JOHANNES KEPLER
UNIVERSITY LINZ

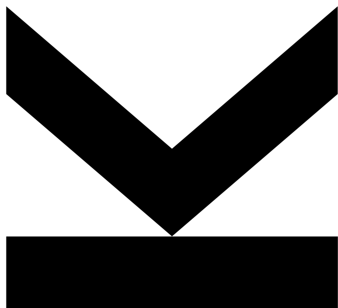
Author
Manuel Vujakovic

Submission
Institute for Systems Software

Thesis Supervisor
Dipl.-Ing. Dr.
Markus Weninger

July 2023

Automatic Detection of Data Structures in Reconstructed Heap States



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Automatic Detection of Data Structures
in Reconstructed Heap States

Student: Manuel Vujakovic

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: February 2022

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
P +43-732-2468-4361
markus.weninger@jku.at

AntTracks, developed at the Institute for System Software at the Johannes Kepler University, is a memory monitoring tool that is used to detect and analyze memory anomalies such as memory leaks. Often, memory leaks are related to data structure misuse, e.g., programming errors leading to growing data structures (such as lists, maps, sets, etc.). Yet, memory analysis tools are by default not aware of the concept of data structures but only see the heap as an object graph, i.e., separate objects that reference each other (thereby keeping each other alive). Since modern applications may contain more than 100 million objects in their heap, analysis on the object level is often too fine-grained and complex.

The goal of this project is to automatically identify data structures in the heap, for example, to automatically identify that `LinkedList` objects are data structures that internally consist of `LinkedList$Node` objects that point to data objects. Such detected data structures can then be used to perform less fine-grained analyses, instead providing a more top-level view of the memory behavior of the application. It is not a goal of this thesis to develop new analysis approaches bases on detected data structures.

To become familiar with automatic data structure detection, the student should thoroughly explore and summarize existing literature on data structure detection and aggregation. Existing approaches often classify heap objects based on their roles in data structures, such as "recursive backbones" or "array backbones" (for example "*Patterns of Memory Inefficiency*" by *Chis et al., https://doi.org/10.1007/978-3-642-22655-7_18* and various work by *Mitchell et al.*). Yet, such approaches often rely on external information which tells them, for example, where data structures start, i.e., which objects are "data structure heads" (for example "*Analyzing Growth Over Time to Facilitate Memory Leak Detection*" by *Weninger et al., https://doi.org/10.1145/3297663.3310297)*.

The thesis should contain a comprehensive presentation of existing approaches, discussing their similarities and differences. Based on these observations, the student should devise at least one own novel detection algorithm that focuses on not relying on external information. The algorithm should use suitable heuristics to decide where data structures start, which other objects are "internal" parts of the data structure, and where data structures end. Ideas for such an algorithm should regularly be discussed with the supervisor. Finally, the algorithm should be implemented as a prototype in AntTracks to show its feasibility and applicability.
The thesis should present how the new algorithm works in general, and which kinds of data structures can be detected with it. More specifically, corner cases should be discussed. For example, the thesis should highlight how the devised algorithm behaves when deciding which object is a data structure head (for example, in Java, a `HashSet` internally contains a `HashMap` to store its data, and existing approach often detect the `HashMap` as data structure, but not the overall `HashSet`). Also, the thesis should present how the algorithm handles data structures that are contained in other data structures (such as a HashMap that in turn contains `ArrayLists` as values).

<u>Modalities:</u>
The progress of the project should be discussed at least every three weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 30.09.2022.

Abstract

Memory is one of the most important and limited resources for developers. Data structure detection is used by developers to either discover memory inefficiencies or to gain a better understanding of complex run-time interactions. In order to comprehend inefficient or consistently crashing software behavior, the most straightforward approach is to directly examine the memory's activity. Existing approaches utilize additional knowledge, such as dictionaries including data structure heads, to be able to detect data structures. It follows that, in order for the approaches to work as intended, mentioned information needs to be included. This makes the process more tedious and less flexible. In this thesis, we present an approach for automatically detecting data structures without the need for additional knowledge. We based our approach on similarities and differences we found in related work. However, we also addressed some shortcomings that existed in these approaches. Eleven data structure-specific functionalities called roles were introduced to fully and accurately describe data structures. These roles were created to identify common structures as well as unique cases that the current were unable to do without the help of the additional information. The entire approach is outlined and explained in this thesis and has also been implemented as a prototype in AntTracks[19], a memory monitoring tool, to demonstrate its feasibility.

Kurzfassung

Arbeitsspeicher ist einer der wichtigsten, aber auch einer der beschränktesten Ressourcen für Entwickler. Die Datenstrukturerkennung wird von Entwicklern entweder genutzt, um Speicherineffizienzen zu entdecken oder um ein besseres Verständnis für komplexe Laufzeitinteraktionen zu erhalten. Der einfachste Weg, um ineffizientes oder wiederholt abstürzendes Softwareverhalten zu verstehen, ist die Aktivitäten des Speichers direkt zu untersuchen. Existierende Ansätze nutzen zusätzliche Informationen, wie beispielsweise Listen mit Datenstrukturköpfe, um Datenstrukturen erkennen zu können. Damit der Ansatz wie vorgesehen funktionieren kann, müssen diese genannten Informationen erstellen und eingebunden werden. Dies macht den Prozess langwieriger und weniger flexibel. In dieser Bachelorarbeit stellen wir einen Ansatz zur automatischen Datenstrukturerkennung vor, der auf keine zusätzlichen Informationen angewiesen ist. Unser Ansatz basiert auf Übereinstimmungen und Unterschiede, die wir aus ähnlichen Arbeiten entnommen haben. Jedoch wurden auch einige Schwächen dieser Ansätze behoben. Die Einführung von elf datenstrukturspezifischen Funktionalitäten, auch bekannt als Rollen, ermöglicht eine umfassende und präzise Beschreibung von Datenstrukturen. Diese Rollen wurden entwickelt, um sowohl allgemeine Strukturen als auch spezifische Einzelfälle zu identifizieren, die bisher ohne zusätzliche Informationen nicht erfasst werden konnten. Die vorliegende Bachelorarbeit enthält eine umfassende Beschreibung und Erläuterung des gesamten Ansatzes, der auch als Prototyp in AntTracks[19], einem Tool zur Speicherüberwachung, implementiert wurde, um seine Machbarkeit praktisch zu demonstrieren.

Table of Content

Contents

Abstract **i**

Kurzfassung **ii**

1 Introduction **1**

2 Background **4**
- 2.1 Memory Inefficiencies 4
- 2.2 Object Reference Graph 5
- 2.3 Dominator Relation 5
- 2.4 Data Structure Roles 7
- 2.5 Running Example 7

3 Existing Approaches **11**
- 3.1 Data Structure Health 12
- 3.2 Backbones 13
- 3.3 Container or Contained 15
- 3.4 Domain-Specific Language 18
- 3.5 Similarities and Differences 19
- 3.6 General Problems 20

4 Approach **22**
- 4.1 Roles 22
- 4.2 Role Assignments 23
- 4.3 Benefits 27
- 4.4 Testing 28

5 Future Work **30**

6 Conclusion **32**

Literature **34**

1 Introduction

Memory is a crucial resources in modern times. Depending on the size and complexity of the software, there tends to be an increase in memory usage. The heap stores all live objects that are needed for the software to function. In fact, many tens of millions of objects are stored in memory for any sizeable program. Hence, it is very important to use memory as efficient as possible. Yet, unintentional memory inefficiencies introduced by developers lead to an increase in usage. Such inefficiencies are difficult to uncover because they require knowledge of how objects are managed at run-time [11].

The easiest way to comprehend what happens in an application is to look at the heap state. It shows which objects are stored without having to understand all of the operations in the program itself. Detecting the underlying structure of heap objects would greatly aid in removing these inefficiencies. For starters, it would allow us to view the heap by grouping and expressing the enormous amount of objects depending on their structure. This visual representation would aid in the discovery of problems or inefficiencies, or act as a starting point for problem-solving [2, 5]. The question now is how to find these inefficiencies among the millions of nodes stored in the heap. The solution lies is data structure detection.

The operation begins by collecting all of the heap objects used by the application into *heap traces* or *heap dumps*. The only difference between them is that a heap dump contains the heap state at a single point in time, while the heap traces contain heap state over a period of time [9, 20]. In any case, the most important information we can get from from any object is a unique ID, a type name, and the references the object has to other objects. This allows us to identify the data structure (DS) that these objects are a part of or define.

Two main concepts are currently applied for data structure detection. The first one consists of creating a domain-specific language (DSL) [20]. This DSL is then used to create definitions that describe all the data structures contained within the program. This method's main advantage is that it is accurate as long as the desired DS definitions are provided. Needing a definition also makes this approach adaptable. No matter what DS is written or used, even if it is a framework, a fitting definition can be provided. At the same time, this also creates the biggest drawback. Without any definition, this method is not able to identify any structural elements. The second approach, is an automatic detection, that uses structural information to assign roles to a object. We will discuss this in more detail in Section 3. As this approach relies on various heuristics to decide the roles, it is also less accurate in its detection than the previous approach. One significant disadvantage of the current automatic approaches is that they depend on *additional knowledge*, such as a dictionary containing the names of the data structure heads, to define the start of an DS or to be able to handle corner cases. The main benefit is that it can detect data structures automatically, not needing the clear definition of all the classes, but it also comes with the drawback of reducing the accuracy.

As a result, both the existing DSL and the automatic concepts of data structure detection rely on the use of additional information. This is especially troublesome in the case of automatic data structure detection, as classes from frameworks or custom-made ones would be unable to be discovered because they would require additional knowledge. Thus, we created a general approach that does not rely on any additional information. This approach could be easily implemented in a variety of memory analysis tools. By doing that, the user experience would improve

as no additional information needs to be created and entered for data structure analysis. The implementation into other memory analysis tools would also not limit or change the functionality of the analysis tools itself. This feature is crucial as it helps memory analysis tools create clear visualizations of memory usage and identify the reasons behind memory inefficiencies. For example, with the help of data structure detection, we know what objects are part of the DS as well as what roles they play within them. With that information, we could calculate the overhead of the DS. A data structures overhead is based on how many objects contain genuine data compared to how many objects are need internally to build the DS [14, 4]. This lets developers make appropriate changes to the given DS to fix the memory inefficiency[2, 5].

The contributions of this thesis are:

- An in-depth summary of existing detection approaches, including their functionality, roles that are used for data structure detection, as well as their application to a running example.
- The identification of similarities and differences between existing approaches, as well as the overall flaws they contain.
- A thorough description of our automatic data structure detection approach, including the roles, their assignments, the complete pseudo code, and testing procedure.

The thesis is organized as follows: Section 2 gives important background information for our work as well as an introduction to the running example used in this thesis. Section 3 discusses existing approaches, in particular how they function and how they detect data structures in the running example. Additionally, the section also includes similarities and differences between the approaches, along with their general problems. Section 4 thoroughly describes our approach in all its parts. Section 5 outlines possible future work, and Section 6 concludes the thesis.

2 Background

This section provides a thorough explanation of the fundamental concepts and theories that serve as the foundation for this thesis.

2.1 Memory Inefficiencies

There are a wide variety of memory inefficiencies. However, certain inefficiencies, such as churn[1], are not observable using memory structural information alone. So, we will focus on the two kinds of inefficiency that can only be found by looking at structural information: Memory bloat and memory leaks.

Memory bloat is defined as a sudden increase in memory usage due to unnecessary memory usage [15], as showcased on the left hand side of Figure 1. In most cases, this happens when data structures have a large overhead. *Overhead* is a substantial difference between the number of data-containing data structures and the overall objects. In other words, the structure takes up a lot more memory than the data itself does. Changing the DS to one that can store them more efficiently is the best method to address memory bloat.

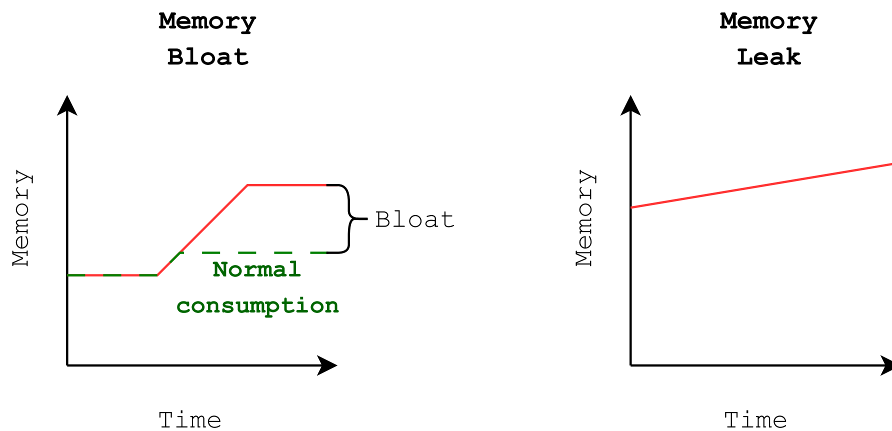


Figure 1: Memory consumption of Bloat and Leak.

Memory leak is the other type memory inefficiency. A memory leak is a steady rise in the amount of memory being used due to objects being kept alive that are not needed anymore [16], as seen on the right hand side of Figure 1. The cause most often is that there exists a garbage collector that automatically collects memory objects that are no longer kept alive by so-called garbage collection roots [1]. Objects that are directly or indirectly referenced will not be collected. An example for garbage collection roots are static or local variables. A memory leak is caused by those objects that are *needlessly* kept alive. They can be found by examining if objects point to empty values such as null. The developer must then either change how the object is declared or release it when it is no longer needed.

[1]Memory churn is time-specific. It describes high-frequent allocation of objects shortly followed by their deallocation. For example, creating objects inside a loop that are collected before the loop ends.

2.2 Object Reference Graph

In the introduction, we mentioned that heap traces and heap dumps offer all the information needed for data structure detection. The most crucial information they provide is [7]:

- **a unique ID:** This ID is very important, as there are many objects of the same type. Without it, we could not distinguish between these objects.
- a **type name** or **class name:** Current detection algorithms make use of the type name to increase detection quality. It is also needed to help the user identify which type of data structure is causing memory inefficiencies or occupying the memory.
- **references to other objects:** References to other objects are required for the construction of an object reference graph. They reveal the interactions between objects within a data structure and aid in program understanding [21].

Moreover, there is a handful of other information that is provided for every object, such as its size or memory address. These can also be used, but they are significantly less essential than the three information types stated above. However, an important question still remains: *how can we combine this information in an meaningfully way to be able to use heuristics on it?* The most common solution comes in the form of the object reference graph. One of the first occurrences was by Zimmermann and Zeller in the paper, *Visualizing Memory Graphs* [21].

The basic idea behind the Object Reference graph is that each object represents a node in the graph, and every reference is an edge to another node. This creates a directed graph that shows all the information provided. It is worth mentioning that not all objects in memory that are collected are necessarily linked with each other. As a result, the graph is not a connected graph. Additionally, the graph is not a flow graph, which means that there is no dedicated root node from which all other nodes can be reached.

The object reference graph also has some shortcomings. One of the most noticeable ones is that the structure can be quite complicated. Because nodes can contain multiple edges, including back edges, it makes it challenging to traverse the graph.

2.3 Dominator Relation

Many current approaches rely on the extremely helpful concept of the dominator relation [8]. It establishes a hierarchy between the nodes by defining the following two rules [18, 3]:

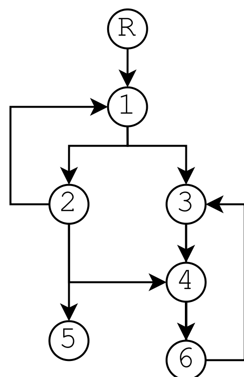
- **(strict) Dominator:** a node **d** dominates a node **n** iff for every path (from the start node) **d** must be visited to reach **n**. Additionally, node **d** and **n** can not be the same node.
- **Immediate Dominator (idom):** a node **i** dominates **n** but does not dominate other nodes that dominate **n**.

Furthermore every node, except the artificial starting node (GC roots), has exactly one immediate dominator.

These two rules are very powerful and can therefore be quite confusing. It is more easily explainable with an example. In Figure 2, we can see an object reference graph and the corresponding dominator tree. The node Ⓡ represents the Start node, or more precisely, in this case, the artificially added root node of the graph. Using the dominator rule, we see that to reach any of the other nodes ① through ⑥ we need to visit Ⓡ. Therefore, Ⓡ dominates all the other nodes of the graph. Now ① is the idom of of ②, ③ and ④. For nodes ② and ③ this is obvious, as from every path of the start node, to reach them the path needs to go through ①. In addition, it also does not violate the second part of the rule as it does not dominate Ⓡ. On the other hand, for ④ it may not be that understandable at first glance. However, when we consider that ④ can be reached by either ② *or* ③, they both disqualify from dominating the node. This means that the only node that can still fulfil these conditions is ① for the same reason as with nodes ② and ③. The rest of the nodes follow the same logic, leading to our complete dominator tree.

With that the dominator relation creates a directed tree. This is helpful because eliminating the back edges makes it *easier to traverse*. At the same time, it also rearranges the structure of the object reference graph, removing important structural information. As we saw in our last example, ① in the object reference graph never directly refers to ④, but in the dominator tree, it does. Data structure detection may be impacted by these reorganizations and the elimination of back edges. However, in addition to converting a graph to a tree, it also offers another useful feature, particularly in the context of memory analysis. When we take our example nodes to represent parts of a data structure or the data structure itself, the dominating node owns the nodes below. Since ② dominates ⑤ it also implies that ② has ownership over ⑤, indicating that if ② is collected by the GC, ⑤ will be collected as well [1, 6].

Object Reference Graph **Dominator Tree**



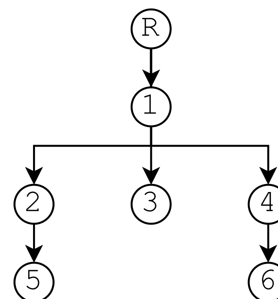


Figure 2: Object Reference graph to dominator tree.

2.4 Data Structure Roles

A (data structure) "role" is defined as a label provided to objects[2] depending on their specific purposes. The label `Primitive Array` can, for example, be assigned to any object that contains primitives of the same type. Additionally you could also introduce a role `Container` that requires the object that should receive the role to point to an object assigned the `Primitive Array` role.

Any role can be assigned to an object or to a node within an Object Reference Graph or Dominator Tree. Essentially both have the same meaning as in this case, any node serves as a representation of the heap object itself. The concept or roles, is important because it helps to identify other objects referred by the role-labeled object or that refer to objects labeled with a specific role. Each approach provided in this thesis has its own definitions and names for an assortment of roles. Some crucial concepts remain the same or have just minor differences, but others are unique to their approach.

2.5 Running Example

This section introduces a running example. We will use the running example to demonstrate each existing approach as well as our own. Applying different approaches to the same example helps to better understand each technique. As they can be compared via the result, further similarities, differences, benefits, and drawbacks can be seen. Additionally, we included as many edge cases as we could without making the example too difficult to comprehend. The edge cases are interesting because they highlight faulty behavior in some of the approaches.

The running example consists of very popular data structures included in the Java Collection framework. Since most applications contain self-developed classes, we also provide custom classes in the example. Therefore, three classes were implemented, each containing essential constructs. The first class, called *Company*, is a basic class that will function as a container for other data structures. The second class, *Department*, represents a custom recursive structure. The final class, *Employee*, contains a primitive array as well as strings. These three classes cover the most useful functions that any custom class will fulfill. For a better understanding of the example construction, we also included some Java like pseudo-code for all the classes in Listing 1.

The setup of the example shown in Figure 3 is not designed to be efficient in terms of operation, space, or memory. Its primary goal is to include many standard programming structures. In this specific example, we have a main container company that consists of an ArrayList and a LinkedList. The LinkedList stores all the company's employees. Each employee has a name and an email that are both saved as strings. Additionally, the employee also contains a character array containing the overall access rights. The companies ArrayList, on the other hand, contains the departments of the company. Each department consists of a HashSet containing the employees that work in that department as well as a reference to another department they work with.

[2]That themselves can be a Data structure.

Additionally, we have also included the Dominator Tree of our running example, as shown in Figure 4. The reason for its inclusion lies in the fact that some of the approaches we present in this thesis utilize the dominator tree to identify data structures.

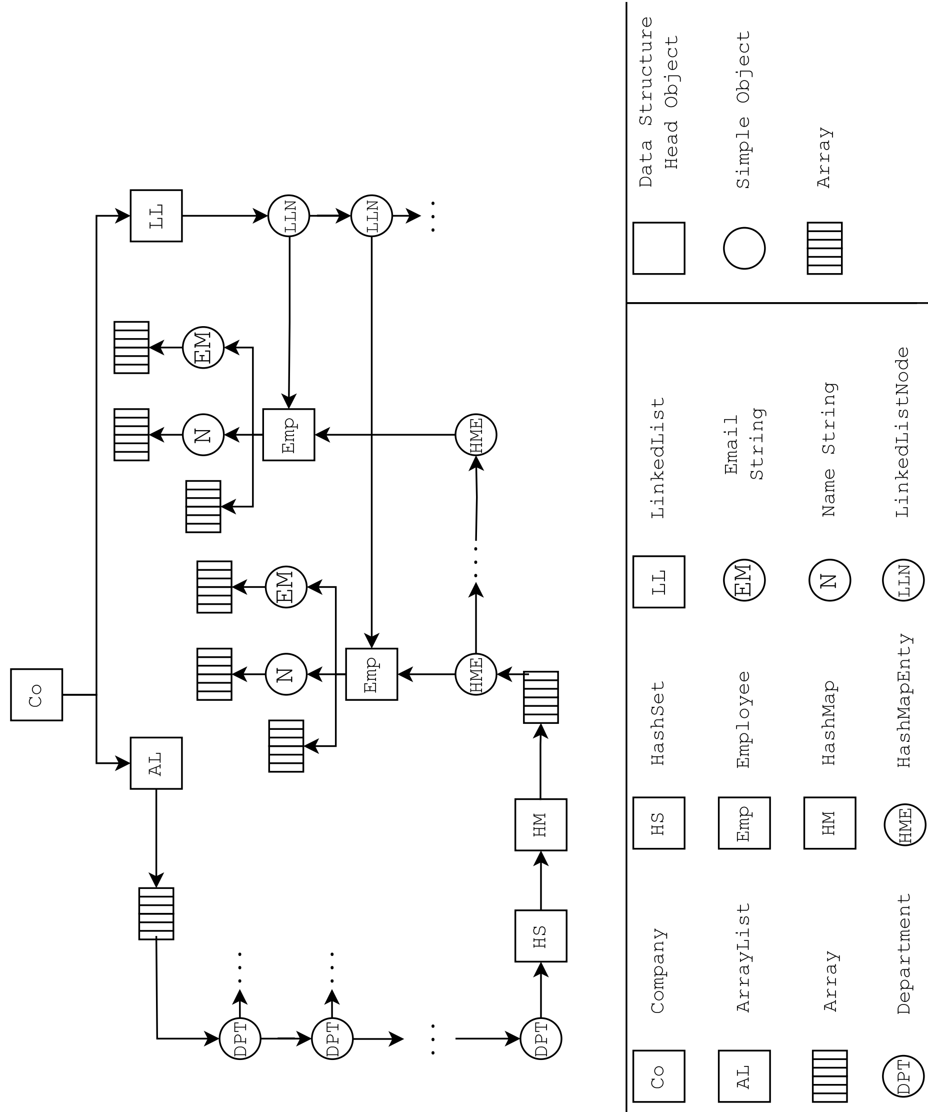


Figure 3: Running Example of a Company.

Listing 1: Running Example Code

```
1  class Company {
2           ArrayList<Department> departments;
3           LinkedList<Employee> allEmployees;
4
5           // ...
6  }
7
8  class Department {
9
10          HashSet<Employee> departmentEmployees;
11          Department associatedDepartment;
12
13          // ...
14 }
15
16 class Employee {
17          String name;
18          String email;
19          char[] accessRights;
20
21          // ...
22 }
```

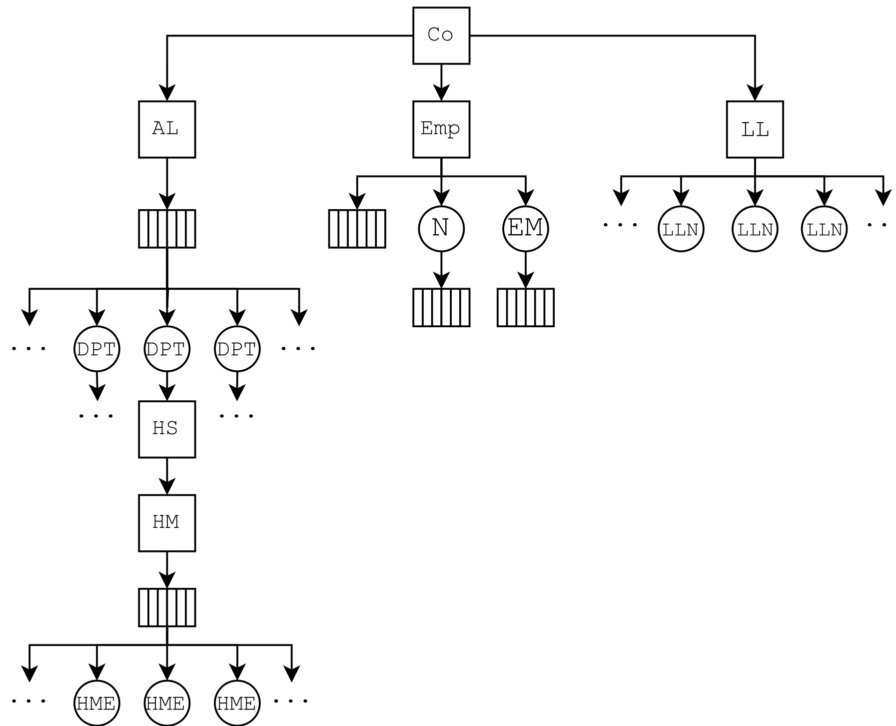


Figure 4: Dominator tree of our Running Example.

3 Existing Approaches

Before discussing the current methods, we wanted to briefly touch on two key components of any detection. That is the detection of array elements and recursive elements. Both of them are determined in the same way in every approach that we will present (even our own).

Arrays are identified by their type, which follows the format `elementType[]`. They can contain either primitives of the same type or reference objects/instances of the same type. We often differentiate between primitive and reference arrays. Primitive arrays exclusively store the language-defined primitive data types' values directly in memory. On the other hand, reference arrays store only the references to objects in memory.

The recursive nodes, on the other hand, are identified by checking if the node contains a reference to an object of the same type as itself. This rule is effective for any kind of recursive data structure. Both can be seen below in Figure 5.

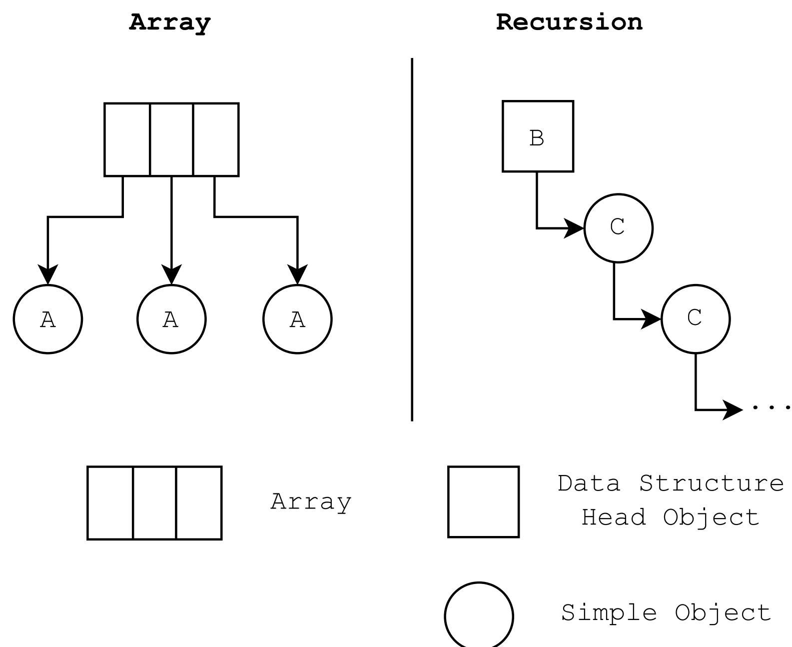


Figure 5: Memory representation of recursive structures and arrays.

These two components are important as the combination of them is nearly contained in any other DS.

3.1 Data Structure Health

This approach by Mitchell et al. [13, 14] focuses on determining the health of data structures i.e., they provide a variety of heuristics and metrics to determine and compute memory bloat in data structures. The identification of data structures is essential in order to accurately assess the memory overhead of data structures.

Four roles were developed for this approach. The first two roles are quite self-explanatory, as they both were already described at the beginning of Section 3. One would be the `Array` role, which is assigned to every reference array. The other would be the `Entry` role that is given to recursive structures. The next one is the `Head` role, which is used with objects that point to either to primitive arrays or to `Array`/`Entry` roles. Primitive arrays themselves are not marked as primitive or `Array` but instead are defined by the role of its immediately dominating node. The `Head` role denotes the beginning of a DS, with the nodes below being part of that DS. The last role is referred to as `Contained`. It serves as a general role for all the nodes that were not given a role.

They start their data structure detection approach by creating a spanning forest from the object reference graph. This forest is based on the dominator relation, thus also gaining all the benefits that we already mentioned. Furthermore, each spanning tree in the spanning forest is a proper dominator tree. Now, the roles can be applied inside each dominator tree to start the data structure detection. To ensure that roles are assigned correctly, there is a role hierarchy (`Array` , `Entry`) > `Head` > `Contained`. That means that the `Array` and `Entry` roles are assigned first, then the `Head`, and lastly, the `Contained` role.

Figure 6 shows the result of the data structure detection approach. It is worth to mention that we created a dominator tree out of the running example and mapped the results back to the original object reference graph. Most of the things we see in the figure are predictable, with only a few unexpected assignments. One of them is the *Company* class being identified as a `Contained` node. This is easily understandable when we look at the rules of the roles. Another example we would have seen, if no additional knowledge was used, was that the HashSet would also be detected as `Contained`. However, in the actual implementation, the authors note that each class from the Java Collections framework has been added specifically so that it would be appropriately assigned the best fitting role. The last thing that is interesting to mention is that *Name* and *Email* nodes are both counted as the head of a DS.

After a complete DS has been identified, each object is further examined to determine how many primitives, headers, pointers, or null pointers it contains in order to improve the bloat calculation. We will not go into more detail, as it is no longer part of the data structure detection but part of the health calculation for the data structures.

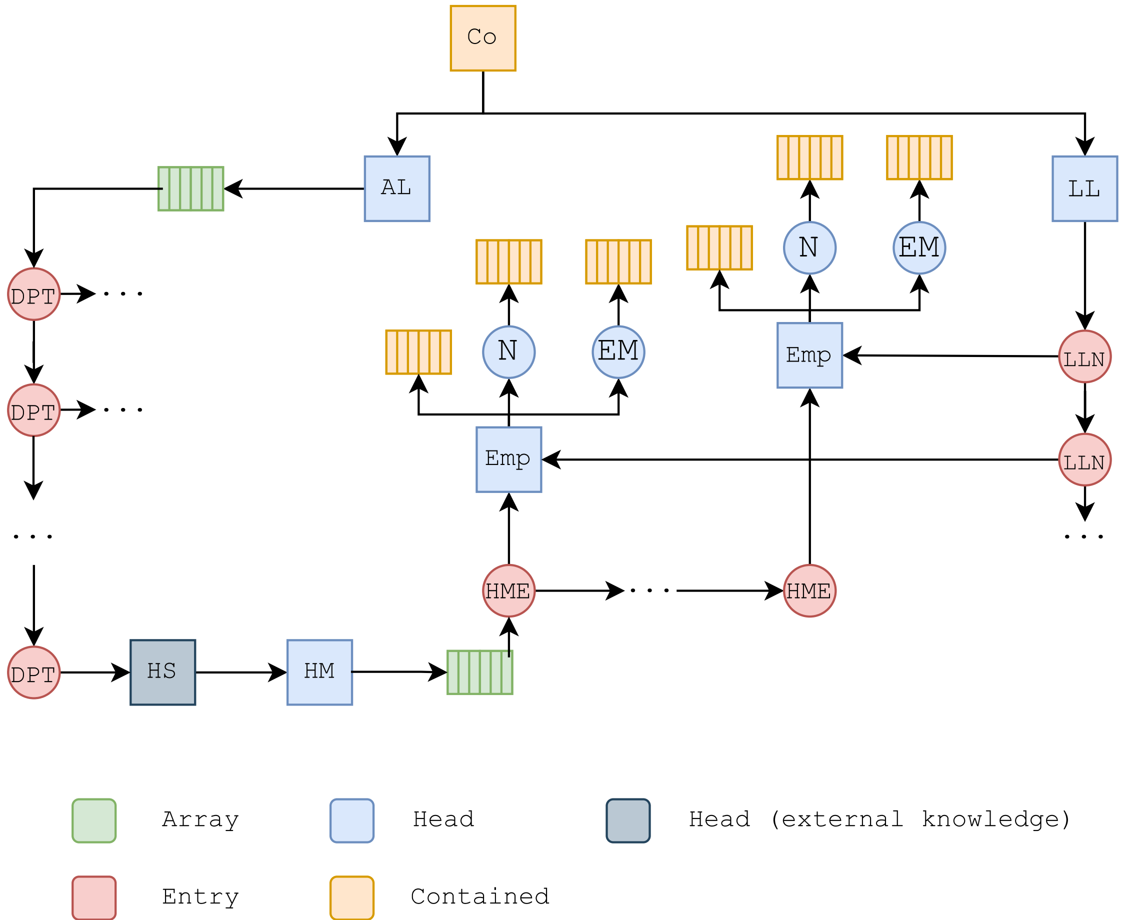


Figure 6: The Running Example identified by the Data Structure Health Approach.

3.2 Backbones

Nick Mitchell first introduced this strategy in his paper titled "The Runtime Structure of Object Ownership" [10]. Its main objective was to create a condensed summary of the heap dump. This swiftly produced report would state the most significant areas of the memory footprint in terms of memory usage. Developers could then make use of this information to alter data structures in order to reduce memory usage. Aside from that, it also aids in understanding how applications behave during runtime. Thus, a large part of the paper is also about graph summerization, which will only be mentioned briefly in this section.

This approach uses a wide variety of so-called graph edits which are algorithms that are applied to the object ownership graph to simplify it. They are even used multiple times in a certain order to achieve better results. They are specifically made to reduce huge numbers of nodes into a few representative nodes. This resulting reduced graph is called the Ownership graph. Each node of the Ownership graph is either a dominator tree of nodes or of other dominator trees. The content within the trees can be summarized based on their assigned role.

The paper introduces the following six roles:

- `Array backbone:` Is assigned to any node that is a reference array. It describes the horizontal growth of the DS.
- `Recursive backbone:` Is given to any node that is identified as recursive. It describes the vertical growth of the DS.
- `Non-Recursive backbone:` Every node that lies sandwiched between `Recursive backbone` nodes.
- `Container:` The node that is above any other role is assigned the `Container` role. The container is regarded as the main head of the DS. It implies that all the nodes below are part of the DS that starts with the nearest container node.
- `Container Sandwich:` Similar to non-recursive, it describes nodes that lie on a path that starts with a `Container, Array,` or `Recursive role` and ends with a `Container` node.
- `Data:` Marks every other node, mainly those directly referenced by a `Recursive` or `Array backbone`. They dominate the actual data of the DS. For completeness this role also acts as a safety role that is assigned to any node that was not given a node.

The general priority of the roles is (`Array, Recursive`) > (`Container, Non-Recursive`) > `Container Sandwich` > `Data`. This approach also coined the term *backbones*. They are the key parts of any DS, as they describe the parts of a DS that allow it to grow and shrink.

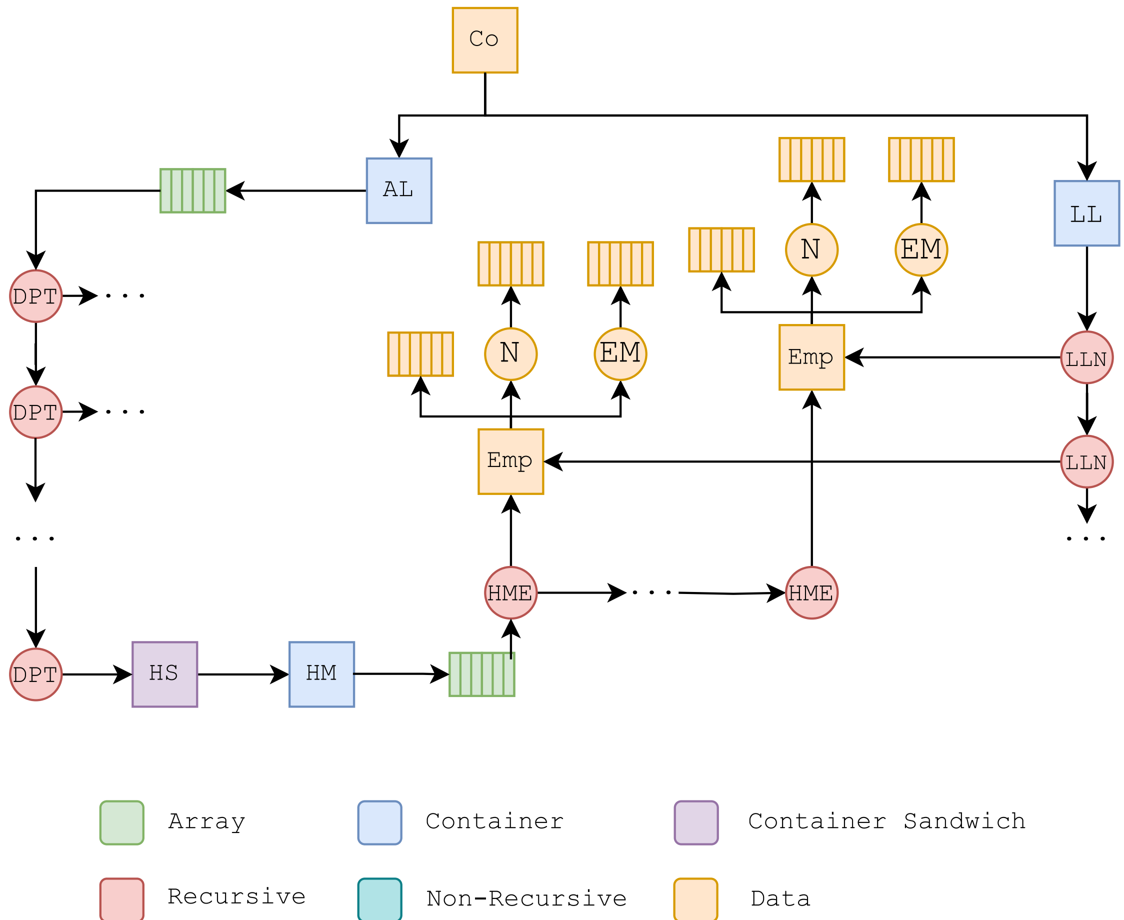


Figure 7: The Running Example identified by the Backbone Approach.

In Figure 7, we have the corresponding running example (again mapped from the dominator tree as the so called *Dominator edit* would create one). Some of the objects are identified as expected. ArrayList and LinkedList are both recognized as a `Container`. The backbones are also identified as they should be in our custom recursive class *Department*. The HashSet is where we start to see some changes, compared to Section 3.1. Only the internal HashMap is identified as a `Container` while the genuine head, the HashSet, is marked as a `Container Sandwich` node. Finally, we can notably see that all of the other objects, namely *Employee*, *Name*, *Email* and also our *Company* objects, have been assigned the `Data` role as there are no other heuristic present to further identify them.

3.3 Container or Contained

The approach from Chris et al. [4] is also mainly focused on using data structure detection to remove memory inefficiency. This is done by detecting patterns in the heap that only present themselves if memory inefficiencies occur. Finding and gathering patterns that met this criteria was therefore a significant portion of their effort. The discovery of data structures that display

these patterns and their aggregation makes up the other essential component.

The method locates the head of a Java Collection class by using additional knowledge in the form of a dictionary. As soon as a node is identified as the head, the nodes below are assigned one of six roles.

The `Container-Contained Transition` role is first assigned to each recursive node. After that, the role is also given to any reference array that does not point to a `Container-Contained Transition` role itself. Every node that is pointed to by a `Container-Contained Transition` node is now assigned as `Head of Contained`. On the other hand, nodes that point to reference arrays or recursive objects are marked as `Head of Container`. Nodes that point to primitive arrays are assigned the `Points to Primitive Array` role.

The last two roles are not explained in great detail by the authors. The first one, `Collection Implementation Details`, is intended to identify nodes that belong to a Java Collection class. A HashMap node, for instance, will point to an array of nodes of type HashMap$Node that, in turn, contains all the data kept in the HashMap. This reference array would be regarded as a collection implementation detail. However, the paper does not explicitly state whether these collection implementation details are determined only through additional knowledge or not. It is possible to construct fairly accurate approximations that would work for the Java Collection framework. For instance, starting from the `Head of Container`/root node, all nodes until a reference array or `Container-Contained Transition` object are marked as collection implementation details.

The other role, `Contained Implementation Details`, is defined as everything else that is contained in the Java Collection class that has not yet received a role. The complete role hierarchy would look like this:
`Container-Contained Transition` > (`Head of Contained, Head of Container`) > `Points to Primitive Array` > (`Collection Implementation Details, Contained Implementation Details`).

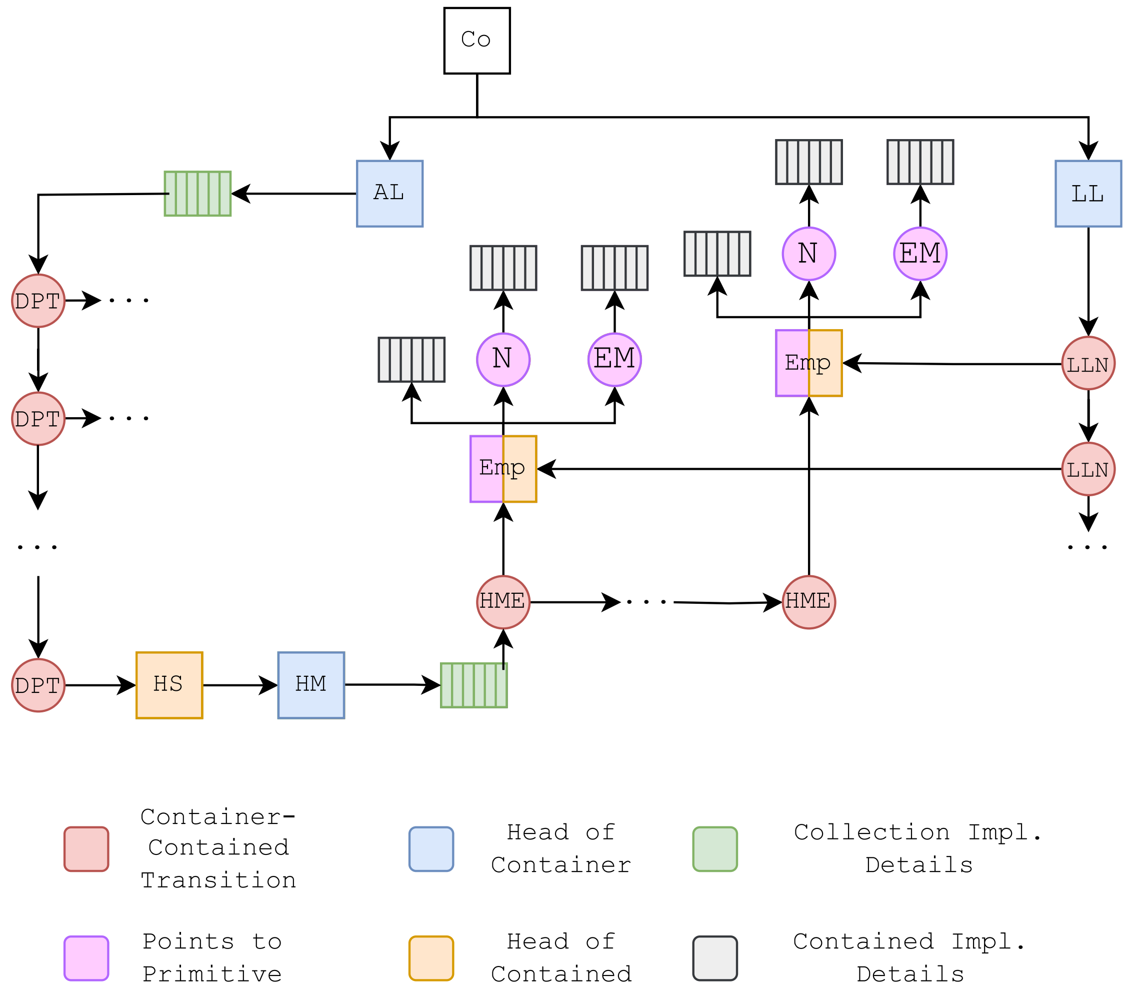


Figure 8: The Running Example identified by the Container or Contained Approach.

In Figure 8, we can see how our running example would be classified with the given heuristics. There are some noticeable differences that we have not observed before. The first one is that for the*Employee* node, we can see that it is assigned two colors. This expresses that the object was assigned two different roles. The HashSet containing the HashMap is also only partly assigned right, as the HashSet is considered the `Head of Contained` while the HashMap inside is considered `Head of Container`. Given their role definition it would be more appropriate to consider the HashSet the both `Head of Container` and `Head of Contained` at the same time while the HashMap functions just as an `Head of Contained` or even an `Collection Implementation Detail`.

The last major difference we can see is that the *Company* node was not assigned any role at all. This is to be expected, as the detection only starts with a Java Collection class. The authors mention that this was specifically done so because their findings indicated that most of the memory inefficiencies are closely related to ineffective use of Java Collection classes.

3.4 Domain-Specific Language

The domain specific language approach was mentioned briefly in the introduction. But since it differs greatly from the others, we felt the need to go into further detail about how it operates.

To begin with, the DSL is a core concept that needs to be developed. It needs to be able to describe any kind of DS as well as its components. Furthermore, it should be able to mark a construct as the head of the DS and allow representation of recursiveness. To demonstrate how such a DSL would look like, we will use the one developed by Weninger et al. [20]. Listing 2 shows an example of how our custom classes as well as the LinkedList would be described as.

Listing 2: A DSL example describing all the structures of the running example.

```
1  DS Company {
2      java.util.ArrayList;
3      java.util.LinkedList;
4  }
5
6  DS Department {
7      java.util.HashSet;
8      Department;
9  }
10
11 DS Employee {
12     java.lang.String;
13     java.lang.String;
14     java.util.Arrays;
15 }
16
17 DS java.util.LinkedList {
18     java.util.LinkedList$Node;
19 }
20
21 java.util.LinkedList$Node {
22     java.util.LinkedList$Node;
23     (*);
24 }
25
26 ...
```

The optional keyword `DS` marks the element as a data structure head. Following that is the name including the complete path. After that, each of the enclosed data structures is listed in between the curly brackets with its complete path, followed by a semicolon.

The final element that is still used is the asterisk. It can be used as either a wildcard within the name of a pointed type or if it is encapsulated in round brackets, it denotes a leaf[3] of any type.

All this functionality fulfills the requirements for the DSL mentioned at the beginning. It even has the added quality of life functionality such as wildcards and leaf nodes.

Now that the definitions are provided (for everything that should be detected), we come to the data structure detection. This approach begins by identifying the DS heads using the type information. It does not require heuristics as it can simply look up which types are considered heads in the definition. After declaring a node as a head, it iterates recursively through each pointer to determine whether it is still a member of the DS. As long as the pointed objects overlap with the given definitions, it continues to assign them to the current DS. Also, if an object is defined as a leaf, it is not only assigned but also marked as a leaf, which stops the recursive descent. With that, all the nodes can be grouped into their respective data structures.

We excluded the running example from this approach since, provided the definition is stated correctly, it always perfectly detects the data structures. In fact, this method of detection is the most precise one. The requirement for definitions and the fact that detection is impossible without them are the only real drawbacks.

3.5 Similarities and Differences

One of the core similarities we found in all the existing heuristic-based approaches was the detection of arrays and recursive structures. They contain some very useful features that make them essential. First of all, is that they are simple to detect. At the start of the Existing Approaches Section 3, we showed how they are identified and this generally was the go-to way for each approach. Furthermore, they are truly essential parts of any DS as they control horizontal and vertical growth.

Even with the more complicated roles, there were some similarities between them. The node above[4] any reference array or recursive structure is always assigned as the head of the DS. From the perspective of general programming practices, it makes a lot of sense to be that way. Since their main purpose is to hold other objects that a DS could use in an effective way, they are rarely structures that stand on their own. It is also common to see roles that are a bit more generalized that contain data or even data structures. Their precision varies in between the approaches. The Backbone and Health approaches keep them extremely vague, not making a difference if the data structures themselves are contained or not. The Container or Contained approach has more well-defined definitions. It differentiates between external (`Head of Container`) and internal (`Head of Contained`) structures. Additionally, it also identifies some basic objects or structures within with the inclusion of roles such as `Points to Primitive Array` and `Container/Contained Implementation Details`. It should be noted that the Health approach does a little bit of extra identification too. It should be noted that the Health approach also includes some extra identification between the objects, such as identifying primitive arrays. They do not, however, assign these roles during data structure detection and only use them during bloat computation.

[3] Is regarded as a node that holds data and does not need further examination.

[4] The node that references to the current object that is also not an array or recursive structure

Another interesting similarity between the approaches is the usage of the dominator relation. Both the Health and Backbones approaches identify structures use the dominator relationship. Interestingly enough, the Container or Contained approach, on the other hand, decided not to use the dominator relation. They concluded that the benefits did not outweigh the drawbacks particularly the decrease in detection accuracy.

3.6 General Problems

Generally, many of the approaches have the same shortcomings. The most notable one is the reliance on additional knowledge. It is time- and effort-consuming to have to define all the data structures that are used or at least keep a dictionary of their data structure heads. Even more so when we think about how applications can have many different frameworks, libraries and custom classes that also need to be defined.

Another notable shortcoming is that the detection is often done only on the basis of popular framework classes. Not that it is done without a reason. In most cases, even custom classes will include some types of framework classes that will have a majority of the saved data within them. Even so, using a more broad strategy may not significantly alter the detection accuracy. Many principles are embedded in the principles of object-oriented programming. Arrays and recursive structures are the cornerstones of programming. It is very important to find heuristics that explain their purpose and how they relate to the other nodes around them. A good example that we could see by the results of the approaches we presented was that they all failed to properly identify the HashSet containing a HashMap without using additional knowledge.

Lastly, Dominator relation-based methods also have a major weakness. Although it is frequently employed to preserve unique ownership while simultaneously getting rid of back edges and simplifying the graph to a tree structure. This simplification also has a huge drawback when considering shared ownership. We provided an example of how shared ownership is handled in Section 2.3. At first glance, this may not seem so significant. But when the primary means of detection relies on structural indicators, this becomes problematic. Artificial connections are formed between previously unrelated objects. While, on the other hand, genuine connections are removed. This can cause a misinterpretation of how data structures truly interact with one another. As a result, memory analysis tools in general become less accurate and could even mislead the developer using them.

4 Approach

The other main part of this thesis was the development and identification of roles that are able to describe data structures. We analyzed the existing approaches and extracted important concepts from them to develop our own roles.

4.1 Roles

We wanted to be able to assign one or more roles to each node that indicate their current function while also providing more general information. Therefore, we created a total of eleven roles that are split into basic and complex types.

The four basic types are:

- `Recursive:` The `Recursive` role is assigned to any node that has a from or to pointer of the same type as the node itself (one hop recursion).
- `Reference Array:` Is assigned to objects that are reference arrays.
- `Transition:` Is assigned to `Recursive` objects or `Reference Arrays` that do not point to `Recursive` objects.
- `Primitive Array:` Is assigned to nodes that are arrays containing the same primitive types.

These four roles are extremely important as they build the backbones of data structures as well as contain important information about where actual data begins.

The seven complex roles are:

- `MContainer: MContainer` stands for main container and represents the stand-alone class. Any node that is not referenced at all but references another node is assigned this role.
- `Container:` The `Container` role represents a DS that either contains other data structures or data itself. In other words, it represents the head or the start of a DS. Nodes that point to `Recursive`, `Reference Arrays`, `IContainer`, `True Data`, or even `Container` nodes themselves are assigned this role. They have a further restriction in that they need to be referenced by at least one other node.
- `IContainer: IContainer` stands for intermediate container. It performs a similar function to the Collection Implementation Details role from the Container or Contained approach. Because it contains objects that are technically containers but are simply utilized as a preliminary step. For example, a HashSet always directly contains a HashMap that contains a reference array containing all the data. The data structure head is the HashSet, and the reference array contains the data, but the HashMap itself is only an intermediate structure.This role is assigned to every node that points to exactly one node of the type `Recursive`, `Reference Array`, `IContainer`, `Container`, or `True Data`. Additionally, the node itself has to be pointed to by only one other node.

- `HContainer`: `HContainer` stands for head container and represents a more self-sufficient DS. However, it is not entirely independent like the `MContainer`. It is assigned to nodes that are referenced by `Reference Arrays`, `Recursive` or MContiner objects. Furthermore, it has to have objects that it refers to that are not completely empty.

- `EStructure`: Empty structures are considered to be objects that cannot be identified properly. Neither the nodes they reference nor the nodes they are referenced by give further information on their purposes. They are quite literally an empty data structure.

- `Container Impl. Detail`: Container Implementation Details is a single node without any other nodes to refer to. Additionally, it must directly be pointed by a node of the container type, such as `Container`, `IContainer`, `HContainer` or `MContainer` or alternatively, by `Reference Arrays` or `Recursive` objects. They are either objects that represent commonly found data types (integer, string, etc.) or class instances.

- `True Data`: `True Data`, as the name suggests, contains the data that is stored in a DS. This role is given to nodes that directly reference objects that are `Primitive Arrays`. It should also only reference a single other node.

We found that these roles do a great job of describing data structures and all their parts. They also provide a useful indicator of what the purpose of a DS is.

4.2 Role Assignments

It is hard to understand the roles completely by only seeing the basic definitions of the roles. Therefore, we use this section to show how the detection approach would behave step-for-step as well as what the priority hierarchy of the roles looks like. Additionally, we have put the entire algorithm as a pseudo-code for better understanding in Listing 3. For the sake of code readability in this example, we did not include filtering of the to and from pointers. Specifically, self-pointers and null-pointers need to be filtered as they do not provide any meaningful information.

We start off by checking if the node should be assigned a basic role. `Recursive` and `Reference Array` roles have the highest priority as they are the backbone of data structures. `Primitive Arrays` are also in the same priority ranking as there is not much difference between their detection approaches. However, the `Transition` role is slightly below the other in terms of priority as its detection relies on the backbones already being detected. (`Recursive`, `Reference Array`, `Primitive Arrays`) $>$ `Transition`.

In Table 1, we see some examples that are expected to receive these roles.

<table>
<tr><td>Recursive</td><td rowspan="2">Transition</td><td>LinkedList$Node,
HashMap$Node,Department</td></tr>
<tr><td rowspan="2">Referance Array</td><td>String[], int[][]</td></tr>
<tr><td></td><td>HashMap$Node[], Thread[], Hashtable$Entry[]</td></tr>
<tr><td>Primitive Array</td><td></td><td>char[], int[], boolean[], double[]</td></tr>
</table>

Table 1: Common examples of basic roles

If the node is not a basic role, we know that it must be a complex role (Line 32-34). Therefore, all the roles the node references are being identified recursively to see what type of complex role the node should be assigned.

The complex roles are all of less priority than the basic roles. Of the complex roles, the highest priority is given to the `MContainer` role. It only needs to check that it is never referenced but contains references to other objects of any kind (Lines 44-46). The next steps in terms of ranking would be the `Container`, `True Data`, and the `IContainer` roles. Their detection relies on the existence of the basic roles. Nodes that have referenced nodes that are either `Recursive`, `Reference Arrays`, `IContainer`, `Container`, `True Data` or `ContainerImplDetail` are assigned either the `Container` or `IContainer` role. As long as they are exactly referenced by one node and only reference one other node, they are considered an `IContainer`. If it does not fulfill that condition, they are assigned the `Container` role (Lines 49-61). The `True Data` role is assigned to nodes that reference an object of the type `Primitive Array` but also only that one object (Lines 63-65). If it references multiple objects and only one of those is considered a `Primitive Array`, they are assigned the `Container` role instead (Line 67). Finally, if the referenced nodes cannot be identified or are identified as an `EStructure`, we also assign the current node the `EStructure` role (Line 70-78).

Now that we have extracted the maximum amount of information from the referenced nodes, we try to gather information from the objects that refer to our current node. If the current node was not given a role until now, we gather all the basic roles from the nodes above (Lines 83-88). Otherwise, we gather all the roles from the nodes above. Now that we have them, we first find `HContainer` nodes. If the object is referenced by either `Recursive`, `Reference Array`, or `MContainer` objects, the current node receives the `HContainer` role (Lines 91-94). Additionally, we want to mention that the current node should not be of type `True Data` or `EStructure`.

The final role we can assign is the `Container Impl Detail` Role, which is given to the current node if it is referenced by either a `MContainer`, `IContainer`, `HContainer`, `Container`, `Reference Array`, or `Recursive` role. In addition, the node is not allowed to reference any other node. If everything is fulfilled, the current node is assigned the Container Implementation Detail and `HContainer` or `EStructure` roles are removed if they exist (Lines 97-104). The roles are saved with the correct id as the final step, so they can be reused if the node is mentioned again in the future (Line 108). A listing of the general role hierarchy is not clearly definable due to the recursive nature.

Listing 3– Pseudo-code for the whole role assignment algorithm

```
function getBasicRoles(heap, id){
 toPointers = heap.getToPointers(id)

 if (detected recursion){
  return (Recursive, Transition)
 }

 if (detected reference array){
  if(toPointers OF id IS Recursive){
   return (RefArray, Transition)
  }

  return RefArray
 }

 if (detected primitive array){
  return PrimitiveArray
 }
}

function getRoles(heap, id, stops){
 if (id IN processed){
  return processed[id]
 }

 if (id IN stops){
  return
 }

 stops.add(id)

 roles.add(getBasicRoles(heap, id))

 if (roles.size IS 0){

  toPointers = heap.getToPointers(id)
  toNodeRoles = for point
   IN toPointer
   CALL getRoles(heap, point, stops)

  fromPointers =
   heap.getFromPointers(id)

  if (fromPointers.size IS 0
   AND toPointers.size IS 0){
   roles.add(MContainer)
  }

  if (toNodeRoles IS Recursive
   OR RefArray OR IContainer
   OR Container OR TrueData
   OR ContainerImplDetail){

   if (fromPointers.size IS 1
    AND toPointer.size IS 1
    AND heap.getToPointers(
     fromPointers[0]).size IS 1){
    roles.add(IContainer)
   }else{
    roles.add(Container)
   }

  }else if (toNodeRoles IS PrimitiveArray){
   if (toPointers.size IS 1){
    roles.add(TrueData)
   }else{
    roles.add(Container)
   }

  }else if (toNodeRoles.size IS 0
   OR toNodeRoles.size IS 1
   AND toNodeRoles IS EStructure){
   roles.add(EStructure)
  }

  if(toNodeRoles IS EStructure
   AND roles.size IS 0){
   roles.add(EStructure)
  }

  fromNodeRoles = null

  if(roles.size IS 0){
   fromNodeRoles = for point IN fromPointers
     CALL getRoles(heap, point, stops)
  }else{
   fromNodeRoles = for point IN fromPointers
     CALL getBasicRoles(heap, point, stops)
  }

  if (fromNodeRoles
   IS (Recursive OR RefArray OR MContainer)
   AND roles IS NOT TrueData OR EStructure){
   roles.add(HContainer)
  }

  if (fromNodeRoles IS MContainer
   OR IContainer
   OR HContainer OR Container
   OR RefArray OR Recursive){
   if (fromPointers.size IS 0){
    roles.add(ContainerImplDetail)
    roles.remove(HContainer)
    roles.remove(EStructure)
   }
  }
 }
 processed[id] = roles
 return roles
}
```

We did not include examples of what objects would be given what roles as it is very context specific. For instance, a HashMap could be considered either an `IContainer`, `HContainer`, or only a `Container` depending on the context. The Hashmap would be counted as an `IContainer` if it fulfills only an intermediate role, most commonly in the form of a HashSet. On the other hand, if it is part of a nested data structure such as a HashMap within a HashMap it would be counted as an `HContainer`. Or if it was part of a class with other data structures within the class, it would only receive the `Container` role. In either case, we include the running example to be able to show how it would identify a specific example.

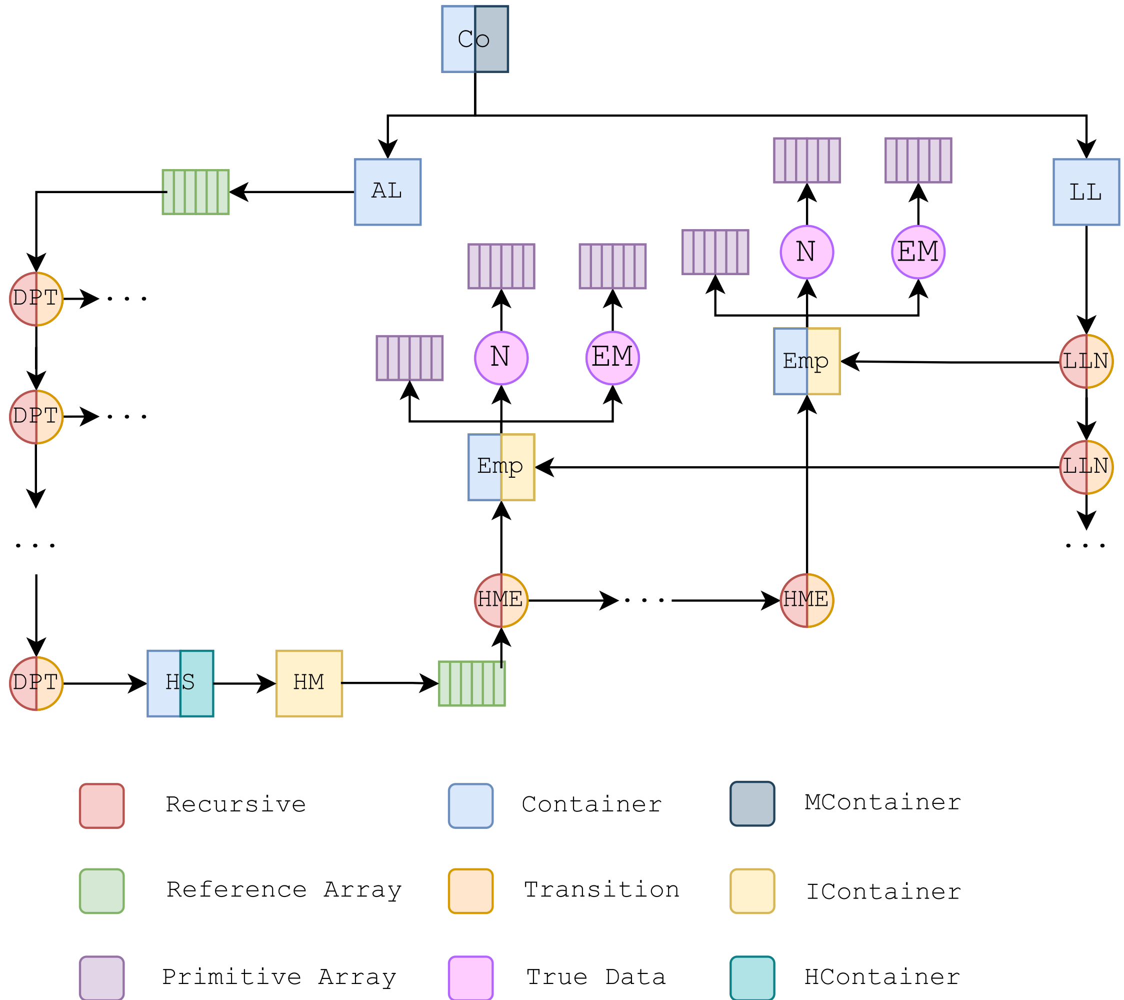


Figure 9: The Running Example identified by our Approach.

In Figure 9, we can first of all see that every object was assigned a role. Some objects even received multiple roles displayed in the form of a two color scheme. The *Company* object received the `MContainer` as well as the `Container` role. In our example, the *Company* node is a self-standing structure, explaining the `MContainer` role. The `Container` role is also expected as it contains two other `Container` nodes. The LinkedList is a `Container`, as it directly

references `Recursive` (as well as `Transition`) nodes. The ArrayList and its content are also self-explanatorily, but it gets interesting when considering the content of our self made recursive class *Department*.

The HashSet was assigned both the `HContainer` as well as the `Container` role. However, the HashMap inside it was only given the `IContainer` role. The reason is only an internal part of the complete data structure, which is the HashSet. The final interesting part can be seen in the *Employee* objects as they are assigned the `HContainer` and `Container` roles. As it is referenced by `Recursive` nodes, the `HContainer` role is typical. The `Container` role is assigned as multiple references to `True Data` exist. The *Name* and *Email* of the *Employee* are assigned as `True Data` as they both only reference a `Primitive Array`(in this case char[]).

4.3 Benefits

Our main goal with our approach was to eliminate the general problems we mentioned for the existing approaches. Therefore, we thought up roles that are able to identify data structures that do not depend on additional knowledge or the dominator relationship. In this section, we want to mention what features actually help in removing these shortcomings.

In the existing approaches, we see that the HashSet is either misidentified or only correctly identified by using additional knowledge. This happens due to the special structure that a HashSet has, as the genuine regarded head of the data structure only directly refers to another data structure head, the HashMap. This fake data structure head will be referred to as a proxy object. In fact, this proxy principle is quite common, with any class that should only change a few properties of an existing data structure. Since proxy objects only have one reference to and from another object, they can be identified solely based on structural data. Even shared ownership does not change this heuristic, as the proxy object is not referenced in that case. Either the node that directly references the proxy is shared or the nodes the proxy references but never the proxy itself. In our approach, we introduced the `IContainer` role that represents such proxy objects. The rules for its detection are also based on this defined proxy structure.

The other difficulty that the existing approaches had with detection was the *Company* class. It is only a simple container class that does not have any specific data itself but only enhouses other data structures. This is primarily due to the fact that the existing approaches use data structure detection to identify memory inefficiencies. This missing correspondence can be accepted regarding how much aggregation is done. On the other hand, we went as far as possible with the data structure detection to make it as complete as we could. Even if it is later aggregated, this inclusion of extra roles such as the `MContainer` role can be helpful. Our roles are typically much more fine-grained than those of others to allow for more defined data structure detection. Of course, this has the disadvantage of being computationally intensive in contrast.

4.4 Testing

We implemented a prototype in the AntTracks Analyzer [19] to aid in the development process and demonstrate the feasibility of our approach.

Throughout the development process, we produced a number of quick Java applications that provided edge-cases for our detection to be tested on. The workflow for these edge-case tests was always the same. We can take the running example that was used throughout this thesis as an example. Firstly, the basic Java classes such as *Company*, *Employee*, and *Department* were developed. A method that generates test data is a further crucial part that must be included.

In the current example, this test data method creates multiple *Employees*, assigns them to a *Department*. After that, it creates multiple Sub-*Departments* and *Departments* that can then be assigned to the *Company*. The last step now is to create a main method and initialize and generate the needed classes. As this is all written in Java, we are also able to call the garbage collector with the `System.gc();` command to ensure our instances are collected at an expected point in time [6, 17].

Now that we have a functioning program, we are able to export the Java application as a runnable JAR file. This file is then used in combination with the AntTracks VM [19] to generate memory traces and encode them into a trace file. This trace file can then be opened by the AntTracks Analyzer, which then provides the data needed to apply our approach.

The output that our approach generates is a text file where each line contains the object id, type name, and the assigned role. Existing references to the object are printed directly below with an indent to show their relationship. In addition, this pairing algorithm also prints shared objects so the given roles can be understood more easily. The whole printing algorithm is a simple recursive function that marks printed objects and is called for each referenced object. By also determining if the printed object was assigned the `Container` or `IContainer` role, an exception can be made when printing already printed nodes — in other words, shared data.

Nevertheless, we also included tests on a trace file created from the large-scale Dynatrace easyTravel[5] application. A variety of selected and random objects were taken to apply our approach to. Depending on the size of the result, we checked every object for correctness.

When there were too many objects or lines[6] to examine them all individually, we performed a rough check of the overall assignment before moving on to check complex structures and complex role assignments.

Overall, we were satisfied with the assigned roles and confirmed that they accurately portray data structures. The only observation we made that would need further development in the future was the case of auto-generated objects. Build-in Class Loaders are a prominent example of such auto-generated objects. In the case of Java, they are created to dynamically load classes into the Java Virtual Machine. In a sense, they obstruct the result, as they are not really implemented by the developer but recognized as data structure heads. Detecting them without additional knowledge is a difficult task and an interesting topic for further study.

[5] `https://community.dynatrace.com/t5/Getting-started/easyTravel-Documentation-and-Download/td-p/181271` (last accessed September 24, 2022)

[6] They can easily reach hundreds of thousands to millions of lines.

5 Future Work

There were a couple of interesting ideas that were created while researching for and developing this thesis.

The first would be the inclusion of metrics to increase data structure detection accuracy and even rule out misleading or auto-generated classes such as class-loaders. Existing approaches still use additional information to be able to remove these auto-generated classes from the detection. It would also be a useful addition to our approach, because it currently cannot distinguish between these artificial constructs and regular classes. The concept of these metrics came up when reading a paper by Mitchell and Sevitsky [12] which used metrics to detect leaking data structure heads. An adapted version particularly developed for these language specific constructs would remove the need for additional knowledge. For example, class loaders typically contain incredibly many references to other heads of data structures. The identification of these auto-generated classes could also provide a good starting point to apply our approach to. As at the current state, the only way to identify the whole heap state would be to begin at the GC roots. Developing appropriate metrics for such features would require extensive research, as they would also need to include class-loaders from frameworks or even custom-made ones.

The other aspect of the future work would be the complete integration of our approach into memory analysis tools. Currently, we have only shown the feasibility with a prototype in AntTracks [19]. Our implementation could be integrated using a number of tools, including AntTracks, VisualVm[7], and Eclipse Memory Analyzer[8]. In the integration process, a couple of refinements would be applied to minimize the memory consumption of the application. For example, switching from the map containing the roles of the processed nodes to multiple BitSets for each role to save memory. It would also be useful to improve the efficiency of the approach itself. Making the method parallel-processable using suitable algorithms, such as divide and conquer, could accomplish this.

[7] `https://visualvm.github.io/` (last accessed June 30, 2023)
[8] `https://www.eclipse.org/mat/` (last accessed June 30, 2023)

6 Conclusion

Data structure detection is used in a range of applications to help reduce memory usage and provide developers with better insight into the applications' runtime behavior.

Existing approaches rely on external knowledge such as dictionaries with class names, to be able to detect data structures heads. This reduces adaptability and user experience as this knowledge needs to be created and applied. To address these issues, we thoughtfully summarized existing approaches and identified similarities / differences between them as well as their general shortcomings. Using this information as a basis, we developed our own approach that only relies on purely structural information. Our approach incorporates eleven roles that can exhaustively define data structures and their individual components. They are directly identified in the Object Reference Graph by recursively traversing references, completely removing the dependency on the dominator relationship. In addition, we extensively discussed how these roles are identified and assigned, providing the entire pseudo-code and a working example. The developed automatic data structure detection approach has the benefit of not relying on external knowledge, which makes it more general applicable as no definitions of frameworks, libraries, or custom classes need to be provided. Popular algorithms, such as the dominator relation, were removed to create more precise identifications.

With some efficiency improvements and additional logic for recognizing auto-generated objects, our approach could be applied to a variety of other memory analysis tools to provide all of the benefits without sacrificing much detection accuracy.

List of Tables

1 Common examples of basic roles 23

List of Figures

1 Memory consumption of Bloat and Leak. 4
2 Object Reference graph to dominator tree. 6
3 Running Example of a Company. 8
4 Dominator tree of our Running Example. 9
5 Memory representation of recursive structures and arrays. 11
6 The Running Example identified by the Data Structure Health Approach. 13
7 The Running Example identified by the Backbone Approach. 15
8 The Running Example identified by the Container or Contained Approach. . . . 17
9 The Running Example identified by our Approach. 26

Listings

1 Running Example Code 9
2 A DSL example describing all the structures of the running example. 18
3 Pseudo-code for the whole role assignment algorithm 25

References

[1] Exact roots for a real-time garbage collector. pages 77–84, 2006.

[2] A. F. Blanco. Effective visualization of object allocation sites. *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 43–53, 2018.

[3] S. Blazy. Validating dominator trees for a fast, verified dominance test. pages 84–99, 2015.

[4] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of memory inefficiency. In *European Conference on Object-Oriented Programming*, pages 383–407. Springer, 2011.

[5] A. Choudhury. Interactive visualization for memory reference traces. *Computer Graphics Forum*, 27, 2008.

[6] H. Grgic, B. Mihaljević, and A. Radovan. Comparison of garbage collectors in java programming language. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1539–1544, 2018.

[7] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 51–62, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM computing surveys (CSUR)*, 51(3):1–34, 2018.

[9] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 115–124, New York, NY, USA, 2010. Association for Computing Machinery.

[10] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming*, pages 74–98. Springer, 2006.

[11] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming*, pages 77–97. Springer, 2009.

[12] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *European Conference on Object-Oriented Programming*, pages 351–377. Springer, 2003.

[13] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 245–260, 2007.

[14] N. Mitchell, G. Sevitsky, P. Kumanan, and E. Schonberg. Data structure health. In *Fifth International Workshop on Dynamic Analysis (WODA '07)*, pages 1–7, 2007.

[15] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. *SIGPLAN Not.*, 44(6):397–407, jun 2009.

[16] W. D. Pauw. Visualizing reference patterns for solving memory leaks in java. pages 116–134, 1999.

[17] P. Pufek, H. Grgić, and B. Mihaljević. Analysis of garbage collection algorithms and memory management in java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1677–1682, 2019.

[18] V. Sreedhar. Incremental computation of dominator trees. pages 1–12, 1995.

[19] M. Weninger and E. Gander. Anttracks – memory monitoring using accurate and efficient object tracing for java applications. Linz, Austria, 2019. Christian Doppler Laboratory Monitoring and Evolution of Very-Large-Scale Software Systems. `http://mevss.jku.at/?page_id=1592`.

[20] M. Weninger, E. Gander, and H. M"ossenb"ock. Analyzing data structure growth over time to facilitate memory leak detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 273–284, 2019.

[21] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization: International Seminar Dagstuhl Castle, Germany, May 20–25, 2001 Revised Papers*, pages 191–204. Springer, 2002.