Heap and Lifetime Visualization in a Memory Monitoring System

Master Thesis

to obtain the academic degree of
Diplom-Ingenieurin

in the Master’s Program
Computer Science
Zusammenfassung

AntTracks ist ein Tool zur Speicheranalyse für die Java Hotspot™ VM. Um es BenutzerInnen zu ermöglichen, speicherbedingte Performance Probleme in ihren Applikationen aufzudecken, bietet AntTracks verschiedene Ansichten, welche die Investigation bestmöglich unterstützen.


Zusätzlich dazu wurde eine neue Ansicht für AntTracks entwickelt, welche, basierend auf der Heap Ansicht, den Speicher im Zeitverlauf visualisiert.

Die Analyse der Funktionen der neuentwickelten Ansicht zeigten dass diese das AntTracks Analysetool mit neuartigen, nützlichen Fähigkeiten ausstattet. Leistungsanalysen zeigten dass die Applikation den Prozessor und Speicher nicht mehr als nötig belastet und für die Visualisierung von großen Datenmengen in Form von großen Heaps gut gerüstet ist.

Abstract

AntTracks is a memory monitoring tool for the Java Hotspot™ VM. In order to analyze and track down memory-related performance problems in applications, AntTracks offers various views that aid the user in their investigation.

This thesis describes the flaws and issues of the graphical heap view in AntTracks, as well as the complete re-development of the view, considering new requirements as well as taking into account capabilities of other state-of-the-art tools. Besides from providing a stable and rich application with useful features, it was also necessary to focus on the ability to deal with large amounts of data without running into memory shortage. In order to achieve this, a novel strategy of memory saving was developed.

Additionally, a new view which is based on the heap view and enables the user to visualize the heap over time was added.

Functional analysis of the newly developed heap view showed that it adds supplementary capabilities to the AntTracks analysis tool. Performance analysis showed that the application does not put any unnecessary pressure on the CPU or memory and is well equipped to handle very large amounts of data.
## Contents

1 Introduction 1

2 Background 3
   2.1 Memory Management in the Hotspot™ VM 3
   2.2 AntTracks 6
   2.3 Filters and Classifiers 10

3 State of the Art 12
   3.1 AntTracks Heap Visualization 12
   3.2 Related Work 17
      3.2.1 GCSpy 17
      3.2.2 Oberon Heap Inspector 20

4 Requirements 24
   4.1 Fast and Efficient Implementation 24
   4.2 Accurate Heap Visualization 24
   4.3 Support for User-Defined Classifiers and Filters 25
   4.4 Automatic Color Generation 25
   4.5 Automatic Cluster Size Adaption 26
   4.6 Detailed Object Info 26
   4.7 Zoom 26
   4.8 Pointer Visualization 27
   4.9 User-friendly Interface 28
   4.10 Heap Visualization over Time 28

5 Implementation 29
   5.1 Architecture 29
   5.2 Object-based Visualization Data 30
      5.2.1 Data Generation 30
      5.2.2 Automatic Color Generation 33
      5.2.3 First Storage Approach 34
      5.2.4 Second Storage Approach 36
      5.2.5 Final Storage Approach 37
      5.2.6 Swapping with Memory-Mapped Files 43
   5.3 Pixel Map 44
      5.3.1 Image Buffer 45
      5.3.2 Shared Code 46
      5.3.3 Object View 47
      5.3.4 Byte View 48
   5.4 User Interface 51
      5.4.1 Heap Panel 52
1 Introduction

Automatic memory management through garbage collection, featured in common programming languages such as Java and C#, relieves programmers from the inconvenient task of freeing memory manually. Moreover, manual memory management is more prone to errors [2] and will eventually produce fragmented memory [12]. The drawback of automatic memory-managed systems is that the memory behavior of the application is completely hidden from the programmer. This is usually not a problem if applications behave as expected. It becomes an issue when memory-related performance degradations arise. To identify and eliminate possible bottlenecks, information about the actual behavior of the garbage collector is of interest. This information is usually not at all or only partially available. Therefore, profiling tools which are able to record information about events in memory, such as allocations, objects movements or object deallocations, are necessary.

The usefulness of such monitoring tools is determined by two major factors. To begin with, the first factor is the quality of the collected data about the application. The collected data ought to be complete, meaning that the provided information should be sufficient to identify possible memory-related performance problems and track down their origins. At the same time, the profiling tool should keep its influence on the application to a minimum in order to not distort the behavior of the program under inspection [10]. The second factor is the way the collected information is represented to the user. A memory-monitoring tool is useless if the user is not able to take a look at the data. Therefore such tools always provide some kind of interface, ranging from command-line output to highly sophisticated graphical user interfaces. Such interfaces should be intuitive and shall offer useful features.

An approach to trace Java applications accurately and efficiently called AntTracks is introduced in Lengauer [10], Lengauer et al. [12] and Bitto [2]. AntTracks offers different views on the monitoring data it collects from applications. Bitto [2] describes a table-based as well as visual view on the heap (memory) of the inspected application. Weninger et al. [18] describes a user-centered approach to group, filter and analyze the data which was elaborated into the table-based view. The goal of this thesis was to re-develop the graphical heap view (“Heap Visualization”) of AntTracks as well as to extend it to visualize the heap over time. The new heap visualization should integrate the user-centered analysis approach [18] and introduce other new features as well as overcome the performance-related and functional issues of the old visualization.

This thesis is structured as follows: Section 2 provides background information on memory management in the Java Hotspot™ VM, AntTracks in general and in particular AntTracks’ user-centered analysis approach. Section 3 describes the old
AntTracks heap visualization as well as related work. Section 4 defines the requirements for the new heap visualization. Section 5 provides a detailed description of the implementation of the new heap visualization. Section 6 shows some examples that prove the accuracy and usefulness of the new heap visualization. Section 7 analyses the new heap visualization in terms of run-time performance and memory consumption. Section 8 discusses possible future work and Section 9 concludes this thesis.
2 Background

This section provides some background information on the memory management of the Java Hotspot™ VM and on AntTracks.

2.1 Memory Management in the Hotspot™ VM

Memory management in the Java Hotspot™ VM is heavily coupled with garbage collection. There is a set of different garbage collectors available that can be enabled via VM flags [2]. The default collector in Java 8 as well as previous versions [10] is the so-called ParallelOld GC. It is a generational stop-the-world collector. Stop-the-world means the whole application is halted that during a garbage collection and resumed when the collection is done. Generational means that the heap memory is split into different regions. The idea is that every heap portion contains objects of a certain age, exploiting the fact that applications usually allocate many short-living and only few long-living objects [13]. This makes it possible to use different collection strategies for each region. The generational garbage collection has been in use since JDK 1.3 [9].

The heap is split into two generations, i.e., the young generation and the old generation. This is an implementation of the parallel generational scavenging technique [17]. The young generation is divided once more into the Eden space and two survivor spaces. A run-time efficient algorithm (Scavenge, minor GC) is used on the young generation only, while the so-called Mark and Compact algorithm (major GC) is used for collection of all spaces. Both algorithms leave the heap unfragmented and are highly parallelized.

New objects are usually allocated into the Eden space. A new object is simply appended at the end of the memory that is already in use and therefore the space never gets fragmented. However, since there are usually multiple threads that might want to allocate objects, they would be racing for the end of the used space and synchronization would be necessary. In order to avoid this, every thread uses its private thread-local allocation buffer (TLAB) located in the Eden space. If a new object doesn’t fit into a TLAB, the thread gets assigned a new TLAB in the Eden space. The remainder of the old TLAB is filled with an int[] in order to keep the Eden space unfragmented. This filler object is not referenced and will therefore not survive the next garbage collection. Once in a blue moon, an object is directly put into the Eden space without a TLAB, for example when it is not possible to allocate a new TLAB. This allocation strategy, known as bump pointer allocation, guarantees O(1) run-time complexity for every allocation [10].
Figure 1: Full survivor space, full Eden space and empty survivor space before garbage collection

Figure 2: Eden space and two survivor spaces during an ongoing minor GC

Figure 3: Empty survivor space, empty Eden space and a filled survivor space after a minor GC

Figure 4: Eden space and two survivor spaces during an ongoing minor GC

Figure 5: Empty survivor space, empty Eden space and a filled survivor space after a minor GC
Figure 4: A space before a major GC

Figure 5: A space after the mark phase of a major GC

marked as being alive if they are reachable via the root pointers, shown in Figure 5; (2) during the summarize phase, the new addresses of live objects are computed; (3) in the compact phase the objects are moved to their new locations and pointers are adjusted. The result of such a Mark and Compact garbage collection is shown in Figure 6. The major GC works also in parallel.

The Mark and Compact algorithm is based on the so-called Mark and Sweep algorithm. In this algorithm, the summarize and compact phases are replaced with a single sweep phase. Instead of compacting the memory, the algorithm declares the memory blocks where the dead objects reside as empty by inserting them into a free list, thus, leading to a fragmented memory [10]. Figure 7 shows the result of a garbage collection when the Mark and Sweep algorithm is applied instead of the Mark and Compact algorithm.

Oracle’s Concurrent Mark and Sweep Collector uses the same generational heap layout as the ParallelOld GC. It also uses the Scavenge algorithm for the minor GC. However, there is also a minor GC which performs a Mark and Sweep algorithm on the old generation only [10]. Thus, the Java heap may be partially fragmented when using this garbage collector.

Another garbage collector for the HotSpot™ VM is Oracle’s Garbage First (G1) Collector, first proposed by Detlefs et al. [4, 5]. Like in the ParallelOld GC, there is a young generation divided into Eden space, survivor from space and survivor to space and an old generation. The algorithms for cleaning these spaces are also the same.
Figure 7: A cleaned space after the execution of the *Mark and Sweep* algorithm

![Figure 7: A cleaned space after the execution of the *Mark and Sweep* algorithm](image)

Figure 8: Heap allocation of the *G1 Garbage Collector* [1]

![Figure 8: Heap allocation of the *G1 Garbage Collector*](image)

as in the *ParallelOld GC*. The main difference between these two is their heap layout. While the *ParallelOld GC* has only four spaces which it may resize if necessary [10], the *G1 GC* splits its spaces into multiple small, non-resizable spaces, so-called *regions*. They vary in size from 1 to 32 MB [1]. A sample *G1* heap allocation is shown in Figure 8. Besides *Eden*, *survivor* and *old regions* there are also so-called *humongous regions*, which are designed to hold objects that are 50% the size of a standard region or larger [1]. It is discouraged to create such large objects, as the collection of humongous regions is not optimized [1].

A *minor GC* in the *G1 GC* will use the *Scavenge* algorithm only on those regions that require cleaning. The *major GC*, which occurs only very rarely, cleans the entire heap using the *Mark and Compact* algorithm.

### 2.2 AntTracks

AntTracks, first introduced in [12], is a low-overhead memory monitoring tool for Java applications. It uses the approach of a modified Java VM, the so-called AntTracks VM. Monitoring on this VM can be enabled via the flag `-XX:+TraceObjects`. The VM then logs all relevant events that occur in the memory of this application during execution to a file, the so-called trace file. When this flag is not enabled, the AntTracks VM behaves just like the normal Java HotSpot™ VM.

Some prominent examples for events in a trace file are object allocations, object movements caused by garbage collections and space creations or redefinitions. After
executing the Java application, the resulting trace file can be analyzed. Therefore, AntTracks’ approach is to analyze offline. Other state-of-the-art tools use approaches where the application can be examined while running. Techniques to achieve this often result in distorting the applications memory behavior or in introducing considerable run-time or memory overheads [10]. AntTracks, on the other hand, introduces an average run-time overhead of 4.68%, which appears to be fast enough to monitor in production mode [12].

To completely reconstruct the memory of an application offline, the trace would need to contain an abundant amount of events. This would result in enormous trace file sizes and possibly a bad run-time overhead. In order to be fast, AntTracks omits information which can be reconstructed from other events [2, 10, 12]. For every object, AntTracks is able to reconstruct the following information from the trace file [2, 10, 12]:

- **Address**
- **Allocation Site**
  The location in the code where the allocation took place, i.e., the class, method and byte code index.
- **Kind**
  Whether the object is an instance or an array.
- **Type**
  The Java type (i.e., class) of the object.
- **Size**
  The object size for instances or array length for arrays.
- **Thread**
  The thread which allocated the object.

Information that is missing here are the contents of the objects, i.e., the values of their fields. In terms of detection of memory-related performance problems, the values of primitive fields of objects are not of great interest. However, to identify memory leaks, it can be helpful to be able to see how objects are kept alive. This could be reconstructed if the values of object fields (pointers) were available. Lengauer et. al [11] and Lengauer [10] describe the integration of pointer tracing to AntTracks. This can be enabled via the additional VM flag `-XX:+TraceObjectsPointers`.

To log all object allocations in the AntTracks VM, the components in the Java HotSpot™ VM which allocate Java objects had to be instrumented. There are four
of these components (allocators), i.e., the Interpreter, the C1 and C2 compiler and
the VM itself.

Similarly, components which move Java objects had to be instrumented to log object
movements. These components are the garbage collectors. Since the HotSpot™ VM
offers multiple garbage collectors, every one of them needs to be instrumented indi-
vidually. Currently, AntTracks only fully supports the ParallelOld GC. The G1 GC
and the Concurrent Mark and Sweep GC are partially supported, i.e., pointer tracing
only works experimentally or does not work at all.

Trace files are analyzed with the so-called AntTracks Tool. The AntTracks Tool
is a Java application, developed using JRE SE 8 [2], which is able to process and
parse trace files generated by the AntTracks VM. It has an intuitive graphical user
interface which offers several different views to analyze the data.

By opening a trace file, a parser process is started. The trace file is parsed in parallel
[2, 10]. The current state of the memory of the inspected application is stored in a
data structure which is mutated during the parsing process. While the scanning and
parsing is still ongoing, said resulting data structure is already used to generate four
overview plots (see Figure 9).

When clicking on a point in time in any of these plots, the point is also selected in
the other plots (see Figure 9). After selecting such a point in time, further operations
which analyze the heap at the selected time can be started. In order to create the
data structure for the desired point in time, another parser process needs to be
started. Note that not any arbitrary point in time may be selected, but only points
immediately before (peaks) or after (minimums) garbage collections.

In terms of memory consumption, retaining a Java heap from a trace file is a challenge
[3]. In order to keep the memory consumption of the parser and furthermore, the
analysis tool, low, a special data structure is used. It enables the AntTracks tool to
parse trace files using only a fraction of the memory that was used by the original
application which was inspected [3]. Lengauer [10] argues that it is guaranteed that
trace files can be parsed with the same heap size as the original application.

The data structure is implemented by the Heap class. The organization of data in this
class is modelled after the Java heap [2, 3]: a Heap contains a list of Spaces which
themselves contain Labs which finally contain the actual objects, represented by the
class ObjectInfo. The Heap class offers methods to iterate either single-threaded
or multi-threaded over all objects [18]. It also offers a method which returns the
Space for a given address via binary search. This resulting Space object can then
be used to retrieve the ObjectInfo for the object at a specified address.
Figure 9: Generated overview plots of a trace file
2.3 Filters and Classifiers

Weninger et al. [18] states that state-of-the-art memory monitoring tools often serve a specific purpose well, but lack to retrieve additional information from their available data. To overcome this in AntTracks, *user-defined object classifiers* and *filters* were implemented.

**Filters.** When inspecting a heap, usually not all objects are of interest. In order to focus on the investigation of a memory-related performance issue, it is helpful to exclude objects which are clearly not related to the problem. We also may want to restrict analysis to certain objects, e.g., only analyze the memory behavior of certain packages. The Heap class offers a method to set a filter, which restricts further iteration functions to only process objects which are not excluded by the filter [18].

**Classifiers.** A classifier portions objects according to a certain criterion. While the classifier describes the method by which the objects are grouped (*classified*), we refer to a concrete output of a classifier as a *classification*. An example for such a classifier is the *Type classifier*. Examples for classifications as result of this classifier are `java.lang.String`, `java.lang.Object` and `int[]`. The results of applying a classifier can be represented to the user in various different ways. For instance, it can be determined for each classification by how many objects it is carried and the numbers can be displayed to the user. Thinking in terms of a graphical visualization, every classification could be assigned a color and the according objects can be drawn in that color.

Weninger et. al [18] distinguishes between *one-to-one*, *one-to-many* and *one-to-hierarchy* classifiers. The output of a *one-to-one* classifier is one object (typically a String), while the outputs of the other classifiers are arrays. In a *one-to-many* classifier the resulting keys in the classification array do not have a relationship, while in a *one-to-hierarchy* classifier the keys are organized in a hierarchical fashion (*i.e.*, keys with a parent-child-relation [18]).

AntTracks offers a range of useful pre-defined filters and classifiers. Figure [10] shows a graphical user interface component which allows to select classifiers and filters. Additionally, the concept of *user-defined object classifiers* contributes to achieve flexible and user-centered information retrieval in the AntTracks tool. As described by Weninger et. al [18], *user-defined object classifiers* do not have to be defined at load time. By using Java Service Providers (SPI) and in-memory on-the-fly compilation they can be added by users at run-time. This can be achieved by pressing the “+”
Figure 10: GUI component which allows users to select classifiers and filters

![GUI component](image)

Figure 11: Dialogs which for creating user defined filters and classifiers

![Dialogs for creating filters and classifiers](image)

Buttons in the GUI component (see Figure 10). This will open the dialogs shown in Figure 11.

As shown in Figure 10, multiple classifiers as well as multiple filters can be selected and applied.
3 State of the Art

This section thoroughly describes the old heap visualization with all its drawbacks and also takes a look at related work.

3.1 AntTracks Heap Visualization

In order to visualize the heap in AntTracks, users can select an arbitrary point in the execution time of the application in the overview plot (as described in Section 2.2) and examine it. A new tab with a table-based view opens up (see Figure 12). From there, the visualization can be opened by pressing the “Visualize” button. The view shown in Figure 12 has already been replaced by a better table-based view which incorporates the concept of user-defined object classifiers.

![Old table-based statistics view which allows to start visualization](image)

The visualization tab is shown in Figure 13. On the left side, there are three panels which visualize the Eden, survivor and old spaces. Each black pixel in the panels represents a cluster of objects in the heap. As Bitto [2] states, clustering was introduced because heaps may be several gigabytes in size, thus, visualizing one object per pixel will lead to a very large image. The default setting is 5 objects per pixel. This can be adjusted by the user in the configuration panel at the top-right. There, the user
may also select to visualize the heap in terms of bytes instead of objects. As shown in Figure 14, a byte cluster may be several kilo-, mega- or gigabytes large. When the object option is selected, the value of the unit box is ignored.

Below the configuration panel, there’s a panel which allows the user to select filters. The term filter is misleading, because these filters do more than plain filtering. According to the terminology introduced in Section 2.3, these filters are both filters and classifiers at the same time. The user can select one from seven pre-defined filters. Then the objects are classified according to the criterion described by the filter. Every classification is automatically assigned a color. However, there are only five pre-defined colors, which is not much compared to the number of classifications some filters may give as result. In order to overcome this problem, the user may also
select the colors for individual classifications themselves. In order to filter the heap, the user may also remove classifications. The corresponding pixels which contain filtered objects then appear as black. Previously removed classifications may also be re-added by the user.

After selecting a filter, the filter is applied to all three panels. Figure 15 shows the visualization of the Eden panel. For some unknown reason there is a gap between the image and the vertical scrollbar. Additionally, the scrollbar is not even necessary because the image does not fill the entire panel.

Every object has an assigned color, but a cluster pixel may contain multiple objects with different colors. Bitto [2] describes how this problem was solved: If a cluster contains only objects of one color, the pixel is painted with the corresponding color. If it contains objects of different colors, the color of the objects which appear the most is used, but the alpha value is adjusted so that this pixel appears more pale.

The user may also zoom into a desired area in order to analyze specific parts of the heap in detail. By pressing the left mouse button and drawing a rectangle over the area of interest, the user can select a sector (see Figure 16). When releasing the mouse button, zooming is applied to the selected area [2].
As described in Bitto [2], zooming is done in two steps (Two-step zooming level): Firstly, the pixels within the selected area are merely enlarged (see Figure 17 top). By pressing the button labeled with a brush, the second zooming step is triggered. There the pixels are dissolved into individual objects or bytes (see Figure 17 bottom), which corresponds to a cluster size of 1. By pressing the button labeled with an un-magnified glass, the zooming mode is exited. The previously selected area remains selected.

We do not know why the magnified area does not fill the entire panel and why the enlarged pixels after the first zooming step are drawn in rectangular shapes rather than squares, which causes the impression of a distorted image. The user also does not know what actually happens when the second zooming step is applied, there is no information or user feedback which indicates that the cluster size in the zoomed area has been set to 1.

Bitto [2] admits that not all corner cases in the zoom have not been fully covered yet. It is stated that if the marked area is too big to be enlarged, the application remains in the non-zooming mode. Since the zoom magnifies the selected area such that it uses the width and height of the panel (or even less than that, as Figure 17 shows), it is possible that dissolution of the cluster pixels to a size of 1 object/byte leads to an image much larger than the available screen space, especially if the cluster size was large. Bitto [2] suggests a step by step zooming functionality which adjusts the cluster size gradually to overcome this issue.
This visualization assumes a ParallelOldGC heap, as the subdivision into three individual space panels shows. Lengauer [10] claims that the AntTracks trace format is portable in sense that the AntTracks tool does not need to make any assumption about the underlying VM implementation or the GC algorithm that generated the trace. We go further and argue that the tool must not make any of these assumptions in order to support all garbage collectors, even those that might be supported by AntTracks in the future. The visualization clearly violates this requirement. One might argue that the subdivision into Eden, survivor and old spaces is common among all garbage collectors in the HotSpot™ VM and therefore does not really create a problem. This is true, but there might be garbage collectors in the future which organize their heap differently. Furthermore, the G1GC, which is already supported, uses this subdivision but does not have continuous spaces (cf. Section 2.1).

We used the visualization to look at a heap generated by the G1GC in order to investigate the behavior. As shown in Figure 18 we used the Allocating Subsystem filter, but the coloring is only applied to the third panel, the pixels in the other panels remain black. Something is clearly not working, so a G1GC trace breaks this visualization. Also, for the user it is not clear whether the visualization in the panels shows all regions of the corresponding space type or whether it just visualizes the first region of that type.
Additionally, the visualization suffers from another major drawback. The implementation iterates over the Heap object every time when painting the pixels [2]. Re-painting has to be done quite frequently, e.g., when zooming in, when using a scrollbar and even when just resizing the window. As the Heap iteration is fairly slow, users do experience waiting times when scrolling around in a panel or resizing the main window. This is a serious problem when the heap size is large, but even for very small heap sizes it is already noticeable, especially if the selected cluster size is small.

3.2 Related Work

Lengauer [10] describes 14 state-of-the-art memory monitoring tools for Java. Some of these tools do not have a graphical user interface or their collected data (e.g., garbage collection times) cannot be used to visualize the memory. The other tools do also not contain any sort of visualization that could be compared to the AntTracks heap visualization. Therefore we examined two different tools which offer similar visualization features.

3.2.1 GCSpy

**GCSpy** is an architectural framework for the collection, transmission, storage and replay of memory management behaviour. It can be used on any application, runtime environment or programming language and can easily be adapted for use with any new memory management system [15].

This flexibility is achieved by the fact that GCSpy is designed and implemented as a client-server application [7, 15]. The server component of the tool runs directly within the observed application or its virtual machine. Via a TCP/IP socket connection the GCSpy client connects to a server component. The server then sends snapshots of the heap to the client when certain events occur, e.g., garbage collector runs. The GCSpy client interprets and visualizes the received heap data and also maintains an index with a history of received snapshots. This allows the user to view the state of the heap at previous points in time. The GCSpy framework provides server infrastructure libraries in C, C++ and Java. The client is a Java application with a Swing user interface. For integrating a new memory monitoring system, only the server part needs to be implemented. The client will work with every server due to an abstract representation of the heap [7].
Hofer [7] describes the abstracted data which GCSpy collects and visualizes:

- **The Heap** is the complete set of data that is observed by GCSpy.

- **Events** are significant points during the execution of the observed application at which the heap state may be collected and visualized. Examples for events are the start or end of a garbage collection.

- **Spaces** are typically regions of managed memory, but can also represent lists of sets (e.g., a free list). The young and old generations of a generational garbage collector would qualify as separate spaces. The server also may define the full heap as one space which can then be visualized. Spaces consist of blocks.

- **Blocks** represent the managed units of a space. They are displayed as equally sized, adjacent tiles in the GCSpy client. However, their actual size, location and order in memory may differ. Other than chunks of a memory area, blocks can be used to represent entries of a free list, remembered sets or other entries.

- **Streams** are the attributes of the blocks or a space. There are two types of streams:
  
  - **Value streams** have integer values that lie within a specified range. The client uses the value relative to the range boundaries to visualize these streams with vertical filled bars or fading colors. The preferred display color can be set of the server side, but it can be changed in the client tool. An example for a value stream is the number of objects per block.
  
  - **Enumeration streams** specify a set of possible named values with distinguishable colors or patterns and displays the assigned descriptions. Colors for individual values cannot be set by the user, only the general visualization (i.e., color or pattern) can be set. Enumeration streams are typically used to specify the type of a block or boolean values such as marked and not marked.

Figure [19] shows the GCSpy client main window visualizing three spaces. By clicking on tiles, information about the block which the tile represents is displayed. Examples for such information are block size, the used space or the number of objects. The stream that is to be visualized can be selected from a list of available streams. Multiple streams may also be visualized at the same time. In order to do so, the tiles are divided. Figure [20] shows the visualization of three different streams.
A purely optical zoom ("magnification") exists, but there is no way to zoom into a tile to examine the individual objects that are contained within the block represented by the tile.

It is possible to apply filters. However, filters only apply to events, therefore, the user can select not to receive heaps triggered by certain events from the server. There is no way to filter blocks within a heap.

The tool is not optimized for large heaps. Initially, the space views did not have any scrolling capability. The space view would only display as many tiles as could fit into the available area. Hofer [7] pointed this problem out and improved the GCSpy client in order to support scrolling in spaces.

---

1University of Kent: The GCspy heap visualisation framework, [https://www.cs.kent.ac.uk/projects/gc/gcspy/](https://www.cs.kent.ac.uk/projects/gc/gcspy/) (2017/06/08)
3.2.2 Oberon Heap Inspector

Oberon is a general-purpose programming language that evolved from Modula-2 [19]. It is an imperative, structured, modular and object-oriented language that shares common ground with Java in many aspects. The Oberon system is not a virtual machine, but rather a custom operating system which allows to load and start Oberon programs ("modules"). Just like Java, it offers automatic memory management with a garbage collector. The system itself is implemented in Oberon, as well as all its components (e.g., garbage collector, compiler). Oberon programs are not interpreted, but directly translated to machine code by the compiler.

The Oberon Heap Inspector is an Oberon module which visualizes the entire Oberon heap. As a consequence of running within the Oberon system, the memory occupied by the inspector itself is also visualized. Only the current state of the memory is visualized, there is no way to record heap snapshots and go back to visualize them. The visualization ("HeapMap") is updated with the current heap through user interaction.

Figure 21 shows a heap visualization with the Heap Inspector. At the top, the heap is visualized in terms of bytes. One pixel in the heap map represents the minimum object size in Oberon (16 to 32 bytes, depending on the machine on which the Oberon system is running). Therefore all objects and even areas of free space are aligned to that size. The lightest used shade of gray represents free space. The darker shades up to black are used for objects. This coloring is only used to distinguish neighboring objects and is therefore arbitrary. However, a different kind of coloring can be switched on. The Colorize Default Blocks feature, as shown in Figure 22, assigns chromatic colors to certain objects (e.g., arrays with or without pointers, typedescriptors, etc.). These colors are not customizable by the user. It is also not

---

Figure 20: Visualization of three streams with GCSpy [7]

---

possible to define different criteria by which the objects could be colored or to filter objects.

The user can select a pixel by pressing the left mouse button. As shown in Figure 23, some information about the selected pixel (kind, size of object/free space) is displayed at the top of the window.

The user may also add additional components. For instance, the feature Show Reachable opens a second heap map which only displays objects that are referenced by the object that was selected in the original heap map. This is shown in Figure 24. There is also the inverse feature which only shows objects that reference the
selected object. Another feature shows objects that are likely to be collected during the next GC run because they are no longer referenced.

By pressing the right mouse button an area can be selected (see Figure 25 top) and by releasing it the selected area is zoomed into (see Figure 25 bottom). This is only an optical zoom. Within a zoomed area, the user can select another area too further zoom into. Unfortunately, there is no information about the current zoom factor and zooming back is only possible in two pre-defined steps (either completely or -30%). If an additional view (e.g., the Show reachable view) is open, the zoom still works and operates on all open views.

Every time an operation is issued by the user (e.g., zoom, display of object information), the Heap Inspector iterates over the heap in order to find the desired block(s). This works very well and does not cause any waiting times for the user. However,
Figure 25: Zoom feature in the Oberon Heap Inspector

The default size of an Oberon heap by the time of developing this tool was 4 MB. We heavily doubt that this approach is feasible for today’s heap sizes.
4 Requirements

This section defines the requirements that we defined for the new heap visualization.

4.1 Fast and Efficient Implementation

As described in Section 3.1, one major drawback of the old visualization is that users often experience waiting times when scrolling. When examining a large heap, the delay is often so drastic that the visualization cannot be used at all. Therefore, the main goal of this thesis is to rebuild the current heap visualization in a more efficient manner.

We understand that the main cause for delays in the old visualization is the underlying Heap data structure (not to be confused with the heap data structure based on a tree). Although it offers direct object access via binary search, it is still too slow to be used for a visualization. We do not aim to change this data structure as it is very memory-efficient and serves perfectly for other views in the AntTracks tool. Rather, we want to design our own data structure which is specifically designed for a visualization.

We also understand that such a data structure, which would have constant access time in the best case, might have a very high memory consumption. This also needs to be considered, therefore we might also need to think of how memory could be saved without sacrificing run-time performance.

4.2 Accurate Heap Visualization

As described in Section 3.1, the old visualization splits up the heap into three independent panels. We also pointed out that this causes problems when visualizing a G1 GC heap and might also cause problems with future garbage collectors. The new visualization should be able to show every heap, regardless of which garbage collector was used.

Furthermore, users may not only be interested in the contents of the spaces, but also in their locality. This can only be seen if the whole heap is visualized in one image. In order to produce an accurate image of the heap, we also need to visualize free space. This only makes sense in the byte view, where the size of an object is proportional to the number of pixels it spans. In the object view, one object can span at most one pixel. If we applied this system to free spaces, every free space,
regardless of its size would be at most one pixel large. The object view aims at giving an overview in terms of object counts ("Which objects were allocated the most?"), whilst the byte view reflects the actual memory layout and gives other insights on memory consumption ("Which objects take up most memory?").

We assume that objects are displayed in the correct order (i.e., ascending according to their addresses) in the old visualization. However, in order to guarantee an accurate view of the memory we will need to make sure that the object order is always preserved, no matter what garbage collector was used.

4.3 Support for User-Defined Classifiers and Filters

As described in Section 3.1, the "Filters" in the old visualization are very limited. The new visualization should incorporate the approach of user-defined classifiers and filters (cf. Section 2.3). This approach is flexible and extensible and is also used by the table-based statistics view, which is complemented by the visualization. We can use existing GUI components (cf. Figure 10 and 11 in Section 2.3) for filter and classifier selection.

4.4 Automatic Color Generation

As described in Section 3.1, the old visualization only uses five pre-defined colors to distinguish classified objects. If the number of classifications exceeds five, various classifications get assigned the same color. The only way to avoid this is to change the colors manually, which is a very tedious task if the number of classifications is large.

In order to avoid this, the number of colors that are automatically assigned should be as large as possible. Additionally, the colors ought not to be just be random colors but possibly complementary colors with high contrast in order to be distinguishable by the user.

Pre-defined color schemes with high-contrast colors usually only contain three to four different colors. Instead of relying on a set of pre-defined colors we need to find an algorithm that generates a large amount of colors that - ideally - are distinguishable by the human eye. Additionally, we still want to give the user the opportunity to select the colors if they wish to.
4.5 Automatic Cluster Size Adaption

As described in Section 3.1, the clustering mechanism was introduced in order to produce a small overview image if the heap size is large. The default cluster size, which is selected when the visualization is first started, is 5. This is an rather arbitrary choice - if a heap is very large, the image might still get big, on the other hand, if the heap is very small, the cluster size could be further reduced without resulting in a big image.

There is no one-size-fits-all cluster size for all kinds of heaps. The ideal cluster size depends on two factors: (1) the number of objects in the heap (or the number of occupied bytes in the heap if the byte view is selected), (2) the available space on screen. Both factors can be identified quite easily.

The cluster size should be automatically determined when the user first opens the visualization tab. It should be computed in a way that the resulting image fills the available screen space, i.e., no scrollbars are necessary. Additionally, the cluster size could also be adapted when components or the whole window is resized. Since users can adjust the cluster size manually, they also might want to reset it to the “ideal” cluster by issuing a command.

4.6 Detailed Object Info

Unlike other state-of-the-art tools, the AntTracks heap visualization does not provide any detailed information about the individual objects represented by one or more pixels (cf. Section 3). It’s possible to classify objects and color them, but we cannot infer any more information than that from a pixel.

An intuitive approach would be to display the available information (i.e., address, allocation site, type, etc.) when the user clicks on a pixel.

4.7 Zoom

As already described in Section 3.1, the zoom is not fully functional and should be replaced by a better zooming functionality.

The first problem we identified with the zoom is the way the zoom area is selected. As shown in Figure 16 in Section 3.1, the user draws a rectangle with the mouse. The problem with this way of selection is that the selected objects are arbitrary.
The first row of selected pixels contains adjacent objects, but the next row has no relationship to the first row. Therefore we suggest to use a selection mechanism like the one used in the Oberon Heap Inspector, where the entire rows between the first and the last selected pixel are magnified (see Figure 25 top in Section 3.2.2). By using this way of selection, all objects in the magnified area are adjacent.

The current zoom functionality does not allow to zoom in without having selected an area first. Such a zoom could be useful if the user wants just to zoom optically (i.e., increase the size of the pixels). Furthermore, in the current visualization, neither the current zoom factor nor the zoomed cluster size (is always 1) are reported back to the user.

The two-step zoom is not useful, but just a compromise due to the bad implementation of the old visualization: dissolving the cluster takes a long time, therefore the second zoom step is only done when the user issues a command. As we already remarked in Section 3.1, the first zooming step often leads to distorted images as pixels are magnified as rectangles rather than squares.

With the new zooming functionality, users should be able to issue a zoom command, optionally after a zoom area was selected. If such an area was selected, the pixels will be dissolved to a smaller cluster size, such that the resulting image fills the available screen space and magnified if possible. Without a previously selected area, only an optical zoom will be issued. Within a zoomed image it should be possible to zoom further down to individual objects. All zooming steps should be stored somewhere, so that the user can go back to the previous zoom adjustment. Like in the old visualization, after going back, the previously selected area should still be marked. The current zoom factor and cluster size should be shown to the user.

4.8 Pointer Visualization

As described in Section 2.2, AntTracks can also trace object pointers. If pointer information is available from the trace file, this could also be visualized graphically. The old visualization completely lacks this feature.

Like in the Oberon Heap Inspector (cf. Section 3.2.2), objects referenced by an object that was previously selected by the user (e.g., through clicking on it) could be displayed. This could be further extended: objects, that are referenced by objects referenced by the selected object could also be displayed. This could be continued until no more references are found or a certain level of indirection, set by the user, is reached. That way we could visualize the amount of memory that is kept alive by a certain or multiple objects.
The other way round, we could also display objects that reference the selected object. This could also be done for a certain amount of iterations or until no more referees are found.

4.9 User-friendly Interface

As shown in Figure 13 in Section 3.1, the old visualization contains many controls. We especially want to point out the existence of two “Apply” buttons. The new visualization should contain as few controls as possible. As the new visualization will also contain more features, it will be a balance act between providing all necessary controls and not making the window too crowded. In this respect we will favor committing changes in controls directly rather than using “Apply” buttons.

As already pointed out earlier in this section, information about changes in magnification or cluster size should be reported back to the user.

The old visualization shows the semantics of the currently displayed colors in a table style. When the number of colors is large, the user has to scroll around to find a certain color. Even worse, since a color might have multiple meanings they will never know how one particular object was classified. If no zoom is active, the colored pixels are very hard to distinguish for the human eye. To overcome both those problems, the new visualization should contain a small key component which updates while hovering over the image. It should display a magnified version of the pixels under the mouse cursor and some additional information on the used colors.

4.10 Heap Visualization over Time

Since it is possible to generate heap images for arbitrary points in the execution time of the application, it would also possible to generate these images all at once and let the user scroll through them. The user should be able to select images at points in time from a timeline or take a look at an animation that shows the development of the heap over time.
5 Implementation

This section describes the implementation of the new heap visualization which incor-
porates all requirements described in Section 4.

5.1 Architecture

As this application offers a graphical user interface and operates on a data structure
which is used by other components in the application, it is designated for the use of
the Model-View-Controller user interface paradigm [8]. The underlying data struc-
ture, the Heap class can be clearly identified as the model. However, as we already
pointed out in Section 4.1 this data structure is not suitable as visualization model
due to performance reasons. Therefore we need to add another layer of model which
grants fast data access. We can also incorporate the concepts of classifiers and fil-
ters into the new model. Methods of the classifier/filter classes are executed on the
objects represented by the Heap class and the results (i.e., classifications) are stored
in the new model layer. Information from the Heap class which is not necessary for
the visualization is omitted. The additional layer serves as a model, but imposes a
view on the underlying Heap class. Therefore these two model layers also share a
model-view relationship.

Figure 26 shows a schematic representation of the architecture. In order to achieve
more flexibility, we added another model/view layer to the architecture. The gener-
ation of the first model layer (class ObjectVisualizationData) can take up much
time, as iterating over the Heap is fairly slow. Drawing the image, on the other
hand, is usually a rather fast task. If the user changes the cluster size or switches
from object to byte view, the underlying object data (i.e., classifications and filtered
objects) does not change. Therefore, the cluster level and cluster size can be stored
in the next layer (PixelMap). Additionally, this class preserves an image buffer which
helps to increase the performance further (cf. Section 5.3.1).

We acknowledge that without the additional layer, we could sometimes omit some
data in the ObjectVisualizationData class. For instance, for the object view, we
would not need to store information about free space (“gaps”). However, we think
that shorter waiting times are a major contribution to usability, while the amount of
memory that can be saved otherwise is negligibly small.

The last layer (HeapPanel) is the actual graphical view. HeapPanel is the core of
the graphical view, which is comprised of some more classes, that serve as view and
controller at the same time. This layered architecture can be filed as an example of
the layered architectural pattern [16].
5.2 Object-based Visualization Data

This section describes the implementation of the class ObjectVisualizationData and related classes.

5.2.1 Data Generation

The data to be stored needs to be generated based on the Heap class. This class offers various methods to iterate over all objects. As defined in Section 4.2, we need to make sure that the objects are displayed in ascending order according to their addresses. To avoid sorting the objects after traversing the heap, it would be convenient if the heap iteration happens in correct order. Experimental evaluation showed that this is already the case. The iteration methods iterate over the Spaces, which happen to be in the right order. They are ordered by the IDs assigned by the garbage collector, which apparently allocates the spaces consecutively. However, if a new garbage collector would start giving descending or random IDs, the order would no longer be preserved. Therefore we created an additional iteration method which guarantees ascending iteration by ordering the Spaces according to their starting addresses before starting the iteration.
An object in the visualization is represented by its classification. As the user may apply multiple classifiers, the outcome may also be a set of classifications. Every unique classification or set of classifications is assigned a `java.awt.Color`. This mapping from classification(s) to Color is stored in a `java.util.Map<K, V>`, which is perfectly suitable for this purpose. When a new classification or classification set is encountered, a new Color is generated and a new map entry is added. The algorithm used to generate the colors is discussed in the following section. We cannot store the Color for the objects alone, because we want to keep the classification information in order to display it to the user (cf. Section 4.9). In order to retrieve the classification information more efficiently, we do not just store the Color object as value in the map, but an object (class `PixelDescription`) that contains both Color and classification object.

As described in Section 2.3, user-defined classifiers may have single objects (one-to-one classifier) or arrays (one-to-many and one-to-hierarchy classifiers). For the visualization, we do not need to distinguish between the different classifiers types. However, we need to take special care if classifiers return arrays as results. Arrays do not offer implementations of `equals()` or `hashCode()` based on their contents, but use the default implementation of `Object` which only considers two objects equal if they are the same. However, we consider two classification arrays equal if their contents are equal.

To overcome this problem, we created a new class that wraps around an array and provides it with content-based `equals()` and `hashCode()` methods as well as the `toString()` method (class `ArrayWrapper`). This is an implementation of the Adapter design pattern, also known as Wrapper pattern. For the `hashCode()` and the `equals()` method, the methods `hashCode()` and `deepEquals()` from the class `java.util.Arrays` were used. Arrays and the `ArrayWrapper` class are also used to create classification sets. If there are multiple classifiers selected by the user, they are consecutively applied and their outcomes (i.e., classifications) are merged into an array which is then wrapped.

For the visualization, filters can not be applied as Weninger et al. intended (cf. Section 2.3). If we apply a filter to the heap before iterating, we will not see the objects excluded by it. However, we do want to see them as we want to still draw them (e.g., as gray pixels). Therefore, we apply the filters manually.

---

3 Oracle: Color (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/Color.html](https://docs.oracle.com/javase/8/docs/api/java/awt/Color.html) (2017/06/12)
4 Oracle: Map (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/util/Map.html](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) (2017/06/12)
5 Oracle: Arrays (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) (2017/06/12)
The iteration itself is implemented as the Visitor design pattern [6]. When starting a heap iteration, one has to specify an object that implements the interface `ObjectVisitor` and its method `visit(SpaceInfo space, long addr, long[] pointedFrom, long[] pointsTo, ObjectInfo obj)`. For every object, this method is invoked. The class `ObjectVisualizationData` implements the interface. Within the method `generateData()` the heap iteration is started: `heap.toObjectStream().sorted().forEach(this)`

The method `visit()` in the class `ObjectVisualizationData` performs the following steps:

1. **Gap detection and storage.**
   By subtracting the address of the previous object, which is kept in a field, from the address of the current object and comparing it to the size of the previous object we can detect whether there is a gap between the previous and the current object. If there is one, we store the gap which resides at the address lastAddress + lastSize in a data structure for gaps.

2. **Filter application.**
   The currently selected filter(s) is/are applied to the current object. If the object is filtered, we assign it a special classification which denotes a filtered object and go to step 4. If the object is not filtered, we proceed to classifier application.

3. **Classifier application.**
   The currently selected classifier(s) is/are applied to the object and the outcomes are wrapped into `ArrayWrappers` if necessary. If there are multiple classifiers selected, the outcomes are merged and wrapped.

4. **Classification storage.**
   The resulting classification object is stored in a designated data structure. If the classification object occurs for the first time, it is assigned a new color which is then stored in the color map.

5. **Address storage.**
   The address of the current object is stored in a designated data structure.

6. **Preparation for next iteration.**
   The current address and object size are stored in the fields `lastAddress` and `lastSize` in order to be available for the next iteration.
5.2.2 Automatic Color Generation

As described in Section 4.4, we aim to generate as many high-contrast colors as possible. Two colors that share maximum contrast are considered complimentary colors. Such colors are located opposite to one another in the color wheel, i.e., 180° apart (see Figure 27).

Java’s java.awt.Color stores a color in the RGB color space. There is no straightforward way to compute the complimentary color of an arbitrary color stored as R, G, B values without converting it into another color model. The HSI family of color models uses three different parameters to store a color: hue (H), saturation (S) and intensity (I). The hue is a measure of the spectral composition of a color [14], i.e., it represents the chromatic value. It is stored as the angle in the color wheel and can therefore have a value between 0° and 360°. Consequently, we can get the complimentary color of an arbitrary HSI color by shifting the hue value by 180° and leaving the other values untouched.

We decided to use the HSL (hue, saturation, lightness; often also referred to as HLS) color system of the HSI family. Java does provide support for various color spaces and color space conversion, but does not provide an implementation for the HSL system.

---

color space. Therefore, we had to implement the conversion from RGB to HSL and vice-versa from scratch.

We developed a color generator (class ColorGenerator) that works like a random number generator: it is instantiated with a seed (starting color) and then consecutively generates new values (colors) when invoking a method (getNextColor()). The first call of this method returns the seed color. By increasing the hue of the starting color by 180° we would get the perfect complimentary color. However, if we call the method a third time, we would get the starting color again. Therefore, we need to find an angle which makes to consecutive colors distinguishable, but does not lead back to the original color quickly.

We settled for an angle of 55°. The least common multiple of 55° and 360° is 3960°, which means that after reaching this angle, we’re back at 0°, our initial seed color. Therefore we can generate 72 (3960°/55°) unique colors by using this algorithm. This may seem little compared to the number of possible classifications (cf. Table 1 in Section 5.2.5), however, we also need to acknowledge that the range of colors is finite and that many of them are hard to distinguish for the human eye. Compared to the five pre-defined colors that were offered by the old visualization (cf. Section 3.1), we consider this a good result.

Because of the well-structured, modular design of the color generation it is possible to easily exchange the color generator. In principle it is also possible to offer different color generators that create various color schemes and let the user choose which one to use. This can be considered part of future work (cf. Section 8).

5.2.3 First Storage Approach

Our aim was to create a model which guarantees constant access time. For every object, we need to store a classification. We do not need to store an own classification object for every heap object, it is sufficient to store all unique classification objects and just hold references to these objects.

Since every object in the heap needs a representation in the form of its classification, the data structure which stores this data can be considered the memory bottleneck of the entire visualization.

The heap can be considered to be a list of objects. Eventually, the visualization will display a two-dimensional image, where the first pixel is related to the first object in the heap and the last pixel is related to the last object. From a pixel coordinate (x,y) we can easily infer the index of the underlying object in the heap. Our first approach was to use the java.util.HashMap<K,V> data structure, which is a hash table based
implementation of the java.util.Map<K,V> interface that provides constant-time performance for the basic operations\(^7\). As keys (K), we used the indices of the objects in the heap. Since Java does not support the use of primitive data types for generics, we used the wrapper class java.lang.Integer\(^8\). As values we used the according classification objects of type Object.

By using this approach, we quickly ran into memory problems. The following calculation illustrates the problem. Assume we want to visualize a heap of 8 GB size with roughly 250 million objects. The selected classifier has only two possible outcomes, therefore, we need to store two classification objects of 16 bytes size each, which make up for the values. We need to store a key for every object of type Integer, which is 16 bytes large. The key-value relationship in a HashMap in Java is modeled by objects of type HashMap$\langle K,V\rangle$. Such objects are 32 bytes large. These objects are stored within the HashMap as an array. Object arrays contain pointers to objects and therefore one entry in such an array must be as large as the address width, namely 8 bytes on a 64-bit machine. However, the Java Hotspot™ VM offers the possibility to use so-called Compressed Oops\(^9\), which make it possible to store pointers with 4 bytes. In today’s Java versions, Compressed Oops are used by default if they can be used. They cannot be used for heaps that have more than four billion objects (not bytes), or a heap size of up to about 32 gigabytes.

In any case, the memory consumption of the HashMap can be roughly computed as follows: \(250 \cdot 10^6\) pointer size bytes for entries array + \(250 \cdot 10^6\) 32 bytes for entry objects + \(250 \cdot 10^6\) 16 bytes for key objects (Integer) + 2 \(\cdot\) 16 bytes for classification objects. For a pointer size of 8 bytes this is approximately 13.03 GB, for a pointer size of 4 bytes (using CompressedOops) it is 12.11 GB. We also need to keep in mind that this is just the memory consumption for the classifications, we have not even considered the object addresses which also need to be stored.

Other data structures from the Java 8 library, such as lists, suffer similar and additional problems, such as slow access time or slow resize operations. Therefore we decided to use a custom data structure, tailored to our needs.

\(^7\)Oracle: HashMap (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html (2017/06/22)
\(^8\)Oracle: Integer (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html (2017/06/22)
5.2.4 Second Storage Approach

The simplest data structure that maps an index to a value is an array. Instead of using a complex structure, we can just store all object classifications in one large Object array and, correspondingly, all object addresses in one large long array. Using this approach, we can reduce the memory consumption for our sample heap (cf. Section 5.2.3) in the worst case (no Compressed Oops) drastically: 250 mio. · 8 bytes object pointers to classification objects + 2 · 16 bytes classification objects ≈ 1.86 GB. The addresses require the same amount of memory, since the long data type is the same size as pointers (8 bytes).

Figure 28 shows a stripped-down sketch of this storage approach filled with sample data, showing the large Object array and the Map that is used to store the colors for each classification (wrapped in PixelDescription objects).

Arrays in Java are indexed with ints, i.e., an array can have at most Integer.MAX_VALUE ( = 2³¹ ≈ 2.1 billion) entries. However, there is no guarantee that a heap contains fewer objects than that. The only assured upper limit on
the number of objects in a heap is the range of `long`, since this range is equal to the address range on modern machines (i.e., 64 bits).

While this issue can be regarded as a rather rare corner case, we would run into another problem quite frequently when using one large array. The AntTracks tool is executed with the G1 garbage collector. As discussed in Section 2.1, it is discouraged to create large objects when using this garbage collector. In order to tackle both problems, we can use multiple small arrays for the object classification data and store them by introducing a second array dimension. We decided to create arrays of 512 kB size (i.e., 50% of the minimum region size). Therefore, the array length for the second dimension (actual data storage) is 512 kB / 8 bytes = 65536. However, we of course also need to limit the length of the first dimension in order to avoid the creation of a large object. With two dimensions, we can store $65536 \times 65536$ object classifications. This is more than `Integer.MAX_VALUE`, but still way beyond the upper limit. In order to be safe, we need four dimensions, which can store $65536^4 = 2^{64}$ object classifications.

In most cases, however, it will not be necessary to allocate a four-dimensional array, for most average-sized heaps two dimensions are completely sufficient. We only want to allocate as many dimensions as we really need. Instead of storing the array in a field of type `Object[][][][]`, we just store it in a field of type `Object`, which can be assigned any array. In order to allocate the right type of array before iterating over the heap, we equipped the `Heap` class with a method that returns the number of objects. We also need to remember the number of dimensions we allocated (“array level”), therefore we added another field of type `int`. With this information we can type-cast the `Object` field to its correct array type.

When writing to or reading from the array, we first need to perform the type-cast and then access the array at the correct position. The index of the classification is only given as one number of type `long`. We use a method which computes the individual indices in the different array dimensions from the given index and returns them as a array of length 1, 2, 3 or 4. The implementation of such a method will be described in the following section.

5.2.5 Final Storage Approach

Although the approach presented in the previous section reduces the memory consumption drastically, the memory demand of 8 (or 4 using Compressed Oops) bytes per object is unreasonably high. We were able to devise an approach which has a memory consumption of 1 byte per object in the best case.
The foundation of this storage approach is an observation about the number of classifications in relation to the number of objects in the heap.

Table 1 compares the number of objects in given heaps from real-world applications to the number of unique classification combinations for various selected classifiers. As shown here, some classifiers generate only very few classifications, as their possible outcomes are limited (e.g., *Space Type* and *Allocating Subsystem*). For other classifiers, the number of outcomes depends on the application itself (e.g., *Type* and *Allocation Site*). Even for those, the number of unique classifications is several orders of magnitude smaller than the number of objects. We also observe: the larger the heap, the larger the chasm. From this data we conclude that the number of objects usually outweighs the number of unique classifications by far. Section 7 provides more detailed data to back up this claim.

Every unique classification combination can be assigned a unique ID. Instead of storing a object reference of 8* bytes size, we store the ID. The number of required bytes depends on the number of classification combinations, in the overwhelming majority of cases this number will fit into an int (4 bytes), short (2 bytes) or even a single byte. In the worst case, we would also need 8 bytes for a long.

However, we also need to store the relationships between ID and the actual classification object. Therefore, the memory consumption would increase if we used long IDs and should not be used if the number of classifications exceeds the int range. Otherwise, we can use an array that maps from ID to classification object, simply by using the array index as ID. The ID will always be within the int range, but it might
<table>
<thead>
<tr>
<th>Objects</th>
<th>Combinations</th>
<th>Data type</th>
<th>Memory consumption 1st approach</th>
<th>Memory consumption 2nd approach</th>
<th>Saved memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>764,151</td>
<td>2,218</td>
<td>short</td>
<td>5.83 MB</td>
<td>1.47 MB</td>
<td>74.71 %</td>
</tr>
<tr>
<td>764,151</td>
<td>2,412</td>
<td>short</td>
<td>5.83 MB</td>
<td>1.48 MB</td>
<td>74.68 %</td>
</tr>
<tr>
<td>72,113,513</td>
<td>3,239</td>
<td>short</td>
<td>550.18 MB</td>
<td>137.57 MB</td>
<td>74.99 %</td>
</tr>
<tr>
<td>72,113,513</td>
<td>3,746</td>
<td>short</td>
<td>550.18 MB</td>
<td>137.57 MB</td>
<td>74.99 %</td>
</tr>
<tr>
<td>72,113,513</td>
<td>490</td>
<td>short</td>
<td>550.18 MB</td>
<td>137.55 MB</td>
<td>74.999 %</td>
</tr>
<tr>
<td>72,113,513</td>
<td>4</td>
<td>byte</td>
<td>550.18 MB</td>
<td>68.77 MB</td>
<td>87.499999 %</td>
</tr>
<tr>
<td>344,046,043</td>
<td>8,862</td>
<td>short</td>
<td>2.56 GB</td>
<td>656.28 MB</td>
<td>74.99 %</td>
</tr>
<tr>
<td>344,046,043</td>
<td>4</td>
<td>byte</td>
<td>2.56 GB</td>
<td>328.11 MB</td>
<td>87.499999 %</td>
</tr>
</tbody>
</table>

Table 2: Memory consumption of first and final storage approach compared

be larger than our limit of 65536. We can also apply the strategy of appending an additional dimension on demand and in this case two dimensions will suffice.

The required memory for this approach is: number of objects in the heap · (1, 2 or 4 bytes) + number of classifications · (8 or 4) bytes. We computed the amount of possible memory savings for the worst case (without CompressedOops). Table 2 shows that this approach pays off and that the amount of memory needed for the additional array is negligibly small, especially if the number of objects is large. The amount of saved memory in percent is very close to the theoretical maximum (75% for short or 87.5% for byte without CompressedOops). With CompressedOops, the additional array becomes even smaller, and we can ideally save almost 75% or 50% memory, respectively.

There is no way to know which data type is needed before iterating over the heap, as the number of classifications is only known after we finished a heap iteration and classified all objects accordingly. Therefore, we developed a data structure which is able to dynamically adapt to a larger data type. The object IDs are stored in an instance of this collection (class ByteCollection). This class offers a method to append a number (which could be an ID or an object address) to the end of the collection (writeUnknown(long value)), as well as a method to read a number at any index (readLong(long index)). To the calling class, ByteCollection behaves like one contiguous array that automatically grows and is able to hold up to Long.MAX_VALUE entries. Additionally, it offers a method to retrieve its size (the number of written values) and a special variant of a binary search which is useful for the byte visualization (cf. Section 5.3.4).
Conceptually, the `writeUnknown()` method operates on one contiguous byte array. The `ByteCollection` stores the current data type that is written, which is initially byte and may be enlarged but not downsized. By keeping three indices that denote the start of the short, int or long area the collection is able to interpret the written bytes correctly. It is also possible to skip data types, e.g., switch directly from byte to int if necessary.

Figure 29 sketches a sample `ByteCollection`. The `value index` is increased every time a number (value) is written to the collection and does not necessarily coincide with the `array index`, which is the actual index in the underlying byte array. Writing to the array is handled by the internal method `write(byte value)`, which just appends value at the end of the array.

Since the data type cannot be downsized, it is most likely going happen that we write down numbers wastefully, e.g., when the number to be written fits into a byte but the current data type is int. An approach similar to ours, Varints\(^1\), does not suffer from this problem. However, this approach is not feasible for us as we want to guarantee constant access time. When using Varints, we would need to read `n` numbers to access the number at the `value index` `n`. With our approach, we can simply compute the `array indices` from a `value index` by performing few case distinctions and simple arithmetic.

The `writeUnknown()` method first checks whether the value to be written fits into the current data type or whether the data type needs to be enlarged. Then, depending on the new or current type, either a helper method (`writeShort()`, `writeInt()` or `writeLong()`) or the `write()` method directly is invoked. The helper methods take care of enlarging the data type by checking whether the value to be written fits

---

\(^1\)Google Developers: Encoding. Protocol Buffers, [https://developers.google.com/protocol-buffers/docs/encoding#varints](https://developers.google.com/protocol-buffers/docs/encoding#varints) (2017/09/07)
into the current data type. They also check whether the method was called in an illegal state. Finally, they invoke the method write() for the individual bytes of the value to be written, using shift commands.

Note, that we do not use the whole range of the data types, because Java does not provide unsigned types. We decided that the possible benefits of using the entire ranges is not worth the time and effort of manually interpreting bits and converting them back to actual signed Java types.

The write() method takes care of the actual byte array. Here we used the approach of a multi-dimensional array introduced in Section 5.2.4. For the last dimension (i.e., the actual byte arrays), we can increase the array length from 65536 to 65536 · 8. To be able to store Long.MAX_VALUE bytes, we still need four dimensions. Since we do not know the number of values to be stored beforehand, we dynamically allocate new arrays and dynamically create new dimensions if necessary.

Figure 30 shows how the storage array (field data) is transformed into a two-dimensional array when the first dimension is full. The original array is assigned to the first entry of the newly allocated first dimension and for the second entry a new, empty byte[] is allocated. The third entry is allocated as soon as the second entry is full. The expansions from two to three dimensions and three to four dimensions work in the same way.

The class ByteCollection offers two methods for reading: readLong() and readInt(), both take an index of type long and return a value of type long or int. Additionally, readInt() will throw an exception if the value to be read was stored as long.

The parameter index in the read methods refers to a value index. The methods compute the actual array indices from index and then invoke the internal method
read(long index) with the corresponding array indices. Then they restore and return the value by performing bit-wise logical and shift operations on the read bytes.

Since the read() method uses array indices of type long, we see that the number of values written to the conceptual contiguous byte array can be at most Long.MAX_VALUE. Therefore, the entire ByteCollection can only store that many values if they are all of type byte. In the worst case (only long values stored), the ByteCollection can store at most Long.MAX_VALUE/8 values. This is equivalent to $2^{60}$ entries, while a Java heap on a 64-bit machine may have up to $2^{61}$ objects in theory (due to the 8 byte alignment). Considering today’s memory sizes we dismiss this scenario as highly unrealistic and claim that this is a feasible solution for Java running on 64- or less bit machines.

The conversion from an array index to the actual indices of the different dimensions of the array, which is needed for reading and writing, is done by a method. This computation is straightforward and intuitive for arrays with one or two dimensions but gets trickier when more dimensions are involved. We derived a general formula that works for arrays with $n \in \mathbb{N}$ dimensions. The lengths of the individual dimensions do not need to be the same (however, all arrays in one dimension need to be of the same size). Assuming a function len(dim) that retrieves the length of a specified dimension $dim$, any index $d$ in a dimension $d$ for a given array index can be computed using the following formula:

$$index_d = \left(\frac{arrayindex}{\prod_{i=0}^{d-1} \text{len}(i)}\right) \mod \text{len}(d)$$

In the class ObjectVisualizationData we use a ByteCollection to store the classification IDs per object. If we run into the case that the number of unique classifications exceeds the int range, we dismiss the ByteCollection, revert the data we stored and fall back to the storage approach discussed in Section 5.2.4. Note that we have never encountered an application where this had to be done.

Additionally, another ByteCollection is used to store the object addresses. To save at least some memory here, we store the addresses relative to the first object address which is kept in an additional field of type long. The gap addresses are also stored within a third ByteCollection relative to the first address. We also exploit the fact that addresses are aligned and omit some bits by performing a division (e.g., divide by 8 for 64-bit addresses).
5.2.6 Swapping with Memory-Mapped Files

Although the approach presented in the previous section cuts down on the visualization’s memory consumption, visualizing the heap remains a memory intensive task. Furthermore, users may want to visualize different heaps from the same or different applications at the same time, which means that multiple instances of ObjectVisualizationData reside in memory. This also happens when using the heap visualization over time (cf. Section 4.10), where visualization data for all available heaps in the application is generated.

An approach to reduce the pressure on the memory without sacrificing the constant access time is to swap data to the disk. Conventional I/O is rather slow because it requires almost always one or more copy operations to move the data between filesystem pages in kernel space and a memory area in user space. However, there is a special type of I/O operation supported by most operating systems that allows user processes to take maximum advantage of the page-oriented nature of system I/O, which results in fast file access. A memory-mapped file pretends to be entirely in memory and can be accessed by treating it as a very large array\textsuperscript{11}.

In Java, the package java.nio offers support for memory-mapped files. The class java.nio.MappedByteBuffer\textsuperscript{12} is a byte buffer mapped to a file. It offers random access, bytes can be written (put()) or read (get()) to/from arbitrary positions within the buffer. Since the class ByteCollection also writes and reads bytes, this constitutes a viable alternative to storing the bytes in arrays.

Since a ByteCollection in memory is a sufficient solution for many cases, we only want to use memory-mapped files if really necessary. In order to support both storage variants, we adapted the class ByteCollection and turned it into an abstract class. All methods that conceptually operate on a byte array remained in the class, only the internal methods that deal with writing to/reading from the actual array were left to be implemented by the subclasses. MemoryByteCollection works with arrays, while MMFByteCollection operates on memory-mapped files.

In order to create a new MappedByteBuffer, we need to specify a file to which it should be mapped. We create a temporary file for the memory-mapped file and request that the file is deleted after the application is terminated normally. If the JVM terminates abnormally, the delete request is not issued. By creating temporary files we can ensure that these files are at least deleted when the computer is

\textsuperscript{11}Java NIO - Memory-Mapped Files with MappedByteBuffer - HowToDoInJava, \url{http://howtodoinjava.com/java-7/nio/java-nio-2-0-memory-mapped-files-mappedbytebuffer-tutorial/} (2017/06/26)

\textsuperscript{12}Oracle: MappedByteBuffer (Java Platform SE 8), \url{https://docs.oracle.com/javase/8/docs/api/java/nio/MappedByteBuffer.html} (2017/06/26)
rebooted. We also need to specify the length of the mapped byte buffer, which we set to the maximum, Integer.MAX_VALUE\(^{13}\), which means we can store that many bytes in the buffer. As a consequence, we need to introduce further dimensions. To be on the safe side, we need once again four dimensions. Since the class MappedByteBuffer behaves like an array, the largest construct we might create would be of type MappedByteBuffer[[],[]]

An important question is how to decide when to create which type of ByteCollection. We equipped the class ByteCollection with a static method which instantiates a corresponding subclass by considering an estimated memory consumption. In order to determine the presumable amount of free memory, we used methods from the class java.lang.Runtime\(^{14}\). In case the creation of a MMFByteCollection fails due to I/O problems, a MemoryByteCollection is used.

The memory estimation needs to be performed by the caller, which is the class ObjectVisualizationData. We assume that the number of unique classification combinations will be within the short range, therefore we need 2 bytes for every object in the heap for the first ByteCollection. For the addresses we will eventually need 8 bytes per object. Note that we need to add the demand of the previous collection, since it was created but not filled with data yet. For the gaps we use the very pessimistic estimation of 1024, which would be sufficient for 128 gaps stored with 8 bytes each.

5.3 Pixel Map

The Pixel Map constitutes the next level in our layered architecture (cf. Section 5.1). The important concepts on this level are the cluster size and the cluster level, as well as an image buffer. While the cluster size is just an ordinary number, the cluster level impacts the way the resulting image is drawn. PixelMap is an abstract class that provides an interface to GUI classes. It has two subclasses, ObjectPixelMap and BytePixelMap, which implement algorithms used to visualize the heap in terms of objects or in terms of bytes. Common code is present in the superclass, while view-specific implementations are part of the subclasses.

The class PixelMap provides methods to retrieve the image buffer, cluster size and the pixel count (the number of drawn pixels - this is not necessarily equal to the prod-

\(^{13}\)Oracle: FileChannel (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileChannel.html#map-java.nio.channels.FileChannel.MapMode-long-long](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileChannel.html#map-java.nio.channels.FileChannel.MapMode-long-long)

\(^{14}\)Oracle: FileChannel (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html)
uct of image width and height, since the last painted row does not have to be entirely filled). Further basic operations are the drawing of the heap to the image buffer with the desired cluster size as well as a method to retrieve a PixelDescription for a pixel at a specified position (x, y) in the image.

5.3.1 Image Buffer

Experimental evaluation showed us that we can optimize the actual painting of the heap image to the screen by using an image buffer. When drawing pixel per pixel to a java.awt.Graphics object in a javax.swing.JComponent surrounded by a javax.swing.JScrollPane, the users will experience waiting times when scrolling around. These waiting times completely disappear when directly painting a BufferedImage with the same amount of pixels to the Graphics object. The delay is not caused by our underlying model, as this effect is observable even when the pixel colors are not looked up anywhere but randomly generated. The painting process itself is the problem.

The class BufferedImage has some limitations, the width and the height of the image are stored as int and are therefore confined. This limitation is even stricter, the product of width and height must not exceed Integer.MAX_VALUE. This is a limitation of a class which is used by BufferedImage. We do not consider this limitation to be problematic. The cluster size can always be adapted in a way that the heap fits on such an image and if the user wants to investigate further, they can zoom in to an area of interest and the cluster size is automatically decreased. However, we do need to consider these limitations in the implementation.

An instance of BufferedImage needs to be kept in memory. By using an image buffer we impose an additional load to the memory. In order to avoid this, we decided to use an external class, BigBufferedImage. It has the same limitations as the class BufferedImage, but it stores the image on the disk, so we do not have to worry about any memory limitations. It is implemented as a subclass of BufferedImage,

---

15Oracle: Graphics (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html](https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html) (2016/06/26)

16Oracle: JComponent (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/JComponent.html](https://docs.oracle.com/javase/8/docs/api/java/awt/JComponent.html) (2016/06/26)

17Oracle: JScrollPane (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/JScrollPane.html](https://docs.oracle.com/javase/8/docs/api/java/awt/JScrollPane.html) (2016/06/26)

18Oracle: BufferedImage (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/image/BufferedImage.html](https://docs.oracle.com/javase/8/docs/api/java/awt/image/BufferedImage.html) (2017/06/26)

19Oracle: SampleModel (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/awt/image/SampleModel.html](https://docs.oracle.com/javase/8/docs/api/java/awt/image/SampleModel.html#SampleModel-int-int-int-int-) (2016/06/26)

so we can just initialize our image buffer with objects of this class. If the creation
of a BigBufferedImage fails for some reason (i.e., I/O problems), we fall back to
conventional bufferedImages.

5.3.2 Shared Code

The painting of the heap to the image buffer is a task that needs to be handled by the
subclasses of PixelMap. However, some actions are independent of the current view
and can therefore be shared. Specifically, the method which handles the painting
needs to perform the following steps:

1. Set the current cluster size.
2. Compute the width and the height of the image buffer.
3. Instantiate the image buffer.
4. Paint the heap to the image buffer.

Steps 1 to 3 are handled by the class PixelMap, step 4 by its subclasses.

Setting the cluster size. The cluster size can be set to any valid cluster size by
the user via the UI. Additionally, PixelMap is able to compute the ideal cluster size
automatically. This is done if the constant PixelMap.UNDEFINED_CLUSTER_SIZE is
passed as new cluster size, in order to meet the requirement of automatic cluster
size adaption (cf. Section 4.5). Cluster sizes may vary from 1 to 1024 (=1k), 1k
to 1024k (=1M), 1M to 1024M (=1G) and 1G to 1024G. The cluster size is stored
as one long value. A value of 2 resembles a cluster size of two, a value of 2048
resembles a cluster of 2k and a value of 2049 is considered invalid.

The ideal cluster size is computed by dividing the object or byte count by the avail-
able space on the screen. This value needs to be retrieved by a method which is
implemented in the subclasses, as they determine whether which of those two counts
is relevant. The available space on the screen is the product of the width and the
height of the UI component which displays the heap image. Unfortunately, we need
to access the view from a model layer here, but for this purpose this is inevitable.
After performing the division (clusterSize = ⌈count/(width · height)⌉), we check
whether the computed cluster size is valid and increase it to the next valid size if
necessary. We also need to further increase the cluster size if the resulting image
exceeds the pixel limit of the class BufferedImage (this also needs to be checked
and done if the cluster size was set manually).
Computing width and height of image buffer. When the cluster size is set, we can calculate the actual size of the resulting image. We need to divide the pixel count assuming a cluster size of 1 by the actual cluster size to get the actual pixel count. The width is given by GUI component which displays the image and the height can also be calculated by dividing the actual pixel count by the width. Unlike in the old visualization (cf. Section 3.1), we only want to display a vertical scrollbar if the image’s height exceeds the panel’s height. We also want to avoid horizontal scrollbars. Therefore, the width of the image must be reduced if the vertical scrollbar is displayed. This can be easily determined: if the computed image height exceeds the displaying panel’s height, we subtract the scrollbar width from the image width and re-calculate the image’s height.

Instantiating the image buffer. The implementation is described in Section 5.3.1.

Painting the heap to the image buffer. This is completely handled by the subclasses ObjectPixelMap and BytePixelMap and described in the following sections.

5.3.3 Object View

As hinted in the previous section, we need a method in the class ObjectPixelMap which returns the total object count of the heap to be displayed. This constitutes a special case of a method with a broader application, namely computePixelCount(long clusterSize). When setting clusterSize to the minimum (i.e. 1), we get the object or byte count. The object count is retrieved from ObjectVisualizationData, which offers a method that returns this value. The method computePixelCount() just returns \[\lceil\text{objectCount}/\text{clusterSize}\rceil\]. The public method getPixelCount() performs the same calculation, but uses the current cluster size of the PixelMap instance instead of a parameter.

The painting of the heap in the object view is trivial. We iterate over all objects and retrieve their PixelDescriptions from the underlying ObjectVisualizationData instance. A cluster size of 1 constitutes a special case which is handled specifically. Within every iteration, we retrieve the PixelDescription and then immediately draw the pixel. If the cluster size is larger, we need to aggregate the PixelDescriptions of the objects within the cluster before drawing. A pixel is drawn in every clusterSize\(^{th}\) loop iteration. The color of the PixelDescription with most occurrences is used. Unlike in the old visualization, we do not adjust
the alpha value of the pixel (cf. Section 3.1), because we do not think that this contributes to usability. However, the code is well-structured and modular so that this behavior can be easily modified if desired.

The method which retrieves a PixelDescription from a given x,y coordinates works in a similar fashion. First, we compute the index of the pixel at (x,y). The pixel at (0,0) is the first pixel, the pixel at (1,0) the second, and so forth. If the cluster size is 1, we just retrieve the PixelDescription at the pixel index from the underlying ObjectVisualizationData instance and return it. If the cluster size is larger than one, we retrieve clusterSize PixelDescriptions from pixelindex \cdot clusterSize to pixelindex \cdot clusterSize + clusterSize (exclusive) and aggregate them in the same fashion as when painting. We only need to be careful when dealing with the last pixel, as it may represent less than clusterSize objects (if clusterSize is not a divisor of the object count).

5.3.4 Byte View

We had to add some functionality to the class ObjectVisualizationData in order to support a byte view. Retrieving the byte count is fairly easy: byteCount = lastAddress + lastSize - firstAddress (all of these are fields of the class ObjectVisualizationData).

In the object view, we retrieve a PixelDescription for an object at a specified index during painting and when retrieving information about an individual pixel. In the byte view, we need to retrieve PixelDescriptions for a byte at a specified index. The first byte in the heap belongs to the first object, i.e. the object starting at firstAddress. The following bytes up to the size of the first object also belong to the first object, the next bytes to the second object and so forth. In the byte view, we also need to take the gaps into account, which are stored in a separate collection.

We also added a method which retrieves a PixelDescription for a specified byte index. The byte index can be viewed as an address. Since we store the addresses relative to the first address (cf. Section 5.2.5), the first address in the ByteCollection instance is 0. We also need to consider that we omit the address bits that do not contain information by performing a division with the heap word size, therefore we also need to perform this division on our address (byte index).

We use the binary search method of ByteCollection (cf. Section 5.2.5) to retrieve the starting address of the object which contains our byte. As shown in Figure 31, this binary search variant returns the index of the value if it was found or insertion point - 1 otherwise. The insertion point is the index in the array where the search key would need to be inserted in order to keep the array sorted.
Since the first entry in the ByteCollection which stores the addresses is always 0 and our search key can never be less than 0, the smallest possible address index we can find is 0. If the byte we are looking for is within the address range before the first gap, the binary search of the collection that stores the gaps will return -1. In this case, we can simply return the PixelDescription for the object at the index we found in the object address collection. Otherwise we need to read the addresses at the found indices from the object address as well as the gap collection and compare their distance to our address (index). If our address is closer to an object address, we return the PixelDescription of that object, otherwise we return a constant PixelDescription that represents a gap (no classification, color white).

The described method is used if we want to retrieve the pixel description for one single byte. If the cluster size is set to one, we have to call the method once for every pixel. For larger cluster sizes, we would need to call this method clusterSize times and start that many binary searches. This would cause a bad performance for large cluster sizes and is also not necessary, because the bytes within a cluster pixel are always adjoining. In many cases, all or most cluster bytes belong to the same object. Therefore, we added another method, which returns a set of PixelDescriptions for a specified amount (size) of consecutive bytes starting at a specified index. The PixelDescription for the starting is retrieved using the technique described previously. Then, we calculate the remaining bytes of the object by subtracting our address (index) from the closest next address (object or gap). If this is equal or greater to size, we are finished, otherwise we go to the next object/or gap and continue until we have size bytes. The method returns a Map instance that maps a PixelDescription to a number, which denotes the number of bytes with that PixelDescription. This Map is interpreted by the class BytePixelMap.

For the painting itself it is not reasonable to use a binary search, as we need to iterate over all objects and gaps anyway. Therefore, we equipped the class

\[
\text{search key} = 0x10 \quad \text{insertion position} - 1
\]

\[
\begin{array}{cccccccc}
0x00 & 0x04 & 0x08 & 0x0C & 0x12 & 0x16 & 0x1A & \ldots \\
\hline
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]

Figure 31: Binary search in ByteCollection
ObjectVisualizationData with a method that returns an Iterator\(^{21}\) (cf. *Iterator* pattern \(\text{[6]}\)). This *Iterator* returns objects of the type BytePixelDescription, which just contains a PixelDescription of a corresponding object or the gap PixelDescription and the size of the object or gap. The iteration starts at the very first address and the object or gap size is computed by subtracting the current address from the closest next address.

The painting itself also becomes a bit more complicated compared to the object view. A cluster size of 1 constitutes a special case once again. In a loop, we retrieve the next BytePixelDescription instance using an *Iterator* and paint as many pixels as the object size in bytes. However, we also need to consider line breaks within the image, therefore we need a nested loop which paints the pixels of the current object row-wise.

If the cluster size is larger, we potentially have to deal with multiple objects in one pixel, like in the object view. However, an object may also span multiple pixels. Like with cluster size 1, we need a main loop which iterates over the heap and a nested loop that spreads the bytes of the current BytesPixelDescription over pixels. We need to keep two counters: the remaining bytes of the current BytesPixelDescription and the number of bytes we already read for the current pixel. We store the PixelDescriptions for the current pixel in a Map along with a byte count. Within the inner loop, we need to distinguish three cases:

1. All remaining bytes of the current BytesPixelDescription fit into the current pixel and the current pixel is not full yet. This will set the remaining byte count to 0 which will cause the inner loop to terminate and let the outer loop retrieve the next BytesPixelDescription.

2. All remaining bytes of the current BytesPixelDescription fit into the current pixel and the current pixel is full. This will also set the remaining byte count to 0 and additionally paint the current pixel.

3. The number of remaining bytes exceeds the capacity of the current pixel. This will cause the current pixel to be drawn, but the inner loop will not terminate yet as the remaining byte count is decreased but not set to 0.

---

\(^{21}\) Oracle: *Iterator* (Java Platform SE 8), [https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) (2017/06/28)
5.4 User Interface

The graphical user interface represents the last layer of our visualization architecture (cf. Section 5.1). Figure 32 shows the new visualization tab.

At the very top of the tab there is the GUI component for classifier and filter selection (see Figure 10 and 11 in Section 2.3). By pressing the “Configure” button the component can be expanded and the classifier and filter selection can be modified. The “Select Colors” button enables the user to select the classification colors (cf. Section 5.4.6) and the “Export as .png” button lets the user export the current heap image as PNG file (cf. Section 5.4.7).

The large panel in the middle displays the heap image (cf. Section 5.4.1). Below this panel, there are three smaller panels: the key panel (cf. Section 5.4.2), the configuration panel (cf. Section 5.4.3) and the object info panel (cf. Section 5.4.5).

At the very bottom there is a small panel which displays information about any ongoing operations (cf. Section 5.4.4).
5.4.1 Heap Panel

The main task of the heap panel (class HeapPanel) is to display the image buffer from the underlying PixelMap instance. HeapPanel is a subclass of JComponent and overrides the method paint() in order to display the heap image. If the image is larger than the available space on screen, only the portion of the image that is currently visible is painted.

Additionally, the class is equipped with a java.awt.event.MouseAdapter\(^{22}\) in order to react to mouse clicks and movements. Since these events may trigger actions in other components, we introduced a custom listener (implementation of the Observer design pattern \([6]\)) which forwards mouse events. This listener is used by the key components as well as the object info panel.

5.4.2 Key Component

We implemented a small key component as described in Section 4.9 to increase usability. Figure 33 shows a screenshot of the key. The component updates when the user is hovering over the heap panel and shows 5·5 enlarged pixels. The pixel in the middle is the one which the mouse currently points to.

Next to the enlarged pixels, there is a list of the occurring classifications and the according colors. Horizontal and/or vertical scrollbars appear if necessary. In order to operate a scrollbar, the user needs to hover back to the key component, which would mean the key component updates again and the current pixels in the key would disappear again. In order to avoid this, the user can perform a click within the heap panel. The key component then freezes and stops updating when the mouse is moved. The component un-freezes when the mouse is moved back into the heap panel after it had been moved out of it before.

---

\(^{22}\)Oracle: MouseAdapter (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseAdapter.html (2017/06/28)
The key component is implemented using the method from the class PixelMap which queries PixelDescriptions for a x,y coordinate. Because of the fast underlying data structure in the class ObjectVisualizationData and the efficient implementations in the subclasses of PixelMap the key component updates smoothly.

5.4.3 Configuration Panel

In the top row of the configuration panel (see Figure 34) there are controls to manipulate the cluster size. The first three controls are identical to those in the old visualization (cf. Figure 14 in Section 3.1). Unlike in the old visualization, selecting a unit (k, M, G) from the unit box also has effects when the object view is selected, e.g. selecting 2k objects will lead to a cluster size of 2048 objects per pixel. The additional button labeled with a unmagnified glass resets the zoom (back to 100%, show the entire heap) if there was any and lets the application determine the ideal cluster size. If the cluster size was automatically adapted, the according values are set in the size spinner and the unit box.

The components in the bottom row are used for the zoom and pointer visualization.

5.4.4 Background Working Threads

Generating a new instance of ObjectVisualizationData or painting the heap to the image buffer are tasks that take up some time. If we would issue them in the GUI thread, the complete user interface would freeze until the corresponding task was completed. In order to avoid this, we need to let this operations run within own threads.
The AntTracks tool uses the concept of workers for this purpose. The class $\text{AntWorker}<R,P>$ is a subclass of $\text{javax.swing.SwingWorker}<T,V>$ which takes care of the background execution of tasks. We created two implementations of $\text{AntWorker}$ for the visualization: $\text{VisualizationWorker}$ and $\text{VisualizationPaintWorker}$.

A $\text{VisualizationWorker}$ returns an $\text{ObjectVisualizationData}$ object and is executed when the classifier and/or filter selection was changed. When such a worker is finished, it immediately triggers the execution of a $\text{VisualizationPaintWorker}$, which executes the task of painting the heap to an image buffer. When the user manually changes the cluster size or cluster level, issues the command to automatically adapt the cluster or the heap panel component was resized, a $\text{VisualizationPaintWorker}$ is issued as well. This is sufficient since a change in cluster size does not change the underlying data.

Initially, when the visualization tab is opened, a $\text{VisualizationWorker}$ is started. The first worker which then paints the heap needs to create a new $\text{PixelMap}$ object. If the classifier selection is changed afterwards, we only replace the $\text{ObjectVisualizationData}$ object of the current $\text{PixelMap}$ with a new one. If the user changes the cluster level, we need to create a new $\text{PixelMap}$ object of the opposite type (object vs. bytes) and use the existing $\text{ObjectVisualizationData}$ instance.

As we required in Section 4.9, the new visualization contains no “Apply” buttons (cf. Figure 32), all changes in components are immediately committed, i.e., a corresponding worker is started. If a user wants to select two new classifiers, they first select the first classifier and a new worker is started. While this worker is still running, the user selects the second classifier and another worker is started. At this point, the result of the first worker is useless, as the user already changed the selection. We therefore wish to abort the first worker.

Instances of $\text{SwingWorker}$ can be canceled, if the task that is executed by them supports it. It offers a method $\text{cancel()}$. In order to support cancellation, we added an additional parameter to methods in the classes $\text{PixelMap}$ and $\text{ObjectVisualizationData}$ of type $\text{AntWorker}$, which denotes the worker that executes the methods. We instrumented the methods to check whether the worker was canceled in every iteration and throw a $\text{java.util.concurrent.CancellationException}$ if that is the case. The exception then causes the worker to abort, by catching it we can perform some cleanup tasks if necessary.

---

23Oracle: SwingWorker (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html (2017/06/29)
24Oracle: CancellationException (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CancellationException.html (2017/06/29)
The heap panel is only updated if the painting was successfully finished. While a background task is running the user can still perform operations on the image that is currently shown. If a user selects a new classifier and then de-selects it before the worker that was just started finished, it is actually not necessary to start a second worker because we already generated the data for the current set of classifiers. Therefore it is also checked whether a new set of settings is equivalent to the set of settings that was applied in the image that is currently displayed.

5.4.5 Detailed Object Info Panel

When the user clicks on a pixel in the heap panel, the object info panel shows detailed info about the object(s) represented by that pixel (cf. Section 4.6). The detailed information comprises the object address, object type, allocation site, object size and the classification. As most of this information is not stored in our visualization model, we need to query it from the underlying heap data structure. Said class performs a binary search to retrieve the object info for a object at a specified address. This is feasible if we retrieve information for few objects. However, for larger cluster sizes, the number of objects represented by one pixel becomes larger and therefore performance degradations may arise. Therefore, we limit the amount of displayed object information to the ten first objects, but indicate that there are more objects in this pixel. The user can decrease the cluster size or zoom in if they want to investigate further.

The currently displayed object(s) are marked in the heap panel, as the corresponding pixel(s) are painted as black in the heap panel. We define the corresponding object(s) to be selected. In the key component the original color appears. In this case it is not necessary to re-paint the entire image buffer, we can just draw the black pixel(s) in the heap panel at the click position over the image buffer. For the object view, this is straight-forward: we just need to draw on pixel at the position where the user clicked.

A selection in the object view always contains one or more entire objects. In the byte view, we also want to select the entire object, which may span multiple pixels. If the pixel that was clicked on contains multiple objects, we want all of those objects to be selected and therefore mark the area from the start of the first object in this pixel to the end of the last object in the pixel. Therefore we equipped the class PixelMap with a method which returns the area, the objects in a specified pixel, span. The implementation in the class ObjectPixelMap is trivial, while the implementation in the class BytePixelMap requires some computations. If the user clicks on a gap in the byte view or on a filtered object, no selection is performed.
Similarly, we need to retrieve the number of objects in the clicked pixel in order to display this in the object info panel. For that purpose, we also added a method which returns the number of objects at a specified pixel. The implementation for the object view just returns the current cluster size.

Figure 35 shows the object info panel displaying some information in the byte view, as well as a clipping from the heap panel, showing the selected object which spans multiple pixels in black.

5.4.6 Color Selection

Figure 36 shows the color selection dialog, which allows the user to manually select the desired colors for the individual classifications. Even if the user does not want to change the colors, the dialog is useful as it displays a complete list of existing classifications in a table.

The table columns may be sorted and re-ordered. When clicking on a color, a color picker dialog opens which allows the user to change the current color. As soon as a new color is selected, it is set in the model (in the color map in the class ObjectVisualizationData) and a new worker that repaints the image buffer is started.
5.4.7 PNG Export

The heap image can be exported as PNG image to the disk. The user needs to set two of the following three parameters: image width, image height, cluster size; the third parameter is then inferred. A second BufferedImage object with the specified and/or computed width and height is instantiated and the heap is painted to that image using the same methods that paint to the original image buffer. The BufferedImage is then stored as PNG file, the user can specify the file path and name.

5.5 Intelligent Zoom

The zoom functionality in the new visualization fulfills all requirements described in Section 4.7.

5.5.1 Pixel Map Adaptions

In order to support a multi-level zoom, we need to use a data structure which can stack multiple zoom levels. The obvious choice here is a stack data structure. When zooming in further, a new zoom level is pushed onto the stack, when zooming out, the last zoom level is popped from the stack. Using this data structure also ensures that while zooming out, the user is lead back along the same path on which they entered the zoom.

Two things may be altered when zooming in: the cluster size and the pixel size. The pixel size denotes the optical zoom factor. Therefore, we need to store these two values on the zoom stack. Since we also want to zoom into parts of the image, we need to store the currently selected as well. The area is stored as two indices, referred to as start index and end index. These values denote the pixel indices of the first and the last pixel that are displayed and are stored as pixel indices of the previous zoom image. Figure 37 illustrates this. In order to compute the absolute start index (i.e., the pixel index within the entire heap image with cluster size 1), we need to traverse the zoom stack. The following formula shows the computation of an absolute starting index using the zoom stack. The length of the zoom stack is given as \( len \), \( startIndex_i \) and \( clusterSize_i \) are queried from the zoom stack.

\[
absoluteStartIndex = \sum_{i=1}^{len-1} startIndex_{i-1} \cdot clusterSize_i
\]
Besides cluster size, pixel size, start index and end index, we also store the selected area of the current image. This is used to generate the indices of the next zoom level when a zoom command is issued. Since we can zoom in without selecting an area first, the selected area may also be empty. We store the selected area as two two-dimensional coordinates \((x, y)\). That way, when a zoom level is popped from the stack, the UI component can just request the selected area from the new top of the stack and display it (so that the area remains selected when zooming out).

The zoom stack has at least one entry, which describes the state of no zoom: the pixel size is 1 and the area is the entire heap (from 0 to the last pixel). If the size of the zoom stack is one, we can not zoom out any more. When zooming in, we create a new object of type `ZoomStackData` and push it onto the stack. If an area in the current image is selected, we compute the start and end index and also decrease the cluster size so that the area fills the available space on the screen, just like we do when we automatically determine the ideal cluster size. If there is still space left, we also zoom optically and increase the pixel size. If no area is selected, the entire area of the current image is the area of the next zoom level and we only zoom optically. We limited the maximum zoom level to a pixel size of 40 (i.e, zoom factor of 4000\%). The number of zoom iterations (i.e., entries on the zoom stack) until this zoom factor is reached is variable.

The pixel size affects the way we compute the ideal cluster size, the way we paint the heap to the image buffer as well as the way we compute pixel indices from coordinates. Instead of working with the width and height directly, we need to work with rows and columns.
The index computation is view-independent and therefore implemented in the class `PixelMap`. However, we need to consider the implications this has on the byte view: one object may spread around multiple pixels. If the user selects the starting pixel to be in the middle of an object, we would only draw a part of the object. Since the smallest unit of interest on the heap is an object and not a byte, we think drawing objects only partially is misleading, therefore we would need to go back to the pixel where the object starts and draw from there.

Things become even worse when objects are spread around multiple pixels and pixels contain multiple objects at the same time. An example to illustrate this: the cluster size is 4 bytes and the selected starting pixel contains the 2 last bytes of one object and the 2 first bytes of the neighboring objects. We could now just forget about the end bytes of the first object and just draw the second or we could find the starting address of the first and start the zoom area from there. However, there are strings attached with both solutions. If we just draw the last objects represented by this pixel, we deliberately omit information. This is particularly problematic if the cluster size is large. As for the second strategy, the first object could be very large and span multiple pixels. The user may select a relatively small area, but get a unreasonably large zoom image as result. This behavior would be confusing to the user. Therefore, we developed a better solution: we allow the user only to select pixels as starting pixels where the first byte of the pixel is the first byte of an object. Gaps constitute an exceptional case, within them, the user may select any arbitrary pixel.

5.5.2 Adaption in the User Interface

In order to enable selection, we needed to modify the class `HeapPanel`. We compute and set the selected area in the class `PixelMap` when the user clicks on a pixel and drags the mouse. In order to restrict selection only to allowed pixels (for the byte view), a method in the class `PixelMap`, implemented by its subclasses, sets the coordinates of the selected area to the first legal starting pixel before the original starting pixel and the end pixel to first legal ending pixel after the original ending pixel.

The selected area is retrieved from the underlying `PixelMap` instance. If such an area exists, it needs to be painted darker. In order to achieve this, we retrieve the color from the image buffer for every pixel within the selected area, create a new color which is darker than the original and draw the re-draw the pixel with the darker color.

With this implementation, the selection automatically jumps to the correct selection points and the user immediately sees what happens.
Figure 38 shows the visualization in zoomed mode after some zooming steps. The current optical zoom factor is 300%, and the dissolved cluster size is 1. This is reported back to the user in the configuration panel. We prohibit the user from manually setting the cluster size when in zoomed mode. Although technically possible with our implementation, we think that the cluster size dissolution should not be performed manually. This also makes sure that the cluster size decreases or remains the same throughout the zoom stack.

The key component and object info panel also work as usual in the zoomed mode. It is possible to change the classifier selection in zoomed mode.

The buttons labeled with magnifying glasses in the second row of the configuration panel (cf. Figure 38 and Figure 34 in Section 5.4.3) are used to zoom in and zoom out. If the maximum or minimum zoom level is reached, the button to zoom in or zoom out is disabled.

We modified the paint worker to issue 3 possible commands: zoom in, zoom out and paint. The paint command is issued for all paint operations just like before, the zoom in command executes the corresponding method in pixel map and is issued by the zoom in button, the same applies for the zoom out command.

5.6 Pointer Retrieval and Visualization

This section describes the implementation of the pointer visualization according to the requirements defined in Section 4.8.
5.6.1 Multiple Object Selection

As described in Section 5.4.5, we already implemented a mechanism in the graphical user interface that allows the selection of individual objects by selecting a pixel. In order to use the selected objects as starting points for pointer retrieval, we need to store them in the underlying PixelMap instance. This is implemented through an array of type long which stores the object IDs (i.e., their index in the heap). We added a method which selects the objects at a pixel \((x,y)\) to the class PixelMap. The method is implemented by the subclasses.

However, for pointer visualization it may be interesting for the user to select more than just the objects residing in a specified pixel. The user should also be able to select object groups that share certain traits. The concept that corresponds to this are classifications, therefore, it should be possible to select all objects that share a classification combination. It should also be possible to select multiple classification combinations.

In order to implement this, we extended the class PixelMap to store a list of selected classification objects (i.e., a subset of the key objects in the color map, cf. Section 5.2.1). We added two methods that allow to add or to remove classifications.

Since there will be multiple objects selected that spread across the heap, we can no longer use the approach of drawing black pixels over our existing image. We need to consider selected objects while drawing the image. We added a method to the class PixelMap which returns whether a specified object is selected via its classification. We want to display a pixel in black if it contains at least one selected object. For a cluster size of 1 (in both views) this is straight forward, if the object at the pixel is selected it is drawn black. However, when multiple objects are represented by a pixel, we cannot rely on the existing aggregation mechanism which determines the pixel color according to the classification that occurs most frequently. We need to store a boolean flag which determines whether any of the objects in the pixel is selected. As long as there is no selected object (i.e., classification) in the pixel, we need to keep testing, if we already found a selected object, we can stop.

If a new classification is selected or deselected, a new paint worker is started. Clicking on a pixel to select it requires no worker, as we still use the concept of drawing the pixel over the existing image.

The classification selection and de-selection is performed within the key component. Selected classifications are then displayed with a dark background within the key component. Additionally, a button labeled "Reset selection" in the configuration panel allows to reset all selected classifications. When hovering over the button, a
5.6.2 Pointer Retrieval

Selected objects, either via pixel selection or classification selection, are used as starting points for the pointer retrieval. As this is a run-time intensive task, this is handled by a new type of worker, the class PointerWorker. The class PixelMap offers a method to generate pointers. A parameter level determines how many levels of referenced objects should be displayed and therefore retrieved (cf. Section 4.8). A boolean flag toPointers signals to retrieve the “to”-pointers, i.e., pointers from the selected objects to other objects, i.e., their fields. If this flag is false, the “from”-pointers are retrieved, i.e., pointers from other objects to the selected objects. We acknowledge that this terminology may be misleading, but we stick to it as it is used by the Heap class. A parameter worker is needed to cancel the worker that performs the pointer retrieval if necessary (cf. Section 5.4.4).

The method that generates pointers in the class PixelMap just calls a method in its underlying ObjectVisualizationData instance. The latter method takes the same parameters as the one in the class PixelMap plus a parameter of type long[] which contains the indices of the objects that should be used as starting points for the pointer retrieval. We already have such an array, i.e., the array that contains the selected object IDs that were selected via pixel selection. However, we also need to add the IDs of the objects that are selected via classification selection. If there are any classifications selected, we iterate index-wise over all objects by retrieving their classification from the underlying ObjectVisualizationData instance. If the classification object for an object at a given index is selected, we add the index to a list of selected indices. When the iteration is finished, we merge the selected indices array with the selected indices list. If there are no selected indices and selected classifications, the method just terminates without retrieving any pointers.

We have an array of indices as starting points, but the pointer retrieval mechanisms in the underlying Heap operate on object addresses. Therefore, the first step is to
convert the object indices to their object addresses. The resulting address array is referred to as \textit{starting addresses}. From these starting addresses, the retrieval process is started. The implementation is an instance of the bread-first search algorithm (BFS) \cite{20}. We use a set data structure which holds all object addresses that are pointed to/from (depending on the \textit{toPointers} flag) including the starting addresses. At the very end of the method, this set is assigned to the corresponding field \textit{toPointers} or \textit{fromPointers} of the \texttt{ObjectVisualizationData} object.

5.6.3 Pointer Visualization

Pointer visualization in the graphical user interface can be enabled (only if pointers are available in the current trace file) via the checkbox labeled “Show Pointers” in the configuration panel. When this checkbox is enabled, two sliders appear at the bottom of the window. These sliders control the pointer level of the \textit{to} pointers (left) as well as the level of the \textit{from} pointers (right). The lowest setting on the sliders (i.e., 0, labeled as “None”) will only show the objects at the starting addresses, the consecutive settings are 1, 2, 3, etc. up to 10, while the highest setting (labeled “All”) refers to an infinite number of levels, i.e., the pointer retrieval terminates when no more new pointers are found.

Within the heap image, objects that are part of the \textit{pointer set} are drawn in their usual chromatic colors (or black if they are selected, i.e., starting objects), while objects that are not part of it are painted in grayscales. Figure \ref{fig:ptr_vis} shows the pointer visualization for a selected classification.

Just like with the selected pixels, a pixel should be painted in a chromatic color if it contains at least one object that is part of the pointer set. However, the color which is then displayed will be still the color of the classification that occurs most
frequently, which may also be of objects which are not part of the pointer set. A different implementation would be misleading to the user who can always decrease the cluster size or zoom in to investigate the objects in a specified pixel.

Pixels that contain exclusively objects that are not part of the pointer set are painted in a grayscale version of their actual color. Unlike selected (black) objects, gray objects do appear in the key component as gray, but no description text is displayed for these pixels.

5.7 Timelapse

The timelapse tab is the implementation of a heap visualization over time (cf. Section 4.10). We start a new parser process for the current trace which supplies us with all available Heaps of the application that generated the trace. By using the same user-defined classifiers and filters for every available Heap we generate according ObjectVisualizationData and PixelMap objects, and then display the resulting images to the user.

Figure 41 shows the timelapse tab, which can be started from the plot overview (cf. Figure 9 in Section 2.2). At the top of the tab, there is the component which allows classifier and filter selection. Below, in the center, there is a panel which displays the current image. Further below the panel there is a slider, which indicates the position of the currently displayed image within the timeline of the inspected application. The user may use the slider to select a different position. Below the slider, a label displays the current slider value. Located next to it, there is a button labelled with a “Play”-symbol, which starts an automatic animation of the timelapse. While the animation is running, the label on the button changes to a “Pause”-symbol and pressing the button suspends the animation. Next to the button, there is a spinner for the cluster size and a unit box. Both components are disabled, as we currently only support automatic cluster size detection. The very last box in the row allows to select byte or object view.

Although the heap iteration, pixel map generation and drawing of the image can be re-used from the conventional heap visualization, we needed to make some adoptions. Every instance of ObjectVisualizationData holds its own color map and color generator. However, we do want the objects of same classifications to have the same colors across the entire timelapse. Therefore, we need to share the color map and the color generator. Additionally, we need to consider that when using the memory saving strategy introduced in Section 5.2.5 we also need to share the array that maps from ID to classification, as the IDs are also stored in the PixelIDescription objects of the color map.
In the byte view, we want to visualize the growth and shrinking of the heap and heap areas. The heap may be enlarged by adding objects or spaces to the very end, but also to the front (depending on the garbage collector). Therefore, every heap image needs to start at the minimum address of all available heaps. We equipped the class ObjectVisualizationData with the possibility to start data generation with a pre-defined starting address. If the address coincides with the first address of the used heap, nothing changes. If the given first address is smaller than the heap’s first address, we found a gap. Therefore, a heap may start with a gap and the contrary assumption we made in Section 5.3.4 does not hold anymore. Some minor adoptions had to be made.

In order to retrieve the starting address, we equipped the Heap class with a method that returns the very first object address of the heap. Before generating any ObjectVisualizationData objects, we need to find the starting address. Similarly, before drawing any heap image, we first need to generate all ObjectVisualizationData objects to determine the cluster size. The cluster size is automatically determined so that the largest image fits onto the screen.

In order to save memory, we only keep the actual images (BufferedImage) to display in the timelapse and throw the instances of Heap, ObjectVisualizationData and PixelMap away. This makes it impossible to use zoom, a key component or allow the user to change the cluster size without re-iterating the heaps. Additional features and re-consideration of the way the timelapse is generated and stored are considered part of future work (cf. Section 8).
6 Functional Evaluation

This section describes three use case-scenarios for the heap visualization.

6.1 Memory Layout Visualization

In order to visualize the heap layout generated by different garbage collectors, we ran the same application three times on the AntTracks VM, using all three garbage collectors currently supported by AntTracks (ParallelOld GC, G1 GC, Concurrent Mark and Sweep GC). The application we used was the fop benchmark from the DaCapo 9.12 Bach benchmark suite.²⁵

Figure 42: Overview plots for the ParallelOld, G1 and Concurrent Mark and Sweep GCs

Figure 42 shows the overview plots with the memory consumption in bytes for the three different garbage collectors. Like expected, the garbage collector phases differ, but the number of bytes at the very end of the application (last peak) is about the same for every garbage collector (roughly 50 MB). We selected the last spike and visualized the heap at that time for every garbage collector.

6.1.1 ParallelOld GC

In order to visualize the heap layout, we selected the Space Type classifier and switched to byte view. Figure 43 shows a screenshot of the visualization. Additionally, we opened the color selection dialog to get a complete list of all classifications and their colors. The automatic cluster is set to 5 kB.

As expected, there are only three space types: Eden, survivor and old space. The heap starts with the old space at the very beginning, followed by a very large gap.

The next space is the Eden space, followed by a smaller gap and then finally the survivor space. All spaces are continuous without any fragmentation. Since the ParallelOld GC uses exactly one space per space type, we expect the space type to coincide with the individual spaces. In order to verify this, we added the Space classifier which classifies by individual spaces rather than space types. After adding the additional classifier, the image remains the same and the number of unique classifications remains the same, as can be inferred from the color selection dialog (cf. Figure 44).

### 6.1.2 G1 GC

Figure 45 shows the visualization of the G1 GC heap with the Space Type classifier selected. The automatic cluster size is only 239 bytes, compared to 5 kB in the ParallelOld GC heap. This seems impossible since the heap sizes shown in the overview plots coincide for all three garbage collector. However, we need to keep in mind that the overview plots only show the amount of actually used memory (i.e., memory occupied with objects), while the heap visualization paints the entire
memory area. This kind of information is only visible in the new heap visualization and has previously been hidden from the AntTracks user.

Like in the ParallelOld GC, the heap starts with an old region followed by a large gap. However, the gap here is much smaller than in the ParallelOld GC heap. This gap is also responsible for the different heap sizes. After the gap, there is a portion of an Eden space, directly followed by a survivor space region followed again by Eden. This shows us immediately that the spaces are not consecutive.

Since the G1 GC spans space types across multiple regions, we do not expect the results of using the Space Type and Space classifier to coincide. We removed the Space Type classifier and used the Space classifier instead. Figure 46 shows the result, there are very many small spaces. A look at the color selection dialog shows that there are 49 of them.
Figure 47: Visualization of the Concurrent Mark and Sweep GC heap with selected Space Type and Space classifiers and the open color selection dialog

6.1.3 Concurrent Mark and Sweep GC

For this garbage collector, the results of the classifiers Space and Space Type should also coincide. Therefore we selected them both, the result including the color selection dialog is shown in Figure 47. The automatic cluster size is 2 kB, which indicates that the major gap in this heap is smaller than the one in the ParallelOld GC heap, but larger than the one in the G1 GC heap.

Unlike in the other heaps, the very first space in blue is not the old space, but the Eden space. This consecutive space is followed by a consecutive survivor space, followed by the large gap. The very last space after the gap is the old space in red. The reason why the colors differ from those in the other heaps is the way the automatic color generation is implemented. The user could always re-assign the colors with the color selection dialog on order to make to better comparable to the other heaps.

In the old space we can see white spots. This shows the fragmentation that occurs due to the Mark and Sweep algorithm that is used in the old space only. We selected this space with the mouse (see Figure 48 top) and issued a zoom. The intelligent zoom decreased the cluster size to 34 bytes. We issued another zooming step to zoom optically, which led to a zoom factor of 200% (see Figure 48 bottom).
6.2 Pointer Visualization

In order to demonstrate the accuracy, we created a toy example. The package test contains the following classes:

- **A** contains one object field of type B and one of type C
- **B** has only one primitive field of type int
- **C** has only one primitive field of type int
- **MyLinkedList**<E> is a generic implementation of a linked list data structure. It contains two objects fields, head and tail of type MyNode<E>. It offers a method `add(E value)` to append a value at the end of the list and a method `iterator()` which returns an iterator for the list.
- **MyLinkedList$MyNode**<E> is a public static inner class of MyLinkedList and represents a value node of the linked list. It contains two objects fields, E value and MyNode<E> next.
- **MyLinkedList$MyListIterator** is a private inner class of MyLinkedList and implements an iterator for the list. It has one object field of type MyNode<E> which points to the current node in the iteration process. It offers the methods `boolean hasNext()` and `E next()`.

In our sample application, we created a new instance of MyLinkedList<A>. Within a loop, we appended objects of type A which contain two fields with new objects of
type B and C each. Next, we iterate over the list with the iterator and sum up the int values of the B and C objects referenced by the A objects in the list. We print out the sum and then issue a manual garbage collector call to make sure that AntTracks is able to record the object and pointer data. We executed this application with the ParallelOld GC.

Figure 49 shows the visualization of the application in object view with a cluster size of 1 and a zoom factor of 700% with the Type classifier. We opened the color selection dialog to get a complete list of all types. There are many objects of types that are completely unrelated to our application. These types are automatically allocated by the Java VM and Interpreter. Since we are only interested in our own object, we created a user-defined filter that only shows objects from the package test.
Figure 50 shows the visualization with the open color selection dialog in the front after selecting the filter. Filtered objects are drawn in gray. The very first object is the \texttt{MyLinkedList} object, right next to it is the \texttt{MyIterator} object. After some unrelated, filtered objects, a bulk of \texttt{MyNode}, A, B and C, allocated alternately in this order follows until the end of the heap, sometimes interrupted by more filtered objects. Even without showing pointers, we can learn much about object locality using the heap visualization. This kind of information is also visible nowhere in AntTracks except the visualization.

Next, we selected the \texttt{MyLinkedList} object by clicking on it and enabled the “Show Pointers” checkbox. The default setting shows one level of to pointers. The result is shown in the topmost image in Figure 51. We can see that the object itself and the two objects it points directly to, namely the head and tail pointers to objects of type \texttt{MyNode}. The rest of the objects is grayed out.

In the next step, we increase the to pointer level by one by moving the slider to the right. The result is shown in the second image in Figure 51. Now three additional objects have appeared: the two A objects that are referenced by the head and tail objects as well as an additional \texttt{MyNode} object. This object is the next pointer of the head object.

By increasing the to pointer level further, six new objects become visible (third image in Figure 51). Four of these objects are the B and C objects referenced by the A objects that are the values of the head and tail nodes. The A object referenced by the new node object which appeared in the previous setting as well as a new node object that is the next pointer of said node.

Continuous increase of the pointer level will make more objects appear gradually. By setting the level to the “All”-setting, all objects that are kept alive by the \texttt{MyLinkedList} object are shown (see last image in Figure 51). This shows that the \texttt{MyLinkedList} object keeps alive all objects that are not filtered, except for the \texttt{MyIterator} object, which still appears as gray.

We can also perform the opposite operation and find out how an object is kept alive. Therefore, we select and arbitrary object within the bulk and set the to pointer level to “None” and the from pointer level to 1. The result is shown in Figure 52 top. The selected object is of type C and therefore kept alive by an A object.

By setting the level to “All” setting we can find the root pointer of the object, so to speak. Figure 52 bottom, shows the result. The A object is kept alive by its neighboring \texttt{MyNode} object. This is referenced by another \texttt{MyNode} object which is referenced by yet another node. Following the chain backwards until the very first \texttt{MyNode} (head), we end up with the \texttt{MyLinkedList} object which keeps the entire
object chain alive. We also notice that the MyIterator object appears. This is due to the fact that it is an inner class object of MyLinkedList, which automatically has and needs a reference to the surrounding object.

Figure 51: To pointer visualization with the MyLinkedList object as starting point and pointer levels 1, 2, 3 and infinity

Since we know the relationships between our objects in this example, we can see that the pointer visualization works accurately.
6.3 Memory Leak Detection

Lengauer [10] describes how AntTracks can be used to identify and track down the cause of a real-world memory leak found in the beta version (beta0) of the *tradebeans* and *tradesoap* benchmarks of the *DaCapo* benchmark suite.

We want to show that the cause of the memory leak can also be tracked down with the heap visualization, using the pointer visualization. In order to get a trace, we executed the *tradebeans* benchmark like described in Lengauer [10]: limited heap size of 100 MB, 14 iterations and using the *ParallelOld GC*. Figure 53 shows the

---

26 The DaCapo Benchmark Suite / Bugs / #9 memory leak in tradebeans, [https://sourceforge.net/p/dacapobench/bugs/9/](https://sourceforge.net/p/dacapobench/bugs/9/) (2017/07/12)
overview plots for the trace. They show that there is a memory leak, as the heap size steadily approaches the maximum size of 100 MB while continuously performing garbage collections that fail to free memory.

We selected a spike on the overview plots (cf. Figure 53) and selected to examine the heap at that point in time. By taking a look at the table-based statistics view (see Figure 54), which appears first, we can gain an insight about the underlying problem. The objects that are allocated the most are of type `java.util.HashMap$Node`. Therefore, we opened the visualization, selected the `Type` classifier and then use the key component to select all objects of that type.

Then, we enabled the pointer visualization. Since we need to find out which objects keep the node objects alive, we selected level 1 on the right slider. Additionally, we wanted to see what is kept alive by the nodes, so we selected the level “All” of the left slider. In Figure 55 we can see that some mint green objects appear, which are of type `java.util.HashMap$Node[]`. These must be the objects which hold references to our nodes, since the nodes in the hash map are obviously kept in an array. As for the objects kept alive by the nodes, we see many different objects appear. However, a great deal of them are colored pink and in a light shade of
Figure 56: Visualization of *from* pointers for objects of type `java.util.HashMap$Node` with level 1, 2 and 3.

These objects are of type `java.lang.String` and `char`. Since Strings in Java are backed by character arrays, there must be a relationship between these kind of objects. By observing this we can conclude that most of the nodes, and therefore, the nodes that cause the memory leak, contain references to `Strings`. This also coincides with the insights table-based statics view (cf. Figure 54), where we see that the second and third-most allocated objects are of types `char[]` and `String`.

In order to investigate further on the objects that allocate the nodes, we reset the left spinner to “None” and increased the value on the left spinner by one. Figure 56, top, shows that the arrays that hold the nodes are actually referenced by `java.util.HashMap` objects (in blue). Therefore, by increasing the level by one again, we should be able to see the objects that hold these maps and are therefore the culprits of the memory leak. As shown in Figure 56, bottom, there are mainly two types of objects that appear. The first ones, that appear in magenta, are of type `java.util.HashMap$Value`. Since these are related to the map itself, these cannot be the cause of the memory leak.

The other objects, which appear in orange, are of type `org.apache.geronimo.gebean.runtime.GBeanInstance`. As described in Lengauer [10], the memory leak is indeed caused by classes in the
org.apache.geronimo package. When looking up the fix, it is revealed that the geronimo package was indeed keeping String objects (backed up by char arrays) alive by adding them needlessly in a HashMap but not removing them.

This example shows that the graphical heap visualization is complementary to the table-based statistics view and enables the user to track down memory-related performance problems in applications. For some users a graphical view is probably more intuitive to use.
7 Performance Evaluation

This section describes how the heap visualization behaves in terms of run-time behavior and memory consumption.

7.1 Run Time

The main asset of the new visualization, compared to the old visualization, in terms of run time is the fact that the visualization runs smoothly once the data is generated and the user does not experience any waiting times when scrolling through the image.

The waiting time for the user is caused through the background working threads which iterate over the heap and draw the image. The table-based statistics view also uses a background working thread to classify the objects in the heap according to the selected classifiers, the results are then displayed in a table. In order to show that our worker that performs the heap iteration (VisualizationWorker) does not introduce any unreasonable run-time overhead, we compared it to the worker of the table-based statistics view. We knew beforehand that the VisualizationWorker would be slower, because unlike the other worker, it does not iterate the heap in a multi-threaded way. This is due to the fact that we need to iterate in order (which is not necessary for the table-based view). Of course, multi-threaded iteration would also be possible, but with mechanisms that make sure that the object order is preserved during iteration or re-established after iteration, which may introduce temporary additional memory consumption. We consider the development of a multi-threaded iteration for the visualization part of future work (cf. Section 8).

In order to compare the workers, we executed all 14 benchmarks of the DaCapo benchmark suite with their largest input size on the AntTracks VM. Then we opened every trace and selected the point in time with the largest memory consumption in terms of objects, as the iteration time depends solely on the number of objects. We then opened the table-based statics view and then the visualization. In both views, we selected the following classifiers in the following order: Type, Allocation Site, Space, Allocating Subsystem, Array Length, Object Kind. All measurements were run on an Intel®Core™i3-2348M CPU @ 2.30GHz×4 on 64-bit with 8 GB RAM running Ubuntu 16.04.2 LTS Xenial Xerus with the Kernel Linux 4.4.0-77-generic. All unnecessary services were disabled, but in order to assure realistic conditions we executed the entire AntTracks tool in graphical mode. For every benchmark, the shown result (referred to as average) is the arithmetic mean of 15 runs.

Figure 57 shows the run time comparison between the background worker of the table-based statistics view (DetailedGroupingWorker) and the
Figure 57: Run time comparison between DetailedGroupingWorker and VisualizationWorker. Like expected, DetailedGroupingWorker outruns VisualizationWorker in every benchmark. However, the introduced overheads, which range from 8.75% to 54.50% (arithmetic mean 38.89%), are not unreasonably bad considering the different preconditions of the workers. Most importantly, the additional waiting times introduced by VisualizationWorker are deemed acceptable for the users.

7.2 Memory Consumption

As described in Sections 5.2.4 and 5.2.5, we developed two data storage approaches, whereas the first is only used as backup if the second approach fails due to an unreasonably large heap with some unrealistic conditions.

The foundation of the second approach is the observation that the number of objects in the heap usually outweighs the number of unique classifications by far. Table 1 in Section 5.2.4 shows this for 3 heaps with different selected classifiers.
In order to show that the observation holds for general use cases, we analyzed the heaps with most objects for every benchmark from the DaCapo benchmark suite used in the experiment in Section 7.1. Like in the previous experiment, we selected multiple classifiers in the following order: Type, Allocation Site, Space, Allocating Subsystem, Array Length, Object Kind. Then we compared the number of objects in the heaps in question to the number of unique classification combinations.

All selected classifiers are useful for the visualization and likely to be selected by users. However, we doubt that the selection of six different classifiers makes sense and that users will usually select as many classifiers. For most use cases, one to two classifiers suffice (cf. Section 6). We assert that this classifier selection constitutes the worst case of a typical use-case scenario.

Figure 58 shows the relationship between the number of objects in the heap (x-axis) and the number of unique classifications (y-axis). The bottom line is that there is no direct relationship, the number of unique classifications depends on the individual application and a larger heap size does not necessarily lead to more classifications.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Objects</th>
<th># Combinations</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>1,125,132</td>
<td>3,971</td>
<td>short</td>
</tr>
<tr>
<td>batik</td>
<td>733,407</td>
<td>12,811</td>
<td>short</td>
</tr>
<tr>
<td>eclipse</td>
<td>12,287,987</td>
<td>26,352</td>
<td>short</td>
</tr>
<tr>
<td>fop</td>
<td>1,005,733</td>
<td>14,297</td>
<td>short</td>
</tr>
<tr>
<td>h2</td>
<td>36,315,335</td>
<td>4,566</td>
<td>short</td>
</tr>
<tr>
<td>jython</td>
<td>16,552,160</td>
<td>11,084</td>
<td>short</td>
</tr>
<tr>
<td>luindex</td>
<td>335,857</td>
<td>4,694</td>
<td>short</td>
</tr>
<tr>
<td>lusearch</td>
<td>1,413,405</td>
<td>3,071</td>
<td>short</td>
</tr>
<tr>
<td>pmd</td>
<td>4,712,763</td>
<td>10,049</td>
<td>short</td>
</tr>
<tr>
<td>sunflow</td>
<td>9,128,987</td>
<td>3,591</td>
<td>short</td>
</tr>
<tr>
<td>tomcat</td>
<td>5,567,449</td>
<td>1,0357</td>
<td>short</td>
</tr>
<tr>
<td>tradesbeans</td>
<td>21,201,257</td>
<td>18,831</td>
<td>short</td>
</tr>
<tr>
<td>tradesoap</td>
<td>15,611,251</td>
<td>19,444</td>
<td>short</td>
</tr>
<tr>
<td>xalan</td>
<td>4,512,256</td>
<td>4,584</td>
<td>short</td>
</tr>
</tbody>
</table>

Table 3: Number of objects in the heap vs. number of unique classifications

We also see that the number of objects is by far larger than the number of unique classifications. This can also be seen by looking at the plain numbers in the complete data table (see Table 3). Most importantly, the number of unique classifications always fits in to the range of short.

In Table 2 in Section 5.2.4 we computed the possible memory savings when using the final storage approach. The percentages shown there refer to the memory consumption of the ObjectVisualizationData object. In order to show that the final storage approach reduces the memory consumption of the entire application, we conducted another experiment.

We compared the memory consumption of the final storage approach (cf. Section 5.2.5) and the previous approach (cf. Section 5.2.4). In order to accomplish this, we used the 14 heaps from the DaCapo benchmark suite used in the experiment in Section 7.1. Then we opened every trace and selected the point in time with the largest memory consumption in terms of objects and then opened the visualization. We selected the same six classifiers as previously in the same order. In order to measure the memory consumption, we retrieved the currently used memory from the Java VM and printed it. In order to make the values comparable, we issued a manual garbage collector call first. We executed the experiment twice: once with the normal AntTracks tool with memory saving, and once we turned the memory saving off manually. In order to make sure that the data resides in the memory, we turned off the creation of MMFByteCollections (cf. Section 5.2.6). All measurements were run on an Intel®Core™i3-2348M CPU © 2.30GHz×4 on 64-bit with 8 GB RAM.
running Ubuntu 16.04.2 LTS Xenial Xerus with the Kernel Linux 4.4.0-77-generic. The AntTracks tool was executed with the default heap size (which is 2 GB on the used machine). For every benchmark, the shown result (referred to as average) is the arithmetic mean of 5 runs.

Figure 59 shows the comparison of the two storage approaches. For all benchmarks, we can reduce the memory consumption noticeably, in a range from 1.74% up to 14.07% (arithmetic mean: 7.07%). Perhaps the most noticeable thing about this is that the reduction becomes larger when the heap grows larger. For small heaps, the general memory consumption in the visualization is small compared to the overall memory consumption of the application. However, when the heap grows, the visualization needs more memory, while the rest of the application’s memory consumption remains relatively stable. Therefore, the reduction of the memory consumption in the visualization is of importance when dealing with large heaps and makes a difference.
8 Future Work

This section discusses possible future work.

8.1 Color Generation

As discussed briefly discussed in Section 4.4 the used color generator could be easily exchanged by a better color generator thanks to the modular design. We also mentioned the possibility of offering various color generators at the same time and letting the user choose a color scheme. This requires some more, but minor changes in the current system. The color objects can no longer be stored directly in the color map, but some kind of ID or identity object instead. The actual color can then be retrieved by querying the color with the specified ID from the color generator. Additionally, the color generator also needs to offer the three special reserved colors that are in use now (white for gap, black for selected and gray for filtered). Ultimately, a component in the graphical user interface which enables the color scheme selection needs to be added.

8.2 Timelapse

As discussed in Section 5.7 the timelapse uses single, static heap images that are generated through the usual heap visualization. For applications that run for a long period of time and therefore have many heap images, this may put much pressure on the memory.

The features of the timelapse are currently limited, which is also a result of the increased memory consumption.

Since the heap visualization is optimized for displaying a single heap image, some adaptions to it had to be made in order to support the timelapse.

While the heap visualization offers very good and perhaps ideal solutions for single images, we are not entirely satisfied with the timelapse. A complete refactoring with new data structures, inspired by the single heap visualization but focused on the timelapse, would help to overcome the mentioned issues.
8.3 Multi-threaded Heap Iteration

As discussed in Section 7.1, the heap iteration for the visualization is always slower than the heap iteration for the table-based statistics view. This is due to the fact that the visualization does not use the multi-threaded parallel heap iteration. In order to overcome this and speed the entire heap iteration for the visualization up, it is necessary to develop a parallel iteration that is able to preserve the object order without introducing too much additional memory consumption. At the moment, a parallel iteration is in conflict with our current method of data storage. However, it may be still be possible to use parallel iteration and sequential storage by using some kind of intermediate data structure or reserving according space in the sequential data structure if the chunks of objects that are handled in parallel are consecutive and their length is know beforehand.
9 Conclusions

We re-developed the AntTracks heap visualization from scratch and equipped it with improved versions of the existing features and many additional, new features. We incorporated the existing mechanisms of user-defined classifiers and filters into the new visualization as well as the pointer visualization.

The new heap visualization equips the AntTracks tool with the previously non-existent capability of visualizing the heap in an accurate and realistic manner and complements the already existing views graphically.

In terms of usability and performance the new visualization outruns the old visualization by far. In order to preserve both usability and run-time performance, the visualization puts a burden on the memory, which may result in memory-related performance problems when dealing with large heaps. In order to overcome this, we developed a novel strategy of memory saving that does not sacrifice run-time performance. A byproduct of this storage approach is a specialized data structure, which could also be used for arbitrary other applications. Performance analysis confirmed that the memory saving strategy works and enables the visualization to handle large heaps.

We developed a rich application with a various range of different features by incorporating well-known programming paradigms, patterns and algorithms as well as own new ideas. The result is a well-structured, expandable and stable running tool which integrates smoothly into the existing AntTracks tool.

To conclude this thesis, the new heap visualization is a precious addition to the AntTracks tool which helps to increase the analysis capabilities of AntTracks.
References


Curriculum Vitae

Personal Information
Name: Christina Rammerstorfer, BSc.
E-Mail: christina.r@outlook.at
Date of birth: 11.12.1988
Marital status: Married, two children (born 2009 and 2011)
Nationality: Austria

School
Matura with distinction (GPA 1.0), Thesis: Universelle DLL

Studies
2016 October – today Master Studies (Software Engineering) at the Johannes Kepler University Linz
2009 October – 2016 June Bachelor Studies (Computer Science) at the Johannes Kepler University Linz with distinction, Thesis: A Benchmark Suite to Evaluate the Bottlenecks of AntTracks

Work Experience
2014 March – today Tutoress at the Institute for System Software, Johannes Kepler University Linz
2006 July Intern at New Technology Systems (NTS), Wilhering
2005 July Intern at Siemens, Linz

September 7, 2017
Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.
Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Christina Rammerstorfer, BSc. (Linz, September 7, 2017)