Adaptive Datastructures for Dynamic Graph Analysis

Master-Thesis von Nico Haase
Juni 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
P2P

Adaptive Datastructures for Dynamic Graph Analysis

Vorgelegte Master-Thesis von Nico Haase

1. Gutachten: Benjamin Schiller
2. Gutachten: Prof. Dr. Thorsten Strufe

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. Juni 2014

(Nico Haase)

Abstract

Choosing the right datastructures can heavily influence the performance of any program, both in terms of runtime and memory consumption. Usually, the choice is done manually on compile time, based on the guess that the given configuration yields good performance. We present an approach that uses profiling to count accesses on each datastructure, estimates performance costs, recommends a configuration with better performance and applies these changes automatically in the same process without restarting it. It takes the access characteristics of the used call scenario in account, and the performance of different datastructures, varying with their current size.

Contents

1 Introduction

There are two basic components that heavily influence the performance, both in terms of runtime and memory consumption, of software systems: algorithms and datastructures. Choosing them well makes the difference between long running tasks that might run out of memory just seconds before a problem is solved and a fast, usable product. The end user only seldomly has the possibility to tune this parts, so he is more likely to abandon software with poor performance than trying to overcome this by using more powerful hardware. Designing algorithms that are able to work even with large input data is the simpler task, as the scope of interaction is contained and the interface can be simple. Changing an algorithm from an implementation with many recursive calls to an iteratively working one can directly lead to better performance for a single component. The same task is more different for datastructures, as they can be very long living and passed through numerous algorithms. Each algorithm can use different operations on datastructures, and as long as the different characteristics of each algorithm can not be anticipated, it is impossible to chose just a single datastructure that performs well in all cases.

For now, the programmer decides which combination of datastructures and algorithms solves a problem the "best" way. Thoughtful designed algorithms, with a well structured and simple interface, might get replaced with little work and without negative impact on the other components. Replacing a datastructure by another requires more work, as the interfaces are incompatible: for example, one has to call `put()` on a Java `HashMap` with a hash key and the element to store, `add()` on a `LinkedList`, and a pure array of elements even has no methods to be called. Even if all internal datastructures get wrapped by a common interface, on first sight, the choice of datastructures has to be done on compile time. And if performance problems occur, this usually leads to a long search for the potential bottleneck and manual changes, tailored to individual problems. To apply the changes, a running program needs to be stopped and restarted, which can be annoying on long running software.

We want to overcome this problem by an approach that combines profiling the usage of a running system, recommendations for improvements and dynamic application of a recommended configuration in a running program, removing the need of stopping a process.

This thesis is structured as followed: in Chapter 2, we present related work. In Chapter 3, common notations and terms are defined. In Chapter 4, we give a more thorough statement of the problem to be addressed by our contribution. In Chapter 5, we present our contribution on a high level view, while the implementation details are given in Chapter 6. An evaluation to justify the quality of our contribution is given in Chapter 7. The main part is finished with the summary and concluding words in Chapter 8 and topics for future work in Chapter 9.

2 Related work

Three different approaches for performance optimization, in terms of runtime performance ("How much time is spent to run a program?") or memory performance ("How much memory is consumed to run a program?"), can be distinguished: the usage of efficient datastructures that are specialized to be used in graph analysis, a passive approach where a running program is analyzed and problems are listed from a purely observing position, and an active approach where a program is analyzed and modifications are done in the same process.

2.1 Efficient datastructures for graph analysis

Some frameworks were developed especially to analyze large dynamic graphs. They should be able to hold the basic graph structure (lists of nodes and edges) and additional data like node or edge weights, labels, creation or modification timestamps. McColl et al. give an overview over different datastructures[15].

Bader, Ediger, et al. introduced STINGER[2, 7] in 2009 which is written in C . It covers the objectives of portability of algorithms and optimizations, productivity (researchers should focus on their own problems and not care about the underlying datastructures), and performance (both in terms of datastructures, that perform well under most conditions while not fitted to each single problem, and memory consumption, allowing parallel and sequential algorithms to be run). The datastructure contains fields for all possible applications, such as in- and out-degrees and weights for each edge. It should handle up to 3 million updates on the graph per second.

Bader and Madduri also develop Snap[1, 14], a framework for analyzing dynamic large-scale small-world graphs where the number of nodes and edges exceeds 100 million. The graph metrics are implemented to run parallel in multiple threads, thus it is on the one hand important to choose algorithms that can run in parallel (divide and conquer), and on the other hand implement them with a low locking level such that they can play out parallelity at the most. They optimize their framework for small-world graphs by exploiting their special characteristics (eg. power-law degree distribution) and do not mention whether it can also be used to analyze more general graphs.

Blandford et al. experiment with a compact representation for static graphs that should be at most 9 times faster and using between a third and a sixth less memory than adjecency lists due to better cache characteristics[4, 3]. Weights or other additional information can not be stored in their first implementation, and the graph needs to be seperable. Due to these characteristics, this approach is not suitable for a general graph storage system.

Finally, there are graph databases like neo4j or FlockDB that handle storing and accessing the graphs data like a usual DBMS by using a disk-based storage and not a memory-based. Thus, these systems can be much slower, but on the other hand, the graph data is without additional work persistently stored.

2.2 Passive analysis

Tools for analyzing running programs exist in many variations, mostly for analyzing memory leaks and deadlocks. In most cases, these tools provide information about classes that do not perform well, either in terms of runtime or memory performance. Examples are JVM Monitor or VisualVM, both for Java, which can hook into running programs through the Java Management Extensions (JMX) technology. Pin[13] (for arbitrary binaries on some Intel platforms) or JFluid[6] (for Java) are frameworks for instrumenting (augmenting) existing applications with additional code for analysis.

Brainy[9] uses a more sophisticated analysis. Prior to the execution of the real application, synthetic programs are run to measure the performance of individual operations on a datastructure on that very hardware. Machine learning algorithms are used to create a rule set for optimizations: if an operation is called more than n times, datastructure X is more efficient than datastructure Y (eg. replace a `LinkedList` with an `ArrayList` if there are many random accesses, which are more expensive in a linked list than in an array list).

Most passive approaches can be used broadly. Even if the time spent for writing either augmenting code or analyzing the output of an analyzer, tools and workflows can be generalized and used in more than one use case.

2.3 Active modification

Some frameworks were developed to enhance existing programs with the capability to change the internal datastructures while running the program.

Chameleon[18] provides a semantic profiler specialized on the usage of Java collections. It runs on top of the Java Virtual Machine and thus requires no changes in the profiled application. Accesses to collections are profiled and based

on a fixed set of rules recommendations for optimizations are given. Through wrapper objects, the optimizations can directly be taken into production.

Coco[19] works in a similar way, but requires changes in the source code. The programmer decides where to use the wrapper data types provided by the framework and a static compiler inlines the rule set into the byte code. Thus, the programmer can decide which fields may be optimized and which should not change their concrete type, and the resulting self-optimizing runtime version can be distributed without dependencies.

3 Preliminaries: common notations and terms

A graph $G = (N, E)$ is a pair of nodes N and edges E. The graph can be directed, such that edges are only traversable in one direction, or undirected. Each edge holds a pointer to the nodes it is connected to. In the directed case they are called source and destination node. Each node holds a unique index and a list of connected edges. In a directed node, this list of edges is split up into one list for incoming edges (where all edges' destination nodes are the current node) and one for outgoing edges. Additionally, a directed node holds a list of neighbor nodes.

n denotes the number of nodes within a graph, e the number of edges. $d(n)$ denotes the degree of a node (the number of adjacent edges), $\overline{d}$ the mean degree over all nodes.

Nodes or edges can be labeled with a weight, eg. to represent distances between nodes through an edge weight.

Changes in a graph are grouped together in batches. They contain a set of updates, which can be of six types: node addition, node removal, node weight update, edge addition, edge removal, and edge weight update.

A single analysis run consists of an initial graph and a set of batches that are successively applied on the graph. To avoid that random generations of graphs or batches mislead the analysis, a series executes multiple runs and aggregates the results of each run.

4 Problem statement

Datastructures have a large impact on the performance of software. Even if they provide the same functionality through a common interface, the internal structure is optimized for different use cases. For example, iterating over a linked list can be a very cheap operation as each element has a distinct pointer to the "next" element. Iterating over a pure tree-based structure is rather expensive, as it already needs several operations to find the "first" element. On the other hand, finding a specific element in a linked list can lead to a complete traversal over all elements, while this can be handled much faster in a sorted binary tree. The complexity notation eases a first comparison of datastructures: random access in a linked list has a complexity of $\mathcal{O}(n)$ (the upper runtime boundary grows by the number of elements stored in that datastructure), while the same operation can be achieved in $\mathcal{O}(\log n)$ in a binary tree.

Figure 4.1 shows a runtime comparison for `contains` operations. Two datastructures, one instance of `LinkedList` and one of `HashSet`, are filled with a number of integers. Afterwards, the runtime of a number of `contains` calls is measured. On a linked list, this operation has a complexity of $\mathcal{O}(n)$, on a hash set the cost is reduced to $\mathcal{O}(1)$ - the hash set provides a much faster lookup through the internal usage of hashes at the cost of a higher memory footprint.

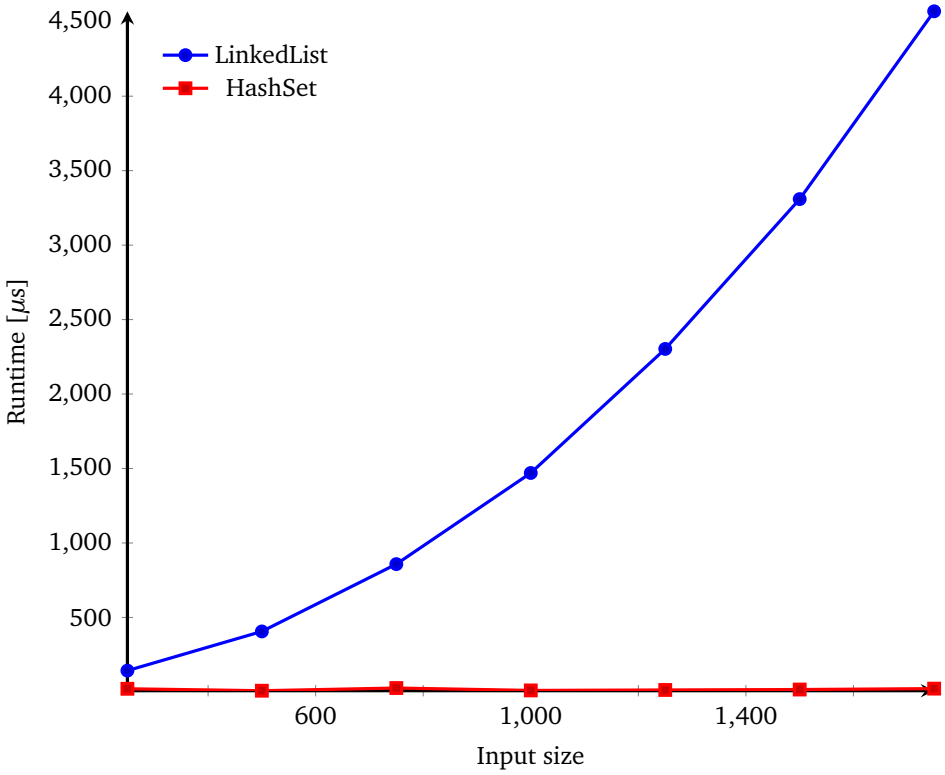

Figure 4.1.: Runtime comparison for a variable number of `contains` operations

But concentrating onto complexities can also be misleading: even if the complexities make you think that two datastructures handle the same number of operations in similar time, this is not true. Figure 4.2 shows a runtime comparison for the basic Java collections `ArrayList`, `LinkedList`, and `HashSet`, that all have a complexity of $\mathcal{O}(1)$ for insertions. Inserting an element into the hash set is up to four times slower than inserting it into one of the other datastructures, as here the hash of the element needs to be taken to store the element at its designated place. This is why we take a total of four cost indicators for performance into account: two terms of costs (runtime and memory consumption) combined with two ways of data gathering (based on complexities and based on benchmarks).

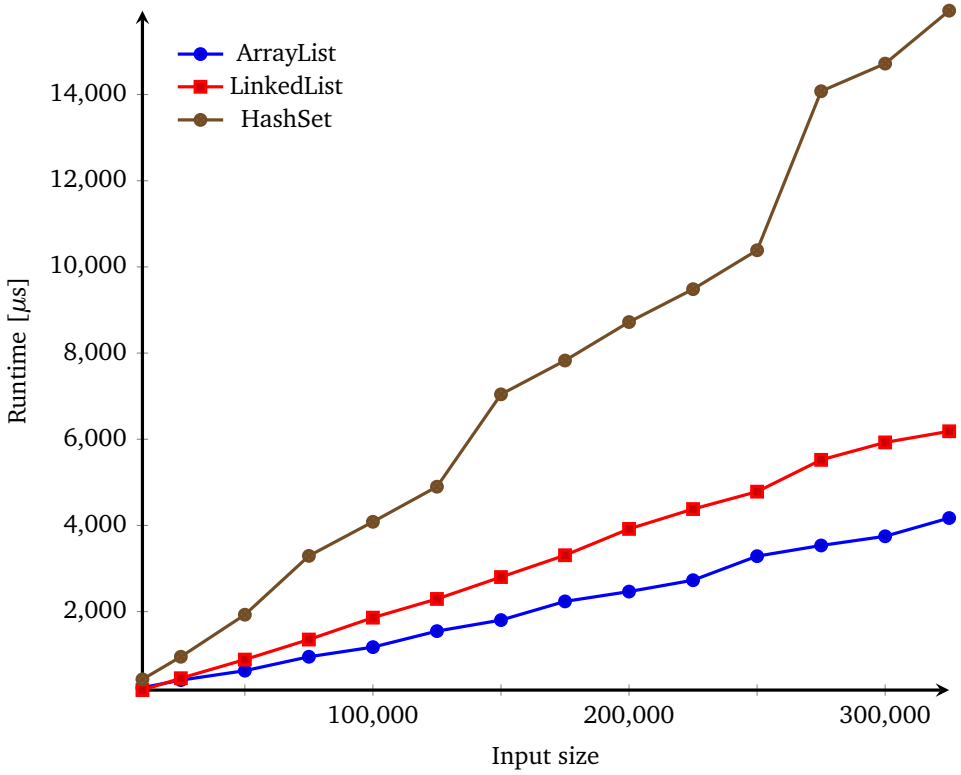

Figure 4.2.: Runtime comparison for adding a variable number of elements to an empty list

In these simple examples, it is already obvious that the choice of datastructures can heavily influence the runtime performance. The question which datastructure is the best to choose can be answered completely different from operation to operation, and this is even more the case if the choice should be done with a whole set of operations in mind: if there are ten `contains` calls and five elements are removed from the datastructure that currently holds 20 elements, which datastructure should be used? In DNA[17], a framework to analyze dynamic graphs, there are two global lists for edges and nodes and node-local lists for storing the connections to other nodes, so we have at most $2 + n$ instances of a datastructure in use. Using a datastructure framework like CoCo, that adds some fields to each datastructure instance to handle the optimization decisions, is not feasible if you want to keep a whole graph in memory.

Manual analysis, where a first run is done to measure the performance, receive a recommendation and change the source code, is also not feasible, as we want to be able to analyze graphs with a large number of initial nodes and edges and many updates. A single run can already take long time and it remains unclear whether the recommended optimization from one run is still a good choice for another run. On the other hand, the transparent optimizations of CoCo have the drawback that the optimizing part needs to be kept under control: if the underlying algorithm should be completely hidden from the main application, how can ubiquitous optimizations be avoided? The runtime saved from using a more efficient datastructure should not be spent completely for many optimization switches between datastructures.

5 Our approach

Our work is based on DNA[17], a framework for the graph-theoretic analysis of dynamic networks. It provides datastructures for holding graphs and their nodes and edges, generators for multiple graph structures (random graphs, scale-free graphs, clique graphs, ring graphs) and update types to simulate different change characteristics (adding and removing nodes and edges, or changing node and edge weights), and metrics to monitor various changes of characteristics (eg. degree distribution, shortest paths, connectivity).

Our work enables DNA to profile (count) the accesses to nodes' and edges' internal datastructures and make recommendations for optimizations. After having computed recommendations in terms of the four cost indicators mentioned in Chapter 4, they can be taken into account directly without the need to recompile the code or even restart a running analysis.

The contribution is split up into five components that are described from a high level view in this Section: the graph datastructure used to store the graph and its parts in several datastructures, the benchmarking process to gather system-dependent performance data, the profiler to create access statistics, the recommender that estimates costs for different datastructure combination and recommends changes to gain performance improvements by comparing the costs, and the hot swap algorithm that performs the recommended changes in the running process. In Figure 5.1, we give a rough overview over the connection between these components.

The Chapter finishes with an example for the data flow of our contribution. It helps to understand how the cost estimation works.

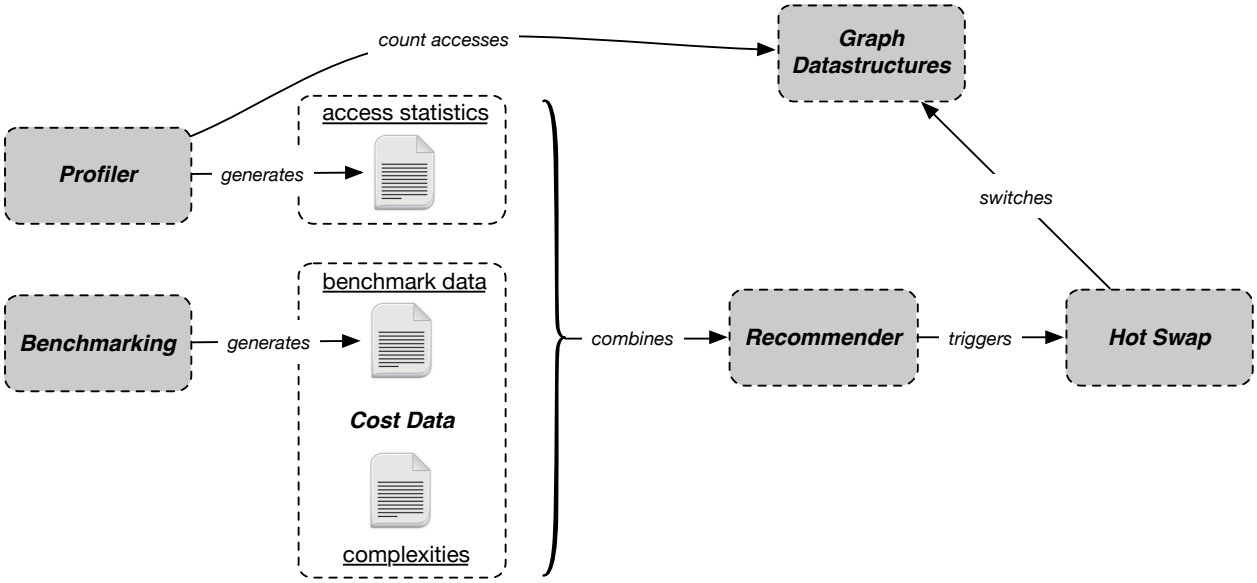

Figure 5.1.: Structural overview over our contribution
(graphic by Benjamin Schiller)

5.1 Graph datastructure

In this Section, we discuss the basic structure of the datastructures that we use in DNA to store nodes and edges, and the meta data that is needed for the cost estimations.

5.1.1 Definition of commonalities of the used datastructures

A central component of our work are the underlying datastructures that store the graph structure in adjacency lists. To be able to use different datastructures for the same purpose - storing nodes or edges of a graph - and exchanging them not only through stopping and changing the program, a common set of access types needs to be defined.

In general, there are two groups of access types: accesses for updating a graph (eg. adding nodes or edges, removing them, or changing node or edge weights) and reading accesses for the analysis (eg. checking whether one node is connected to another, or requesting the degree of each node).

The classes representing graphs, nodes, and edges contain methods that cover these access types and defer the data access to the underlying datastructures. Defining a common interface of methods for these datastructures avoids too strict coupling to the classes that use them: these do not instantiate datastructures directly, but request a factory to return a new datastructure instance.

We identify the following set of general access types which all datastructures should provide:

- `init` to initialize the datastructure once
- `add` to add elements to the datastructure
- `contains` to check whether an element is present in the datastructure
- `get` to receive a specific element
- `getRandom` to receive a random element
- `remove` to remove an element from the datastructure
- `size` to get the number of currently present elements
- `iterator` to receive an iterator over all elements

For the access types `contains`, `get`, and `remove`, we furthermore distinguish between a successful request (returning true resp. the requested element) and a failed request, as the costs can differ strongly. For example, finding an element in an array through iteration takes (in the average case) less time if the element is present, compared to finding a non-existing element.

5.1.2 Distinguishing specific list types

In a graph, different types of lists occur, as introduced in Chapter 3:

- A global node list for storing nodes on a graph-global scope
- A global edge list for storing edges on a graph-global scope
- A local node list for storing nodes on a node-local scope
- A local edge list for storing all edges on a node-local scope
- A local edge list for storing incoming edges on a node-local scope
- A local edge list for storing outgoing edges on a node-local scope

For each list type, a different datastructure may be used. Thus, they can easily adopt to the specific access characteristics. For example, if the set of nodes within a graph is read much more often than it is changed, it is obviously efficient to use a datastructure that is optimized for read access - while using the same datastructure for a frequently changing edge list in the same graph might not lead to a good performance.

5.1.3 Meta data for the datastructures

The recommender needs performance data for each datastructure to compute the occuring costs, in four different type of costs: for both runtime and memory consumption, there needs to be a fixed set of costs in terms of complexities and a system-specific set of performance measurements, computed through the benchmarking tool. After all, one entry for each combination of performance cost, datastructure, and access type must be present. In some cases, the access type needs one more distinction between the stored element type (node or edge). On the other hand, a fallback mechanism helps to avoid redundant entries. An example is shown in Listing 5.1, where all memory complexity costs except those for the access type `INIT` default to 0.

```
RUNTIMECOMPLEXITY_DARRAY_INIT = 1 Static
RUNTIMECOMPLEXITY_DARRAY_ADD_NODE = 1 Static
RUNTIMECOMPLEXITY_DARRAY_ADD_EDGE = 1 Linear
RUNTIMECOMPLEXITY_DARRAY_CONTAINSSUCCESS_NODE = 1 Static
RUNTIMECOMPLEXITY_DARRAY_CONTAINSSUCCESS_EDGE = 1 Linear

MEMORYCOMPLEXITY_DARRAY = 0
MEMORYCOMPLEXITY_DARRAY_INIT = 1 Linear

RUNTIMEBENCHMARK_DARRAY_ADD_NODE = 50=1161.312,1119.687,1200.125,1194.062,1153.062;
    100=701.757,676.333,779.424,746.545,797.212;
    500=1042.22,813.54,1111.3,1151.6,1166.46;
    1000=1815.28,1384.44,1993.8,1978.06,1740.24;
    2000=3581.8,2770.64,4125.96,3247.1,4504.38
```

Listing 5.1: Excerpt from an example file containing costs for `DArray`

5.2 Benchmarking

A benchmarking tool for the system on which the analysis should later run on is the first component of our work. It provides system-specific performance data for all access types, which allows the recommendation step to use real-life costs and not only abstract complexities. This drastically improves the recommendation quality.

For each combination of datastructures, predefined list sizes, and access types, the tool measures the average runtime and memory consumption of calls with randomized input data. This is done in seperated steps, and not in combinations of different access types or datastructures within one measurement. It is important to perform these measurements with different list sizes, as already stated in chapter 4: the runtime performance of the `contains` operation lies in $\mathcal{O}(n)$ for a `LinkedList`, so taking a benchmark for random `contains` calls on a list containing 1,000 elements yields completely different results than on a list containing 1,000,000 elements. Repeating each single benchmark action can level out inacurracies, eg. due to other running processes.

The results are written out to files that can be used by the other components. Results are not aggregated: each and every performance data that is gathered is written to the files. On the one hand, this allows a more flexible access as all computed data is available for detailed analysis later on. Writing analyses that can distinguish the best from the worst case is not possible if the data is directly aggregated. Additionally, this enables us to extend benchmarking results through additional runs later, as new results just have to be appended.

As a complete benchmark run can take several hours, a default data set is provided. Creating an own data set can be helpful if the predefined list of input sizes is too inaccurate for own purposes. The recommender can produce more precise results with a fine-grained set of input sizes, but due to the time it takes to create the set and the storage the results need, the default set uses a generalized range.

It is necessary to create an own complete data set; mixing own results with the default set leads to unusable recommendations. Consider the case where a usually good performing datastructure is added, and benchmarks for it are run on a machine that is slower than the machine the other benchmarks were run on. This leads to high cost estimations for this new datastructure, while other datastructures would also have yielded higher costs on this machine.

5.3 Profiler

To provide recommendations that are based on current access statistics, we need to assemble them from the running program. The profiler obtains this data by counting each access to the datastructures, seperated into access type and list type. Furthermore, the profiler should make a distinction between different accessors to the lists: this helps to not only compute aggregated access statistic for the whole running program, but also for subparts like metrics and updates. Different implementations of the same metric can be compared this way, seen from the datastructure's point of view.

Integrating the profiler into the existing code should be as easy and automated as possible. It is not feasible to add the profiling sections into the main parts (graph, batches, metrics,...) or into the datastructures without inaccuracies. A common place to handle this is desired to minimize the efforts to maintain the profiler and the other parts - neither for introducing new metric nor for new datastructures any profiling parts should be written. Secondly, this minimizes the risk of leaving out accesses from the profiling - a recommendation can only be as accurate as possible if all accesses are taken into account.

5.4 Recommender

The core our our work is the recommendation algorithm. It merges the raw costs for each access with the counted accesses and arrange a list of datastructure combinations and assigned costs. By sorting the list by costs, we can find the combinations with the best overall performance. Afterwards, these results can be used by the hot swap algorithm, and they are written to files for manual analysis.

5.4.1 Calculation of result sets

The standard input for the recommender is a list of access calls, collected by the profiler. For each combination of datastructures, the costs are calculated as the multiplication of the number of accesses by the costs of these accesses for the used datastructure. The sum of these per-access-costs yields an estimation for the total costs that occur for this datastructure combination under the given accesses.

5.4.2 Selecting the proper performance data values

The benchmark results are taken for a set of input sizes, as it is not feasible to run benchmarks on all sizes - it would not only take long time, but also much disk space. Additionally, it lowers the accuracy further due to other processes interfering with the benchmarking and inaccuracies of the benchmarking itself.

The collected data contains a list of values for each combination of input size and action. An example can be seen in Listing 5.1: for the datastructure `DArray` and the access type `ADD_NODE`, multiple measurements of runtime costs for different datastructure sizes are given. To be able to merge this raw data with the number of counted accesses, this data can be accessed in two dimensions: over a list aggregator and a bucket selector.

The list aggregator consumes all values from a single list and condenses them into one value. This can either be the minimum value, the maximum value, or the average over the whole list. The resulting values are put into buckets of a map, using the input sizes they belong to as the key. From the example in Listing 5.1, the values in Table 5.1 are aggregated through the three different aggregators.

List size	Minimum value	Maximum value	Average value
50	1119,687	1200,125	1165,65
100	676,333	797,212	740,25
500	813,54	1166,46	1057,02
1000	1384,44	1993,8	1782,36
2000	2770,64	4504,38	3645,98

Table 5.1.: Aggregated costs for the datastructure `DArray` and the access type `ADD_NODE`, processed from the data in Listing 5.1

When the costs for a single call need to be accessed, the bucket selector is used. A single value is addressed not only through the datastructure and access type, but also through the current mean size of all the datastructures of that list type that are just in use. The bucket selector can now chose to return the value stored in the bucket with the next lower key, with the next upper key, or interpolate the values stored in these two buckets. Figure 5.2 shows a schematic comparision of the different selectors.

With these two parameters, the recommender can be configured to do more of a “worst case estimation” (by using the maximum value for each bucket and selecting them through the next upper key) or an optimistic estimation (by using the minimum value for each bucket and the lower bound selector). From the data in Table 5.1, we can extract the following example costs:

- Using the list aggregator for minimum values and the lower bound bucket selector, the costs for a single access on a datastructure with 50 to 99 elements are estimated with 1119,687.
- Using the same list aggregator, but the upper bound bucket selector, the costs for a single access on a datastructure with 51 to 100 elements are estimated with 676,333.
- Using the same list aggregator, but interpolating values yields size-dependent results: for a current size of 50 elements, the costs are estimated with 1119,687. For a current size of 100 elements, the costs are estimated with 676,333. For a current size of 75 elements, costs are estimated with 898,01 using linear interpolation.

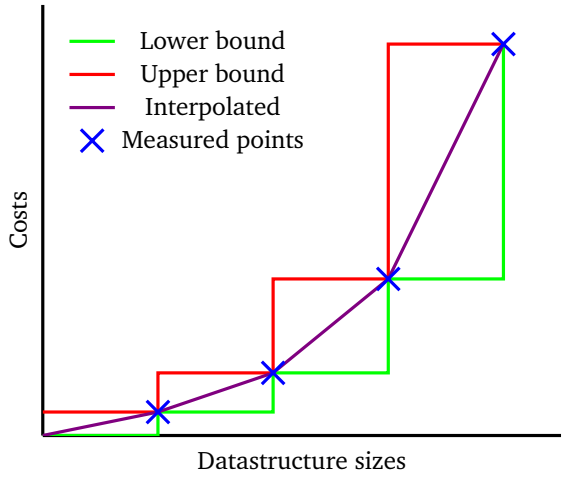

Figure 5.2.: Comparison of bucket selectors for distinctly collected values

5.4.3 Comparison of cost values

Sorting the two lists of costs based on benchmark data is easy, as each cost value consists of a single numerical value. If for one combination costs of $1,000$ are estimated and $1,500$ for another combination, it is obvious that the first one performs better (independent from the category or units - comparing costs from different categories is obviously dangerous, and within the same category, the unit is redundant).

Sorting the complexity costs is a little more sophisticated. We combine two complexity cost values by comparing their parts. Internally, we prioritize the parts in the following manner: static costs (which are independent from the current size) are the cheapest, followed by costs dependent on the degree, followed by costs dependent on the number of nodes. Costs dependent on the number of edges are considered to be the most expensive. Sorting two values is done by descending part costs, taking their factors into account. If the factor is equal, the comparison continues with the next less expensive part. If one entry has no factor assigned for one part, the other one is considered to be more expensive. If the factors are different, the entry with the smaller factor is considered to be cheaper directly.

This approach enables a very simple comparison without the need of expensively condensing the raw costs to a single numerical value that could be compared with less effort. But this also leads to the drawback of some inaccuracies. Some examples:

- $10\mathcal{O}(E) < 15\mathcal{O}(E)$, which is obviously correct
- $10\mathcal{O}(N) + 10\mathcal{O}(E) < 10\mathcal{O}(N) + 15\mathcal{O}(E)$, which is still obviously correct
- $10\mathcal{O}(N) < 10\mathcal{O}(E)$, which makes sense for the case that $n < e$. This is the case for most of the graphs in our analysis.
- By the definition of our comparison algorithm: $100\mathcal{O}(N) < 1\mathcal{O}(E)$, which is where this comparison gets inaccurate.

5.4.4 Combination of result sets

Combining recommendations from two ore more basic cost categories is also possible. While it makes no sense to add their arbitrary results (a recommendation under the terms of runtime complexity can yield a result of $100 * \mathcal{O}(1)$, while a recommendation using the runtime benchmark data can yield a result of $1,127,975.80$), we combine this data based on the positions in the sorted recommendation lists. Through assigning weights W_n to each source category, we can prioritize one category over another. The total cost value is in this case computed as

$$C_c = \sum W_n * position$$

For example, if a combination is at position 2 in all recommendation lists to be combined, it might get a cost value of 2. Putting it at first position in one list that a weight of 5 is assigned to, but on the 10th place in a second list with weight 2 results in a cost value of $5 * 1 + 2 * 10 = 25$. Through such a combination, there is no need to normalize or think about other mathematical ways of combining the raw values.

5.5 Hot swap

Finding candidates of datastructure combinations that could replace the currently used ones to improve the performance of DNA is only the first basic step. Exchanging the datastructures already in the running process lowers the need of configuring datastructures even further. After the recommendations are computed, the hot swap algorithm is able to access the sorted list of recommendations for each cost type. Using one of the candidates in this list, the swap itself is very simple: all datastructure types that are to be changed are initialized and all elements are moved from the old datastructures to the new ones.

But exchanging the datastructures at each potential position is not efficient. Two mechanisms help to find a good recommendation to replace the current datastructures and ensure that these changes do not cost more than they save. The first step is to collect recommendations over multiple batches and put them in a sliding window, such that entries from former batches get overwritten after some time. This allows to monitor whether a recommendation is just given in one batch due to its individual access characteristics, or if it is given multiple times in a row which gives more confidence about it being a good long-term replacement. If a threshold (eg. in terms of "recommendation was present in the last five of six batches") is exceeded, the recommendation is examined further.

Taking not only the recommendation with the least cost from the sorted list, but the first two or three recommendations into account deals with the problem that some combinations can lead to very similar costs. If these recommendations would be on the first positions in the sorted list, but swap the positions among themselves, a single recommendation can not exceed the given threshold and is not considered for the swap.

After having collected a set of recommendations that occured multiple times, they have to pass a second check: a candidate to be used should be efficient over the next batches and amortize the swapping costs. To check this, the access data from some last batches is aggregated. It is used to extrapolate the accesses over the next batches. Costs for theses accesses are computed, both for the currently used datastructure combination and the proposed candidate. Combined with the costs of executing the swap, the candidate must yield better results than the current combination. Only if this is the case, the swap is executed; otherwise, the next candidate is checked. If no candidate passes this quality check, the current combination is retained.

5.6 Example data flow

To make the whole concept of our contribution more obvious, we show a small example of the data flow here. Our examples use the list type L_1, the access types A_1 and A_2 and the datastructures D_1 and D_2.

The first step is to define, either through complexity analyis or benchmarking, costs for each access type on datastructure. We give benchmark results here, as they can not only be used to show the general data flow through all parts, but also to show the differences of the list aggregators and bucket selectors. The costs are given in Table 5.2. The access costs are aggregated before computing costs and are given in Table 5.3.

		Datastructure D_1	Datastructure D_2
Access type A_1	10 elements	1, 2, 2	10, 15, 11
	20 elements	10, 16, 24	20, 22, 21
Access type A_2	10 elements	5, 2, 9	2, 10, 9
	20 elements	5, 4, 5	5, 2, 5

Table 5.2.: Cost matrix for the example datastructures and access types

		First scenario (min values)		Second scenario (max values)	
		DS D_1	DS D_2	DS D_1	DS D_2
Access type A_1	10 elements	1	10	2	15
	20 elements	10	20	24	22
Access type A_2	10 elements	2	2	9	10
	20 elements	4	2	5	5

Table 5.3.: Concrete costs for each access in both scenarios

In two scenarios, we assume to have counted 10 accesses for access type A_1 and 5 accesses for access type A_2, and represent these numbers with $a_1 = 10$ and $a_2 = 5$. The number of accesses are combined with the costs for each access (dependending not only on the datastructure, but also on the current size) to compute the aggregated costs that occur for using each datastructure.

In the first scenario, we want to give an optimistic cost estimation by using the minimum values from each cost list and the lower bound bucket selector. The datastructure has a current size of 15 elements, so we use the values from the buckets for 10 elements. The costs are computed as followed:

$$
\begin{aligned}
Costs(D_n) &= \sum Costs(D_n, A_m, Size) * a_m \\
Costs(D_1) &= Costs(D_1, A_1, 15) * a_1 + Costs(D_1, A_2, 15) * a_2 \\
&= 1 * 10 + 2 * 5 = 20 \\
Costs(D_2) &= Costs(D_2, A_1, 15) * a_1 + Costs(D_2, A_2, 15) * a_2 \\
&= 10 * 10 + 2 * 5 = 110
\end{aligned}
$$

The obvious recommendation would be to use datastructure D_1 in this case.

The second scenario uses a pessimistic cost estimation, using the maximum values from each cost list and selecting the bucket by the upper bound. The costs are computed:

$$
\begin{aligned}
Costs(D_n) &= \sum Costs(D_n, A_m, Size) * a_m \\
Costs(D_1) &= Costs(D_1, A_1, 15) * a_1 + Costs(D_1, A_2, 15) * a_2 \\
&= 24 * 10 + 5 * 5 = 265 \\
Costs(D_2) &= Costs(D_2, A_1, 15) * a_1 + Costs(D_2, A_2, 15) * a_2 \\
&= 22 * 10 + 5 * 5 = 245
\end{aligned}
$$

Using these selection parameters, datastructure D_2 is recommended.

6 Implementation details

While we covered the five parts of our contribution from a high level view presenting how the whole process works in the previous Chapter, we add some details of the implementation to each part in this Chapter. It covers implementation choices and details, and ways of configuring the parts.

6.1 Graph datastructure

In this Section, we cover the implentation details of the graph datastructure, which consists of the datastructures to hold nodes and edges in adjacency lists, the meta data in terms of performance costs, and the factory to instantiate new objects.

6.1.1 Interfaces and abstract classes

As already mentioned in section 5.1.1, the basic functionality to store elements is defined through common interfaces:

- The interface `IElement` and the abstract class `Element` define common functionality for all following elements that can be stored in the graph datastructure, like nodes (represented through the interface `INode`, the abstract class `Node`, and concrete implementations for undirected and directed nodes and their extension through weights) and edges (with a similar structure like nodes).

- The interface `IDataStructure` defines the method signatures all datastructures must implement. This covers initializing the datastructure with a data type to be stored within and an initial size, re-initializing a datastructure with a new size, add, remove and contains methods for `IElement`, requesting the current size, and three housekeeping methods for checking whether a datastructure is able to store elements of a given type, requesting the data type that can be stored, and preparing that datastructure's internal datastructure to be garbage collected.

  Using a size parameter can speed up filling a datastructure with new elements, as multiple succeeding resizements are avoided. After doing a hot-swap, there are usually lots of datastructures that can be deleted through the JVM's garbage collector if they are properly dereferenced. This is handled through the preparation method.

- The interface `IReadable` extends the interface `IDataStructure` with methods for a random access onto elements, and an iterator over the stored elements. Keeping this seperated from the basic interfaces allow the integration of compressed datastructures that do not store pointers, but only condensed information for each element. An example is a bloom filter[5] that uses only one bit for holding each element.

  Both `IDataStructure` and `IReadable` define their methods to work on any instance of `IElement`, independent from concrete node or edge types.

- The interface `IEdgeListDatastructure` extends `IDataStructure` with methods for adding and removing edges, and a method to check whether a specific edge is present in the datastructure.

- The interface `INodeListDatastructure` extends `IDataStructure` with methods for adding and removing nodes, and a method to check whether a specific node is present in the datastructure. Additionally, it defines a signature for asking the index of the node with the highest node index currently present in the datastructure.

- The interfaces `IEdgeListDatastructureReadable` and `INodeListDatastructureReadable` extend the related interfaces with methods for random access to edges resp. nodes.

Having distinct interfaces for node and edge lists has two reasons: Not only does the handling differ (eg. nodes are stored in an array at their node index' position, while edges do not have such unique identifiers and are stored at the next free position in an array), this also allows using a datastructure for storing only one of the two kinds of elements. In Figure 6.1, a class diagram of the interfaces is shown to visualize the available methods and connections.

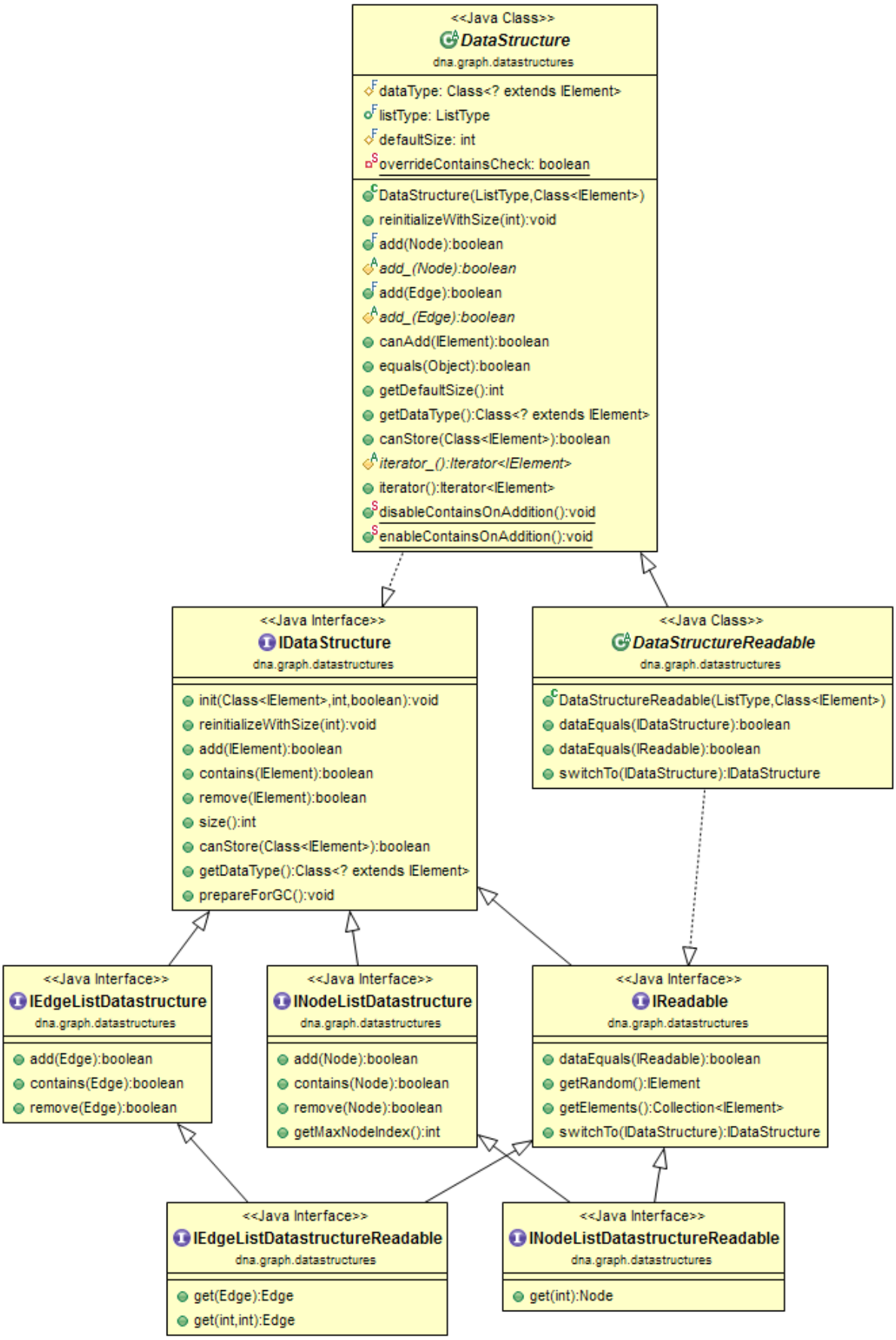

Figure 6.1.: Class diagram of the interfaces for the datastructures, created using the ObjectAid UML Explorer

6.1.2 Concrete datastructures

As an initial set of datastructures, we provide adapter classes for some datastructures that are shipped with the standard Java runtime:

- `ArrayDeque`, wrapped in `DArrayDeque`
- `ArrayList`, wrapped in `DArrayList`
- `HashMap`, wrapped in `DHashMap`
- `HashSet`, wrapped in `DHashSet`
- `HashTable`, wrapped in `DHashTable`
- `LinkedList`, wrapped in `DLinkedList`

The adapter classes do not try to influence the logic of the tethered classes, but leave this to their own responsibility. For example, a `HashMap` is resized if the number of elements exceeds an internal threshold. This is neither controlled nor monitored through the adapter classes.

Additionally, we provide three simple datastructures:

`DArray` provides a plain interface to a Java array. While the usual `ArrayList` also uses an array internally, `DArray` does not use any anticipatory logic. If an element is added to an array list, the capacity is checked. If there is no more room to store another element, the list is resized by an arbitrary factor. Thus, resizing the array is not necessary on each addition. On the other hand, this implementation always reserves some more free slots than currently needed. Our own array class handles this different and resizes the array just as needed for each new element that should be stored.

`DHashArrayList` is based on CoCo[19]. It uses both a hash set and an array list internally. On adding and removing elements, both datastructures need to be changed. For other actions, the datastructure that provides better performance is used, for example the hash set for `contains` operations or the array list for random access on `get`.

`DEmpty` serves a special purpose: datastructure types that are not necessary can be exchanged to use this functionless datastructure. It provides all methods needed for node and edge storage, but does not store anything internally. This allows all actions to run in very little time, consuming very little memory. On the other hand, it provides static results on all operations, so it must be used with caution.

A common problem in hash-based datastructures like `HashMap`, `HashSet`, `HashTable`, and our own implementation `DHashArrayList` is the usage of integer-based hash keys. For each distinct key, at most one object can be stored in these datastructures. Java integers are stored with 32 bits, so in theory, these datastructures could hold up to $2^{32} = 4,294,967,296$ elements before a hash collision occurs (in practice, this only holds for an optimally used key space). For edges, we use a hash code consisting of their node's indices:

$$hashcode = n1.index * 2^{16} + n2.index \tag{6.1}$$

Thus, a collision can already happen in a strongly connected graph with $2^{16} = 65,536$ nodes. In real life, collisions can occur much earlier, which puts the datastructure under danger: if one element with a hash code $\mathscr{A}$ is already present in that datastructure and a second, different element with the same hash code is to be added, what should be done?[1]

The concept of multimaps gives one solution to this problem: for each common hash code, multiple distinct objects can be stored in such a datastructure. As two more additional datastructures, we implemented adapter class to `LinkedHashMultimap` and `DHashMultimap` which are part of the Guava library[8]. For the key set, the first uses a `LinkedHashMap` internally, the second a `HashMap`. In contrast to the usual usage of these maps, where each key points directly to a stored object, a multimap maps each key to a collection of objects, in this case to instances of a `LinkedHashSet` resp. a `HashSet`. The elements are stored in these nested sets.

[1] Even if this looks like a problem that won't happen soon on the first view, consider that it is the task of a hash code to condense lots of information to a smaller set of possible values. The text-book definition of a hash code tells you that clusters of almost identical keys should be broken up[11, p 514], but Java does a really bad job here, as both the strings "BB" and "Aa" have the same hash code 2112

6.1.3 Meta data for the graph datastructure

In Section 5.1.2, we introduced the six different list types that occur in DNA. To ease the work with these lists, we implemented a fallback mechanism. For two basic list types, the `GlobalNodeList` and `GlobalEdgeList`, the datastructure to use must be given, for the remaining, the datastructure can be infered. If no datastructure for the `LocalNodeList` is chosen, the type defined for the `GlobalNodeList` is used. In the same manner, the datastructure for `LocalEdgeList` default to the one for `GlobalEdgeList`; and datastructures for `LocalInEdgeList` and `LocalOutEdgeList` default (transitively) to the one for `LocalEdgeList`.

As a second additional information, each list type holds an indicator to its growth characteristic. This is modeled with the three types `Degree` (denoting that a datastructures average size is proportionally growing with the average node degree), `NodeSize`, and `EdgeSize` (denoting that a datastructures average size is proportionally growing with the number of nodes resp. edges). This is a mandatory information for the recommender: this step needs to differentiate the complexity costs by this fact - it makes a huge difference whether an algorithm on a large graph runs in $\mathcal{O}(n)$ or $\mathcal{O}(d)$ (whether the runtime complexity scales with the number of edges or the average degree).

6.1.4 Performance cost data

Four basic types of performance costs data are introduced in section 5.1.3: runtime complexity costs, memory complexity costs, runtime benchmark costs, and memory benchmark costs. Their raw data is stored in files, for which an example can be found in Figure 5.1 on page 12. Each line consists of a key and an assigned value. The key is built from four categories: the performance data type, the name of the datastructure, the access type, and optionally the stored data type.

A fallback mechanism helps to keep the data maintainable: data is always requested by all four categories. If there is no cost entry present using all keys, the last key is truncated and the search is repeated until only the first two keys (performance data type and name of datastructure) are left. If this leads to no results, the complete search request is repeated in the default section, where each key is prepended by the keyword `DEFAULT`. The default section enables the co-existence of two sets of benchmark costs - one we ship based on a widespread benchmarking run on one of our servers, and an individual set that can be created on a different machine - without having to overwrite any files.

The value obtainable under the keys holds the assigned cost data in two formats: complexity costs are specified using their growth type (`static` for size-independent costs and `linear` for linearly growing costs), which is combined with the growth-characteristic of the assigned list type to calculate the costs per access. The benchmarking costs are represented in a more complex format: this data is available for varying sizes, and for each size, a set of benchmarking results is given.

6.1.5 Factory for graph datastructure instances

Java is not ready for exchanging datastructures on runtime-level. Being able to instanciate classes only through their class name and not through calling their constructor is possible with reflection. The factory for constructing new instances of datastructures strongly builds on reflection.

From the view of the programmer, a graph datastructure is instanciated with a list of datastructures to be used and the names of the node and edge classes to be used. In places where a datastructure is needed (eg. on instantiating a new graph that is in need of a global edge list), it is not directly constructed but the graph datastructure class is requested to do this and return it by calling the method `GraphDataStructure.newList(ListType)` with the list type as an argument. Internally, the datastructure type to be used is pointed out using the fallback mechanism and a new datastructure is instantiated using reflection. The initial size is estimated from the most recent graph generation: initial sizes for both global lists are known from the generation, initial sizes for the local lists are estimated as $\overline{d} = \frac{e}{n}$. These sizes can also be overwritten, if for example the graph is initialized through a graph generator with only few elements, but it is known to grow fast and resizing should be avoided.

Listing 6.1 gives an example for the instantiation of the graph datastructure, including overwriting the default size that is assumed for the global node list, and first elements of the graph from the outer view (the graph itself is instantiated with a name, the initial timestamp, and the information that it will hold 50 nodes and 200 edges after the initial generation has finished). Listing 6.2 gives an example about the initialization of datastructures for a graph object.

```
EnumMap<ListType, Class<? extends IDataStructure>> listTypes =
  GraphDataStructure.getList(
    ListType.GlobalNodeList, DHashMap.class,
    ListType.GlobalEdgeList, DHashMap.class );
```

```
GraphDataStructure gds = new GraphDataStructure(listTypes,
  UndirectedNode.class, UndirectedEdge.class);
gds.overrideDefaultListSize(ListType.GlobalNodeList, 100);

Graph g = gds.newGraphInstance("Undirected graph", 0, 50, 200);
Node node1 = gds.newNodeInstance(1);
Node node2 = gds.newNodeInstance(2);
Edge e = gds.newEdgeInstance(node1, node2);

g.addNode(node1);
g.addNode(node2);
g.addEdge(e);
```

Listing 6.1: Sample code for instantiating a new graph datastructure

```
public Graph(String name, long timestamp, GraphDataStructure gds) {
  this.name = name;
  this.timestamp = timestamp;
  this.nodeDatastructure =
      (INodeListDatastructure) gds.newList(ListType.GlobalNodeList);
  this.edgeDatastructure =
      (IEdgeListDatastructure) gds.newList(ListType.GlobalEdgeList);
  this.gds = gds;
}
```

Listing 6.2: Sample code for instantiating lists for a new graph

6.2 Benchmarking

To run benchmarks, we use Perfidix[12]. It allows running tasks repeatedly (to level out random inaccurricies) and defining different meters to measure different aspect. For our work, runtime costs and memory consumption is measured. One benchmarking process is run for each datastructure. Thus, we can run either a complete suite of benchmarkings over all datastructures, or extend an existing set with more detailed results or results for a newly implemented datastructure.

In the class `BenchmarkingExperiments`, each combination of access type and used graph element is covered in a seperate method. It is kept seperated from the datastructure to be benchmarked to ensure that for each datastructure a completely equal set of operations is executed and new datastructures can be implemented without changing the benchmarking code.

Each benchmark method needs to operate on datastructures with different sizes. Initially, a default set of nodes and edges is generated to be used later on, and for each and every call of a benchmark method, a clean datastructure with the needed size is generated to avoid side-effects. In addition to the general set of nodes and edges to be used, lists with randomly chosen nodes and edges that are sure to be, or sure not to be, in the used datastructures are generated in advance. This avoids the selection within the benchmarking method, which would weaken the results.

To be able to run benchmark methods with different input parameters (in our case: the datastructure and the current size), we extended Perfidix with the ability to run parameterized. The benchmark executor uses reflection to collect all methods annotated with `@Bench` and execute them repeatedly. Our contribution to Perfidix allows calling a data provider, a method within the same class that computes a set of input parameters. Listing 6.3 shows the complete setup for the data generation for the benchmarks, and Listing 6.4 shows an example benchmarking method.

All benchmarking methods are called repeatedly for each tuple within this set. Once more, this helps avoiding duplicated code and even allows to configure the benchmarks to be done on runtime level and not on compile-time level. Configuration for the benchmarks is done through the class `BenchmarkingConf` which reads the input sizes and the number of repetitions from the configuration. The number of repetitions is raised by three to skip outliers afterwards when writing the results to the data files.

Within each benchmarking method, the internally used access method is called several times (usually 50 times; less for smaller datastructure sizes, as it for example makes no sense to remove 50 elements from a datastructure that holds only 10 elements), once more to enhance accuracy: running it only once can trigger internal actions that are only executed from time to time, or on the other hand leaving these actions completely out of scope. Repeating calls can take this into account for a comprehensive benchmark result.

The benchmark method for the `Iterator` access type contains a special case. In this scenario, it is not sufficient to benchmark the execution of the iterator initialization only. In the usual run scenario, the iterator is also used to access the elements within a datastructure. As this is not representable through a distinct access type[2], we decided to iterate over all elements in the benchmark method and measure the costs for this action.

```
// classToBenchmark holds a pointer to the currently
// benchmarked datastructure class
static Class<? extends IDataStructure> classToBenchmark = DArray.class

// inputSizes are read from the configuration to define
// the size to run the benchmarking with
static int inputSizes = new int[]{ 50, 100, 200 };

// operationSize holds the number of executions to be run
// for each benchmark, dependent from the inputSize
int operationSize;

// Simple arrays to hold a general set of nodes and edges
INode[] nodeList;
IEdge[] edgeList;

// Arrays to hold node indices and nodes that are surely (not)
// stored in a datastructure
Integer[] randomIDsInList;
Integer[] randomIDsNotInList;
INode[] randomNodesNotInList;

// Arrays to hold edges that are surely (not) stored in
// a datastructure
Edge[] randomEdgesInList;
Edge[] randomEdgesNotInList;

// Data generation before the execution of the
// very first benchmarking method
@BeforeFirstRun
public void setupGeneralLists() {
  EnumMap<ListType, Class<? extends IDataStructure>> list = GraphDataStructure.
    getList(ListType.GlobalEdgeList, DArray.class,
      ListType.GlobalNodeList, DArray.class);
  GraphDataStructure gds = new GraphDataStructure(list,
        DirectedNode.class, DirectedEdge.class);

  Node node, formerNode;
  Edge edge;

  node = gds.newNodeInstance(0);
  nodeList[0] = node;

  for (int i = 0; i < maxListSize; i++) {
    formerNode = node;
    node = gds.newNodeInstance(i + 1);
    nodeList[i + 1] = node;

    edge = gds.newEdgeInstance(formerNode, node);
    edgeList[i] = edge;
  }
}
```

[2] For this case, it would be necessary to monitor calls to `Iterator.next()` which AspectJ does not support in a simple way

```
// Generation of the testInput in form of a matrix of classes and
// input sizes
public static Object[][] testInput() {
  Object[][] inputSet = new Object[inputSizes.length][2];
  int counter = 0;
  for (int inputSize : inputSizes) {
    inputSet[counter] = new Object[] { classToBenchmark, inputSize };
    counter++;
  }
  return inputSet;
}

// Initialization of the graph datastructure
public void setUpGds(Class<? extends IDataStructure> datastructureClass,
  Integer setupSize) {
    EnumMap<ListType, Class<? extends IDataStructure>> list = GraphDataStructure.
      getList(ListType.GlobalEdgeList, datastructureClass,
        ListType.GlobalNodeList, datastructureClass);
    gds = new GraphDataStructure(list, DirectedNode.class,
      DirectedEdge.class);
    this.operationSize = config.getOperationSize(setupSize);
}

// Initialization of an edge list to be used for the benchmark method,
// also containing data for the edges that surely are stored and
// surely are not
private void initForEdgeList(Class<? extends IDataStructure> datastructureClass,
  int initialSize) {
    if (edgeListToBenchmark == null || edgeListToBenchmark.size() != initialSize) {
      // Create a new list on the first run
      edgeListToBenchmark = (IEdgeListDatastructure) gds.
        newList(ListType.GlobalEdgeList);
      for (int i = 0; i < initialSize; i++) {
        edgeListToBenchmark.add(edgeList[i]);
      }
    }

  int rand;
  IEdge e;

  randomEdgesInList = new Edge[operationSize];
  randomEdgesNotInList = new Edge[operationSize];

  HashSet<Edge> tempEdgesInList = new HashSet<Edge>();

  for (int i = 0; i < operationSize; i++) {
    do {
      rand = Rand.rand.nextInt(initialSize);
      e = edgeList[rand];
    } while (tempEdgesInList.contains(e));
    tempEdgesInList.add((Edge) e);
  }

  int listCounter = 0;
  for (int i = 0; listCounter < operationSize; i++) {
    e = gds.newEdgeInstance((Node) nodeList[i], (Node) nodeList[i]);
    if (!edgeListToBenchmark.contains(e)) {
```

```
        randomEdgesNotInList[listCounter] = (Edge) e;
        listCounter++;
      }
    }

    tempEdgesInList.toArray(randomEdgesInList);
  }

  // Common setup method that defers the action
  public void setUp(Class<? extends IDataStructure> datastructureClass,
    Integer setupSize) {
      setUpGds(datastructureClass, setupSize);
      int initialSize = (int) (setupSize - Math.ceil(operationSize / 2));

      if (INodeListDatastructure.class.isAssignableFrom(datastructureClass)) {
        initForNodeList(datastructureClass, initialSize);
      }

      if (IEdgeListDatastructure.class.isAssignableFrom(datastructureClass)) {
        initForEdgeList(datastructureClass, initialSize);
      }
    }
```

Listing 6.3: Sample code for data providers for the benchmarking

```
@Bench(dataProvider = "testInput", beforeEachRun = "setUp")
public void ContainsFailure_Edge(Class<? extends IDataStructure>
  datastructureClass, Integer setupSize) {
    for (i = 0; i < operationSize; i++) {
      edgeListToBenchmark.contains(randomEdgesNotInList[i]);
    }
  }
```

Listing 6.4: Sample code for benchmarking `contains` on a edge list

After all benchmarks are run and the measured costs are handled, the three largest results from each set are discarded. As the Java Virtual Machine usually runs optimizations on frequently called methods (by compiling them from byte code to faster running machine code), we previously saw each set containing a small number of results far apart from the average. Controlling this mechanism is not possible in an obvious way, so running a “warmup phase” or discarding outlier results is common in Java benchmarking. The cleaned results are written to files in a structure that is shown in Listing 5.1 on page 12.

As already mentioned, taking the benchmarks can take some time. Performing 10 runs on a set of 24 input sizes between 10 and 200,000 elements takes between one hour for `DEmpty` and 29 hours for `DArray`. While nearly all datastructures can be benchmarked in less than two hours each, it is only `DArray` and `DLinkedList` (20 hours) that need much more time. To overcome too long computation times, the benchmarks can be run in parallel. But it should be strictly avoided to overload the benchmarked system, as this leads to wrong results.

6.3 Profiler

To build a profiler that is easy to maintain, but does not lack functionality, we use AspectJ[10, 16] which enables us to write the code for the profiler seperated from the main code. With a special compiler, the profiler code is woven into the byte code compiled from the main code. This keeps the seperate aspects seperated: a class representing a metric contains only the logic for computing the metric, a datastructure only the logic to store data.

6.3.1 Profiling aspects - gathering the data

We want to count the number of accesses onto each datastructure, split up into the callee information ("Who accesses data?"), the access type ("Which method is called..."), and the accessed datastructure ("...on which datastructure type?"). Listing 6.5 shows the code responsible for monitoring the graph generation part: a pointcut defines the entry point for any modification to be done (in this case: watch out for the execution of the method `generate` on any sub classes of `IGraphGenerator` and keep a pointer to the class on which the method is called on) and an advice contains the code to be woven into the original code (in this case, an `around` advice is chosen to wrap the advice around the whole existing method which is executed through `proceed`). The field `currentCountKey` is used to hold a key representing the current callee. A stack is used to properly allocate nested calls.

Listing 6.6 shows the code responsible for reacting on an access. The pointcut has the same structure as above. The advice calls the static method `Profiler.count` to actually count the access, providing the three key facts about the access.

```
private static Stack<String> formerCountKey = new Stack<>();

pointcut graphGeneration(IGraphGenerator graphGenerator) :
  execution(* IGraphGenerator+.generate()) && target(graphGenerator);

Graph around(IGraphGenerator graphGenerator) : graphGeneration(graphGenerator) {
  formerCountKey.push(currentCountKey);
  currentCountKey = ((GraphGenerator) graphGenerator).getName();
  currentGraph = proceed(graphGenerator);
  currentCountKey = formerCountKey.pop();
  return currentGraph;
}
```

Listing 6.5: Pointcut and advice to monitor graph generation (excerpt)

```
pointcut contains(DataStructure list) : call(* IDataStructure+.contains(..))
   && target(list);
pointcut size(DataStructure list) : call(* IDataStructure+.size())
   && target(list);

boolean around(DataStructure list) : contains(list) {
  boolean res = proceed(list);
  if (res)
    Profiler.count(this.currentCountKey, list.listType,
      AccessType.ContainsSuccess);
  else
    Profiler.count(this.currentCountKey, list.listType,
      AccessType.ContainsFailure);
  return res;
}

after(DataStructure list) : size(list) {
  Profiler.count(this.currentCountKey, list.listType, AccessType.Size);
}
```

Listing 6.6: Pointcut and advice to monitor datastructure accesses (excerpt)

6.3.2 Profiler storage - handling the data

The aspects described in the former section gather the access data through aspects and hand them over to the class `Profiler` that stores them in an integer array and aggregates them later. This data is still present in raw format, cost data is assigned in a later step. To distinguish in which phase of the graph analysis the access occured, a three-bucket system is used: during one batch, all accesses are counted in the batch's local bucket. After the batch is completed, these access countings are added to the run's bucket and the batch bucket is emptied. And after completing a whole run, the run's bucket is added to a series-wide bucket and the run's bucket is emptied. This allows to compute recommendations based on a whole run or even over all current series.

To provide data about mean list sizes, each list initialization and each insertion and removal of elements is counted seperately, split up into the different list types. The recommender uses this information for the estimation of benchmark-based costs.

On each finished batch, run, and series, the counted accesses are written to files, seperated in their distinct callee types and aggregated over the whole file. Listing 6.7 shows an excerpt of such a file.

```
NR.GlobalNodeList_Init=0
NR.GlobalNodeList_Add=0
NR.GlobalNodeList_ContainsSuccess=0
NR.GlobalNodeList_ContainsFailure=0
NR.GlobalNodeList_GetSuccess=0
NR.GlobalNodeList_GetFailure=0
NR.GlobalNodeList_Random=0
NR.GlobalNodeList_RemoveSuccess=5000
NR.GlobalNodeList_RemoveFailure=0
NR.GlobalNodeList_Size=0
NR.GlobalNodeList_Iterator=0

aggregated.GlobalNodeList_Init=2
aggregated.GlobalNodeList_Add=70000
aggregated.GlobalNodeList_ContainsSuccess=1189412
aggregated.GlobalNodeList_ContainsFailure=140000
aggregated.GlobalNodeList_GetSuccess=200100
aggregated.GlobalNodeList_GetFailure=0
aggregated.GlobalNodeList_Random=1005785
aggregated.GlobalNodeList_RemoveSuccess=5000
aggregated.GlobalNodeList_RemoveFailure=0
aggregated.GlobalNodeList_Size=205181
aggregated.GlobalNodeList_Iterator=0
```

Listing 6.7: Profiling results file (excerpt)

6.4 Recommender

To compute recommendations for all available datastructures, we need to keep track of them. As Java reflection does not covers a feature to get all classes that implement an interface or extend another class - this way we could request a list of all classes that implement the interface `IDataStructure` -, we use the class `ClassPointers` to hold some sets of pointers to such classes. It is for example used to build the list of datastructure combinations used by the recommender.

At the time of writing, we have implemented eleven datastructure classes, and each of them can be used for storing nodes and edges. They can be used for six different list types, so there is a maximum number of $11^6 = 1,771,561$ combinations. Calculating costs is done in four categories, and the results need to be sorted. It is obvious that such a large set of data can not be handled without causing a huge slowdown to the whole analysis.

Through two mechanisms, this number is reduced. The first trick is not to define classes for all six list types, but only the first four. Classes for the last two list types, where in a directed graphs incoming and outgoing edges of a node are stored seperatedly, would get the class assigned through the fallback definition from the class assigned to the local edge list. This lowers the number of combinations to a maximum of 14,641.

The second reduces the set before calculating the costs by eliminating combinations where classes are assigned to unused datastructures. For example, it makes no sense to consider combinations where the local node list is not set to

DEmpty in an undirected graph - it never makes use of instances of this list type. Even if the costs for all combinations where everything but this one list type is fixed are equal, the list of recommendations would be unnecessarily filled up. This lowers the number to 10,001 for directed graphs (where five of the six list types are in use) and 1,001 for undirected graphs (where only three list types are in use).

In the same second step, we allow to filter out combinations using another configuration parameter. It controls the usage of hash-based datastructures like DHashMap, DHashSet, DHashTable, and DHashArrayList for the global edge list in graphs with more than 65.000 nodes, as hash collisions are known to cause trouble here.

To ease the cost computation and avoid repetitions, this is not done on the full set of combinations: as a single datastructure will occur as a candidate for the same list type in multiple combinations, this would mean calculating these costs multiple times. Instead, these costs are generated seperatly in a first step, such that the costs for each combination of datastructure and list type are known. Afterwards, the costs for each full combination of datastructures to a complete list are generated by summing up the single costs.

After all costs are estimated, an additional filtering step allows to filter out combinations where estimated memory costs exceed a given bound.

The resulting costs are written to the same files that already hold the profiler statistics shown in Listing 6.7. Listing 6.8 shows an excerpt of such a file, covering the profiler output.

```
Aggr for RuntimeComplexity: 182.556*O(1) + 1.214*O(d) + 18.657*O(N) + 182.531*O(E)
 Recommendations:
  GlobalNodeList=DArray;GlobalEdgeList=DHashMultimap;LocalNodeList=DEmpty;
    LocalEdgeList=DHashSet;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
    202.425*O(1) + 458*O(E)
  GlobalNodeList=DArray;GlobalEdgeList=DLinkedHashMultimap;LocalNodeList=DEmpty;
   LocalEdgeList=DHashSet;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
    202.425*O(1) + 458*O(E)
Aggr for MemoryComplexity: 2.402*O(d) + 1*O(N) + 2*O(E)
 Recommendations:
  GlobalNodeList=DLinkedHashMultimap;GlobalEdgeList=DArray;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
   2.402*O(d) + 1*O(N) + 2*O(E)
  GlobalNodeList=DHashTable;GlobalEdgeList=DHashTable;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
    2.402*O(d) + 1*O(N) + 2*O(E)
Aggr for RuntimeBenchmark: 195.402.642,38
 Recommendations:
  GlobalNodeList=DArray;GlobalEdgeList=DHashTable;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
   193.740.406,50
  GlobalNodeList=DArrayList;GlobalEdgeList=DHashTable;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty:
   194.449.611,13
Aggr for MemoryBenchmark: 842.861.268.025,08
 Recommendations:
  GlobalNodeList=DLinkedHashMultimap;GlobalEdgeList=DHashMultimap;
   LocalNodeList=DEmpty; LocalEdgeList=DArray;LocalInEdgeList=DEmpty;
   LocalOutEdgeList=DEmpty: 830.307.018.446,04
  GlobalNodeList=DHashMultimap;GlobalEdgeList=DHashMultimap;
   LocalNodeList=DEmpty; LocalEdgeList=DArray;LocalInEdgeList=DEmpty;
   LocalOutEdgeList=DEmpty: 830.371.788.647,84
Aggr for CombinedBenchmark: 7,35
 Recommendations:
  GlobalNodeList=DArrayList;GlobalEdgeList=DHashTable;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty: 5,00
  GlobalNodeList=DArray;GlobalEdgeList=DHashTable;LocalNodeList=DEmpty;
   LocalEdgeList=DArray;LocalInEdgeList=DEmpty;LocalOutEdgeList=DEmpty: 5,40
```

Listing 6.8: Results of the profiler output (excerpt)

6.5 Hot swap

In this Section, we cover the implementation details for the hot swap algorithm. The most important part is the configuration, as the quality of all swaps results from the configuration. Swapping too early to a recommendation that has not yet completely proven its efficiency is in the same way undesired as running an analysis on a large graph and performing an expensive swap in every other batch. Afterwards, we present how the automatic hot swapping can be overwritten, and how the efficiency of a swap is improved through simple modifications of the basic approach.

6.5.1 Configuring the hot swap

Four aspects of the hot swapping algorithm mentioned in Section 5.5 are designed to be configurable: the window size, the lower bound after which a recommendation is examined further, the number of batches after which the switch should have amortized itself and the maximum number of swaps to perform during a single run.

Through the window size, the number of batches over which recommendations should be collected can be controlled. It should be set not too low, as the window's task is to keep recurring recommendations; but there is also no need for a too large window: recommendations that are considerable will just be present more often in a larger window. Even if it supports such recommendations, it mostly stores redundant information. We arbitrarily set this number to six batches.

The lower bound gives a number, relative to the window size, that decides how frequent a recommendation should occur to be considered. In our default configuration, it is set to 0.8, so recommendations have to be present in at least $0.8 * 6 = 4.8$ of six batches to reach the efficiency check. A higher bound gives recommendations that do not occur that frequent less chances, but if a recommendation is considerable over some batches, it still surpasses the bound.

Efficiency is the last hurdle to be taken by a recommendation. It is obviously no gain to switch datastructures at very high costs just before the analysis ends. The amortization counter helps to avoid this. In the default, it is set to 10 batches, but this can be overwritten. For example, in a long running analysis with very much batches, it can be fine to run a switch that takes relatively long, but leads to lower costs seen over a total of all batches.

The last parameter controls the behaviour of the hot swapping algorithm on a longterm view. If we expect to improve the performance through performing a swap, it might be desired to hinder the hot swap to harm the performance afterwards. Recommendations are also computed for all batches following a hot swap, and handling them (through searching for the most occuring recommendations and checking if they are efficient) takes some time in each batch. The hot swap can be disabled completely after a number of swaps to allow the recommended datastructure combination to play out the improvement completely.

6.5.2 Manual switching

For directly comparing the performance of two predefined combinations on a single experiment's run, the hot swap can be taken into manual mode. It won't perform swaps anymore on the basis of profiled accesses and given recommendations, but will swap datastructures at given numbers of batches to predefined combinations. This will override all automatic handling: searching for recurring recommendations and performing an efficiency check before the swap is unnecessary. In Listing 6.9, an example for such a configuration is shown.

```
EnumMap<ListType, Class<? extends IDataStructure>> newCombination =
  GraphDataStructure.getList(ListType.GlobalNodeList,DHashMap.class,
    ListType.GlobalEdgeList,DHashMap.class);
HotSwap.addForManualSwitching(10, newCombination);
```

Listing 6.9: Sample code for configuring a manual swap that will switch both the global edge and node list to `DHashMap` after batch 10

6.5.3 Enhancing the efficiency of the swap

The costs of performing a swap should be kept as low as possible. One obvious part of this is to perform swaps only on those datastructures that are changed. The second improvement lies in the handling of inserted elements. As no duplicate entries should be present in the datastructures, each insertion is internally preceded by the check whether the element to be inserted is already held in the datastructure. If this is the case, the insertion is aborted. On swapping a datastructure, we can be sure that all elements are unique, as they had not been inserted in the source datastructures in the first place. No duplicates will be created through just moving them to a new datastructure, so we can disable the duplicate check. Both parts are reflected in the efficiency check to let it estimate these costs as good as possible.

6.6 Configuration

Nobody should be forced to use the parts we developed. Even if they can help finding bottlenecks in analyses, some use cases require to avoid the overhead caused by adding profiling and swapping capabilities - either as the datastructures are already configured to perform well, or switching them is not intended due to other reasons. Profiling accesses and allowing the hot swap algorithm to do its work is explicitly to be enabled. Benchmarking, profiling and hot swapping options can be controled through the following configuration parameters, either in the file `profiler.properties` or through `Config.overwrite`:

`BENCHMARKING_RUNS` Controls how many runs to perform for each benchmarking method

`BENCHMARKING_INPUTSIZES` Controls the input sizes in a list of numbers (eg. `100, 200, 500`)

`PROFILER_ACTIVATED` Set it to `true` to enable counting accesses and writing access statistics to files

`PROFILER_WRITE_ACCESSSTATS_AFTER_EACH_BATCH / _RUN` Set it to `true` to write and plot access statistics at the end of each batch / run

`RECOMMENDER_USE_SIMPLE_LIST` Set it to `false` to use the full list of datastructure combinations and not the reduced one described in Section 6.4

`RECOMMENDER_NUMBER_OF_RECOMMENDATIONS` Controls the number of recommendations to write to logs and the standard output

`RECOMMENDER_GRANULARITY` Controls for which access statistic sections a recommendation is appended to the statistic files. Possible values are: `DISABLED` to completely disable computing and writing recommendations, `ALL` to compute and write recommendations for all following sections, and combinations of `AGGREGATIONONLY` (compute and write recommendations for aggregations at the end of each file only), `EACHMETRIC`, `EACHBATCHGENERATION`, `EACHUPDATETYPE`, `EACHBATCH`, `EACHRUN`, and `EACHSERIES`

`RECOMMENDER_PRINTRECOMMENDATION_AFTER_EACH_BATCH` Set it to `true` to output the last recommendations at the end of each batch

`RECOMMENDER_PRINTRECOMMENDATION_AFTER_EACH_RUN` Set it to `true` to output the last recommendations at the end of each run

`RECOMMENDER_MAX_MEMORY_BOUND` Controls an upper bound for filtering recommendations based on their estimated memory consumption

`RECOMMENDER_FORCE_USAGE_OF_HASHBASED_FOR_GLOBALEDGELIST` Set it to `true` to not filter out hash-based datastructures for the global edge list in graphs with more than 65.000 nodes

`RECOMMENDER_LISTAGGREGATOR` Selects the list aggregator to be used (`MIN`, `MAX`, `MEAN`)

`RECOMMENDER_BUCKETSELECTOR` Selects the bucket selector to be used (`LOWER`, `UPPER`, `INTERPOLATE`)

`RECOMMENDER_COMBINEDCOMPLEXITY_RUNTIMEWEIGHT` Controls the weight of the runtime benchmark results for computing combined costs, as described in Section 5.4.4

`RECOMMENDER_COMBINEDCOMPLEXITY_MEMORYWEIGHT` Controls the weight of the memory benchmark results for computing combined costs

`HOTSWAP_ENABLED` Set it to `true` to enable hot swaps

`HOTSWAP_INCLUDE_FIRSTBATCH_FOR_EFFICIENCY_CHECK` Set it to `true` to also include the very first batch into the efficiency check. As it usually contains the graph generation accesses, it tends to temper the results

`HOTSWAP_WINDOWSIZE` Controls the window size for the collection of recommendations through the hot swap algorithm

`HOTSWAP_LOWER_BOUND` Controls the (relative computed) lower bound a recommendation has to exceed to qualify for swapping (eg. 0.8 means that is has to occur in 80% of the last recommendations)

`HOTSWAP_AMORTIZATION_COUNTER` Controls the maximum number of batches to be used for the amortization check

`HOTSWAP_MAXNUMBER_OF_SWAPS` Controls the maximum number of swaps to be done during analyzing one DNA run

`HOTSWAP_PROFILERDATATYPE_SELECTOR` Selects the cost category to be optimized (eg. `RuntimeBenchmark`, `MemoryBenchmark`)

7 Evaluation

In this section, we cover an evaluation of our contribution, based on some simulations. The common setup:

- All experiments are run with five different profiling configurations:
  1. Initial setup as defined for each experiment, profiling and hot swap enabled
  2. Initial setup as defined, profiling enabled, hot swap disabled
  3. Initial setup as defined, profiling and hot swap disabled
  4. The datastructure combination is changed to the configuration of the first swap in the first experiment, profiling and hot swap enabled
  5. Datastructure combination from the first swap, profiling and hot swap disabled

  The first case is the usual one later on. Comparing it to the second one shows the performance gain through the hot swap. The third case allows us to see a comparison to the performance of the system without any profiling influence. Comparing the first case with the fourth can give an answer to the question whether using a dumb configuration in the beginning and swapping later is much more expensive than using an optimal case from the beginning. Using the fifth case, one can again estimate the runtime overhead of the profiler, compared to the fourth case.
- With each profiling configuration, ten runs are done to level out inaccuracies
- For the results of runtime and memory measurements, the average data from these ten runs is taken
- Each configuration uses its own forked JVM process to avoid side effects (especially in terms of memory usage)
- Garbage collection is forced to run after each batch, for more accurate memory measurements. Automatic invocations triggered through the JVM are neither influenced nor prohibited.

7.1 Real-life use case: runtime optimization for a random growing graph

Setup for Experiment 1	
Graph type:	Undirected random graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	2,000 nodes, 30,000 edges
Updates:	25 batches, each adding 100 nodes and 4,500 edges and removing five nodes and 225 edges
Metrics:	degree distribution and undirected clustering coefficient
Improvement:	based on runtime benchmark

The first experiment initially generates an undirected random graph with 2,000 nodes and 30,000 edges. In each of the following 25 batches, 100 nodes and 4,500 edges are added and five nodes and 225 edges are to be removed. All lists are configured to use `DArrayList` in the beginning, and the metrics degree distribution and undirected clustering coefficient are used. After the fifth batch is finished, a stable recommendation in terms of the RuntimeBenchmark can be given: while continuing the experiment with the current combination would generate costs of 6,847,034,072.89, the recommended entry only accounts for costs of 610,734,401.74 (excluding the swap), saving over 90%.

The datastructures for the node-local node, incoming and outgoing edge lists are changed to `DEmpty`. This causes no swappings, as they are not in use in an undirected graph. The graph-global node list is reconfigured to use a `DHashArrayList`: the access statistic in Figure **??** shows that the access types `ContainsSuccess`, `GetSuccess`, `Random`, and `Size` dominate for this list type, and especially for the second and third access type, the datastructure combined of a hash set and an array list yields good performance. Using a pure `ArrayList` is also suggested, as the estimated costs are very close: 587,305,095.63 for the chosen combination, 587,970,869.94 using the `ArrayList` instead for the global node list.

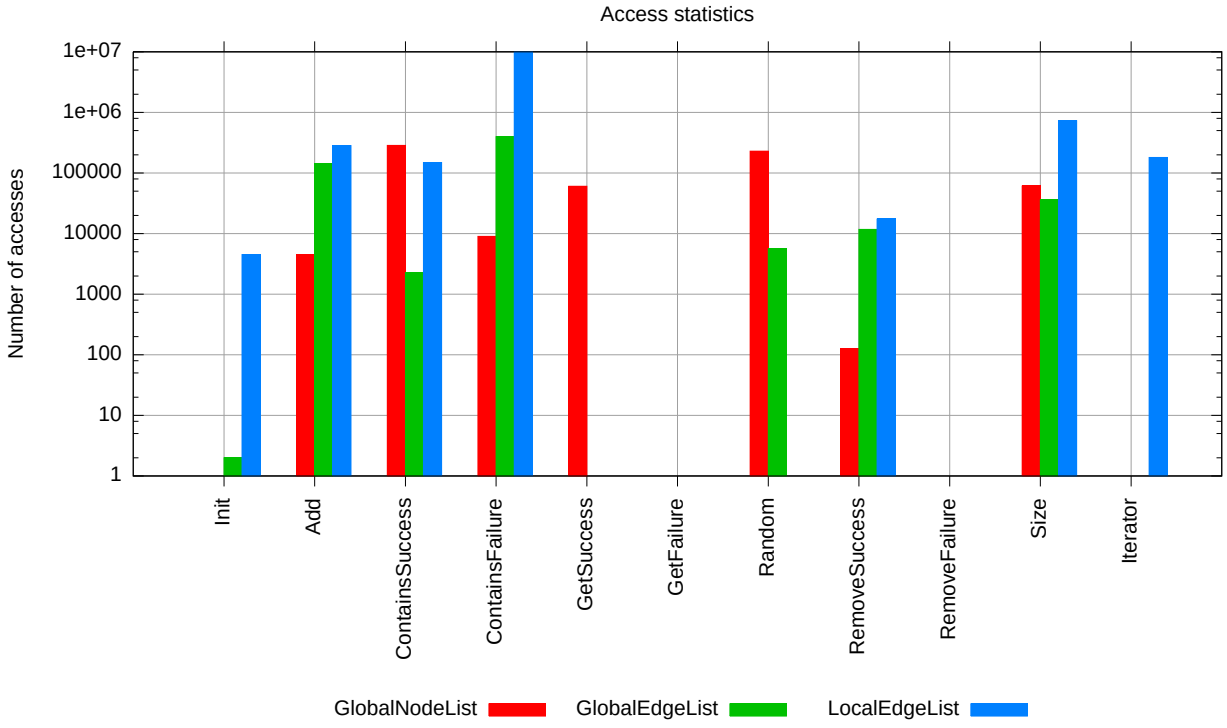

Figure 7.1.: Access statistics for Experiment 1 (log scale for count)

For the graph-global edge list, a list using `DHashSet` is recommended. While it does not perform that well on `Add` operations that occur often as the edge list is growing, it is among the best choices for failing `Contains` and the best choice for succeeding `Remove` requests that also occur often. The node-local edge lists are swapped to `DArray`. While its performance is getting worse with growing lists, it yields by far the best performance on small lists in nearly all access types.

Runtime measurements of our experiment (visualized in Figure 7.2) show that the swap is efficient: the per-batch runtime drops to the runtime of the optimized case. While using the initial combination leads to fast growing runtimes (about 10s for the fifth batch, about 40s after batch 20), the recommended combination yields per-batch runtimes of less than five seconds. While the cumulated runtime of the completely optimized case can not be reached, the additional costs are acceptable: as seen in Figure 7.3, the complete runtime using the initial datastructure combination and swapping is about 90s, while the continued usage of the first combination yields a runtime of about 600s. Starting with the recommended combination, the experiments would have finished after 30s per run.

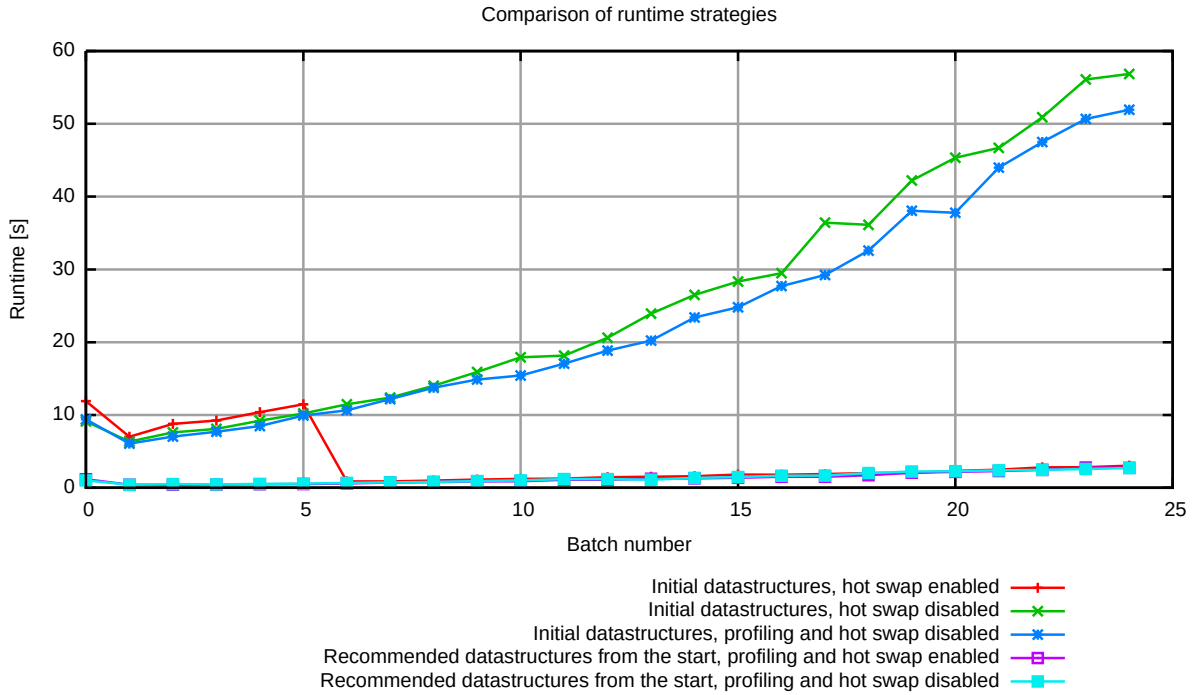

Figure 7.2.: Runtime plots for Experiment 1

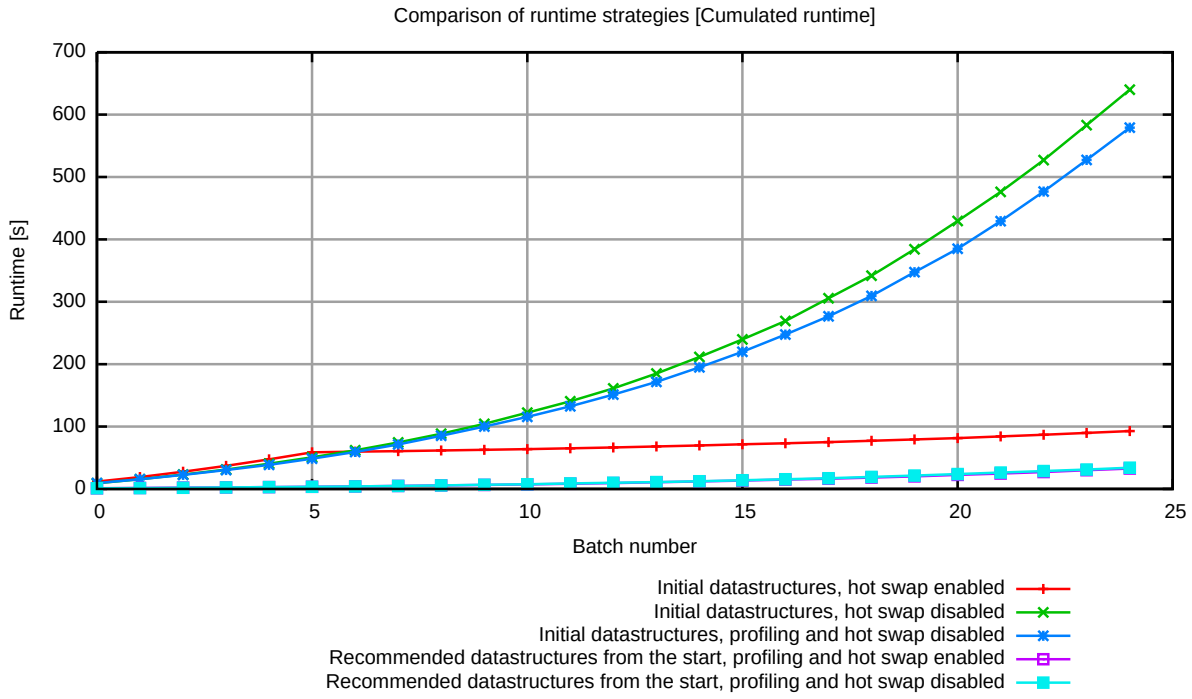

Figure 7.3.: Runtime plots for Experiment 1 (cumulated runtime)

7.2 Real-life use case: memory optimization for a random growing graph

Setup for Experiment 2	
Graph type:	Undirected random graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	2,000 nodes, 30,000 edges
Updates:	25 batches, each adding 100 nodes and 4,500 edges and removing five nodes and 225 edges
Metrics:	degree distribution and undirected clustering coefficient
Improvement:	based on memory benchmark

In our second experiment, we want to optimize the memory consumption. Especially for graphs that are already large and account for a high memory consumption through the data of nodes and edges, we can run out of memory if the datastructures to store these elements need a too large memory overhead. Optimizing the memory consumption can help through recommending datastructures that use as little memory as possible.

The setup of this experiment is the same as in the first one, the only change is the optimization strategy. Three changes are done, after finishing the fourth batch, the ninth batch and the 22nd batch. While the recommendation in the first experiment predicted costs to fall by 90%, the estimated memory lie much closer: at the first swap, costs of $3,506 * 10^9$ are given for the current configuration, and $3,338 * 10^9$ for the recommended one. Comparisons at the later swaps show similar ranges.

By having only so little savings predicted, it is not surprising that the memory consumption does not change that much. In all cases, the experiments take between 44MB and 54MB of memory after huge initial spikes. The largest amount of memory is used in the case where the hot swap was enabled, which can also be seen in the other experiments - this is the memory consumption for the internal data keeping of the hot swap. In the other cases, no obvious difference can be seen. Especially the measured consumptions of the experiments with the initial and the recommended configuration, while having the profiler disabled, do not show differences that justify the swap. Figure 7.4 visualizes these results.

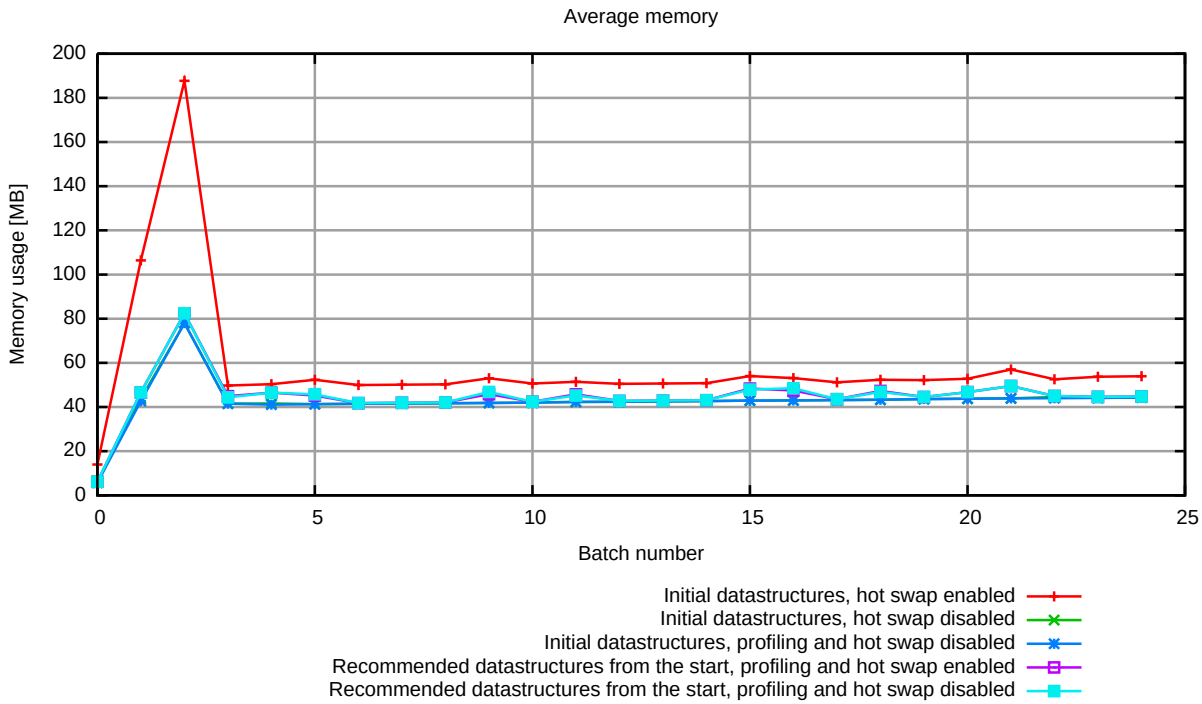

Figure 7.4.: Memory plot for Experiment 2

Setup for Experiment 3

Graph type:	Directed and undirected random graph
Initial datastructures:	`DArray` for all lists
Initial graph size:	7,500 nodes, 37,500 edges
Updates:	20 batches, each adding 7.5% edges (~2.800 in the first batch, ~11,000 in the last batch)
Metrics:	degree distribution and connected components
Improvement:	based on combined benchmark

In Section 6.4, we mention that using the complete set of all available datastructure combinations for the recommender is too costly. The third experiment is to compare the profiler runtimes between a run with a directed graph, where in each case costs for 10,001 datastructure combinations are computed and sorted, and a run with an undirecte graph, where only 1,001 combinations are considered. Surprisingly, there is no obvious runtime difference between these two runs: the profiler is able to do its work in about 0.12s per batch in both cases. Figure 7.5 visualizes the cumulated runtime for the different configurations of this experiment.

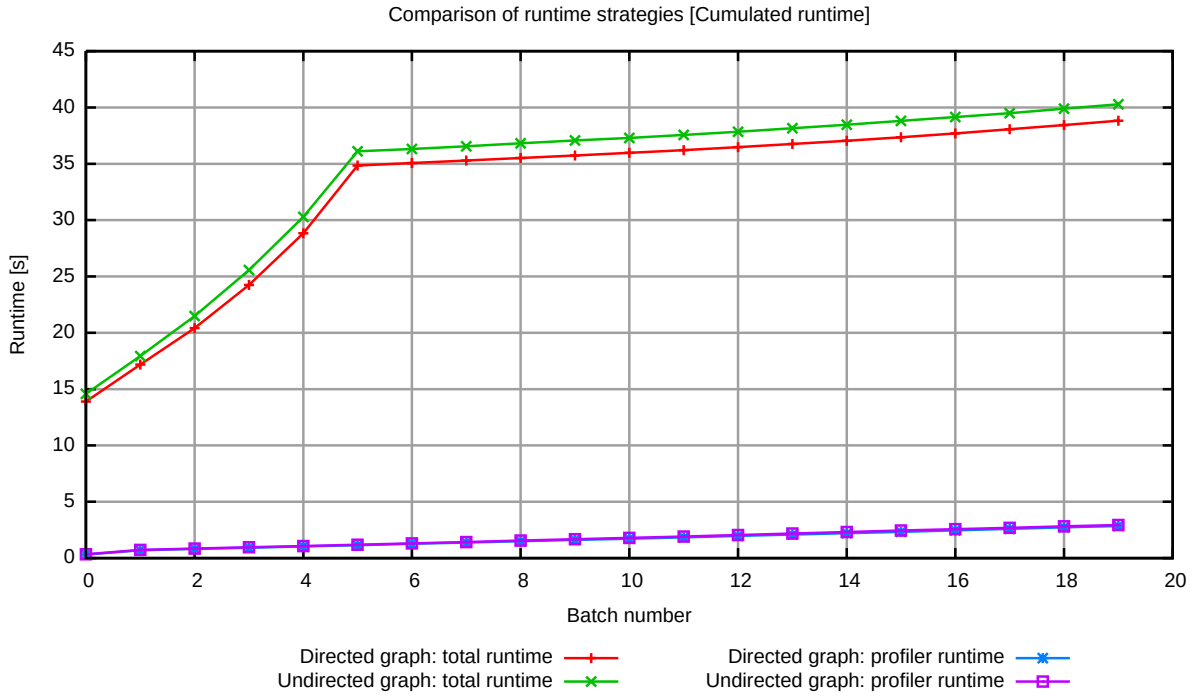

Figure 7.5.: Comparison of the cumulated runtimes of runs with an undirected graph and runs with a directed graph

Setup for Experiment 4

Graph type:	Undirected scale-free graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	1,000 nodes, 15,000 edges
Updates:	20 batches, each adding 20 nodes and 300 edges
Metrics:	degree distribution and betweenness centrality
Improvement:	based on combined benchmark

In this experiment, we evaluate the recommendation quality for a setup with two metrics that make strong use of two distinct access types: computing the degree distribution results in regular requests the size of each local edge list, and to compute the betweenness centrality, a lot of iterations over the global node and local edge lists are done. In Figure 7.6, the access statistics visualize the dominance of two access types. On this basis, the local edge list is told to use `DArray`, both global lists are switched to use `DHashArrayList`.

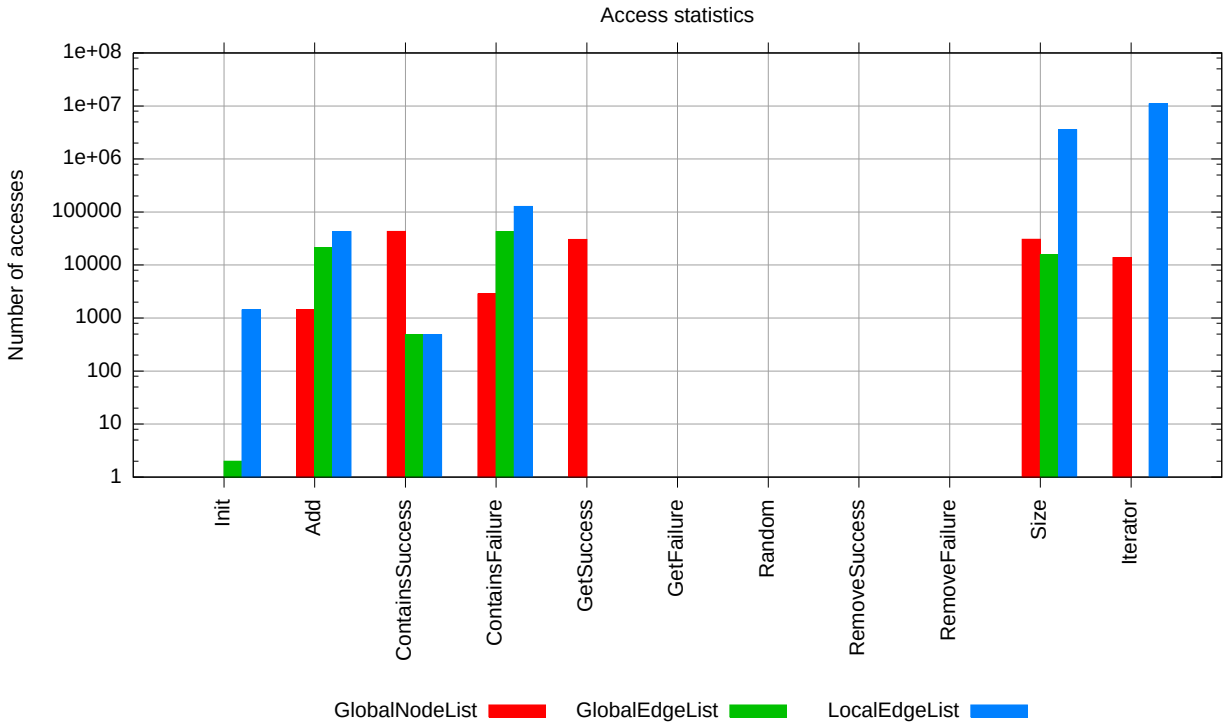

Figure 7.6.: Access statistics for the experiment using a scale-free graph (log scale for count)

But even if the performance of `DArray` is good for iterating and requesting the size, the performance of the whole experiment is not improved: running it takes a total of 820s. Running it using the initial configuration, without making use of profiler, recommender, or hot swap, from first to last batch takes 773s, and starting with the recommended configuration, where the profiler and following parts are also disabled, takes 840s, as plotted in Figure 7.7. Looking at raw runtimes of the single components does not reveal which one is responsible for the differences.

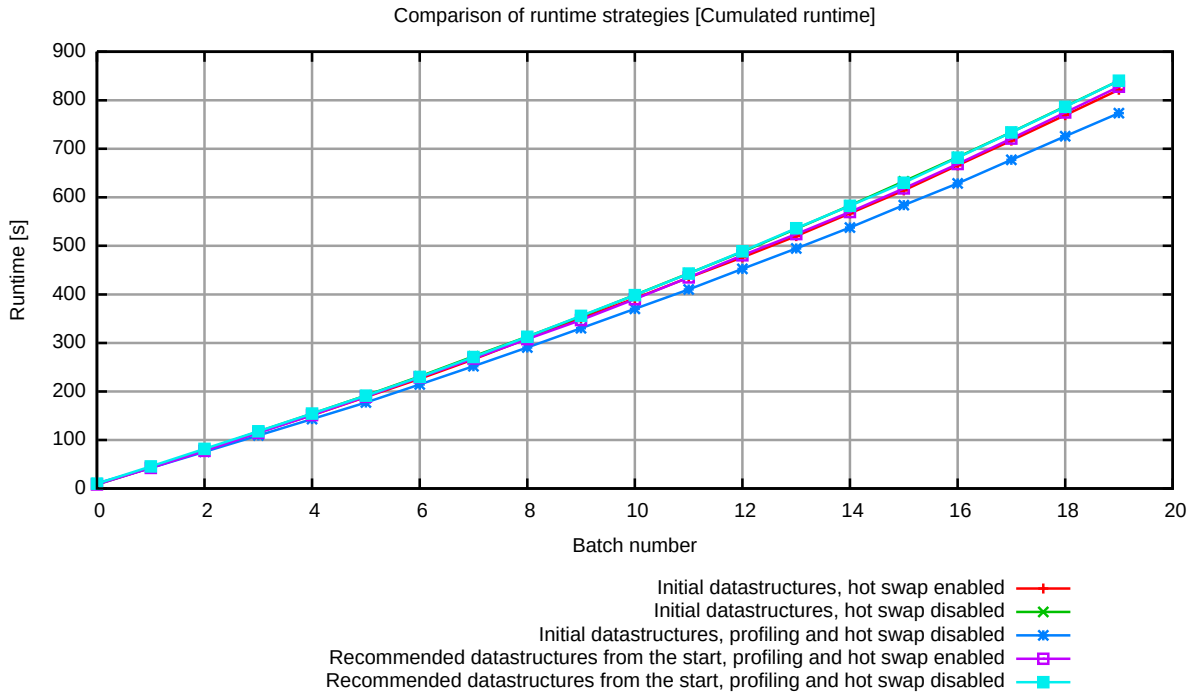

Figure 7.7.: Runtime plots for Experiment 4 (cumulated runtime)

7.5 Experiments with faked metrics

The previous experiments already reflected that the given recommendations can be manipulated through metrics that use only single access types repeatedly and diminish the impact of all other parts that perform accesses within one batch. In this section, we want to exploit this behaviour through faked metrics, that do not run a real analysis on a graph, but each perform only a single access repeatedly. We implemented the following metrics for this purpose:

`ContainsSuccessNodeMetric` using one of the nodes that are added through an update and calling `Graph.containsNode` repeatedly

`GetSuccessEdgeMetric` using one of the edges that are added through an update and calling `Graph.containsEdge` repeatedly

`GetFailureEdgeMetric` using one of the edges that are removed through an update and calling `Graph.containsEdge` repeatedly

`SizeNodeMetric` calling `Graph.getNodeCount` repeatedly

7.5.1 Faked metric `ContainsSuccessNodeMetric`

Setup for Experiment 5

Graph type:	Random graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	10,000 nodes, 50,000 edges
Updates:	25 batches, each adding a single node
Metrics:	`ContainsSuccessNodeMetric`
Improvement:	based on runtime benchmark

In a graph with 10,000 nodes and 50,000 edges, we call `Graph.containsNode` more than 700,000 times per batch. After the fourth batch, a hot swap is performed: `DArray` is the datastructure with the best performance for this access type, so it is recommended and used afterwards for the global node list. All local lists are switched to `DEmpty`. The per-batch runtime drops from about 2.5s to 0.15s, which is visualized in Figure 7.8.

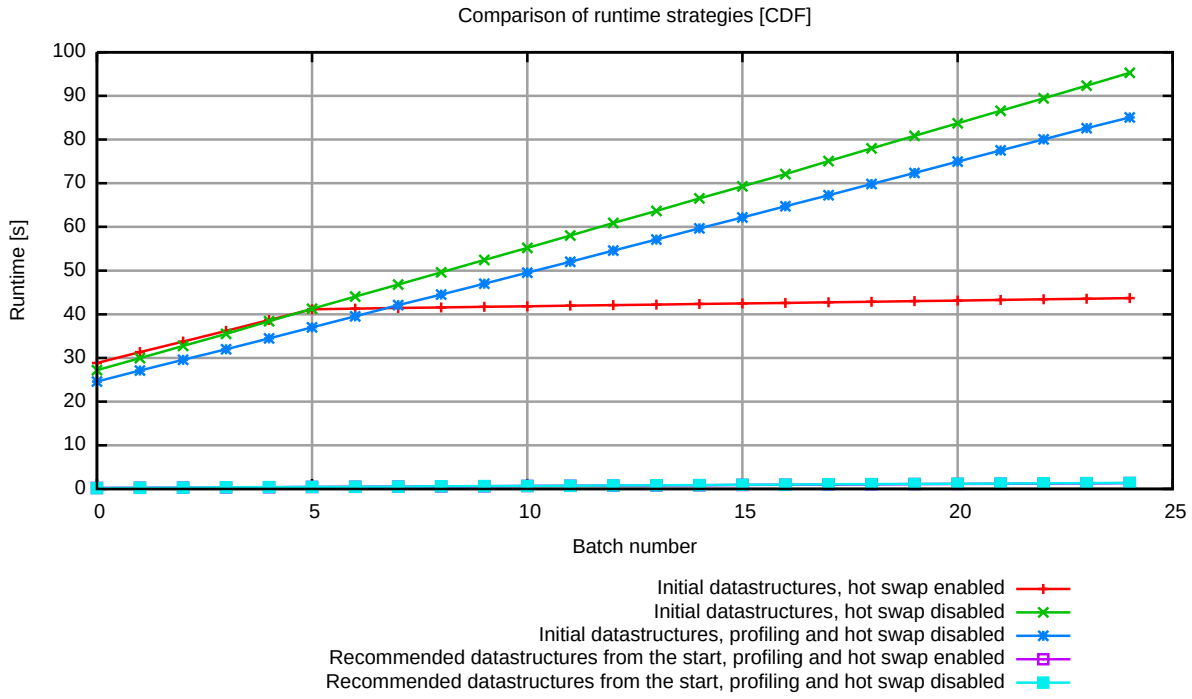

Figure 7.8.: Runtime plots for the experiment using `ContainsSuccessNodeMetric` (cumulated runtime)

7.5.2 Faked metric `GetFailureEdgeMetric`

Setup for Experiment 6

Graph type:	Random graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	3,000 nodes, 75,000 edges
Updates:	25 batches, each removing a single edge
Metrics:	`GetFailureEdgeMetric`
Improvement:	based on runtime benchmark

A second experiment with faked metrics is run on a graph with 3,000 nodes and 75,000 edges, where the metric `GetFailureEdgeMetric` accounts for 300,000 calls of `Graph.containsEdge` for an edge that is no longer present in the graph. In this case, the datastructure `DHashMap` provides the best performance for this access type. Straightforward, a recommendation to use this datastructure for the global edge list is given and applied. The per-batch runtime drops from more than 2 minutes to about 0.2 seconds, which is visualized in Figure 7.8.

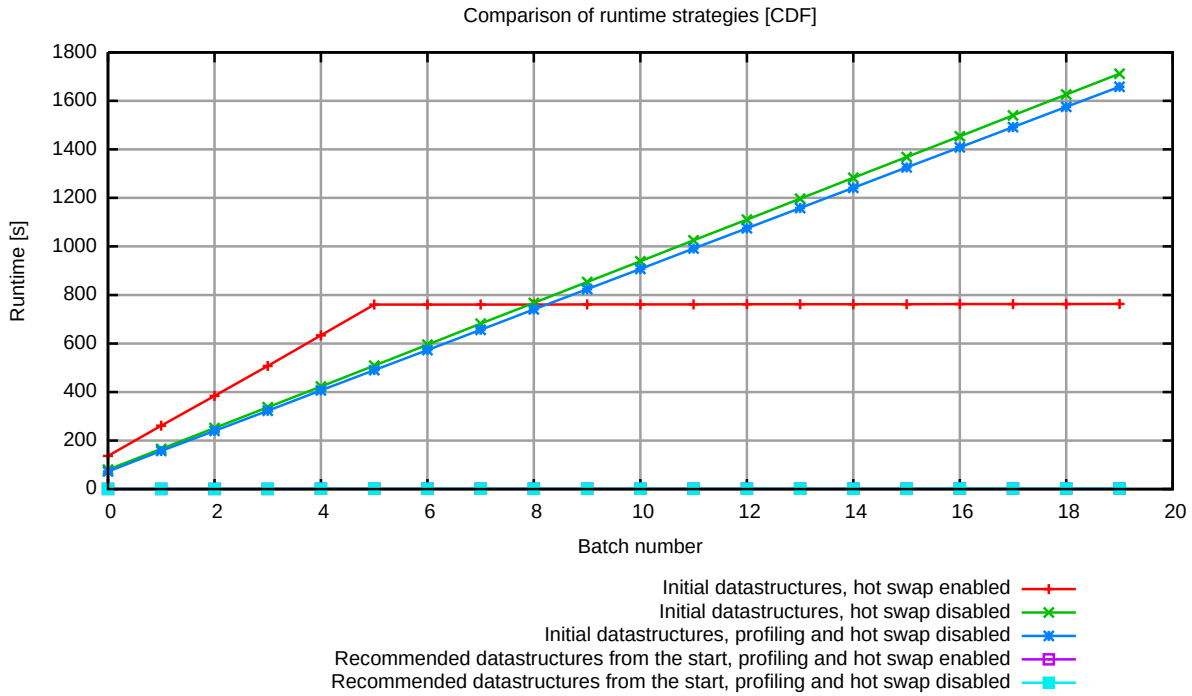

Figure 7.9.: Runtime plots for the experiment using `GetFailureEdgeMetric` (cumulated runtime)

7.5.3 Faked metric `SizeNodeMetric`

Setup for Experiment 7	
Graph type:	Random graph
Initial datastructures:	`DArrayList` for all lists
Initial graph size:	2,500 nodes, 10 edges
Updates:	20 batches, each adding a single node
Metrics:	`SizeNodeMetric`
Improvement:	based on runtime benchmark

In a third experiment, the metric `SizeNodeMetric` is used. In a graph with 2,500 nodes, it triggers a switch of the global node list to use `DLinkedList`, but the runtime results do not justify this as they do not fall after performing the switch. In Figure 7.10, this can be seen clearly: the per-batch runtime is much higher while the profiler is activated, but we would expect to see a difference between running the experiment with the initial combination and running it with the optimized combinations, which is not the case. But considering the action this metric performs, it is no surprise: in most datastructures, the size is not computed on each request, but kept as an internal field. Requesting it results in simply returning a pre-computed value. Over all datastructures, the runtime costs of this access type differ by 10%, so no real performance gain can be expected.

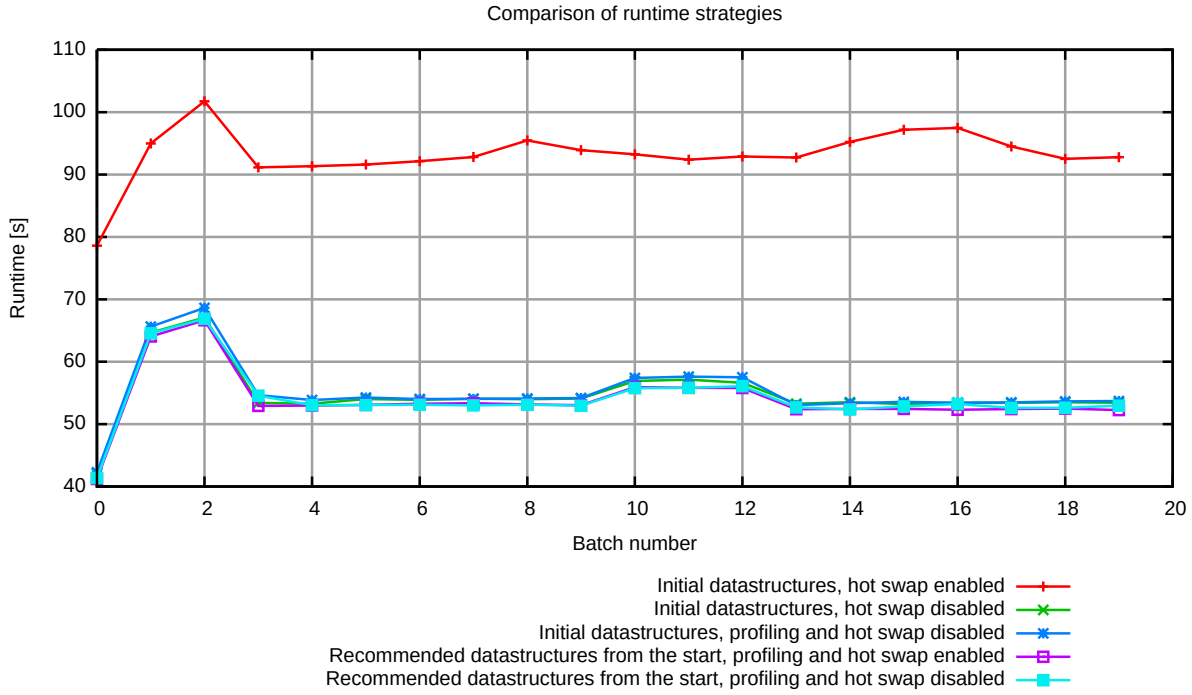

Figure 7.10.: Runtime plots for the experiment using `SizeNodeMetric`

7.5.4 Memory consumption lowered through `DEmpty`?

In the previous experiments with faked metrics, a switch for some lists to `DEmpty` occurs, which should help to lower the memory consumption. But similar to the experiment from Section 7.2, there is no obvious saving in doing this. While the memory consumption even rises very obviously in the first batches of the runs having the hot swap enabled and drops to a usual level again, it does not drop after swapping lists to `DEmpty`. In an example plot from the experiment using `ContainsSuccessNodeMetric` in Figure 7.11, this can be seen clearly.

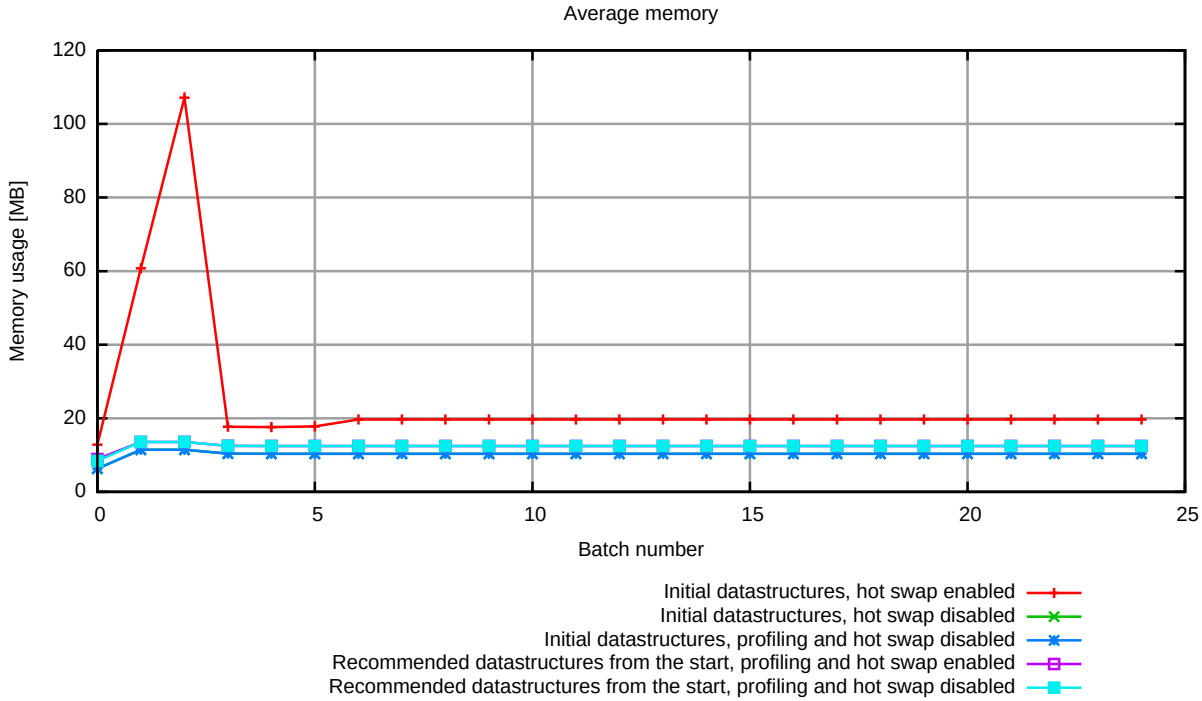

Figure 7.11.: Memory plot for Experiment 21

7.5.5 Changing sliding window sizes

Setup for Experiment 8

Graph type:	Random graph
Initial datastructures:	`DArrayList` for all node lists, `DHashMultimap` for all edge lists
Initial graph size:	5,000 nodes, 25,000 edges
Updates:	50 batches of repeating the following behaviour: five batches adding 250 and removing 25 nodes, followed by five adding 250 and removing 25 edges
Metrics:	GetSuccessEdgeMetric and ContainsSuccessNodeMetric
Improvement:	based on combined benchmark

Using two fake metrics, we want to evaluate the influence of the hot swap window size in this last experiment. With a repetitive batch and according metrics, changing behaviour is simulated.

In a first run, the size of the sliding window is set to 3, so a switch can occur very frequently. The first can be seen after the third batch, and it will exchange the datastructure of the global node list with an instance of `DLinkedHashMultimap`, the global edge list is changed to use `DHashSet` now. The update behaviour will change after the fifth batch. As the recommendation that was used for the first switch could not foresee the new behaviour, the performance is not that good: the metric to perform `get` accesses on the global edge list dominates the access statistics now, and `DHashSet` is the worst choice for this in terms of runtime performance. This one batch alone takes 68s to be finished, which accounts for more than half of the total runtime for this run. Directly after this batch, the datastructures are swapped again: from

now on, the global node list is helt in a `DArray`, the global edge list in a `DHashSet`. Each following batch takes less than 0.5s to finish.

In a second run, an enlarged window of 6 batches is used. The first switch happens after the eighth batch, taking all five batches with the first behaviour and three with the second into account. Thus, the recommender is able to calculate an optimized datastructure combination that fits both behaviours and does not favor just one. As it is the same that is used for the second switch in the first run, all following batches are computed relatively fast.

The runtime plot in Figure 7.12 visualizes: after the sixth batch, the per-batch runtime of both runs is really low. In the first six batches, the larger sliding window accounts for a longer time to come to the first recommendation - instead, using a smaller sliding window, a premature swap leads to a change that had to be corrected some steps later.

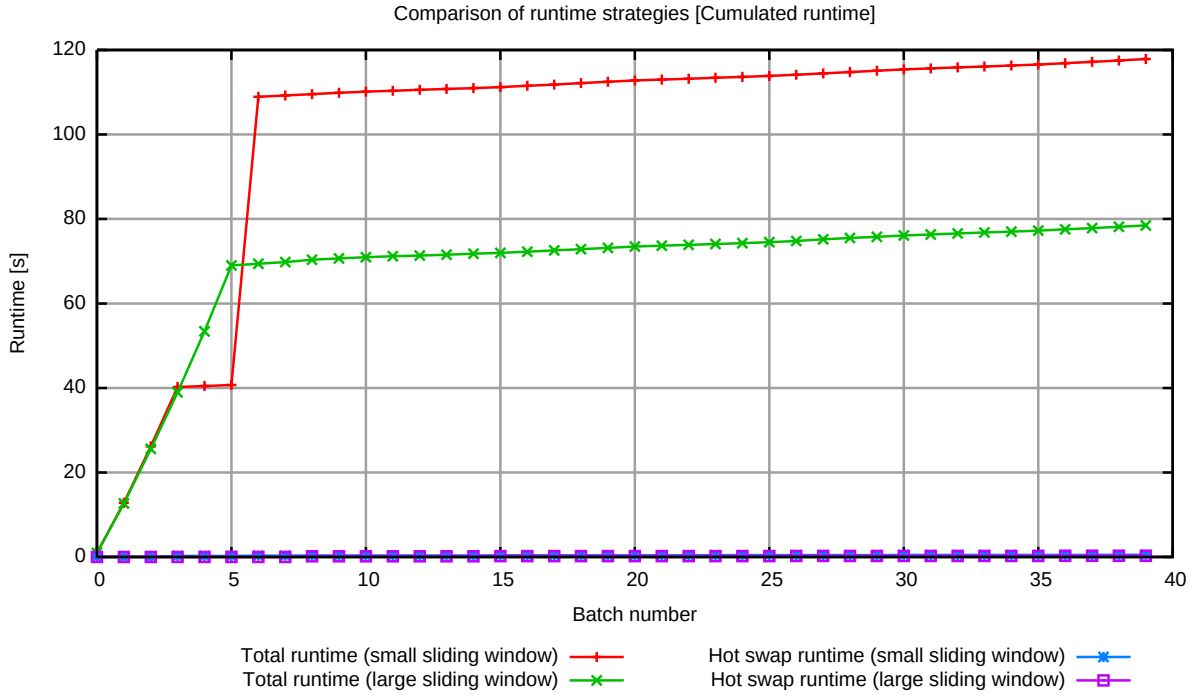

Figure 7.12.: Comparison of the cumulated runtimes of runs with a large and with a small hot swap window size

8 Summary and conclusion

Choosing the right datastructure can heavily influence the performance of any program. In most programs, they occur in different places, and finding bottlenecks can be complicated. Even if a programmer finds the one method that accounts for long runtimes or a high memory consumption, he has to find a sound replacement, recompile the code and restart the program. Approaches like Brainy or CoCo can already help in this place, but do not offer as much flexibility as possible: running synthetic benchmarks makes each result too restricted to changes, and static rules do only approximate the solutions. Having found a replacement based on the comparison in only one category does not guarantee that the recommended datastructure is also a good choice for other accesses.

In our work, we described a system that profiles accesses on datastructures and uses cost data for each access to estimate costs for specific datastructure combinations. Giving these estimations for different cost categories allows to monitor the performance of a running program by different means. Through not only giving these estimations for the datastructure combination that is currently used, but also for a set of others, the system is able to give recommendations for performance improvements and even apply them on runtime through a hot swap algorithm. Our contribution enables a program to do this in a fully automated mode, without a restart or other manual intervention (eg. recognizing that there is a performance bottleneck, looking for a recommendation to fix it, changing source code, and restarting an analysis).

The ability of giving and applying system-dependent recommendations is an advantage over the mechanisms presented in Section 2: our contribution offers more flexibility through a deep integration into the core system and clustered handling over all datastructure instances of the same list type, and lowers the need for manual interventions. Through taking more aspects of the running process into account (eg. which datastructures are in use after all, and how long will the process run after a change was applied), the hot swap algorithm is not too greedy.

In most experiments, the performance gain is obvious. Being able to run a whole analysis in only the fraction of the time a not optimized run takes is a result that justifies our contribution. And making our parts working with only as little performance overhead and configuration as possible gives the user the ability to focus on the core parts of graph analysis again - nobody is interested in the datastructure that is used to store all nodes of the graph as long as this works smoothly.

But our evaluation shows that there still are open issues. Even if the performance was not harmed in any experiment (neither was the analysis slowed down excessively nor did it use much more memory), we did not see the expected results either in some cases. Is there no datastructure combination where we can see an enhancement in terms of memory consumption, like in the experiment in Section 7.2? And did we focus too much on the accesses to datastructures defined for holding the graph and left datastructures the metrics use internally out of sight accidentally? In the experiment in Section 7.4, the given recommendation is not able to enhance the runtime performance, so are we only scratching the surface by analyzing the graphdatastructure performance only and disregarding other datastructure usages? In the next Chapter, we give some ideas for future work in this field to find even more ways of improvement.

9 Future work

During our work, some additional questions arose that we did not cover yet and leave open for future work:

Address hashing problems Even if the problems described in Section 6.1.2 are not completely in our scope, as we cannot change the core of the Java Virtual Machine to use 64 bits in hash code functions, they should be addressed. For random access, hash-based datastructures have huge performance advantages: a hash table of 100,000 edges is able to return a requested edge in $\frac{1}{1000}$ of the time of the fastest datastructure without hashing! But as long as there are problems addressing edges in graphs with more than 65,000 nodes (eg. it is not possible to add a new edge to the graph if another edge with the same hash code is present), these datastructures can not be used.

Enhance the accuracy of profiling The internals of Java's own datastructures can not safely be monitored through the profiling. For example, it could be interesting to keep track of expensive resizings happening in `ArrayList` or `HashTable`. Adding elements to these datastructures can be either cheap (if there is enough space reserved for the new element) or expensive (if the datastructure needs to be resized to reserve space for more elements). For the benchmarks and afterwards for more accurate cost estimations, it would be an advancement to distinguish cheap from expensive accesses.

Improve the results of memory benchmarks The experiments show that estimations of the memory consumption do not correlate with the real consumption. Even if the estimation is only done for the parts of the process that cover the graph datastructure and there can be other parts that could have a much larger impact on the memory usage, we would have expected even small improvements. It remains to be evaluated whether these estimations do not work due to inaccurate benchmarks or wrong usage in the running program. Solving this question might not only give recommendations for the memory consumption (in most cases, it is more interesting to gain runtime performance), but also make a memory boundary for large graphs work: if the analysis already takes a large part of the available memory, it might be undesired to optimize the datastructure solely to runtime performance, as faster datastructures often use more sophisticated accesses that need more memory.

Find fundamentally good combinations In the current state, a user of DNA still has to choose an initial datastructure combination. Our contribution already allows to hide any further configuration and let the user concentrate on the interesting parts. Finding good initial combinations could eliminate the need of datastructure configuration completely.

Make the system more adaptive Currently, we already distinguish list types on a broad level. Being able to use a different datastructure for a graph-global edge list and another one for edge lists assigned to single nodes already takes advantage of the fact that the datastructure performance changes based on its size. For scale-free graphs, it might be interesting to distinguish furthermore between low-degree and high-degree nodes and use different list types for storing edges in a low-degree and a high-degree node, like STINGER already does.

Another field of interest are other datastructures used in the program. Especially metrics that analyze graphs and changes in DNA can be taken into the focus, as they are not only requesting data from the graph datastructure, but also using datastructures internally that are not yet under monitoring or control of our contribution.

Add more datastructures Our contribution makes it easy to add new datastructures to the system, so make use of it! Especially the field of compressed datastructures looks promising to gain even more performance. Probably, there is also the one datastructure that makes profiling and recommending other combinations redundant, as it performs well under all circumstances?

A How to add a new datastructure

This is a short overview about the steps to add a new datastructure. Through the usage of reflection, adding is not that difficult.

- Add a new adapter class in the package `dna.graph.datastructures` for your datastructure
- **Extend** either the class `DataStructure` or `DataStructureReadable` (ee Section6.1)
- **Implement** the interfaces that match the stuff you want to be able to store: `IEdgeListDatastructure`, `IEdgeListDatastructureReadable`, `INodeListDatastructure`, or `INodeListDatastructureReadable`
- Implement all methods
- Add an complexity file containing the runtime and memory complexity costs for your datastructure (see Section 6.1)
- Run benchmarking (see 6.2) – be warned about erroneous recommendations if this is run on another system than the other benchmarks were run on! Mixing data from different systems can lead to recommendations that do not reflect the real performance measurements, as the cost dimensions might differ.
- Add the class name to `dna.graph.ClassPointers` to make it available for recommender and test suite
- Running `dna.tests.DatastructureTester` ensures proper basic functionality. Running more tests (especially `dna.tests.GeneratorsTest` and `dna.tests.BatchTest`) ensures that the datastructure properly interacts with others

B Bibliography

[1] David A. Bader and Kamesh Madduri. "SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks". In: *2008 IEEE International Symposium on Parallel and Distributed Processing* (Apr. 2008), pp. 1–12. ISSN: 1530-2075. DOI: 10.1109/IPDPS.2008.4536261. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4536261.

[2] David A. Bader et al. *STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation*. Tech. rep. Georgia Institute of Technology, 2009. URL: http://cass-mt.pnnl.gov/docs/pubs/pnnlgeorgiatechsandiastinger-u.pdf.

[3] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. "An experimental analysis of a compact graph representation". In: *Workshop on Algorithms Engineering and Experiments (ALENEX)*. 2004. URL: http://repository.cmu.edu/compsci/106/?utm_source=repository.cmu.edu/compsci/106\&utm_medium=PDF\&utm_campaign=PDFCoverPages.

[4] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. "Compact Representations of Separable Graphs". In: (2003).

[5] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. ISSN: 00010782. DOI: 10.1145/362686.362692. URL: http://portal.acm.org/citation.cfm?doid=362686.362692.

[6] Mikhail Dmitriev. "Profiling Java applications using code hotswapping and dynamic call graph revelation". In: *ACM SIGSOFT Software Engineering Notes* (2004), pp. 139–150. URL: http://dl.acm.org/citation.cfm?id=974067.

[7] David Ediger et al. "STINGER: High performance data structure for streaming graphs". In: *2012 IEEE Conference on High Performance Extreme Computing* (Sept. 2012), pp. 1–5. DOI: 10.1109/HPEC.2012.6408680. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6408680.

[8] *Guava: Google Core Libraries for Java 1.6+*. URL: https://code.google.com/p/guava-libraries/.

[9] Changhee Jung et al. "Brainy: Effective selection of data structures". In: *ACM SIGPLAN Notices* (2012), pp. 86–97. URL: http://dl.acm.org/citation.cfm?id=1993509.

[10] Gregor Kiczales, Erik Hilsdale, and Jim Hugunin. "An overview of AspectJ". In: *ECOOP 2001 - Object-Oriented Programming* (2001), pp. 327–354. URL: http://link.springer.com/chapter/10.1007/3-540-45337-7_18.

[11] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Vol. 28. 128. 1974, p. 1175. ISBN: 0201896850. DOI: 10.2307/2005383. URL: http://www.jstor.org/stable/2005383?origin=crossref.

[12] Marc Kramis, Alexander Onea, and Sebastian Graf. "PERFIDIX: a Generic Java Benchmarking Tool". 2007. URL: http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-opus-84446/perfidix.pdf?sequence=1.

[13] Chi-keung Luk et al. "Pin: Building Customized Program Analysis Tools". In: *ACM Sigplan Notices* 40.6 (2005), pp. 190–200.

[14] Kamesh Madduri and David A. Bader. "Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), pp. 1–11. DOI: 10.1109/IPDPS.2009.5161060. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5161060.

[15] Rob McColl et al. "A Brief Study of Open Source Graph Databases". 2013. URL: http://arxiv.org/abs/1309.2675.

[16] David J. Pearce et al. "Profiling with AspectJ". In: *Software: Practice and Experience* 37.7 (2007), pp. 747–777. URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.788/abstract.

[17] Benjamin Schiller and Thorsten Strufe. "Dynamic Network Analyzer - Building a Framework for the Graph-theoretic Analysis of Dynamic Networks".

[18] Ohad Shacham, Martin Vechev, and Eran Yahav. "Chameleon: adaptive selection of collections". In: *ACM Sigplan Notices* (2009). URL: http://dl.acm.org/citation.cfm?id=1542522.

[19] Guoqing Xu. "CoCo: Sound and Adaptive Replacement of Java Collections". In: *ECOOP 2013 - Object-Oriented Programming* 7920 (2013), pp. 1–26. DOI: 10.1007/978-3-642-39038-8_1.