bags in the tree decomposition, generated by the Minimum Degree Algorithm. To make sure that we got a broader insight in how the data structures dealt with such a task, we compared their performance in two different algorithms to compute the separated components. The first made use of a regular Breadth First Search algorithm, the second computed the separated components by making use of many logical operations. We compared the data structure/algorithm pairs on four measures. The time it takes to compute the separated components and the memory required to store the graph, the set of separators, and the set of components that the separator separate. This way we not only obtained a sense of which data structures are efficient time wise, we also get a sense of which data structures are efficient for storing different types of vertex sets.

In the MMD Experiment, we compared the data structures when they were implemented in the MDD algorithm [5]. MMD makes heavy use of the computation of so-called substars, the neighborhood of the components, separated by a vertex with its neighborhood. Note that this operation corresponds closely to the one we studied in the Benchmark Experiment. We compared the data structures on three measures: the time it takes to complete the algorithm, the amount of memory to store the chordal graph, associated with the tree decomposition, and an indication of the maximum memory required during the execution of the algorithm. One of the main questions that we wanted to answer, was whether the performance of the data structures in the Benchmark Experiment could give an indication of their performance in a real world algorithm, that performs a similar computation.

We will first recapitulate our findings on the different implementations of Roaring, then our comparison between Roaring, EWAH, XBitSet, and Ints. We

continue with a couple of words on the predictive behaviour of the Benchmark Experiment, then briefly discuss the graphs that we used. Finally, we will close with some recommendations for future work.

5.1 Roaring Bitmap

When comparing the different version of Roaring, we observed that RoaringRun16 - the version of Roaring with chunk size 2^{16} and which makes use of run optimization - performed best. Both in the Benchmark Experiment and the MMD Experiment, we found that it either outperformed or had similar results as the other versions for almost all of the measures. Only, RoaringRun16P - where the P indicates that it is implemented in the Benchmark Experiment with the Parallel Algorithm - performed poorly. This indicates that RoaringRun16 seems to be the best version of Roaring for storing vertex sets of any kind and performing a regular Breadth First Search. However, for performing many logical operations - as in the Parallel Algorithm - RoaringRun16 does not perform well.

Interestingly, we found that Roaring12P and RoaringRun12P were able to outperform RoaringRun16 in the time performance in the Benchmark Experiment, for a small region of graphs. For graphs of size between 800 and 5000, Roaring12P and RoaringRun12P outperformed both RoaringRun16B and RoaringRun16P. This indicates that if many logical operations need to be performed on small graphs, it could be worthwhile to consider a version of Roaring with smaller chunk size. Here, the chunk size that performs best for a given graph seems to depend on the graph size. We found that Roaring performed best with the Parallel Algorithm for graph sizes, such that it frequently makes use of bitmap containers and before more than one chunk is needed to represent the vertex sets. So RoaringX - where X indicates the chunk size - performs best for graph sizes between 2^{X-4} and 2^X.

It is important to note that RoaringRun16 gave the same results as Roaring16 for all measures in both experiments, except for one. For storing sparse vertex sets they gave similar results, as well as for their time performances. Only for storing the separated components in the Benchmark Experiment, RoaringRun16 outperformed Roaring16. These separated components are frequently almost completely full. This indicates that when very dense vertex sets need to be stored on a large scale, RoaringRun16 is a better option than Roaring16. If this is not the case, we did not observe a significant difference between RoaringRun16 and Roaring16.

However, we do expect to observe another difference between RoaringRun16 and Roaring16, namely for performing logical operations between dense vertex sets. In our experiments the vertex sets that the algorithms had to work with largely corresponded to the neighborhood of vertices. The vertices have a low number of neighbors. Since converting sparse and unstructured sets to run length encoding does not to lead to improvements in memory consumption, this did not happen. The fact that these vertex sets are not converted to RLE chunks, basically meant that we get more or less the same computations for Roaring and RoaringRun. Note that this fact also holds for the performance

in the Parallel Algorithm in the Benchmark Experiment. During the Parallel Algorithm, we did not convert the separated components to consist of RLE chunks. To save time, we only did this afterwards. So we did not perform any logical operations on RLE chunks. However, we do expect to observe a difference in computation time for logical operations between bitmap chunks and RLE chunks. It would be interesting to investigate this behaviour further.

The reason that we observed that Roaring16 and RoaringRun16 performed best has to do with the fact that all versions of Roaring seemed to start behaving relatively poorly, as soon as they reach the threshold at which they start consisting of more than one chunk. So what we observed is that the time it takes to iterate through a set and the memory required to store a set, seems to depend on the number of instantiated chunks. For a similar reason we observed that the behavior of different versions of Roaring converged for large graphs. As graphs get larger, more and more chunks are not instantiated, since chunks are only instantiated if at least one value in that chunk is present. Since we have that this effect is larger for versions of Roaring with smaller chunk size, this effect turns out to converge the results for the different versions of Roaring. Though note, that this is largely due to the fact that we are dealing with very sparse vertex sets. For dense vertex sets we could observe differently.

5.2 Roaring, EWAH, Ints and XBitSet

We start our discussion with the best performance of the data structures. Note that by 'best performance', we mean that we consider the best versions and algorithms for each data structure in the Benchmark Experiment. Though note that this also depends on the implementations that we used. Different implementations could provide different results.

In both experiments the general picture of the best performance of the data structures seemed to be about the same. For large and medium sized graphs, Ints performed best, followed by both EWAH and Roaring, and XBitSet performed worst. In the Benchmark Experiment, we observed that EWAH outperformed Roaring, and in the MMD Experiment we observed oppositely. It is also important to note that, for small graphs we found that XBitSet seemed to perform best. Though, we also observed that XBitSet was not able to deal with the largest graphs, due to the amount of memory needed to have the graphs in memory.

We observed this pattern for nearly all investigated measures: the computation time and the memory required for storing the graphs G and H, the set of separators in the Benchmark Experiment, and the maximum amount of memory consumed during the MMD Experiment. Note that each of these memory measures compare the data structure in their performance on sparse vertex sets. For storing the separated components, which also included dense vertex sets, we observed behaviour that broke with the pattern. Here, Ints performed worst, followed by XBitSet, and both EWAH and Roaring seemed to perform best. This is the first evidence that leads us to conclude that the choice of data structure depends on the vertex sets, it has to represent.

Now, let us first consider the time performance of the data structures in combination with the different algorithms in the Benchmark Experiment. First of all, Roaring, EWAH and Ints performed best with the BFS Algorithm and XBitSet performed best with the Parallel Algorithm. It is interesting to explore the behaviour of the data structure in combination with the Parallel Algorithm a bit more, since we observed different behaviour than when we compared the data structures at their best. We observed that Ints performed very poorly. The others performed better, first EWAH, then Roaring, and XBitSet performed best.

In the Parallel Algorithm, the running time is determined by the speed of performing logical operations between vertex sets. These vertex sets range from very sparse to very dense. For both in the Benchmark Experiment with the BFS Algorithm and the MMD Experiment, however, we have that iterating through the vertex sets, that correspond to the neighborhood of vertices, determines the running time. Note that in these cases we were typically dealing with very sparse vertex sets. This leads us to conclude, that besides the typical vertex sets that the data structures have to deal with, also the typical operations that the data structure have to perform are an important factor in determining which data structure to use. Many logical operations on vertex sets of different density give significantly different results, than iterating through sparse vertex sets. This seems to be one of the key factors that determines which data structure performs best.

We will now dive deeper into the behaviour of the data structures with respect to the vertex sets the have to deal with. To do this we consider the difference between the Benchmark Experiment with the BFS Algorithm and the MMD Experiment. In the MMD Experiment the data structures had to deal with the fill graph H, which turned out to be around a factor 2 denser than the corresponding original graph G, which the data structures had to deal with in the Benchmark Experiment. Note that both these graphs are still quite sparse.

This led to some useful insights about how the data structures handles an increase in density. For starters, we consider what this means for the memory requirements. We observed that XBitSet dealt best with the increase in density, then Roaring, and both EWAH and Ints performed relatively worst. XBitSet was not affected much by this change in graph density, since it always deals with a bitmap which has the size of the graph. Ints had to store one extra Integer number for each extra vertex. Roaring had to store an extra Short number for each extra vertex, and for large graphs it could had to instantiate extra chunks. For large graphs EWAH typically had to store two extra Long numbers, one for the marker word and one for the literal word. For smaller graphs EWAH turned out to be more efficient, since it is frequently the case that two vertices reside in the same literal word or two literal words are consecutive. Note, however, that it is often the case that for small graphs, when almost all EWAH words turn out to be literal words, EWAH turns out to be simply a more complicated version of XBitSet.

We observed similar patterns for the time measurements for iterating through sets. For large graphs we observed that Ints and EWAH were effected most by the change in density, then XBitSet, and RoaringRun16 was effected the least. However, we noted that this relative difference for Ints was due to the fact that it performed so well in the Benchmark Experiment. So we had to conclude that, for iterating through vertex sets, Ints seems to scale well to denser graphs. For small graph we got the same picture, except for the behaviour of EWAH. For small graphs, we observed that EWAH dealt relatively well with the increase in density.

So based on our results, what do we expect for large graphs as they get even denser? First we discuss the memory required for storing the graph and time performance for performing logical operations. We conjecture that RoaringRun16 and XBitSet will start to perform relatively well, similarly to our observations. At some point RoaringRun16 will also start to make use of bitmap chunks. In the current implementation this happens as chunks are filled for at least 6.25%. It could be interesting to figure out if this percentage could be optimized for specific purposes.

For the memory required, EWAH will, for up to a certain density, outperform XBitSet. But at a certain density it will start to resemble the behaviour of XBitSet, as it approaches the state of a single marker word followed by the entire set in literal words. Only when vertex sets get very dense, EWAH will start to perform well again. For performing logical operations, we expect EWAH to perform relatively poorly. We already observed relatively poor performance for EWAH and we believe that larger amounts of marker words will strengthen this behaviour.

We conjecture that at some point, Ints will start to perform relatively worst. In performing logical operations for vertex sets of different density, we observed that it seems to be no match for the others. For storing the graphs, Ints needs to store an Integer number for each extra vertex. As we observed in the measurement for storing the separated components in the Benchmark Experiment, for storing dense vertex sets this weighs heavily on Ints. However, there will be a region of graph densities in which Ints scales better than EWAH, since EWAH scales horribly when it has to store two Long Numbers for each extra vertex.

For time performance of iterating through vertex sets it is more difficult to perform good predictions. Ints seems to be the most efficient for these operations. It has the upside that the vertex set density does not seem to have an influence on the performance, only the size of the set has an influence. Since both EWAH and RoaringRun16 make use of iterators, we conjecture that they are not be able to overtake Ints. Though they might have some performance tricks up their sleeves for denser graphs. We conjecture that XBitSet will perform better for dense sets, than for sparse, since it will be less bothered by the large stretches of zeros. Though we advice more research for more profound conclusions about this.

5.2.1 Comparison with literature

Note that our results seem to be adequately in line with the results of Wang et al. [38]. They compared multiple bitmap compression techniques, with respect to their performance on space overhead, decompression time, intersection time, and union time. Note that they also incorporated inverted list compression techniques into their study. Unfortunately, they did not include our Ints data structure, so it is somewhat difficult to compare those results with ours.

It is important to note is that Wang et al. performed the experiment with sets that were typically denser than in our case. They were also somewhat larger, as they performed their experiments for sets of size a million up to a billion. The fact that they used denser sets is probably the reason that they found that Roaring performed better than EWAH. For our case - as we made use of very sparse sets - we observed that relative performance seemed to depend on the graph density. Similarly to us, they found that Bitset performs poorly for sparse graphs. For denser graphs, they found that Bitset performs better, in accordance with our expectations.

They found that inverted list compression techniques were able to outperform bitmap compression techniques, for space overhead, decompression time, and union time. We observed that Ints performed very well in our experiments in storing and iterating through sets. However, we also observed that Ints seemed to perform relatively poorly in logical operations, since we observed that all other data structure performed better with the Parallel Algorithm in the Benchmark Experiment. Of course, Ints is not one of the data structures that they performed the experiments on. Still, as we suppose that these results could partially translate to our experiment, these seemingly contradictory results are very interesting. We conjecture that this difference can be explained by the fact that the typical size of the set, that the data structures have to perform the union operation on, differs between the experiments. In the Parallel Algorithm vertex sets get very large, as the separated components get to be nearly the full set. It would make sense that it because of these cases that Ints starts to perform very poorly, since for nearly full sets Ints is expected to get outperformed by regular bitmaps. The results from Wang et al. seem to indicate that, when only sparse sets have to be unified, Ints could indeed be able to perform relatively well. It would be interesting to learn more about the performance of Ints in performing logical operations on sparse sets.

5.2.2 Performance in different operations

We will now try to give an indication of the behaviour of each data structure separately, for storing vertex sets, iterating through sets, and performing logical operations.

For Ints, we observed that it is very efficient in storing sparse vertex sets. Though, as they get denser, Ints does not seem to scale that well. When many nearly full vertex sets have to be stored, Ints performs very poorly. For iterating through vertex sets, Ints seems to be very efficient and it seems to scale well.

For performing logical operations, Ints seems a poor data structure, especially when vertex sets get dense.

Ints are to be advised for sparse graphs of all sizes, as long as it is not the case that many logical operations have to be performed, or many large vertex sets have to be stored. As graphs get denser, Ints is expected to start performing relatively worse.

We observed that XBitSet is inefficient for storing sparse vertex sets. However, as vertex sets get denser or if graphs are small, it is expected to perform well. However, if large graphs get extremely dense, bitmap compression techniques might, again, start to prevail over XBitSet. For iterating through vertex sets the same conclusions hold as for storing vertex sets. For performing logical operations, XBitSet seems to be a well performing data structure. As vertex sets get denser, we expect the relative performance of XBitSet to further improve.

XBitSet is advised for small graphs and dense medium-sized graphs. In particular when many logical operations have to be performed, XBitSet seems to be a good data structure. However, for large sparse graphs, XBitSet is not advised, since storing large and sparse vertex sets and iterating through them scales relatively poorly.

EWAH seems to be efficient for storing very sparse and very dense vertex sets, though Ints performs better for very sparse vertex sets. However, for graphs of medium density it tends towards being a more complicated version of XBitSet, without having extra benefits. For iterating through sparse vertex sets, EWAH performs well. But, as graphs get denser, we observe that its relative performance became worse. Logical operations are not EWAH his strength, though it is not seem like a terrible data structure for it.

EWAH does not often seem to be the data structure of choice. Only when an algorithm has to make use of both very sparse and very dense vertex sets, and not many vertex sets in between, EWAH seems to perform best. For extremely sparse vertex sets it seems to outperform Roaring, this is where EWAH could find use.

We observed that Roaring seems to be efficient for storing vertex sets of a wide variety. It handles both very sparse and very dense vertex sets well, though Ints and EWAH outperform Roaring in storing very sparse vertex sets. The strength of Roaring is that is also expected to efficiently handle graphs of medium density, which it can represent as bitmap chunks. For iterating through vertex sets it performs better than XBitSet, though Ints outperforms Roaring. For very sparse vertex sets Roaring got outperformed by EWAH, but Roaring seemed to be able to scale better to denser graphs. For logical operations Roaring performed second best, after XBitSet.

Roaring is able to effectively handle vertex sets of different sparsity and it is both efficient in iterating through sets and performing logical operations. So if many different vertex sets are needed to be stored and different operations are needed, Roaring seems to perform best. However, when mostly dealing with very sparse vertex sets, Ints and EWAH outperform Roaring. For its versatility, we advice Roaring, whenever Ints, EWAH, or XBitSet do not seem suited for one the reasons mentioned above.

We can now answer the question of which data structures are useful for treewidth algorithms on large graphs. Since treewidth algorithm usually operate on sparse graphs [18], we end up with large and sparse graphs. We have seen that XBitSet does not perform well on large problem instances, since it runs into memory problems for storing the graphs. The choice between the other data structures depends on the vertex sets that the algorithm makes use of. We believe that in most cases, as the graphs are sparse, we will have that the typical vertex sets encountered are sparse as well. In these cases Ints seems best. However, if an algorithm also makes use of vertex sets that are very dense or both very sparse and very dense, as was the case for the Parallel Algorithm, EWAH and Roaring perform best. If vertex sets can also have medium density, Roaring is likely to yield the best results.

5.3 The Benchmark Experiment

We conducted the Benchmark Experiment in the hope that it would be able to give good predictions about the behaviour of the data structures, in how they would perform in the MMD Algorithm and the HBT Algorithm. Unfortunately, we were unable to implement the data structures in the HBT Algorithm. However, we will say something about our expectations for the HBT algorithm, based on the relative performance between the Benchmark Experiment and the MMD Experiment.

When we compared the performance of the data structures in the Benchmark Experiment and the MMD Experiment, we observed that the results translated quite nicely. For storing the graphs, separators, and computation time of the algorithms we observed quite similar results. However, we did note a couple of differences, which we attributed to the fact that the typical vertex sets which the data structures have to deal with are a factor 2 denser in the MMD Experiment, than in the Benchmark Experiment. Especially when comparing Roaring with EWAH, we observed that this gave significantly different results.

On the bright side, the fact that the data structures had to deal with a denser graph in MMD Experiment, sparked an interesting discussion about the influence of vertex set density on the performance of the data structures. However, the predictive behaviour of the Benchmark Experiment would have been better, if the typical vertex sets that the data structures had to deal with, would correspond more closely to those in the MMD Experiment. So if one want to improve the predictive performance of the Benchmark Experiment, one needs to carefully pay attention to the typical operations performed. Both the typical vertex sets and typical operations upon them need to be mimicked as well as possible. Since we observed that, besides this explained difference, the results between the two experiments resembled each other quite nicely, we have to conclude that such a Benchmark Experiment seems to be able to yield predictive behaviour of a more complicated algorithm.

Since the substar computation also turns out to be the most time-consuming operation in the HBT Algorithm, we expect that the results of the Benchmark Experiment can also translate to the HBT Algorithm. Though note, that, for

optimal results, the Benchmark Experiment would have to be somewhat adjusted to contain vertex sets of the proper size. We expect that the typical vertex sizes encountered in the HBT Algorithm are somewhat similar to what we encountered in both the Benchmark Experiment and the MMD Experiment. Therefore, we expect that our conclusions on which data structure performs best, translates to the HBT Algorithm. This means that for small graphs, we expect XBitSet to perform best, and for medium-sized and large graphs, we expect Ints to perform best.

It seems that the performance of the data structures has been mostly determined by how they dealt with four operations: storing sparse and dense vertex sets, iterating through sparse vertex sets, and performing logical operations between a sparse vertex set and a vertex set that ranges from sparse to dense. One could argue that, when investigating the MMD Algorithm one could simulate these basic operations, instead of performing a relatively complicated experiment, such as the Benchmark Experiment. Especially, since we found that in the Benchmark Experiment it turned out that the data structures typically had to deal with sparser vertex sets, than was the case in the MMD Algorithm. So the Benchmark Experiment seems to be prone to simulating imprecise operations, if one is not careful about the typical operations required.

If it is clear which operations are needed for an experiment, a less comprehensive approach could very well be a more time efficient alternative. Something like iterating through the neighborhood sets of vertices in a graph, could very well be simulated. However, it is not always clear which operations are typical, as operations can get more complicated. Take, for example, the union operation in the Parallel Algorithm, which deals with a large range of vertex sets. In such cases mimicking the operation correctly gets more complicated and could lead to mistakes, since it is more difficult to determine which vertex sets are typical for the operation. So it could be the case that we are measuring the wrong basic operations and get to conclusions which turn out not to be useful.

When we use an experiment, that performs the most resource intensive operations of an algorithm, we are more likely to compare the data structures in how they operate in that algorithm; that we get the typical operations just right. This way, we will be less likely to come to wrong conclusions. So a more comprehensive benchmark experiment might take more work to implement, but it also leads to more diligent results.

Still, performing the experiments on such an easier benchmark would make it easier to investigate the behaviour of the data structures on a wider range of vertex sets. This could give further insight in their behaviour and in which situation which data structure perform best. This is definitely interesting further research. Likewise, it would also be interesting to compare the predictive behaviour of such an easier experiment with the predictive behaviour of our Benchmark Experiment.

5.4 Partial 40-trees

We performed both experiments on random partial 40-trees. As we observe, this leads to quite steady scaling behaviour for the data structures. This way we were able to nicely distinguish between the performance of the different data structures. For this reason we can recommend using partial k-trees as graph for performing experiments on graphs, when one wants to fix the treewidth.

There are three things to note about the graphs we used. First of all, the fact that we decided to use partial 40-trees can be disputed. For some applications a treewidth of 40 already seems quite high. So if one wants to perform experiments, one could, for example, also decide to use partial 10-trees. Especially when the running time of an algorithm depends on the treewidth, this makes sense when one wants to scale to large graphs. Since we hypothesised that this also holds for our implementation of MMD, this could also be an interesting addition to our experiments.

Another decision that we made, was deciding on the graph density. It would definitely be interesting to see how the data structures would perform on graphs that are denser. We decided to have the focus of the project on as large graphs as possible, this meant that we wanted graphs to be sparse. However, more experiments with denser graphs could definitely be interesting.

Finally, it would also be interesting to compare the performance of the data structures on the PACE instances of the 2017 Treewidth Track. The PACE instances consist of various types of real world graphs, for which treewidth computations are relevant. It would be interesting to see how our results relate to the results on the PACE instances. This could also give further insight in whether using random partial k-trees as benchmark graphs can give good predictive results for real world graph problems.

5.5 Future work

There are a lot of possibilities to further our understanding and the performance of bitmap compression techniques in the field of graph algorithms.

First of all, Roaring Bitmap is a new technique, that performs fairly well. Moreover, it is the first technique in its kind. It could be interesting to look into improving Roaring or let it serve as an inspiration for even better techniques. Similarly as that Byte-aligned Bitmap Compression inspired a large range of techniques, such as WAH and EWAH.

For specific situations, it could also be worthwhile to look into tweaking Roaring to be more optimal for the situation at hand. One could, for example, alter the chunk density at which a array of integers chunk is converted into a bitmap chunk. Or, if it is likely that only small number of chunks are needed, it could be worthwhile to consider using dynamic memory allocation for the bitmap containers, thus saving space.

Something that we observed in the experiments, is that bitmap compression techniques seem to perform well for vertex sets, which show a lot of structure. So

if they either contain large stretches of ones or zeros. One could try to improve structure in the vertex sets, by making use of vertex relabeling algorithms. Usually vertices are labeled randomly. If, however, one could label the vertices in such a way, that neighbouring vertices are somewhat grouped together, this could lead to improved performance.

Consider, for example, the case that we would split the graph through the middle, and we would give each vertex on one side of the split a label in the range $[0, G(N)/2 - 1]$ and each other vertex a label in the range $[G(N)/2, G(N) - 1]$. Suppose now, that we would have to store a subgraph that either contains all of the vertices in the first half, or none of them. In both cases all those vertices could be represented very efficiently by bitmap compression techniques, since they are very efficient in storing large stretches of ones or zeros. Note that both Ints of XBitSet do not benefit from such a vertex relabeling.

It definitely the case that more efficient vertex relabeling algorithms exist than the one describe above. However, it does explain the idea: if vertices are labeled in a smart way, they can be represented and worked with more efficiently by bitmap compression techniques. There have been many vertex relabeling algorithm proposed. For a concise survey on the topic we refer to the survey of Besta and Hoefler on "Lossless Graph Compression and Space-Efficient Graph Representations" [6].

Note that it could also be interesting to observe how the different versions of Roaring would behave in combination with vertex relabeling algorithms. We conjecture that smaller chunks could have a larger benefit of more structured sets.

Besides considering bitmap compression techniques for graph algorithms, it could also be interesting to investigate the behaviour of inverted list compression techniques. Especially since we observed that Ints performed well for sparse and large graphs and that Wang et al. found that inverted list compression techniques are efficient for many operations. The performance of Ints could perhaps be exceeded by considering different inverted list compression techniques.

Finally, as mentioned in our discussion of improving the performance of the MMD Algorithm, it could be worthwhile to look into further improving the substar computation. Both MMD and HBT make use of this procedure. Trying to improve it, by making use of a tree decomposition to limit the area that needs to be searched, could lead to better performance and improved scaling to larger graphs.

References

[1] Example tree decomposition. https://en.wikipedia.org/wiki/Tree_decomposition.

[2] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[3] Yasuhito Asano, Tsuyoshi Ito, Hiroshi Imai, Masashi Toyoda, and Masaru Kitsuregawa. Compact encoding of the web graph exploiting various power laws: Statistical reason behind link database. In Guozhu Dong, Changjie Tang, and Wei Wang, editors, *Advances in Web-Age Information Management, 4th International Conference, WAIM 2003, Chengdu, China, August 17-19, 2003, Proceedings*, volume 2762 of *Lecture Notes in Computer Science*, pages 37–46. Springer, 2003.

[4] Emgad H. Bachoore and Hans L. Bodlaender. Weighted treewidth algorithmic techniques and results. In Takeshi Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 893–903. Springer, 2007.

[5] Anne Berry, Pinar Heggernes, and Geneviève Simonet. The minimum degree heuristic and the minimal triangulation process. In Hans L. Bodlaender, editor, *Graph-Theoretic Concepts in Computer Science, 29th International Workshop, WG 2003, Elspeet, The Netherlands, June 19-21, 2003, Revised Papers*, volume 2880 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2003.

[6] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *CoRR*, abs/1806.01799, 2018.

[7] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[8] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[9] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations i. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[10] Vincent Bouchitté, Dieter Kratsch, Haiko Müller, and Ioan Todinca. On treewidth approximations. *Discret. Appl. Math.*, 136(2-3):183–196, 2004.

[11] Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31(1):212–232, 2001.

[12] Ramon Ferrer I Cancho and Richard V Solé. The small world of human language. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 268(1482):2261–2265, 2001.

[13] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exp.*, 46(5):709–719, 2016.

[14] Zhen Chen, Yuhao Wen, Junwei Cao, Wenxun Zheng, Jiahui Chang, Yinjun Wu, Ge Ma, Mourad Hakmaoui, and Guodong Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, 20(1):100–115, 2015.

[15] François Clautiaux, Aziz Moukrim, Stéphane Nègre, and Jacques Carlier. Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO Oper. Res.*, 38(1):13–26, 2004.

[16] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. *Inf. Process. Lett.*, 110(16):644–650, 2010.

[17] Robert Cummings, Matthew Fahrbach, and Animesh Fatehpuria. A fast minimum degree algorithm and matching lower bound, 2021.

[18] Fabien de Montgolfier, Mauricio Soto, and Laurent Viennot. Treewidth and hyperbolicity of the internet. In *Proceedings of The Tenth IEEE International Symposium on Networking Computing and Applications, NCA 2011, August 25-27, 2011, Cambridge, Massachusetts, USA*, pages 25–32. IEEE Computer Society, 2011.

[19] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6-8, 2017, Vienna, Austria*, volume 89 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[20] Fedor V. Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. volume 32, pages 289–308, 2012.

[21] Linton C Freeman. Visualizing social networks. *Journal of social structure*, 1(1):4, 2000.

[22] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. *CoRR*, abs/1207.4109, 2012.

[23] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. SBH: super byte-aligned hybrid bitmap compression. volume 62, pages 155–168, 2016.

[24] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. Treewidth: Computational experiments. *Electron. Notes Discret. Math.*, 8:54–57, 2001.

[25] Arie Marinus Catharinus Antonius Koster. *Frequency assignment: Models and algorithms*. PhD thesis, 1999.

[26] C Lekkeikerker and Johan Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51(1):45–64, 1962.

[27] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.

[28] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. Roaring bitmaps: Implementation of an optimized software library, 2018.

[29] Daniel Lemire, Cliff Moon, David McIntosh, Robert Becho, Colby Ranger, Veronika Zenz, Owen Kaser, Gregory Ssi-Yan-Kai, and Rory Graves. Javaewah: A compressed alternative to the java bitset class. https://github.com/lemire/javaewah, 2020.

[30] Harry M Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[31] Jürgen Plehn and Bernd Voigt. Finding minimally weighted subgraphs. In Rolf H. Möhring, editor, *Graph-Theoretic Concepts in Computer Science, 16rd International Workshop, WG '90, Berlin, Germany, June 20-22, 1990, Proceedings*, volume 484 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 1990.

[32] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[33] Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017.

[34] Hisao Tamaki. Computing treewidth via exact and heuristic lists of minimal separators. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis of Experimental Algorithms - Special Event, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2019.

[35] Hisao Tamaki. A heuristic use of dynamic programming to upperbound treewidth. *CoRR*, abs/1909.07647, 2019.

[36] Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *J. Comb. Optim.*, 37(4):1283–1311, 2019.

[37] Athanasios Theocharidis, Stjin Van Dongen, Anton J Enright, and Tom C Freeman. Network visualization and analysis of gene expression data using biolayout express 3d. *Nature protocols*, 4(10):1535, 2009.

[38] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang,

and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 993–1008. ACM, 2017.

[39] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 296–303. ACM, 1996.

[40] Kesheng Wu, Kurt Stockinger, and Arie Shoshani. Breaking the curse of cardinality on bitmap indexes. 5069:348–365, 2008.