Efficient data structures for dynamic graph analysis

Benjamin Schiller[1], Jeronimo Castrillon[2], and Thorsten Strufe[1]

[1] Privacy and Data Security, Department of Computer Science, TU Dresden
benjamin.schiller1@tu-dresden.de, thorsten.strufe@tu-dresden.de
[2] Compiler Construction, Department of Computer Science, TU Dresden
jeronimo.castrillon@tu-dresden.de

In the era of social networks, gene sequencing, and big data, a new class of applications that analyze the properties of large graphs as they dynamically change over time has emerged. The performance of these applications is highly influenced by the data structures used to store and access the graph in memory. Depending on its size and structure, update frequency, and read accesses of the analysis, the use of different data structures can yield great performance variations. Even for expert programmers, it is not always obvious, which data structure is the best choice for a given scenario. In this paper, we present a framework for handling this issue automatically. It provides compile-time support for automatically selecting the most efficient data structures for a given graph analysis application assuming a consistent workload on the graph. We perform a measurement study to better understand the performance of five data structures and evaluate a prototype Java implementation of our framework. It achieves a speedup of up to 4.7× compared to basic data structure configurations for the analysis of real-world dynamic graphs.

1 Introduction

There is an emerging application domain that deals with the analysis of dynamic graphs. They serve to model dynamic systems across different disciplines, such as biological [6, 19], transportation [7], computer [11], and social networks [14, 5, 21]. Due to a proliferation of applications and the ever increasing size of dynamic systems, performance has quickly become a major concern [17, 9, 10].

The general application pattern of dynamic graph analysis consists of a sequence of graph modifications followed by a computation of metrics. Several metrics investigate local properties such as local clustering coefficient and motif patterns. Other metrics determine global properties like degree distribution, all-pairs shortest paths, and connected components. Such an analysis serves to better understand a system, improve its design, and perform live analysis.

For performance reasons, dynamic graph analysis is implemented on an in-memory graph representation [9, 10]. There are well understood representations of graphs, such as adjacency lists and matrices. There is also plenty of material about algorithms, data structures, and complexity analysis for the different graph representations. For practical applications, however, it is challenging to find the best suited match of algorithms and data structures. In the case of dynamic graphs, the best match depends on many factors, including graph size and structure, frequency of updates to its topology, and access patterns of metric computation. Different graph representations result in high performance deviations but are challenging for programmers to foresee.

There exist many frameworks for the efficient analysis of static graphs [2, 1, 18]. While they are all built for efficient analysis, the graph representation is fixed and selected by the developers. For representing large graphs over time, many graph databases have been developed [20]. While they allow for complex queries of the graph over time and the storage of additional properties, they are neither suited for a large number of updates nor the efficient computation of graph metrics at specific points in time. A lot of work has been done to develop compact representations of graphs. These approaches do not focus on runtime efficiency but on obtaining a small memory footprint [4]. Often, these approaches are not even applicable to arbitrary graphs as they are developed for specific classes only [3, 25]. For dynamic graphs, some special graph representations have also been developed. Their underlying data structures are tuned for memory [17] or runtime efficiency [10] but cannot be adapted to different scenarios.

Many approaches have been developed for profiling programs to allow for their subsequent optimization. Frameworks like *Pin* [16] or *JFluid* [8] allow the instrumentation of existing programs to collect statistics about various aspects. In addition to this instrumentation, *Brainy* [12] allows for the optimization of the data structures used by a program. Based on benchmarks of available data structures, the approach uses machine learning to generate rules like, e.g., *if operation o is called more than k times use data structure d*. After the analysis of a complete execution of the program, data structures are exchanged based on these general rules that are not fitted to the problem of dynamic graph analysis.

Other approaches attempt to exchange data structures during run-time. *Chameleon* [24] provides a framework for run-time profiling without the need to adapt the program. In case the program uses data structure wrappers provided by the framework, data structures can be replaced during runtime which comes at the high cost of performing a live monitoring of all data structures. Based on fixed rules for exchanging data structures as well, *CoCo* [26] requires the programmer to use wrappers provided by the framework in order to optimize the selected data structures during run-time. With their use of pre-defined rules that do not adapt to the current properties of the graph and read accesses of the analysis, both approaches are not suited for the analysis of dynamic graphs.

In this paper, we focus on automatically selecting the best data representation for an application at compile-time. We introduce our terminology in Section 2 and outline our approach and its components in Section 3. We benchmark and discuss the performance of data structures in Section 4, evaluate our approach in Section 5, and summarize our work in Section 6.

2 Terminology & notation

In this Section, we introduce our terminology and notations for graphs, dynamic graphs, and their analysis. We introduce the different adjacency lists and operations on them and define the problem of selecting the best data structures.

Graphs and adjacency lists A *graph* $G = (V, E)$ consists of a vertex set $V = \{v_1, v_2, \dots\}$ and an edge set E. In undirected graphs, edges are unordered pairs of vertices and ordered pairs in directed graph. The *adjacency list* of an undirected vertex is then defined as $adj(v) := \{\{v, w\} \in E\}$. For directed vertices, *incoming* and *outgoing adjacency lists* are defined by $in(v) := \{(w, v) \in E\}$ and $out(v) := \{(v, w) \in E\}$. In addition, the vertices with bidirectional connections are stored in the *neighborhood list*, i.e., $n(v) := \{w \in V : (w, v) \in in(v) \wedge (v, w) \in out(v)\}$.

Dynamic graphs As a *dynamic graph*, we consider a graph whose vertex and edge sets change over time. Each change is represented by an update of V or E that adds or removes an element. Applying any of these updates $add(v)$, $rem(v)$, $add(e)$, and $rem(e)$ implies the modification of V, E, and adjacency lists.

We consider a dynamic graph at an initial point $G_0 = (V_0, E_0)$ and its development over time: $G_0, G_1, G_2, \dots$. The transition between two states G_i and G_{i+1} of the graph can then be described by a set of updates we refer to as a batch B_{i+1}. Then, the complete transition of a dynamic graph over time can be understood as the consecutive application of batches to it: $G_0 \xrightarrow{B_1} G_1 \xrightarrow{B_2} G_2 \xrightarrow{B_2} \dots$.

Analysis of dynamic graphs Analyzing a dynamic graph means to determine its graph-theoretic properties at certain points in time, e.g., for $G_0, G_1, G_2, \dots$. Examples of such graph-theoretic metrics are degree distribution (DD), clustering coefficient (CC), connected components (C), all-pairs-shortest path (SP), and motif frequencies (M). They can be computed using snapshot- or stream-based approaches. We use snapshot-based algorithms in the following since the problem of modifying and accessing the in-memory representation of a dynamic graph is the same for both.

Storing a graph in memory For directed and undirected graphs, different lists are required to represent the graph and all adjacencies in memory. For both types, the set of all vertices V and the set of all edge E must be stored. For each vertex of an undirected graph, we must store the list of all adjacent edges adj. In the case of directed graphs, separate lists of incoming and outgoing edges (in and out) as well as neighboring vertices (n) must be maintained. Hence, there is a total of 6 different list types which we denote as $\mathcal{L} := \{V, E, adj, in, out, n\}$. Each of these lists stores either edges (e) or vertices (v), denoted as $\mathcal{T} := \{v, e\}$. We refer to the data type of a list by $t : \mathcal{L} \to \mathcal{T}$ with $t(V) = t(n) := v$ and $t(E) = t(in) = t(out) = t(adj) := e$.

Each list must provide operations to modify it or retrieve certain information. To create and maintain a list, it must provide means to be initialized (*init*), add elements to it (*add*), and remove existing elements (*remove*). It must provide operations to fetch a specific element using a unique identifier (*get*) or iterate over all elements (*iterate*). Often, it is also necessary to retrieve a random element from a list (*random*), determine its cardinality (*size*), or determine if a specified element is contained in the list (*contains*).

The execution of *add*, *remove*, and *get* can be successful or fail depending on the current state of the list. Therefore, we distinguish between successful (s) and failed (f) execution and consider a set $\mathcal{O}$ of 12 different operations: $o \in \mathcal{O} := \{init, rand, size, iterate, add_{s/f}, remove_{s/f}, contains_{s/f}, get_{s/f}\}$.

Problem definition In this paper, we consider the problem of selecting the most efficient data structures for representing a dynamic graph during analysis in memory. Hence, we must find the most efficient configuration cfg which maps each list type to a data structure: $cfg : \mathcal{L} \to \mathcal{D}$. For undirected graphs, this means to select data structures for V, E, and *adj* while directed graphs require data structures for *in*, *out*, and n in addition to V and E. In the following we focus on undirected graphs since all results can be transferred to directed graphs.

3 Compile-time optimization - our approach

Our approach for optimizing the data structure selection for dynamic graph analysis is based on the assumption that workload and characteristics of the dynamic graph do not change drastically over time. Then, we can estimate the workload for the complete analysis of the graph based on the first batches.

To understand the performance of data structures, we benchmark their runtimes for the execution of all operations beforehand. This preparation phase must be executed only once for a platform where the application should be executed.

To optimize a given application, the compile-time phase uses these benchmarking results (cf. Figure 1): First, a given application is instrumented to enable profiling. Second, it is executed for some batches to record access statistics for all lists. Third, these statistics are analyzed using the runtime estimations obtained during benchmarking to recommend the most efficient configuration. Fourth and finally, the program is re-compiled to use the recommended configuration.

Benchmarking The runtime of executing an operation $o \in \mathcal{O}$ on a list of type $l \in \mathcal{L}$ depends on the element type $t(l) \in \mathcal{T}$, the data structure $d \in \mathcal{D}$ used to implement the list, and its cardinality $size(l) \in \mathbb{N}_0^+$. To approximate this runtime, we perform measurements for data structures and data types with all operations and list sizes $s \in [0, s_{max}]$. As a result, we obtain a function $m_{d,t}$ that maps each operation o and list size s to a set of runtimes: $m_{d,t} : \mathcal{O} \times \mathbb{N} \to \mathbb{R}^k$.

Based on these runtime measurements, we can define an estimation $e_{d,t}$ of the actual runtime for performing a specific operation on a list of given size: $e_{d,t} : \mathcal{O} \times$

$\mathbb{N} \rightarrow \mathbb{R}$ with $e_{d,t}(o, 0) := 0$ and $e_{d,t}(o, s) := m_{d,t}(o, aggr(m_{d,t}(o, min(s, s_{max}))))$. The aggregation $aggr(M)$ can then be defined arbitrarily, e.g., as the minimum, maximum, average, or median of all values $m' \in M$. Minimum and maximum values would result in an optimistic or pessimistic estimation. Since the average value can lead to distortions, we use the median value in the following.

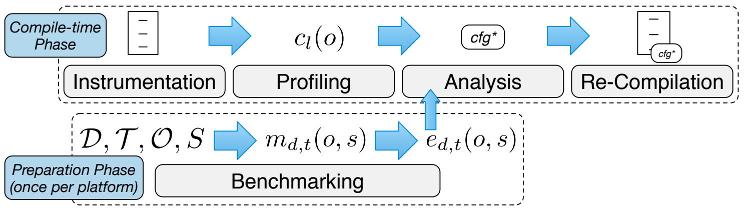


Fig. 1. Compile-time optimization of data structures for dynamic graph analysis

Profiling Two actions are performed during the analysis of a dynamic graph: graph modification and metric computation. Graph modification means that the in-memory representation is changed to reflect the updates that occur in the graph over time. For the computation of metrics, read operations like *iterate* and *contains* are executed on certain lists depending on metrics and algorithms.

During profiling, we record the number of executions of each operation o on a list of type l as $c_l : \mathcal{O} \rightarrow \mathbb{N}$. In addition, we record the average list size as $size(l)$.

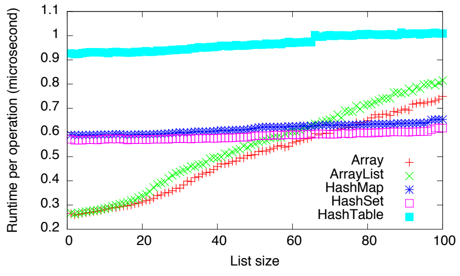
Analysis Given operation count c_l and average cardinality $size(l)$ for a list type l, we can estimate the runtime of any data structure d using the estimation $e_{d,t}$ as follows: $\sum_{o \in \mathcal{O}} c_l(o) \cdot e_{d,t(l)}(o, size(l))$. An estimation of the most efficient data structure $d^* \in \mathcal{D}$ for performing c_l is then given by $d^*(c_l) = \arg\min_{d \in \mathcal{D}} \sum_{o \in \mathcal{O}} c_l(o) \cdot e_{d,t(l)}(o, size(l))$. Hence, the most efficient configuration of all lists l can be estimated as $cfg^*(l) := d^*(c_l)$.

4 Benchmarking, profiling, and analysis results

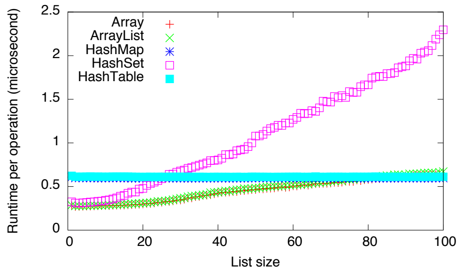
In this Section, we discuss a benchmark of five data structures and give examples of access statistics recorded during profiling and resulting recommendations.

Benchmarking We performed a measurement study to obtain $m_{d,v}(o, s)$ and $m_{d,e}(o, s)$ for sizes $s \in [1, 10^5]$, and five Java data structures that provide the required operations: $\mathcal{D} = \{Array,\ ArrayList,\ HashMap,\ HashSet,\ HashTable\}$. All measurements are executed under Debian with a 64-bit JVM version 1.7.

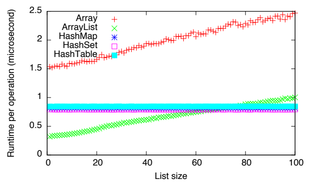
We used implementations of vertices and edges and repeated all measurements 1,000 times. A vertex is identified by a unique index, an edge by the indexes of the two vertices it connects. Selected results for $e_{d,e}$ with $s \in [1, 10^3]$ are given in Figure 2[3]. In addition, the fastest data structure for each operation and list sizes between 100 and 100,000 is given in Table 1.



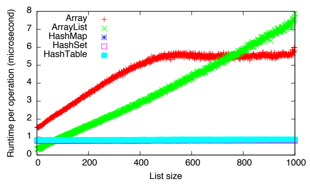
(a) $e_{d,e}(contains_s, s)$



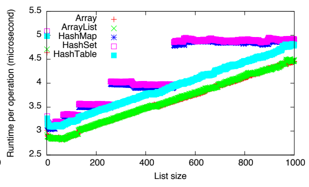
(b) $e_{d,e}(get_s, s)$



(c) $e_{d,e}(remove_s, s)$



(d) $e_{d,e}(remove_s, s)$



(e) $e_{d,e}(init, s)$

Fig. 2. Selected runtime estimations $e_{d,e}$ for list sizes $s \in [1, 10^3]$ and $s \in [1, 10^3]$

The runtime for certain operations differs greatly for data structures and list sizes. For example, *Array* is the fastest data structure for testing the existence of an edge for list sizes ≤ 70 while *HashSet* is the best choice for larger lists (cf. Figure 2a). The retrieval of an edge is fastest on *Array* and *ArrayList* up to list sizes of 80 and faster on *HashTable* and *HashMap* for larger lists while *HashSet* is far slower for all sizes (cf. Figure 2b).

It is interesting to see the extent to which runtimes depend on list sizes which renders an accurate extrapolation impossible. From the runtimes for $remove_s$ on *Array* and *ArrayList* with sizes ≤ 100, is seems intuitive that the runtime grows linearly with the list size in both cases (cf. Figure 2c). Clearly, this is not the case as the runtime for *Array* does not grow linearly for $s > 400$ anymore (cf. Figure 2d). Thereby, *Array* removes edges faster from lists with more than 750 elements. At a first glance, the measurements for *init* appear strange as the runtime of *HashMap* and *HashSet* increase stepwise in contrast to the linear

[3] Measurements for all data structures and larger lists can be found in the technical report: http://bit.ly/1GjiMua

t	s	$init$	add_s	add_f	$rand$	$size$	$iter$	$cont_s$	$cont_f$	get_s	get_f	rem_s	rem_f
v	10	A	A	A	AL	A	AL	A	HS	A	A	AL	A
v	100	A	A	A	AL	A	AL	A	A	A	A	A	A
v	1,000	A	A	A	AL	A	AL	A	A	A	A	A	A
v	10,000	AL	A	A	A	A	AL	A	HS	A	A	A	A
v	100,000	AL	A	A	AL	A	AL	A	A	A	A	A	A
e	10	AL	AL	AL	AL	HM	AL	A	AL	A	A	AL	AL
e	100	A	HM	HS	AL	A	AL	HS	HS	HM	HM	HS	HS
e	1,000	A	HM	HS	AL	AL	AL	HS	HS	HM	HM	HS	HM
e	10,000	AL	HM	HM	AL	AL	AL	HS	HS	HM	HM	HS	HM
e	100,000	HT	HM	HM	AL	AL	AL	HS	HS	HM	HT	HM	HS

Table 1. Fastest data structure depending on operation, type, and size

increase of the other data structures (cf. Figure 2e). The reason is that the number of bits used for the hash function depends on the initial size and thereby the number of hash buckets to initialize grows with $log_2 s$ which explains the steps at list sizes of 2^k.

For storing vertices, *Array* (A) appears to be the fastest data structure overall (cf. Table 1). It performs best for most operations and list sizes $\leq$ 10,000. Only *ArrayList* (AL) appears faster for iterating over the complete list as well as retrieving a random element.

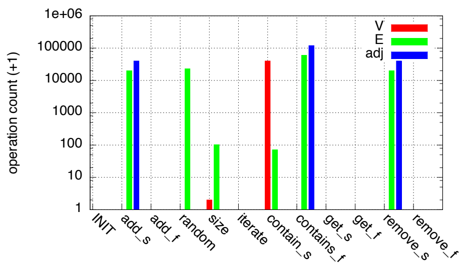
When storing edges, *Array* and *ArrayList* are only fast for small lists of size 10. As the lists grow, the fastest data structure depends on the respective operation and even changes again the more the lists grow (cf. Table 1). For example, *HashMap* (HM) performs best when executing add_s on lists of size $\geq$ 100 while *ArrayList* is fastest for lists of size 10. Similar results can be observed for add_f but for lists of size 100 and 1,000, *HashSet* (HS) is the best choice.

The reason for the differing performance when storing vertices or edges lies in their different identification. Vertices are identified by a unique identifier which can simply be used as the index of *Array* or *ArrayList*. Therefore, performing *contains* or *get* operations translates to a simple lookup at a deterministic location in memory. In contrast, hash-based data structures perform the overhead of looking up this identifier in the corresponding hash table and potentially determining its location in memory. Edges are identified by a hash computed from the two unique indexes of the adjacent vertices. Their lookup in an array-based data structure is time consuming since the complete list has to be scanned. While this is still faster for small lists, hash-based data structures are faster for larger lists as they only need to check for the respective hash in their hash table.

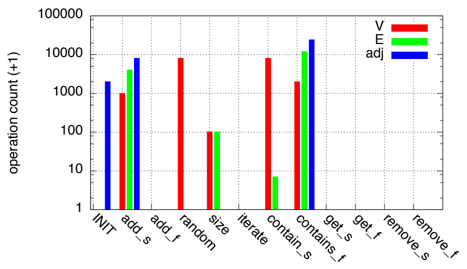
From these results, we assume that array-based data structures should be recommended for storing vertices. Similarly, we see that for storing small edge lists, array-based data structures should be recommended. For larger edge lists with more than 100 elements, there is not a single data structure which appears best. Hash-based data structure perform better than *Array* and *ArrayList* but which one depends on the combination and count of the performed operations.

Profiling We instrumented the graph component of *DNA (Dynamic Network Analyzer)*, a framework for the analysis of dynamic graphs [22], to record c_l and $size(l)$ for all list types $l \in \mathcal{L}$ during graph modification and metric computation using *AspectJ* [13]. Both make up the workload which we analyze and for which we seek the fastest data structures.

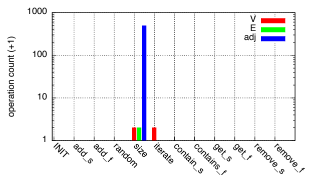
First, we compare c_l for two different workload types of dynamic graphs: static and dynamic workload. We refer to a workload as *static* in case the list sizes do not change significantly over time. In the example shown in Figure 3a, batches are generated by selecting random pairs of edges and exchanging their end-points, i.e., removing two existing edges and creating two new ones. This results in an operation count where random edges are drawn from E and edge removals and additions are performed on E and adj. We consider a workload as *dynamic* in case the list sizes change over time. Such a workload is produced when growing a graph, i.e., adding new vertices and further interconnecting them (cf. Figure 3b). This workload is reflected by add operations on V, E, and adj and the selection of random vertices from V for generating new edges.



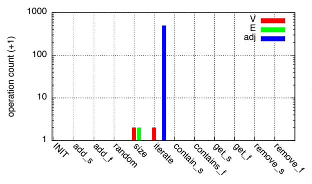
(a) Static workload: edge exchange



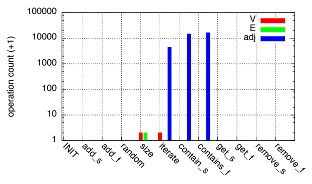
(b) Dynamic workload: growth



(c) Degree distribution



(d) Connected components



(e) Clustering coefficient

Fig. 3. $c_l(o)$ for batch generation/application and metric computation

Second, we observe c_l during the computation of metrics on an instance of a dynamic graph: degree distribution (DD), connected components (C), all-pairs-shortest paths (SP), clustering coefficient (CC), and 4-vertex motifs (M). To compute the degree distribution of a graph, the metric iterates once over V and determines the degree of each vertex using the size operation of its adjacency list adj (cf. Figure 3c). To determine the connected components of a graph, the metric performs a breadth-first search (BFS) by iterating over V and the adjacency lists *adj* (cf. Figure 3d). Similarly, the all-pairs-shortest paths are computed by performing such a BFS from every vertex resulting in a higher count. Computing the clustering coefficient of a graph implies an iteration over all vertices and iterations over all adjacency lists *adj* (cf. Figure 3e). For each neighbor pair, contains operations are executed on the respective neighbor's adjacency list to test if the other neighbor is contained, some are successful, others fail. Similar operations are executed when determining the motif occurrences in the graph. In addition to a higher number of iterations and contains operations, the size operation of *adj* is also executed.

Recommendations Using $e_{d,t}$, we can determine the data structure expected to perform best when executing a set of operations recorded in c_l for a list type l of size $size(l)$. As an example, we estimated the best data structures for computing the five metrics based on the recorded c_l for $size(adj) \in \{10, 100, 1000\}$. For *DD*, *C*, *SP*, and *CC*, array-based data structures are recommended in all cases. Because of the high number of executed *contains* operations performed by *M*, the use of *HashSet* is recommended for list sizes of 100 and 1,000.

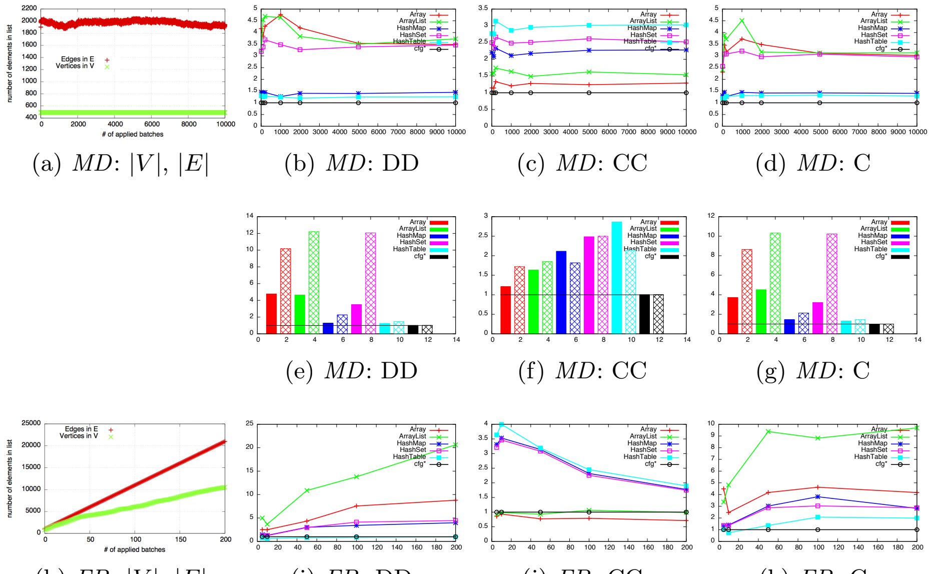
5 Performance evaluation

In this Section, we evaluate our approach on the analysis of two dynamic graphs: one that produces a static workload (*MD*) and a second one that generates a dynamic workload (*FB*). *MD* is the dynamic graph obtained from a molecular dynamics simulation of an enzyme, the para Nitro Butyrate Esterase-13 [23]. The initial graph consists of 491 vertices representing the residues of the enzyme and 1,904 edges. Edges exists between two vertices in case their Euclidean distance is shorter than 7Å. The simulation was performed for 50ns and snapshots taken every 2.5ps. On average, each batch consists of 70 edge additions and 70 edge removals resulting in a static workload (cf. Figure 4a). The *FB* dataset is a friendship graph of Facebook taken from KONECT, the Koblenz Network Collection [15]. It represents users and their friendship relations as a list of edges sorted by the timestamp they appeared. We take the initial graph consisting of the first 1,000 edges and 824 vertices. With each batch, the next 100 edges and corresponding vertices are added creating a dynamic workload. After 200 batches, the graph consists of 10,551 vertices and 21,000 edges (cf. Figure 4h).

For both datasets, we created the initial graph and applied the first five batches. After the application of each batch one of the following metrics was computed: *DD*, *C*, *SP*, *CC*, or *M*. Based on the operation counts c_l of the five batch applications and metric computations, we determine the recommended data structures for V, E, and *adj*.

Then, we perform the same computation with the recommended data structures, as well as configurations where V, E, and *adj* are all using *Array*, *ArrayList*, *HashMap*, *HashSet*, or *HashTable*. For *MD*, we execute between 5 and 10,000 batches. For *FB*, we execute between 5 and 200 batches. For comparison, we compute the runtime of all five configurations relative to our recommended data structures. All results are the average of 200 repetitions.

Static workload (*MD*) For *MD*, our approach recommended the use of *Array* for V, *HashMap* for E, and *ArrayList* for *adj* for all metrics. Since the dataset creates a static workload, we expect that our recommendation is valid independent of the number of batches applied and the speedup achieved compared to basic configurations stays the same.



(a) *MD*: $|V|$, $|E|$ (b) *MD*: DD (c) *MD*: CC (d) *MD*: C

(e) *MD*: DD (f) *MD*: CC (g) *MD*: C

(h) *FB*: $|V|$, $|E|$ (i) *FB*: DD (j) *FB*: CC (k) *FB*: C

Fig. 4. Speedup for *MD* and *FB* depending on total number of batches

For all metrics, we see that the speedup of our recommended data structures compared to basic configurations is independent of the number of batches applied. Our recommended data structures achieve a speedup up to 4.7× and always outperform the five basic configurations (cf. Figures 4b, 4c, and 4d). Using *Array* for all lists achieves the second best performance when computing the clustering coefficient while it is the slowest tested configuration for computing the connected components and the degree distribution. The performance of *HashMap* is contrary: it performs second best for *DD* and *C* but slowest for *CC*.

The fact that our recommended data structures outperform all other tested combinations suggests that our estimation of the actual runtime based on $e_{d,t}$ is accurate and the recommendation valid for all subsequent batches. In Figures 4e, 4f, and 4g, we depict the absolute measured speedup (filled boxes) together with the speedup that we estimated (checked boxes) based on $e_{d,t}$ and c_l. In most cases, our analysis overestimates the speedup by up to 3×. Still, we are able to correctly estimate the order of configuration, i.e., a configuration with a higher estimated speedup also has a higher actual speedup.

Hence, we have shown that our approach achieves speedups over basic data structure configurations in case of a static workload. These recommendations are based on a short profiling phase and the results independent of the duration of the analysis afterwards.

Dynamic workload (*FB*) After profiling for five batches, our approach recommended for *FB* the same data structures as for *MD*: *Array* for V, *HashMap* for E, and *ArrayList* for *adj*. We consider this workload to be dynamic because the sizes of V and E increases with each batch. We expect that this significant change in list sizes renders the initial profiling meaningless for long running analyses. Based on the profiling during the first five batches, we assume a total number of $1,000 + 5 \cdot 100 = 1,500$ edges as input of our analysis. But after 200 batches, E grows to a total of $21,000$ elements, 14× more than the list size we assume based on our initial profiling. Therefore, we expect that the recommendations generated by our approach is not always the best choice, especially in case many batches are applied.

The relative speedup for *FB* depending on the number of analyzed batches is shown in Figures 4i, 4j, and 4k. In some cases, the speedup increases with the number of batches, in others it decreases. For the analysis of the clustering coefficient, even an analysis with 10 batches using recommendations based on 5 batches, the basic configuration with *Array* for all lists is faster. With more batches, this difference increases further.

We have shown that our approach is not able to recommend data structures that achieve better performance than basic configurations in all cases. For dynamic workloads, the speedup changes significantly with the number of batches. Hence, our static optimization is not well-suited for the optimization in case of dynamic workloads.

6 Summary, conclusion, and outlook

In this work, we considered the problem of finding the most efficient data structures for representing a graph for the application of dynamic graph analysis. We proposed a compile-time approach for optimizing these data structures. We performed a measurement study of data structure performance, implemented our approach on top of a Java-based framework for dynamic graph analysis, and evaluated it using real-world datasets. Our results show that our optimization achieves speedups up to 4.7× over basic configurations on real-world datasets.

We observed constant speedups for static workloads, independent of the analysis duration. For dynamic workloads, this speedup changes significantly with the analysis duration. Thereby, our approach is well-suited for improving the analysis of dynamic graphs with a static workload but not capable of adapting to the drastic changes of list sizes that occur in dynamic workloads. Therefore, we propose to investigate possibilities to perform dynamic optimizations that profile an application during execution and allow for hot-swapping data structures.

In future work, we will also investigate the benefits of our approach for larger graphs. Furthermore, we will perform a more extensive parameter study with synthetic workloads and benchmark the internals more comprehensively to gain insights and generalize our results.

References

1. Bader, et al.: Snap, small-world network analysis and partitioning. In: Parallel and Distributed Processing (2008)
2. Batagelj, et al.: Pajek-program for large network analysis. Connections (1998)
3. Blandford, et al.: Compact representations of separable graphs. In: ACM-SIAM Symposium on Discrete algorithms (2003)
4. Blandford, et al.: Experimental analysis of a compact graph representation (2004)
5. Braha, et al.: Time-dependent complex networks: Dynamic centrality, dynamic motifs, and cycles of social interactions. In: Adaptive Networks (2009)
6. Candau, et al.: Structural, elastic, and dynamic properties of swollen polymer networks. In: Polymer Networks (1982)
7. Chabini: Discrete dynamic shortest path problems in transportation applications. Journal of the Transportation Research Board (1998)
8. Dmitriev: Profiling java applications using code hotswapping and dynamic call graph revelation. In: ACM SIGSOFT Software Engineering Notes (2004)
9. Ediger, et al.: Massive streaming data analytics: A case study with clustering coefficients. In: Parallel & Distributed Processing (2010)
10. Ediger, et al.: Stinger: High performance data structure for streaming graphs. In: IEEE High Performance Extreme Computing (2012)
11. Gonçalves, et al.: Characterizing dynamic properties of the sopcast overlay network. In: Parallel, Distributed and Network-Based Processing (2012)
12. Jung, et al.: Brainy: effective selection of data structures. In: SIGPLAN (2011)
13. Kiczales, et al.: An overview of aspectj. In: Object-Oriented Programming (2001)
14. Kossinets, et al.: Empirical analysis of an evolving social network. Science
15. Kunegis: Konect: the koblenz network collection. In: WWW (2013)
16. Luk, et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices (2005)
17. Madduri, Bader: Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In: IEEE PDP (2009)
18. Malewicz, et al.: Pregel: a system for large-scale graph processing. In: ACM SIGMOD International Conference on Management of data (2010)
19. Marti: Dynamic prop. of hydrogen-bonded networks. Physical Review E (2000)
20. McColl, et al.: A brief study of graph databases. arXiv:1309.2675 (2013)
21. Mucha, et al.: Community structure in time-dependent networks. Science (2010)
22. Schiller, Strufe: Dynamic network analyzer - building a framework for the graph-theoretic analysis of dynamic networks. In: SummerSim (2013)
23. Schiller, et al.: Stream - a stream-based algorithm for counting motifs in dynamic graphs. In: AlCoB (2015)
24. Shacham, et al.: Chameleon: adaptive selection of collections. In: Sigplan (2009)
25. Sun, et al.: Compact matrix decomposition for large sparse graphs. In: SDM (2007)
26. Xu: Coco: Sound and adaptive replacement of java collections. In: OOP (2013)