DANF: Approximate Neighborhood Function on Large Dynamic Graphs

Continuously finding changes in node centrality

Master’s thesis in Computer Science

SIMON LINDHÈN
JOHAN NILSSON HANSEN

Computing Science division
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

MASTER'S THESIS 2016:NN

DANF: Approximate Neighborhood Function on Large Dynamic Graphs

Master's thesis in Computer Science

Continuously finding changes in node centrality

SIMON LINDHÈN
JOHAN NILSSON HANSEN

Department of Computer Science and Engineering
Computing Science division
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

DANF: Approximate Neighborhood Function on Large Dynamic Graphs
Continuously finding changes in node centrality
SIMON LINDHÈN
JOHAN NILSSON HANSEN

Company Supervisor: Tobias Ara Svensson, Meltwater
Chalmers Supervisor: Ashkan Panahi, Department of Computer Science and Engineering
Examiner: Peter Damaschke, Department of Computer Science and Engineering

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

DANF: Approximate Neighborhood Function on Large Dynamic Graphs
Continuously finding changes in node centrality
SIMON LINDHÉN
JOHAN NILSSON HANSEN
Department of Computer Science and engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The neighborhood function measures node centrality in graphs by measuring how many nodes a given node can reach in a certain number of steps. The neighborhood function can for example be used to find central nodes or the degree of separation. The state-of-the-art algorithm, called HyperANF (Hyper Approximate Neighborhood Function), can calculate an approximate neighborhood function for graphs with billions of nodes within hours using a standard workstation [P. Boldi, M. Rosa, and S. Vigna, "Hyperanf: Approximating the neighbourhood function of very large graphs on a budget," *CoRR*, vol. abs/1011.5599, 2010]. However, it only supports static graphs. If the neighborhood function should be calculated on a dynamic graph, the algorithm has to be re-run at any change in the graph.

We develop a novel algorithm called Dynamic Approximate Neighborhood Function (DANF) which extends HyperANF to support dynamic graphs. In our algorithm, all relevant nodes are updated when new edges are added to the graph. This allows a constantly updated neighborhood function for all nodes in large graphs. DANF will be used on a real-time data stream supplied by the company Meltwater, where about 2 million news articles are received per day.

Rapidly changing nodes and trends are detected by tracking the nodes whose centrality change by an insertion. This is used to monitor which subjects are getting more or less popular.

Keywords: HyperANF, HyperBall, DANF, node centrality, approximate neighborhood function, neighborhood function, dynamic approximate neighborhood function

Acknowledgements

We want to thank Meltwater for providing us with real-world data, an interesting problem and an opportunity to meet new people working in our area. In particular, we want to thank our supervisor Tobias Ara Svensson, for his professional advice in working methods and helping us to connect to the company systems, and Otto Frost for his valuable input. We also want to thank our supervisor at Chalmers, Ashkan Panahi, for his support and time. Lastly, we want to thank our examiner at Chalmers, Peter Damaschke, for the time and effort to ensure the quality of the report.

Simon Lindhén & Johan Nilsson Hansen, Gothenburg, June, 2016

Contents

List of Figures

List of Tables

1 Introduction

1.1 Aim and background

In the era of big data, mining graphs for information is getting increasingly more popular. An example is to retrieve how central a given node is, which can be measured by node reachability. Node centrality has plenty of real world applications. Consider a graph of companies with edges representing some relations (for example whether they collaborate or not). Node centrality can determine which companies are well established.

The neighborhood function (NF) is used for determining node reachability in graphs. For each node v in a graph, the NF is used to determine how many other nodes v can reach in a limited number of steps. Formally speaking, given a graph $G = (V, E)$, $NF(G, h) = \{|\{v : v \in V, dist(u, v) \leq h\}| : u \in V\}$, where h is the specified maximum number of steps. Due to the dependence on the value h, a variety of graph properties can be expressed by the NF. For example, the diameter of a graph can be expressed as the smallest number d where $NF(G, d) = NF(G, \infty)$. By setting h to a fixed value $h < d$ the fraction of a graph that a node can reach in h steps can be calculated.

The NF can be calculated exactly in either $O(n^{2.38})$ operations and $O(n^2)$ space units or $O(nm)$ operations and $O(m + n)$ space units, where n is the number of nodes and m is the number of edges [1]. However, for very large graphs these polynomial bounds become infeasible. Therefore, recent research has focused on the development of approximation algorithms to calculate the NF for large graphs [1, 2, 3]. HyperANF is a state-of-the-art algorithm, created by P.Boldi et al., that provides an approximation of the NF [2]. HyperANF uses HyperLogLog counters to approximate the number of nodes a given node can reach. HyperLogLog counters are statistical counters that only require $O(\log \log n)$ bits to approximate the cardinality of a multiset up to size n [4]. In HyperANF, each node is given a constant number of counters and hence requires $O(n \log \log n)$ space. The time complexity for HyperANF is not known but it is an extension to ANF which runs in $O((n+m)h)$ operations. A similar time complexity is expected of HyperANF. The authors of [2] provide benchmarks on public data-sets that empirically show that HyperANF is much faster than its predecessors, including ANF. Due to HyperANF's low space complexity, it works well on graphs with billions of nodes. Currently, the algorithm only support static graphs. This paper aims to extend the existing algorithm to supports dynamic graphs. We call our algorithm Dynamic Approximate Neighborhood Function (DANF).

NF for dynamic graphs can be an efficient tool to continuously gather information from very large graphs and see changes in centrality over time. In the above example of companies, the NF can be used to calculate which companies are most central at the moment. If the graph is updated in real time, it is interesting to see how the graph evolves over time as this can give insight into the companies that are growing. Seeing this information might be an advantage for a stock trader, for example. The NF can be used repeatedly to monitor changes as well, but depending on the size of the graph, it can take hours to recalculate. The stock market might have already been closed for the day when the recalculation is finished. Instead, a dynamic NF can provide a continuously updated graph throughout the day with substantially less effort. Then the trader can get updates anytime throughout the day with a very short delay.

1.2 Problem formulation

Let G_i be a directed graph $G_i = (V_i, E_i)$ at time $i = 1, 2, 3, ...$ Let A_i be the set of added nodes in G_i and D_i the set of deleted nodes. At each time step $i > 1$ $G_i = (G_{i-1} \backslash D_{i-1}) \cup A_{i-1}$ and G_1 is the original graph. Given the NF for G_1, our aim is to continuously update the NF for $G_{i>1}$ without performing a complete recalculation of $NF(h, G_i)$. Additionally, the solution should work for very large graphs. Hence, it must be scalable.

The algorithm will be used by Meltwater to find important entities in their data. A graph will be initially built from the data collected by Meltwater. The data comes from sources such as news papers and social networks. Whenever new information is collected, the graph should be updated to maintain relevant information. There are roughly two million new documents received each day and the number of existing documents are in the scale of billions. To be able to handle these magnitudes, the time and space complexity of the proposed algorithm are very limited.

The data stream from Meltwater have some specific properties that should be accounted for in the algorithm: The items are rarely removed from the data. Hence, insertions will be much more common than deletions. The original graph is very large, and the data-stream is relatively small compared to the original graph.

1.3 Limitations

1.3.1 Entity disambiguating

A notorious problem in graph construction is how to disambiguate the nodes. For example, assume that there is a graph consisting of nodes representing individual identities. Suppose that there exists a node with the label Anders Svensson. Next, an article regarding a person with a similar name is added to the graph. Entity disambiguation is required to determine if the article refers to the currently existing Anders or if a new node should be created. Disambiguating entities is important in the construction of the graph, as false results are likely to occur otherwise. Meltwater has an NLP-department which already disambiguates entities with a good performance. Therefore, it is assumed that the entity disambiguation, already in place by Meltwater, is correct.

1.4 Mathematical preliminaries

1.4.1 Graph transpose

The transpose of a graph is a graph with all of its edges flipped. A function T calculates the transpose of a graph G in the following manner: $T((V, E)) = (V, \{(u, v) : (v, u) \in E\})$.

1.4.2 Maximal matching

Given a graph $G = (V, E)$, a matching is a subset $M \subseteq E$ such that all nodes are incident to at most one edge in M. A maximal matching is a matching such that if another edge is added to it, it is no longer a matching.

1.4.3 Approximation algorithms

Approximation algorithms are useful to calculate a non-optimal answer quickly where the optimal solution would take too long to calculate. A C-approximation means that the solution is in worst case C times the size of the optimal solution. For example, if you calculate the minimal vertex cover with a 2-approximation, the solution can be at most twice as large as the optimal solution. If you want to calculate the maximal matching with a $\frac{1}{2}$-approximation the size of the solution can be no less than half of the optimal solution.

1.4.4 Minimum vertex cover

Given an undirected graph $G = (V, E)$, a vertex cover is a subset $S \subseteq V$ such that for all edges $e = (u, v) \in E$, $u \in S \lor v \in S$. A minimum vertex cover is a set S of minimum size. The problem of finding a minimum vertex cover is NP-complete [5].

2

Technical Background

2.1 Neighborhood function

The individual neighborhood function (IN) takes a directed and unweighted graph $G = (V, E)$, a node $v \in V$, and a number of steps $h \in \mathbb{N}$, and returns the number of nodes v can reach in h steps in the graph, that is, $IN(G, v, h) = |\{u|u \in V, dist(v, u) \leq h\}|$, where $dist(v, u)$ is the length of the shortest path from v to u. A brute force algorithm for computing $IN(G, v, h)$ exactly is to perform a breadth-first search that saves every encountered node within h levels. The cardinality of the resulting set will represent v's individual neighborhood.

The neighborhood function (NF) returns the individual neighborhood function for all nodes in the graph. Formally speaking, the neighborhood function is defined as: $NF(G, h) = \{IN(G, v, h)|v \in V\}$.

2.2 HyperANF

HyperANF is an algorithm which calculates the approximate neighborhood function for a graph. The algorithm works in the following way: In the first iteration, let the set of reachable nodes from each node v be $R_0(v) = \{v\}$. Then, for $1 \leq i \leq h$ iterations, let each node take the union of all its neighbors sets $R_i(v) = R_{i-1}(v) \bigcup\limits_{u \in s(v)} R_{i-1}(u)$ where $s(v) = \{u|(v, u) \in E\}$. After h steps every node's reachability set $R_h(u)$ will contain the nodes it can reach in h steps [6]. However, instead of keeping track of the set of reachable nodes, HyperANF uses one HyperLogLog counter per node to approximately count the size of the set of the reachable nodes. The benefit of calculating the set sizes approximately is that it only requires $O(n \log \log n)$ units, in contrast to the exact algorithm which requires $O(n^2)$.

2.3 HyperLogLog

HyperLogLog [4] is the state-of-the-art cardinality estimator of the multisets that are received as a data stream. A multiset M is an arbitrarily ordered set that allows duplicate elements and the cardinality is the number of distinct elements. HyperLogLog takes a multiset in the form of a data stream S and estimates the cardinality.

An intuitive algorithm for calculating cardinality exactly is to save each element received in the data stream into a set S. Then, the cardinality will be $|S|$. With this intuitive algorithm $O(n)$ space is needed to calculate the cardinality of a set with size n. For various applications, this space complexity is not sufficiently low. Therefore, space saving approximation algorithms, such as HyperLogLog, have been developed. HyperLogLog has a space complexity of $O(\log \log n)$. In theory, there exist an algorithm that achieves the optimal asymptotic space and time usage of a cardinality estimator [7]. However, that algorithm is complex and its implementation is impractical [8].

HyperLogLog works by hashing the elements seen in the data stream and storing the maximum number of leading consecutive zeroes in the bits of the hash. The intuition of the algorithm is that hashes with few leading zeroes are more likely to appear than hashes with many leading zeroes. A crucial assumption of the algorithm is that the hashed elements are evenly distributed across the target domain. Hence, every bit of the hash sequence has an equal probability of being zero or one. The probability of only one consecutive zero is $\frac{1}{2}$, the probability of two zeroes is $\frac{1}{2} * \frac{1}{2}$ and the probability of observing k zeroes is $\frac{1}{2^k}$. Assume that $p-1$ consecutive zeroes and a one have been seen. The probability of this outcome is $\frac{1}{2^p}$. The expected number of tries for this probability to occur is 2^p and it can be concluded that approximately 2^p unique elements exist in the multiset [4].

2.3.1 Multiple register

In HyperLogLog counters, registers are the locations where the number of trailing zeroes is stored. If only one register is used, the precision is very bad. The precision can be drastically improved by having several registers and later calculating the mean of the estimates. As a single hash value may be observed multiple times, any given hash must always use the same register so that a number that has been observed and added once before does not affect the cardinality. To ensure this, the first b bits of the hash are used as a register index. This means that the number of registers will be 2^b. By using more registers, a higher precision is achieved, but the time and space requirement is increased. The space complexity is $O(2^b \log \log n)$ [4].

The algorithm with multiple registers works as follows: Given a number x as input; Run the hash function on x so that $x_h = hash(x)$. Remove the b least significant bits from x_h and call them r. Let the number of consecutive trailing zeroes in the remaining bits of x_h be called c_0. If the number in register r is less than $c_0 + 1$: store $c_0 + 1$ in register r [4].

2.3.2 Terminology

Added Adding an element to a HyperLogLog counter means to hash the element, determine the register r, count the consecutive leading zeroes and store that number in r if it is larger than the existing value.

Included An element is said to be included in a HyperLogLog counter if it has been added to it.

Union The union of two HyperLogLog counters is the same as including all elements included in at least one of the two counters into one counter. This is simply performed by taking the maximum of each saved value [2].

Subcounter A counter A is a subcounter of B if all elements included in A are included in B. Subcounter is denoted by $\subseteq$ and $\subset$.

2.3.3 Memory usage

Given that the actual cardinality of the input is n, the hash function needs to target a domain of size at least n. This reduces the number of hash collisions. Only the number of leading consecutive zeroes in the hash is stored. The maximum number of zeroes in the hash sequence is the number of bits required to represent a number n, which is equal to $\log_2(n)$. For example, if $n = 64$, the number to store is at most 6. To store a number of size $\log_2(n)$, $\log_2 \log_2(n)$ bits are needed. If $n = 64$ and the max number to store is 6, only 3 bits are needed. If $n = 10^9$ only 5 bits are needed. This leads to the space complexity of $O(\log \log n)$ for this algorithm [4].

2.3.4 Harmonic means

The harmonic mean of all the registers is used to calculate the cardinality. An important property of harmonic means is that extreme values have less leverage than small ones. This property is mathematically written as $\min(x_1, x_2, ..) \leq H(x_1, x_2, ..) \leq n * \min(x_1, x_2, ..)$ where n is number of elements. This prevents the mean to spike off when there are a few extreme values. The harmonic mean is defined as: $\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}}$

2.3.5 Precision

HyperLogLog is a probabilistic algorithm and understanding its performance require a deep analysis. Only the final conclusions are presented below. For a full proof see [4].

Let E be the estimate produced by the mean of all the registers, $\mathbb{E}_n(E)$ be the expectation of E with an unknown cardinality n, and m be the number of registers used per counter. Assuming $m \geq 16$, the relation $\frac{1}{n} * \mathbb{E}_n(E) = 1 + \delta_1(n) + o(1)$ holds. δ_1 is a small oscillating function with $|\delta_1(n)| < 5 * 10^{-5}$, and has no significant impact on the expected value. This indicates that E is an asymptotically almost unbiased estimator.

Flajolet et al. also provides the standard error [4]. Let $\mathbb{V}_n(E)$ be the variance of E with an unknown cardinality n. Assuming $m \geq 16$, $\frac{1}{n} * \sqrt{\mathbb{V}_n(E)} = \frac{\beta_m}{\sqrt{m}} + \delta_2(n) + o(1)$. δ_2 is an oscillating function with $|\delta_2(n)| < 5 * 10^{-4}$, and β_m is a constant only depending on m. For $m = 16$, $\beta_m = 1.106$ and for increasing m, β_m decreases asymptotically toward 1.03896. It follows that the precision of each sample of the counter depends on the number of registers in the counter. By choosing a large number of registers a better precision is achieved. Chassaing and Gérin proved a lower bound for this approximation problem to be $\frac{1}{\sqrt{m}}$. This shows that this algorithm is nearly optimal [9]. Flajolet et al. compares the precision of HyperLogLog and exact measures to confirm that the standard error is low [4].

2.3.6 Hashing function

For the HyperLogLog algorithm to work, it is necessary that every bit in the hash value have equal probability of occurring [4]. This can be achieved by a hashing function with good avalanche property. The avalanche effect is when changing a single bit in the input leads to a very different output. In addition, for HyperLogLog to be useful for data streams with very large cardinality the hashing function must also be fast. HyperBall, an implementation of HyperANF, uses the hashing function Jenkins [10]. Experimental results show that the Jenkins hash exhibits a good avalanche property. It is also very fast to compute as it only uses a few clock cycles for start up and about one clock cycle per three bytes of input.

2.4 Webgraph

Webgraph is a framework for compressing and analyzing very large graphs [10]. It is developed by Boldi and Vigna, two of the developers of the HyperANF algorithm.

2.4.1 HyperBall

HyperBall is an open-source framework that utilizes the HyperANF algorithm for performing computations related to the neighborhood function. It is a part of the webgraph framework.

2.4.2 Graph compression in Webgraph

The webgraph framework [10] uses many different ways to compress a graph but its principal method is the format BVGraph [11]. It exploits common relational properties of the websites when sorted in lexicographic order and given an index based on their position in the order. A relation from page A to page B exists if there is a hyperlink from page A to B. One common property is that most hyperlinks on websites are navigational, i.e. leads to a different page on the same website. So, if the links are sorted in lexicographic order, many node indices will be close to each other. This property is called *locality.* Another common pattern is that, as most links are navigational, many nodes close to each other have many common neighbors. This property is called *similarity.* The navigational links often refer to pages far down in the site hierarchy and all of these pages will be right beside each other because of the long mutual address prefix. Hence, many of the neighbors form long consecutive sequences of one-increasing indices, a property called *consecutivity.*

To take advantage of *locality*, the neighbors are encoded as the difference from the neighbors preceding them in the neighbor list. This makes the encoded numbers significantly smaller and can take up less space. Another compression method called copy-blocks exploits the *similarity* property and copies neighbors from a node nearby. This saves a lot of space if two nodes have almost the same neighbors. The *consecutivity* property is used by representing large portions of intervals by two numbers: the start node and the interval length [11].

2.4.2.1 Layered Label Propagation

While the properties mentioned above are trivially achieved in a graph of hyperlinks, they might be harder to find in other types of graphs. To be able to use the compression techniques in other types of graphs P. Boldie et al. have developed a method called LLP (Layered Label Propagation) [12] to reorder the node indices to use the compression techniques more efficiently.

2.5 Breadth-first search

Breadth-first search (BFS) is an algorithm which traverses a graph from a given origin node v. It starts by adding the node v to a queue. Then, while the queue is not empty, the algorithm takes the first node u from the queue and adds the previously not seen neighbors of u to the queue. In the first step, v will be visited. In the second step the neighbors of v will be visited, and so on until all reachable nodes are traversed.

2.5.1 Multi source

There exist algorithms that optimize the calculation of several BFS's simultaneously. These optimizations mainly target the ability of the different searches to use each others' computations. The state-of-the-art algorithm for running multiple breadth-first searches on the CPU is called Multi Source Breadth-First Search (MS-BFS) [13].

MS-BFS works by propagating a set of BFSs that have reached the same node at the same time. To begin with, one set is created for every BFS at their respective source node. Each sets contains only the BFS that starts at its corresponding source node. In the next iterations, all sets are propagated to the neighbors of the node they were previously on. If several sets reach a node, the sets are merged. Additionally, the nodes that the BFSs have already reached are tracked and any BFS that reaches a node it has already seen is ignored.

This means that once two breadth first searches meet, they can be considered as one. This decreases the amount of necessary propagations compared to the individual BFS case. The sets of BFSs are lists of bits where the indices of the bits represent the id of the BFSs [13].

2.6 Approximate minimum vertex cover

As minimum vertex cover is NP-complete there have been studies of polynomial time approximation algorithms. The state-of-the-art algorithms give 2-approximations. For general graphs it is hard to achieve a $(2-\epsilon)$-approximation for any $\epsilon > 0$ [14]. Hence, 2 can be the best constant approximation factor achievable.

A simple greedy 2-approximation algorithm maintains a maximal matching M to calculate a minimal vertex cover V. By the definition of maximal matching it is certain that every edge is either in the maximal matching or shares a node with one in it. For all edges $e = (u, v) \in M$, pick both u and v for the vertex cover, and all edges are guaranteed to be covered. For an optimal minimum vertex cover V_{opt}, it holds that for all edges $e = (u, v) \in M, u \in V_{opt} \lor v \in V_{opt}$. This can be proved by a simple contradiction. Assume $u, v \notin V$. This implies that e is not covered by any node, hence V is not a vertex cover $\blacksquare$. Therefore, by picking both nodes for all edges in the maximal matching, a 2-approximation algorithm is achieved.

2.6.1 Dynamic approximate minimum vertex cover

There exists several different algorithms for maintaining a fully dynamic approximate vertex cover [15, 16, 17]. A fully dynamic algorithm supports both insertions and deletions. The main difference between the algorithms is the trade off between the time complexity for insertions and deletions. The state-of-the-art fully dynamic approximate minimum vertex cover algorithms can achieve 2-approximations.

3
Methods

The approximate neighborhood function has been studied by P. Boldi et al. in the webgraph framework [10]. As this framework only supports static graphs, we made extensions to support dynamic ones and an evolving approximate neighborhood function. Our research was mainly focused on the compression techniques of webgraph graphs [11], the HyperANF algorithm [2] and the HyperANF implementation HyperBall [6], including HyperLogLog counters [4]. We also studied general dynamic algorithms.

3.1 Merging graphs

The existing compression methods in webgraph are very sensitive to change. Nodes can copy neighbors from other nodes so that if the neighbors of a node change, all nodes that refer to the modified node have to be found and updated as well. Moreover, the graph is either stored in a file or in a byte buffer. None of these methods support insertion of data at an arbitrary position without expensive reallocation and repositioning of the data succeeding that position. As dynamic graphs are required, we investigated optimal methods of merging two graphs.

3.2 Extending HyperANF to support dynamic graphs

The proposed algorithm is divided into two phases. In the first phase, the proposed algorithm runs the HyperANF on the initial state of the graph. The second phase is the dynamic part, which starts immediately after the HyperANF is completed. In the second phase, the algorithm takes the information produced by the HyperANF and modifies it when new nodes and edges are added or deleted. To be able to share computations, the algorithm is optimized to handle several edge insertions in a bulk.

3.3 Extending HyperLogLog to support deletions

Support for deleting nodes and edges is desired, but requires a method to remove elements from the HyperLogLog counters. We studied methods to delete elements from HyperLogLog counters. However, we observed that a satisfactory result required a more careful analysis, which was deemed out of scope. This seems like a drastic choice to make, but many data-streams are of the type "append-only", in which the resulting graph only allows node and edge additions.

An alternative way of supporting deletions by making some recalculations was investigated. Due to limited time it has not been implemented. The method is mentioned as a suggestion for future study in section 6.2.

3.4 Benchmarks

Several benchmarks on different parts of the algorithm have been performed to ensure that the methods most suitable to our needs are used. The benchmarks included comparing both time and memory usage. On several occasions, a trade-off between memory and computation speed had to be chosen. Speed is the first priority in this study, although the storage has to be kept reasonable.

The proposed algorithm was compared to HyperANF. The comparison was done by inserting different amounts of edges into DANF, while recalculating HyperANF. Different sized graphs are tested to analyze how the comparison scales. For better accessibility, the data-sets used to perform the benchmarks will be publicly available.

4
Development

To extend the HyperANF algorithm, we will use the HyperANF implementation in HyperBall. Initially, the algorithm takes an existing non-empty directed graph as input. This graph can be seen as static and therefore HyperANF can perform a complete calculation on it. After HyperANF is complete, the calculated counters can be used as the initial counters for the dynamic algorithm. For every insertion and deletion in the dynamic graph, these counters are manipulated. If an edge is added, nodes might reach more nodes and their counters should be raised. Similarly for deletions, if an edge is deleted, nodes might reach fewer nodes and their counters should be decreased.

4.1 Preliminaries

4.1.1 Dynamic graphs

To achieve a dynamic graph, the ability to insert new nodes and edges (called entries in this section) to an existing graph is needed. The graph files used by HyperANF are tightly compressed in a byte stream, which makes it an expensive operation to modify the graph at an arbitrary point. This makes it infeasible to insert every additional entry directly into the graph file. Instead, new entries can be stored in some other data structure.

The additional entries can either be stored in a high-level data structure or a byte stream. Keeping them in a high-level data structure will consume a significant amount of storage already at a low number of extra entries. A high-level data structure takes up more memory as each number is always the same size, no matter what the value is. For example, the value 3 can be expressed by two bits but a high level integer always take a fixed amount of bits to express it. Additionally, if each node has a structure containing the neighbors, a significant amount of space will be used by the pointers to those structures.

If a byte stream is used, the same problem as the original one occurs. Even though this byte stream will be smaller, there will be a point where it is unreasonable to resize the stream and reposition the existing entries at each new entry. Regardless of which method is used, the extra entries and the original graph eventually have to be merged.

4.1.1.1 Merging two graphs with webgraph

The webgraph framework has a method for creating a new graph G by taking the union of two subgraphs G_1 and G_2. The graph G works by simultaneously reading from both streams G_1 and G_2 to produce the nodes and their neighbors. The sub graphs can be unions of other graphs as well. This means that an arbitrary number of graphs can be joined using this method. However, having lots of recursive unions create a large overhead in time when fetching nodes and edges [10].

The framework also supports storing a union-graph to disk. This removes the overhead of any union. As the graph express some arcs by references to previous arcs, storing is slow. Ignoring the references makes the graphs larger but it increases the running time of the storeing step.

4.1.1.2 Memory dependent merging

To leverage the benefits of both high level structures and byte streams, we use a mixture of both to achieve dynamic graphs. The original graph is kept as a byte buffer and the additional entries are saved in a high level structure. The high level structure was chosen as additions are faster than in a byte stream graph. We prioritize fast edge additions as the addition time has a larger impact on the overall performance than the iteration time (see Section 5.1.1). Each new bulk of edges is inserted into the high level structure. The union method of webgraph can then be used to union the high level structure and the original graph, creating a dynamic graph.

The overhead of recursive unions can be avoided by limiting the number of unions to one. By inserting each bulk insertion into the same high level structure, and keeping the original graph separated, the current union can be replaced by a new one. However, the problem of high memory overhead when using a high level structure still remains. Our solution is to merge the graphs when the additional entries' memory usage, compared to the byte stream graph, reaches a certain ratio. At this time, the unioned graph is stored to disk. The graph is then loaded as a byte stream and the additional edges become part of the original graph. This resets the memory overhead of the high level data structure. Both the time and memory usage of this technique are nearly optimal (see Section 5.1.2).

4.1.2 HyperLogLog resizing

After new nodes are added to the graph, new counters have to be created for them. The initial counters retrieved by HyperANF are represented as an array of 64-bit numbers. This representation is kept unchanged as it minimizes the amount of wasted space usage. It also minimizes the number of objects created on the heap. However, as new counters are needed, the array must be resized. We use a geometric expansion which gives $O(1)$ amortized time per insertion [18]. Amortized time is the average time spent on each operation. Geometric expansion means that when an element cannot fit the array, the length of the array is increased by a factor. This is how the standard implementation of ArrayList in Java performs dynamic increase. A larger factor implies a small average number of copies per inserted element, but a high amount of unused space. The amortized number of copies per inserted element for a factor r is roughly $\frac{1}{(r-1)}$ and the average unused space is $\log(r) * \frac{r}{(r-1)} - 1$ [18].

Under the assumption that the graph expands slowly relative to its size, resizing events will be sparse and a large factor implies a large amount of unused memory. Therefore, it is more beneficial to let the array increase by a small factor. With the factor $r = 1.1$, an average of 4.8% space is unused and every element will on average be copied 10 times.

4.2 DANF: The first attempt

The following non-optimized version of DANF is the intuition of the algorithm. It will be refered to as the two-BFS algorithm. The two main phases of our algorithm, the collecting BFS and the propagation BFS, are introduced.

After an edge is added to the graph, we update the HyperLogLog counters of all nodes affected by insertion. Figure 4.1 is used as an example. The thin arrows are the edges. The circles are the nodes included in the counters produced by the collection step. The bold arrows represent the propagation step. Let $e = (2, 4)$ be an edge added to the graph. The data with all the nodes reachable from 4 has to be propagated to the nodes that can now reach 4 via 2. This is performed in two phases: collecting BFS and propagating BFS.

The collecting BFS is responsible for creating a HyperLogLog counter for each level in the search and add all nodes in the frontier to the counter. The collecting BFS is performed from the to node of the added edge. The nodes included in each level's HyperLogLog counter are visualized as the circles in Figure 4.1. These counters represents all nodes that 4 reaches in l_4 steps. The collecting BFS stops at level $h - 1$ and returns an h-long list of counters. The h'th step is not needed as not even 2 can reach level h.

The propagating BFS is responsible for updating the counters of all nodes that now reach 4, as they might have been changed by the insertion. To find all nodes that can reach 4, a BFS is performed from 2 in the transpose of the graph. For every level l_2 in the BFS, the frontier of nodes in the BFS can reach all nodes that the collecting BFS reached in $h - 1 - l_2$ steps. To update the counters of the nodes at level l_2, the current counters of the nodes are merged with the counters from collecting steps $l_4 \leq h - 1 - l_2$ retrieved in the collecting BFS. The propagation BFS stops after $h - 1$ steps, when all affected node counters are updated.

After the two-BFS algorithm has completed an update in the graph, the collected counters are thrown away as they might be different in the next update.

We will now prove the correctness of the two-BFS algorithm:

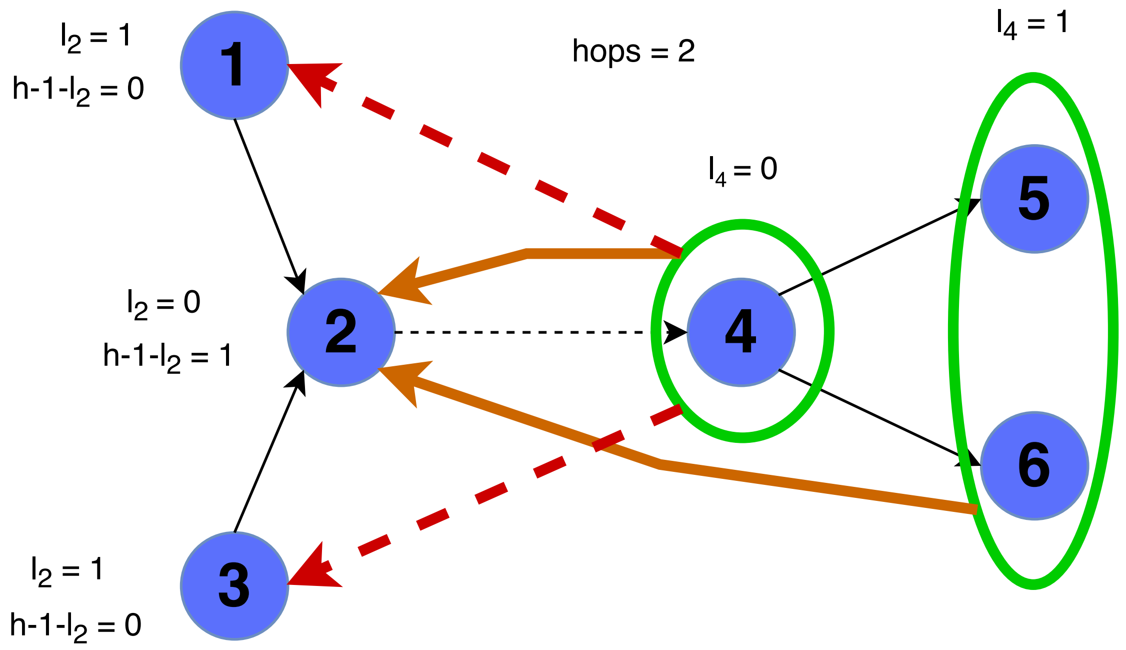

Figure 4.1: Visualization of the collect and propagate steps

Theorem 4.2.1. *Given that the two-BFS algorithm and HyperANF use the same hashing function, the two-BFS algorithm yields node counters identical to a HyperANF recalculation.*

Proof. Let $S_{\text{alg}}(v)_i$ for $alg \in \{\text{2bfs}, \text{hanf}\}$ be the set of nodes that the algorithm "alg" includes in the counter of v after i edges have been inserted. Let $N_{\text{alg}}(v)_i = S_{\text{alg}}(v)_i \backslash S_{\text{alg}}(v)_{i-1}$.

Base case: $i = 0$) As the algorithm uses HyperANF to calculate the initial counters,

$S_{\text{2bfs}}(v)_0 = S_{\text{hanf}}(v)_0$.

Inductive hypothesis: $\forall i \leq n : S_{\text{2bfs}}(v)_i = S_{\text{hanf}}(v)_i$.

Inductive case: $n+1$) $S_{\text{hanf}}(v)_{n+1} = \{u : \mathit{dist}(v,u) \leq h\}$ and $S_{\text{2bfs}}(v)_{n+1} = S_{\text{2bfs}}(v)_n \cup N_{\text{2bfs}}(v)_{n+1}$. Let the added edge be $e = (v,u)$. In the propagation step at a node z of depth $\mathit{dist}(z,v)$ all counters from the collecting step of depth $\leq h-1-\mathit{dist}(z,v)$ will be unioned with the counter of z. As z is $\mathit{dist}(z,v)$ steps away from v, $\mathit{dist}(z,u) = \mathit{dist}(z,v)+1$. This means that z can reach $h-(1+\mathit{dist}(z,v))$ steps into the newly reachable nodes. So z will get all new nodes $\leq h$ steps away, hence $N_{\text{2bfs}}(v)_{n+1} = N_{\text{hanf}}(v)_{n+1}$. By the inductive hypothesis: $S_{\text{2bfs}}(v)_{n+1} = S_{\text{2bfs}}(v)_n \cup N_{\text{2bfs}}(v)_{n+1} = S_{\text{hanf}}(v)_n \cup N_{\text{hanf}}(v)_{n+1} = S_{\text{hanf}}(v)_{n+1}$. As both algorithms use the same hashing function, and the same nodes are included in the counters, the resulting node counters will be identical.

$\blacksquare$

4.3 Optimized Breadth-first search

A problem with the two-BFS algorithm described above is that it only supports single insertions at a time. By bulking the edges, several BFS's can be performed at the same time to speed up the insertion time.

The algorithm MS-BFS [13] does several BFSs at the same time very effectively. MS-BFS is magnitudes faster than a parallel standard BFS implementation and hence will be used in the algorithm.

4.3.1 Visitors

In order to prune paths in the BFS, there needs to be a way to stop the individual BFSs. We developed a method to pass to the MS-BFS a visitor function (not related to the common Visitor design pattern). The visitor will be called every time a group of BFSs reach a node. The group of BFSs reaching the visited node is represented by a list of bits and is passed to the visitor. If the visitor wants to stop the propagation of a BFS from this node it can clear the corresponding bit. No modifications to the algorithm needs to be done, other than adding the visitors, as MS-BFS already use a list of bits to propagate the BFSs.

4.3.2 Travelers

We developed a new concept called travelers. The purpose of the traveler is to bring data along with BFSs which is passed to the visitors. This removes the need to loop through which source nodes reached the visited node. Travelers have to be able to merge when several BFSs reach a node to avoid increasing the space complexity of the algorithm.

4.3.3 Extended algorithm

The extended algorithm is presented in Figure 4.2. It has only slight modifications compared to the algorithm in section 4.1.1 from [13]. The visitor is called on line 16. The neighbors are not inspected if the visitor returns an empty set. The travelers are merged at lines 19 to 22. If the neighbor will be visited from other nodes, i.e. the $visitNext$ bits are not empty, the neighbor's traveler is merged with the traveler from the current node.

```
Input: $G = (N, E)$ a graph containing $N$ nodes with edges $E$
       $S$ = BFS source nodes
       $V$ = Visitor function
       $T$ = Travelers
for each $s_i \in S$
    $seen[s_i] \leftarrow 1 << i$
    $visit[s_i] \leftarrow 1 << i$
for each $t_i \in T$
    $travelers[s_i] \leftarrow t_i$
reset $visitNext$
reset $travelersNext$

while $visit \neq \emptyset$
    for $i = 1, ..., N$
        if $visit[i] = \emptyset$, skip
        $visit[i] \leftarrow V(i, visit[i], travelers[i])$
        if $visit[i] = \emptyset$, skip
        for each $n$ where $(i, n) \in E$
            $traveler \leftarrow travelers[i]$
            if $visitNext[n] \neq \emptyset$
                $traveler \leftarrow traveler.merge(travelersNext[n])$
            $travelersNext[n] \leftarrow traveler$
            $visitNext[n] \leftarrow visitNext[n]$ | $visit[i]$

    for $i = 1, ..., N$
        if $visitNext[i] = \emptyset$, skip
        $visitNext[i] \leftarrow visitNext[i]$ & $\sim seen[i]$
        $seen[i] \leftarrow seen[i]$ | $visitNext[i]$
    $visit \leftarrow visitNext$
    reset $visitNext$
    $travelers \leftarrow travelersNext$
    reset $travelersNext$
```

Figure 4.2: Extended MS-BFS algorithm in pseudo-code

4.4 DANF: The final algorithm

4.4.1 Node history

MS-BFS speeds up the algorithm significantly, but further optimizations can be done to the individual BFSs. To update a counter, the algorithm has to do two BFSs for every edge added. A large portion of the time spent by the algorithm will be consumed by these BFSs. Therefore, the ability to prune BFSs early implies significant time improvement. So far, the collecting BFS must traverse the graph h steps to gather all data needed. This is because every node only keeps track of its registers, resulted by a search in h steps. If all nodes also keep track of their registers in $h-1$ steps, then the collecting BFS need to traverse $h-1$ steps. This as in the $h-1$'st step, the collecting BFS can use the $h-1$ history of all nodes to calculate the h'th step. For every extra history level added, the collecting BFS can stop one step earlier. By saving every history level of all nodes, the collecting BFS only needs to visit the immediate neighbors of a new node v to be able to calculate the HyperLogLog counter of v.

During the HyperANF phase of the algorithm, the intermediate counters in HyperANF can be used to calculate all history levels of all nodes. HyperANF works in iterations. During the i'th synchronization, the counters will represent the i'th level of all nodes' history. So, instead of only using the last level from HyperANF, we save all levels. The h'th level we refer to as the top counter and level 0 to $(h-1)$ as history counters. All levels combined will be referred to as counter collection.

Now, every node has its history which has to be updated with every edge update. For an edge $e=(u,v)$ inserted, the collecting BFS has to gather the node history from all neighbors of v and union them into one. Then, the propagation BFS has to propagate this history in the transpose of the graph.

With node history, the algorithm has $O(hn \log\log n)$ space complexity, as the top counter uses $O(n \log\log n)$ and the history counters use $O((h-1)n \log\log n)$. To insert an edge $e=(u,v)$, the two previous BFSs use $O(2m)$ operations but with node history the collecting BFS uses $O(hd^+(v))$ operations, where $d^+(v)$ is the out degree of node v. It is dependent on h as it has to take the union of each history counter. The time complexity of the node history version is then $O(m + hd^+(v))$. In practice, this means a large improvement as it often holds that $hd^+(v) << m$.

Node history speeds up the algorithm but also uses extra space that, for large graphs, can be quite extensive. To create an algorithm that balances the gained speed versus the extra memory usage, the history can be saved for only a subset of nodes. These nodes should be chosen so that as few nodes as possible need to save their history while keeping the BFS distance as short as possible. The algorithm to determine these nodes has to be very fast to avoid affecting the running time of the overall algorithm. Moreover, it has to be dynamic in order to continuously determine the nodes included in the set.

4.4.1.1 Vertex cover

We realized that saving the history of the nodes that are in a vertex cover (VC) can significantly improve space usage, yet the BFS can still be bounded by at most two steps.

The top counter still needs to contain counters for all nodes, so that the space complexity of the top counter remains $O(n \log\log n)$. As the history counters only consist of counters of nodes in the VC, the history counters' space complexity is reduced from $O((h-1)n \log\log n)$ to $O((h-1)|VC| \log\log n)$. In total, the node history uses $O(((h-1)|VC| + n) \log\log n)$ space.

The trick here is that for all nodes u, it holds that: $u \in VC \vee \forall e = (u,v) : v \in VC$. Then, when the collecting BFS searches from a node v it will take at most one step for nodes not in the vertex cover and two steps for nodes in the vertex cover until they reach a frontier of only nodes in the vertex cover. From this frontier, the collecting BFS can calculate v's history and counter by merging the frontier's node histories. The collecting BFS is now bounded by two steps, resulting in $O(d^+(v) + \sum_{u \in s(v)} d^+(u))$ operations, where $s(v)$ is the set of neighbors of v.

Dynamic minimum vertex cover
At all times, we need to keep track of which nodes are in the vertex cover. This requires a fully dynamic minimum vertex cover algorithm. However, as the minimum vertex cover problem is NP-complete, approximation algorithms must be used. The choice of fully dynamic approximate minimum vertex cover algorithm depends on the ratio between the number of insertions and deletions. A simple greedy algorithm can maintain a 2-approximation in $O(1)$ time per insertion and $O(n)$ time per deletion [15], while another algorithm that partitions the nodes can maintain the same approximation in $O(\log n)$

amortized time per insertion and deletion [16]. In our case, deletions will be very sparse in the data stream, which is why we chose greedy algorithm. The greedy algorithm also have the property that if a deleted edge was not in the maximal matching previously, it deletes the edge in $O(1)$ operations. For dense graphs, only a small amount of the edges will be in the maximal matching, which makes the greedy algorithm perform quickly in deletions as well. In practice, the greedy algorithm performs $30,000,000$ edge insertions per second and $5,000,000$ edge deletions per second (see Section 5.1.3).

Directed graphs
As the standard minimum vertex cover problem is defined for undirected graphs we have to slightly modify the problem for directed ones. The new problem description is as follows; given a directed graph $G = (V, E)$, select a minimum cardinality subset $V' \subseteq V$ such that for all edges $e = (u, v) \in E, u \in V' \vee v \in V'$. The problem is still NP-complete as undirected graphs are a special case of directed graphs.

By the same reasoning as in the undirected case, a maximal matching is a 2-approximation of the generalized problem. The greedy algorithm needs to be modified to support directed edges. The only case affected is when an edge in the maximal matching is deleted. Let $e = (u, v)$ be a deleted edge. Previously, the algorithm removed both u and v from the vertex cover and then scanned the outgoing edges from u and v for edges uncovered due to the removal. With directed edges, both incoming and outgoing edges must be verified. This is solved by scanning the outgoing edges of u and v in both the original graph and the transpose of it. The original graph will give the outgoing edges of u and v and its transpose the incoming ones.

4.4.2 Edge insertion

Edge insertion in DANF is now divided into four steps. The first step is checking if a new edge contains any new nodes. New nodes are added to the graph and the top counter is, if necessary, resized to fit new counters of the new nodes. The second step is to check if any new node needs to be added into the vertex cover. For all new nodes in the vertex cover, memory is allocated in the history counters. The third step is calculating the history of the nodes added to the vertex cover. Lastly, the new history is propagated with a BFS in the transpose of the graph to update the counters of the nodes affected by insertion. After these four steps all nodes will have an approximate neighborhood function in their top counters and all nodes in the vertex cover will have their history counters updated.

4.4.2.1 Partial history calculation

When a bulk of edges is inserted, the vertex cover needs to be modified and the history counters updated. Using a maximal matching [15], which does not delete any nodes upon insertion, the collecting BFS can generate the partial history by searching at most one level. For every node added to the vertex cover, the collecting BFS only needs to retrieve the history of the node before the current bulk insertion. The remaining of the node history will be propagated by the propagating BFS.

The collecting BFS of node v looks through all its neighbors that are in the vertex cover and adds their history to its own. For this, only $O(d^+(v))$ operations are needed, which is an improvement to the time complexity stated in 4.4.1.1.

4.4.2.2 History propagation

When a new edge $e = (u, v)$ is added to the graph, many nodes that have a path to u will contain outdated history. The algorithm works by propagating the history of v through the transpose of the graph. If v is not in the vertex cover, the history needs to be calculated from the neighbor nodes. The algorithm is presented in pseudo code (see Figure 4.3).

```
e = (u,v) //Edge to add
if(isInVertexCover(v))
    $H_v$ = H(v);
else
    $H_v$ = union of the history of the neighbor nodes;
In BFS; source: u, current node: z, at depth: d{
    d = d+1;  // The BFS is performed from u so the actual depth
              // (from v) is one higher
    if(isInVertexCover(z)){
        foreach 0 $\leq$ i < h+1-d{
            H(z,i+d).union($H_v$(i));
        }
    }else{
        H(z,h).union($H_v$(h-d));
    }
}
```

Figure 4.3: History propagation in pseudo-code

To speed up the algorithm, it needs to be modified to handle several edges at once. In this case, the traveler support of the MS-BFS algorithm can be used. This means that the data provided by the traveler can be used instead of looping through all the source nodes. The travelers will contain the counter registers of their respective source node. When the travelers merge they can take the union of their registers. Also They only need to keep the $h + 1 - depth$ top-most counters as the others will not reach further anyway. In the visitor the data from the traveler is joined with the existing counters of the visited nodes. The merge function for the traveler is constructed as in Figure 4.4.

```
Input: t1,t2 = travelers to merge (HyperLogLog counters)
       d = depth of the BFS
Output: a new traveler
tOut = t1.clone
foreach 0 $\leq$ i < h+1-d{
    $tOut_i$ = $tOut_i \cup t2_i$
}
return tOut
```

Figure 4.4: History propagation traveler in pseudo-code

The visitors have to be slightly modified to make use of this traveler. The only difference is that a visitor uses the counter history from a traveler data instead of taking it from the source nodes.

Given that the partial history calculation makes sure that all nodes in the vertex cover have the history they had before the current insertion, we prove that all nodes have the correct history after the propagation step:

Theorem 4.4.1. *Given that HyperANF and DANF uses the same hashing function; after every bulk insertion, all nodes have had all reachable nodes added to their counters.*

To prove this we first need to establish two lemmas:

Lemma 4.4.2. *Assume that the history of all the nodes in the vertex cover in bulk insertion p have had all reachable nodes added to their counters after bulk insertion p. Then, during bulk insertion $p+1$, only one BFS step is needed to collect the history that any node v would have after bulk insertion p.*

Proof. Given a node v whose history should be calculated for bulk insertion $p+1$:

Fact: It holds that $v \in VC \vee \forall e = (v, u) : u \in VC$.
Let $N_p(v, h) = \{v\} \bigcup\limits_{u \in s_p(v)} N_p(u, h-1)$ be the reachable nodes of v between insertion p and $p+1$, where $s_p(v)$ is the set of successors of v in step p. Let $H_p(v, h) = \{v\} \cup \bigcup\limits_{u \in s_p(v)} N_{p-1}(u, h-1)$ be the nodes included in the calculated counter. Let $n_p(v) = s_p(v) \backslash s_{p-1}(v)$.

As no edges are removed: $s_p(v) \subseteq s_{p+1}(v)$.
$H_{p+1}(v, h) = \{v\} \bigcup\limits_{u \in s_{p+1}(v)} N_p(u, h-1) = \{v\} \bigcup\limits_{u \in s_p(v)} (N_p(u, h-1)) \bigcup\limits_{u \in n_{p+1}(v)} (N_p(u, h-1)) \supseteq \{v\} \bigcup\limits_{u \in s_p(v)} N(u, h-1) = N_p(v, h)$.
As $N_p(v, h)$ is a subset of $H_{p+1}(v, h)$, $H_{p+1}(v, h)$ must contain all elements in $N_p(v, h)$. As $H_{p+1}(v, h)$ only uses the history of its neighbors, only one BFS step is needed.

■

Lemma 4.4.3. *Assume that the history of the nodes in the vertex cover in bulk insertion $p+1$ have the history they would have after bulk insertion p. Then, after a bulk insertion $p+1$ all nodes have had all reachable nodes added to their counters.*

Proof. Take any two nodes u and v where u reach v in at most h steps. In bulk insertion $p+1$ there are two cases.

u **reached** v **in insertion** p) As u reached v after the previous insertion, combined with the assumption, v has been added to the counters of u.

u **did not reach** v **in insertion** p) As u reach v in insertion $p+1$, there must be a path P from u to v where an edge has been added. Let $e = (a, b) \in P$ be the new edge of greatest distance from u. As u reach v in at most h steps it holds that $|P| \leq h$. As there are no more new edges between b and v, combined with the assumption, b will contain v. By the algorithm, the propagation step will traverse h steps from b and as $|P_{u \to b}| \leq |P| \leq h$ the propagation from b will reach u and added v to its counter.

■

We are now ready to give the full proof of Theorem 4.4.1:

Proof. This will be proved by induction. Let p be the number of edge bulk insertions.

Base case: $p = 0$) The initial state is produced by HyperANF which has been proven to be correct.

Induction Hypothesis: The history of all nodes are correct after p bulk insertions.

Induction case: Running the partial history calculation is allowed, as the assumption is fulfilled by the induction hypothesis Lemma 4.4.2. The history propagation may then be run as the partial history calculation fulfills the assumption in Lemma 4.4.3. This means that all nodes have had all reachable nodes added to their counter.

■

5
Experiments

5.1 Benchmarks

5.1.1 Comparison of graph structures

A benchmark was performed to compare the performance of a high level data structure and a byte stream (see Table 5.1 and Figure 5.1). The graph in-2004 was used (see Appendix A). If all edges are added in one bulk, the performance measured will only reflect the complexity based on the input. By inserting the edges in bulks of 5000 both the complexity based on input and the existing structure are accounted for. The add time in Table 5.1 is the time it takes to insert edges in bulks of 5000. The iteration time is the time it takes to iterate over all edges in the graph once.

In Figure 5.1 the memory difference can be seen. The high level data structure and the byte stream have the same space complexity, but the difference can be explained by the memory overhead used by the high level data structure.

	Add time (s)		Iteration time (s)	
edges	High level	Byte stream	High level	Byte stream
10^6	1.1	14.0	0.15	0.03
2×10^6	2.3	47.4	0.29	0.04
3×10^6	3.5	100.1	0.40	0.05

Table 5.1: Benchmark of representing the graph as a byte stream and a high level data structure

5.1.2 Memory dependent merging

To find a suitable ratio r between the memory used by the graph part saved as a byte stream and the part saved as a high level data structure, a benchmark was performed to compare the time and the space usage (see Figure 5.2). The benchmark was performed on the in-2004 graph (see Appendix A). 1,000,000 edges were randomly generated and inserted into the graph in bulks of 5,000. The measured time is the time to insert the edges into the dynamic graph and then to perform a complete edge scan. The measured memory is the graph's heap size. The plotted time values are the average of ten bulk insertions.

For reference, the benchmark compares a certain ratio r with the extremes, $r = 0$ and $r = \infty$. With $r = \infty$ the additional entries are never merged with the original graph. This represents the optimal elapsed time. With $r = 0$, the graphs are always merged after each bulk insertion. This represents the optimal memory usage. By choosing $r = 8$, the benchmark shows that near optimality in both time and memory usage can be achieved. The plots were made to visualize near optimality rather than trends.

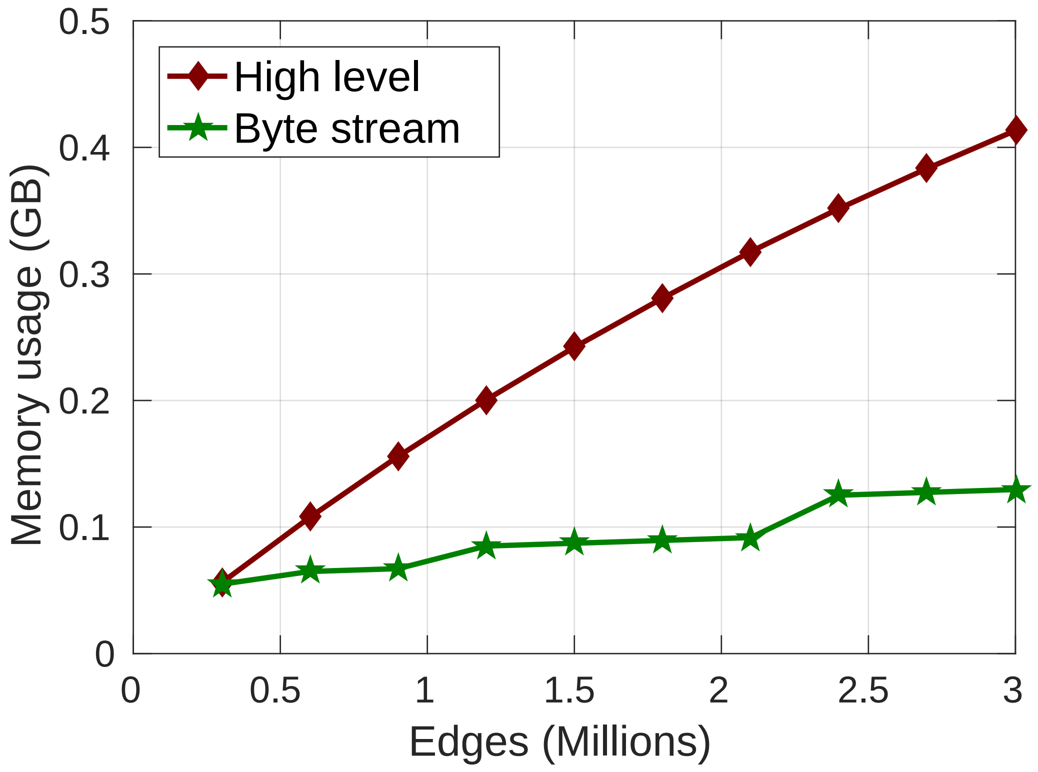

Figure 5.1: Benchmark of memory usage between high level structure and byte stream.

5.1.3 Dynamic Vertex Cover

The selected greedy algorithm was benchmarked with the it-2004 graph (see Appendix A). Starting with an empty graph, every edge was sequentially inserted from it-2004 into the dynamic vertex cover. Both the time and heap size were measured over the number of inserted edges (see Figure 5.3). This shows roughly 29.5 million inserted edges per second. The memory used mainly depends on the number of nodes. The heap size increased to 0.33GB implying an average of 8 bytes per node.

To test the performance of sequentially deleting edges, the it-2004 graph was used (see Appendix A). The test was performed by first inserting all edges into the dynamic vertex cover and then sequentially deleting them (see Figure 5.4). All 1,150,725,436 edges were deleted in 220 seconds, giving a performance of 5,000,000 deleted edges per second. The heap usage is not plotted in the figure, as the algorithm is not designed to downsize the allocated memory for the vertex cover after deletions.

5.1.3.1 Comparison of unoptimized and final DANF

A comparison between the two-BFS algorithm and DANF was performed by adding edges of different bulk sizes (see Figure 5.5 and 5.6). The benchmark was performed on the graph in-2004 (see Appendix A) with 16 HyperLogLog registers per node for $h = 3$ and $h = 8$. The time measured was the time to insert random edges and update the counters. The memory usage of DANF was divided into the four main components while the two-BFS memory usage is for the complete algorithm. The figure shows that DANF scales with the edge insertions bulk size while the two-BFS algorithm's performance is constant. For large h-values DANF is up to 60 times faster than the two-BFS algorithm, but also uses

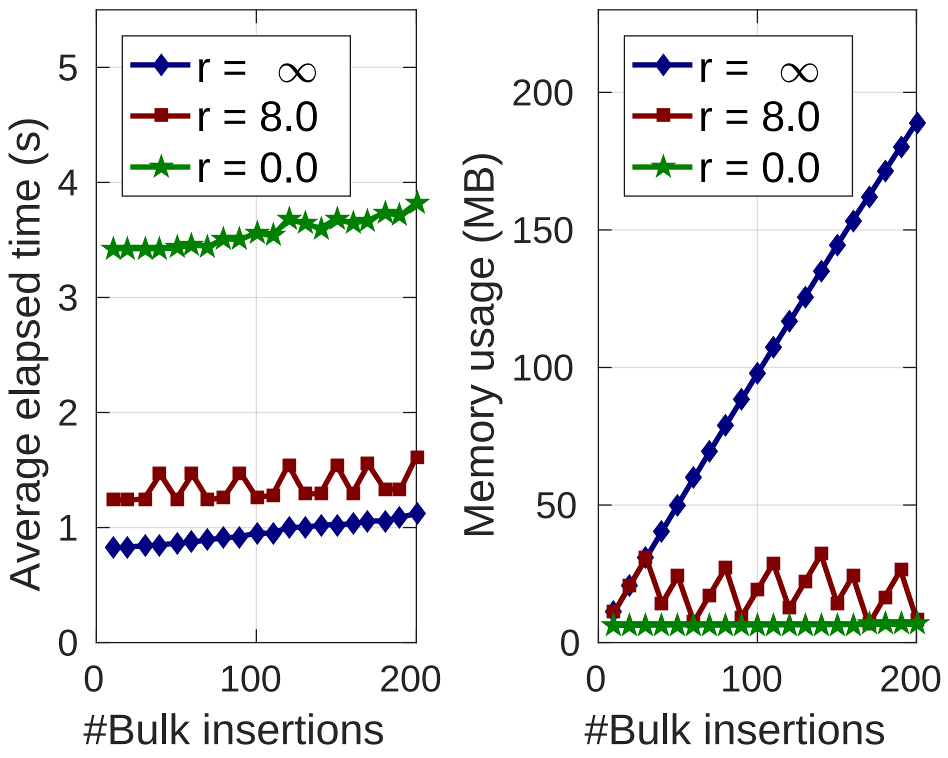

Figure 5.2: Benchmark of different union to stored memory ratio limits

up to 45 times more memory. For small h-values the gained speed of DANF is smaller but so is also the space consumption. Almost all of the memory used by DANF is used by MS-BFS. The vertex cover and graph space usage is almost too small to be seen in Figure 5.6.

5.1.4 Comparison of DANF and HyperANF

A comparison between DANF and HyperANF was made by inserting different amounts of edges into DANF. The maximum number of added edges that was used as the limit was when the time of inserting the edges was the same as recalculating HyperANF. The graph in Figure 5.7 shows the amount of edges added compared to the initial amount of edges. When the initial graph has 70.000 edges, DANF can add more than the existing edges before HyperANF finishes the recalculation on the new graph which is the sample right above 10^0 in Figure 5.7. However, when the initial number of edges increases the ratio of the total number of edges that can be added decreases. When the number of edges grows, DANF converges toward being able to add 0.01% of the initial edges before a recalculation is completed. The ratio of added edges to initial ones decreases as h increases.

5.1.4.1 MS-BFS tuning

The MS-BFS article [13] mentions that if several MS-BFSs are to be performed, the sources should be sorted and partitioned by out-degree. This increases the speed as when the sources have a higher out-degree, more BFSs meet and can join. When the number of edges added to DANF reaches a certain threshold, the edges to be inserted are divided into partitions. Sorting by out-degree improved the performance in the graph in-2004 (see

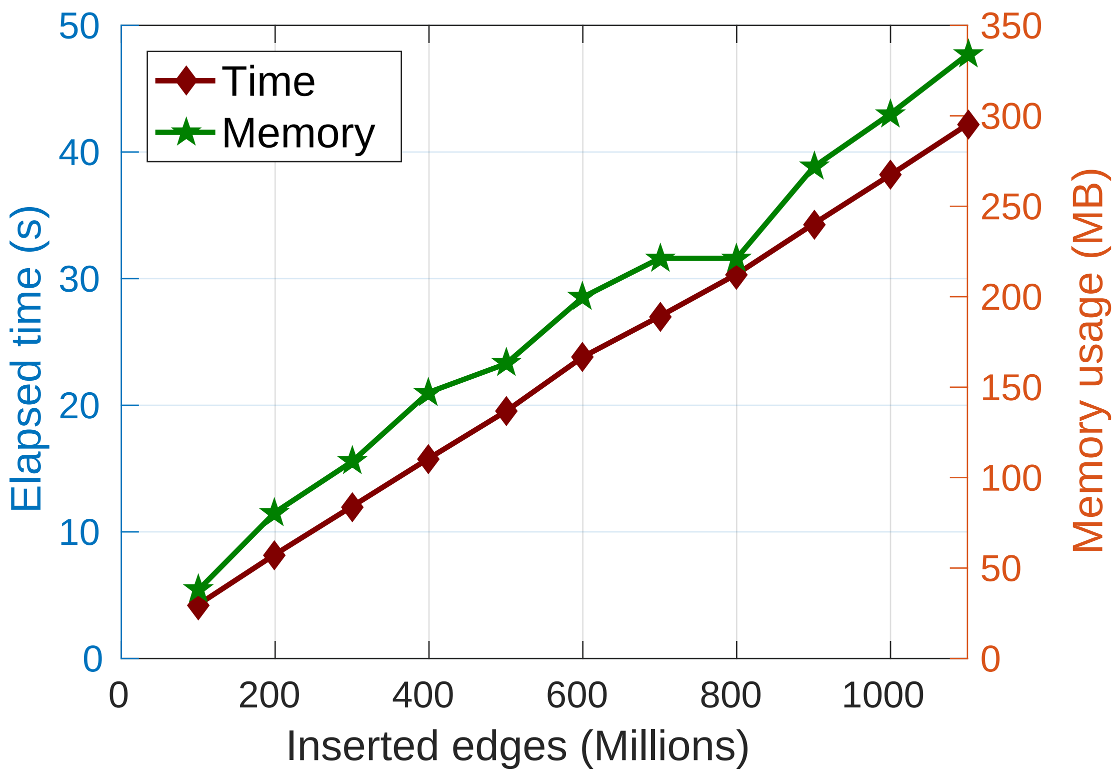

Figure 5.3: Benchmark of sequential insertions in 2-approximate dynamic vertex cover

Appendix A) by up to 37.2% compared to no sorting. As it is preferable to make as many BFSs as possible to meet, sorting by the ANF value improves the performance even more. Sorting by the ANF value improved performance by up to 45% compared to no sorting.

5.2 Experiments on a real-time data stream

To do real-time experiments on a data stream, there are a few more tasks that need to be performed. The three main tasks are: fetch data, produce graph edges and update DANF. To efficiently perform these tasks they are run in parallel. They are implemented as components of a pipeline as seen in Figure 5.8. A component stores its results in a buffer which can be read by the next component in the pipeline. The writing component has write-only access and the reading component has read-only access. Note that Edge producer can be a writer to a DANF updater using another coupling handler. This makes it possible to create a theoretically infinite pipeline with very low coupling between components and allows every component to run in their own threads.

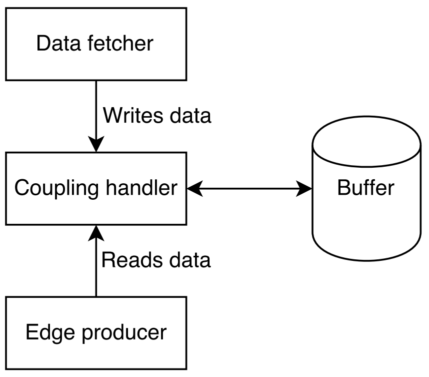

Figure 5.8: Parallel-compatible pipeline layout

As all nodes and edges are handled as plain numbers, a mapping needs to be stored from node index to the data. To avoid the data manager competing with the algorithm for CPU and memory, the data should preferably be stored in a database by a completely

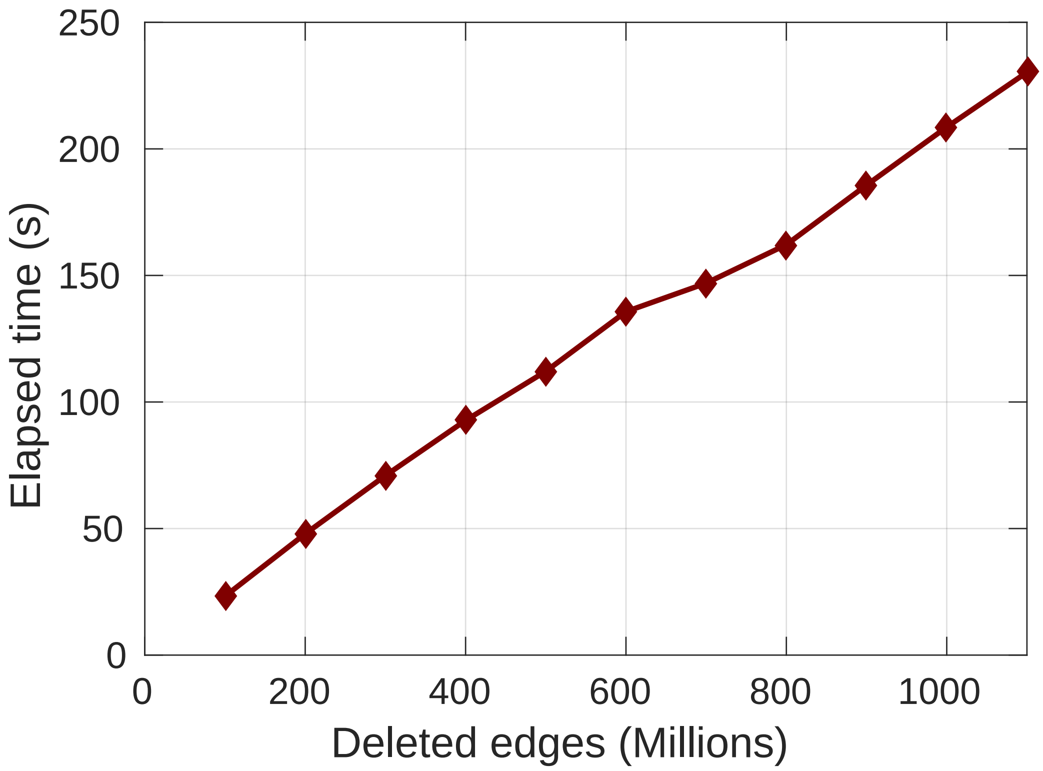

Figure 5.4: Benchmark of sequential deletions in 2-approximate dynamic vertex cover

different machine. As the only data that needs to be saved on the same machine is a mapping from node indices to data indices, which can be saved on disk.

A list A of the calculated value for each node is kept. Another list B keeps track of the value of the nodes that change. After a given amount of time, the difference of the changed nodes is checked and nodes with significant changes are marked as rapidly changing. The elements in B are added to A and B is cleared. This allows the detection of upcoming trends.

Another useful feature is to keep track of the top nodes, sorted in descending order according to their DANF values. Initially, all nodes were tracked. However, this turned out to be unsustainable on a graph with only a few million nodes. Instead, only the top X nodes were tracked. This leads to a drastic speed increase while still allowing to identify the most central nodes.

5.2.1 Graph layout

When performing experiments on the data stream, we used a certain graph structure. The structure was designed to enable us to detect popular subjects, authors and sources. This was achieved by creating three different graph models which, for simplicity, were combined into one (see Figure 5.9). The common node for all models is Document.

The first model is the concept to document part. This model identifies popular subjects by retrieving all concepts that are mentioned in a document and then adding an edge from the concept to the document. The DANF value represents how many mentions a concept have.

The second model is the location to document line in the middle of 5.9. This model can

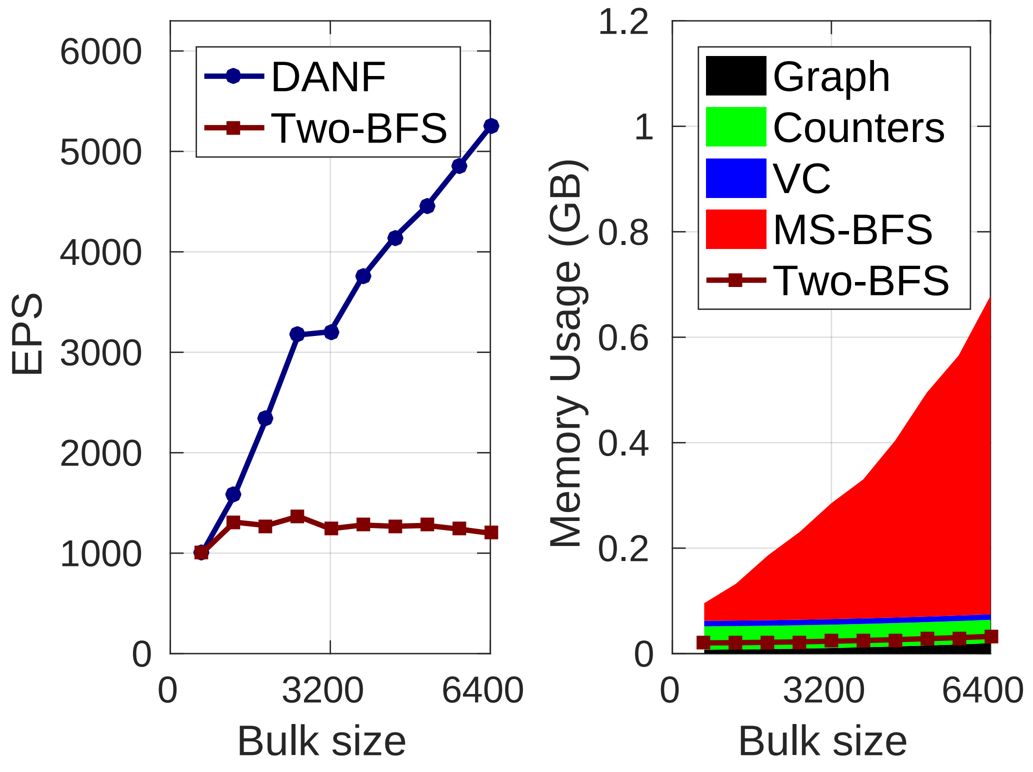

Figure 5.5: Benchmark of DANF and the two-BFS implementation with h = 3

identify which locations, sources (news) and authors that publishes the most documents.

The last model is the named entity part to the right in Figure 5.9. The named entities are general things mentioned in articles; such as countries, people or companies. Each entity has a sentiment. The sentiment specifies if the entity is mentioned in a positive (P), neutral (N) or negative (V) manner. Using the visualized structure for the named entities in Figure 5.9 enables detection of both the number of mentions and trends of how entities are referenced.

5.2.2 Starting with an empty graph

The first experiment performed was done by creating an initially empty graph and setting up DANF to track the neighborhood function on it. A connection was established to the company live stream. The algorithm had no trouble keeping up with the stream of about 11 documents per second (two million documents per day) where about 30 edges were generated per document.

5.2.3 Starting with an arbitrary large graph

Building a large graph from the existing data takes a long time. The data have to be parsed and analyzed to generate edges. Instead of spending a large amount of time to generate the graph, the it-2004 graph (see Appendix A) was used. DANF managed ten documents per second, which means that it could not keep up with the stream. This indicates that roughly one billion edges is the limit of the current implementation. However, the it-2004 graph is denser than the constructed graph. Hence, DANF would have a higher

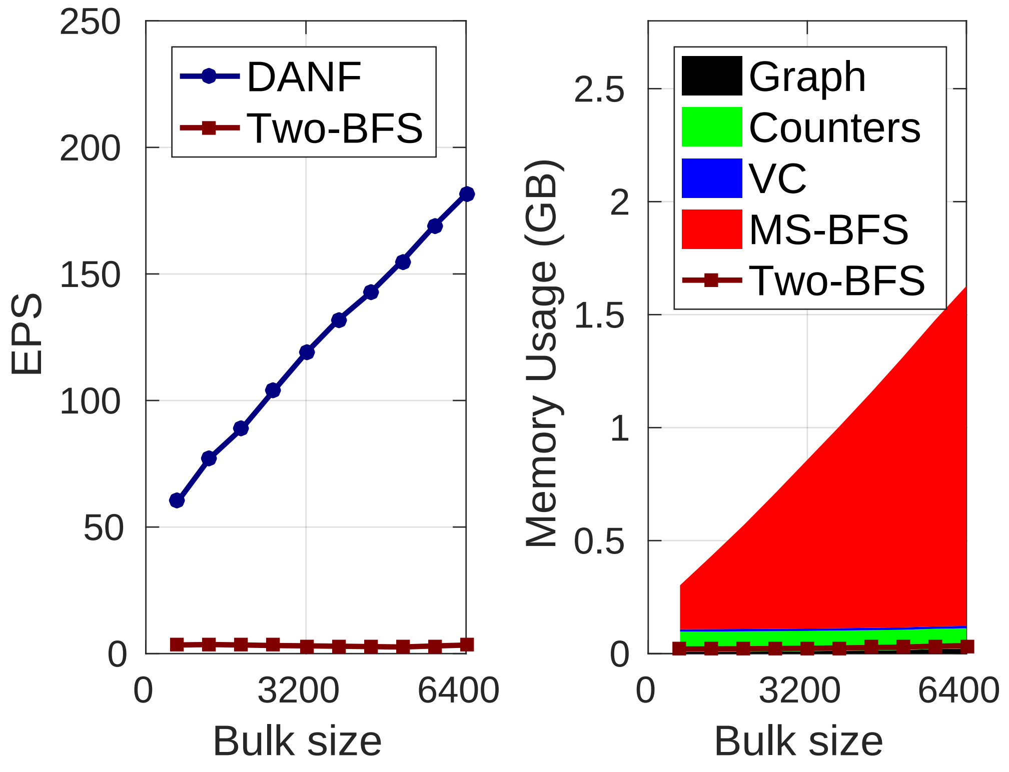

Figure 5.6: Benchmark of DANF and the two-BFS implementation with h = 8

performance if the graph was made from scratch. More optimizations are required to prevent buffer build up in the long run.

5.2.4 Data retrieval

From the experiment, it was evident that both the United States and China were central nodes in the constructed graph. The United States was one of the most central nodes in all of the three models.

In the experiment, a trend concerning North Korea was detected. Short after North Korea released information concerning further nuclear tests, the DANF value of the North Korean concept node increased rapidly. Such trends can easily be detected by tracking rapidly changing nodes.

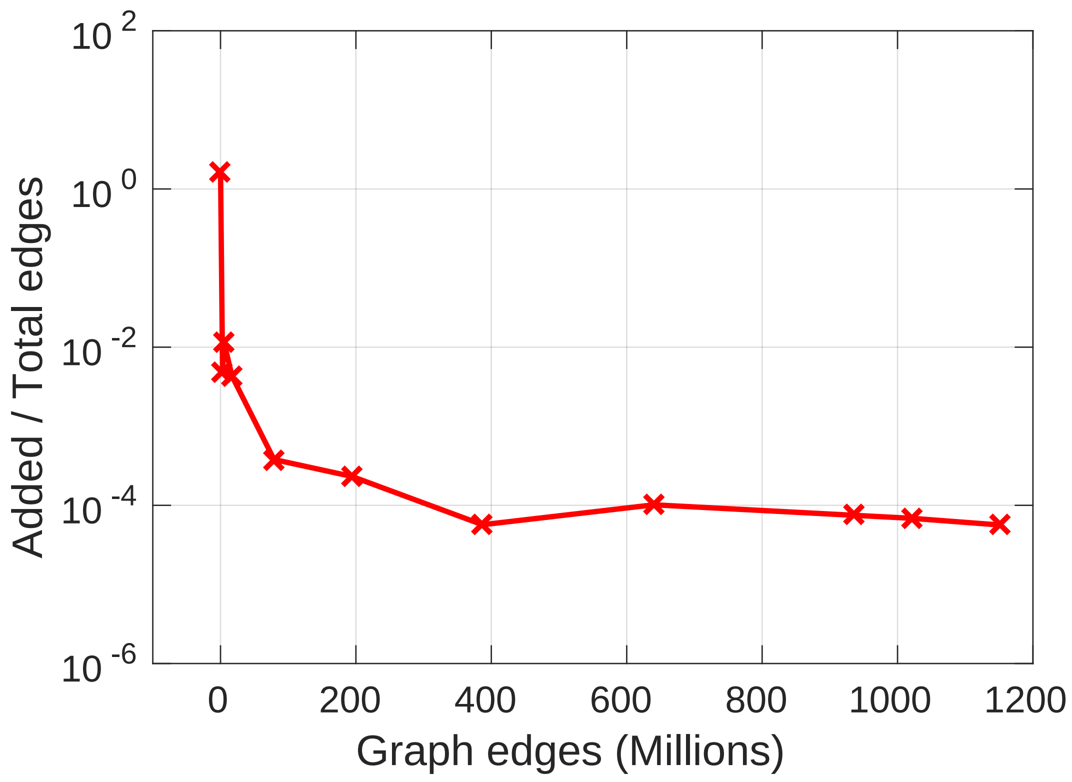

Figure 5.7: Benchmark of the ratio of edges that can be added in the same time as a HyperANF recalculation with h = 3

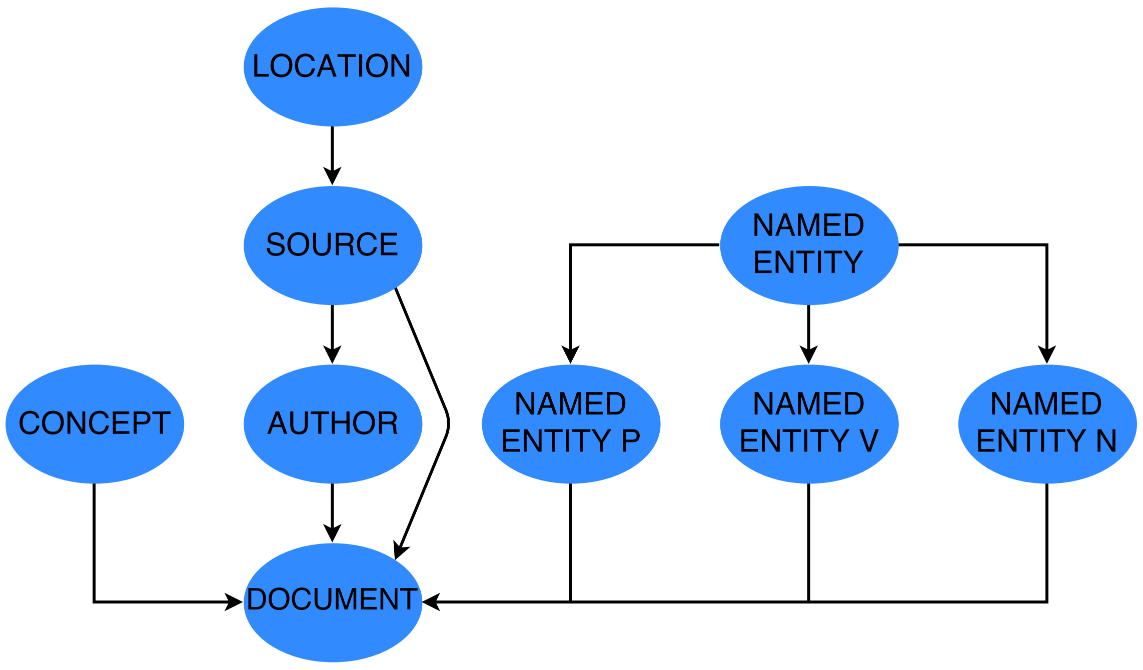

Figure 5.9: Graph layout used in the conducted experiment

6 Discussion

With the DANF algorithm, it is now possible to continuously maintain an approximate neighborhood function for all nodes in a graph that supports insertions. As node centralities now can be tracked over time, graphs can be further mined for data and trends can be detected.

DANF is more efficient than a HyperBall recalculation when the edge insertions' bulk size is small compared to the total size of the graph. However, DANF uses more memory than HyperBall. So a HyperBall recalculation might be proved to be able to handle larger graphs given a certain amount of RAM. When h is large (approaching 10) DANF becomes significantly slower compared to HyperBall.

An unexpected drawback in the algorithm is the space used by MS-BFS. It has a space complexity of $O(ns)$, where s is the number of sources, and in practice it used a lot more storage than anticipated. The percentage of storage used by MS-BFS is presented in Table 6.1. See Appendix A for graph details. The extra memory means that DANF may use 3 times more space than HyperBall. However, the MS-BFS space percentage tends to decrease as the graph size increases. The MS-BFS, vertex cover and HyperLogLog counters are included in the total algorithm space.

edges	uk-2007-05@100000	in-2004	it-2004
640	56.1%	40.3%	23.5%
2560	72.3%	65.8%	29.1%
4480	75.6%	71.7%	37.8%

Table 6.1: Percentage of the algorithm space used by MS-BFS with h = 3 and 16 HyperLogLog registers per node

6.1 Ethics

Internet can be represented as an undirected graph. Routers can be represented as nodes and connections between routers as edges. Applying the neighborhood function to this graph can help identifying the most central routers. Assume a hacker gets hold of this data. The hacker can then target attacks on only these routers to disrupt the maximum amount of people, with minimal effort. Applying the neighborhood function to the Internet graph can also be used to simulate an attack. This can be achieved by first running ANF, then deleting some nodes or edges and then running ANF again. The difference in ANF values give insight of how the deletions affected the graph. With this approach the Internet graph predicts the efficiency of targeting certain routers.

The exact same information above can be used by safety companies to prevent attacks. By identifying critical routers extra precautions can be taken at these routers. The information provided by the neighborhood function can be used for both good and bad purposes, depending on who gets hold of it.

The information retrieved from the neighborhood function may also have ethical issues when applied to a graph that includes personal data. Assume a graph where the nodes are journalists and articles. In this graph an edge represents that a journalist have written a certain article or that an article is referenced by another article. By applying the neighborhood function to this graph central journalists and articles can be identified. A central article is referenced in many other articles and a central journalist is a journalist that has written many central articles. This can be a great help when evaluating the credibility of a journalist. It can also be used to retrieve the most influential articles in the graph. The ability to find the most influential journalists must be used with caution. If the graph only consist of articles that are critical to a certain view, the neighborhood function can identify what journalists to target to silence the criticism. If the influence score is made public, or possible to be bought, it could also affect the career of a journalist. If the score is not perfect, an influential journalist could be incorrectly marked as non-influential which might decrease the chance of getting good stories.

The ability to identify central and influential nodes can be useful in many different situations. When the information is used for realisations that benefits the humanity as a whole, it should always be acceptable. An example of this is when the algorithm is applied to find information for medical purposes, perhaps by finding out central genes that are involved in the development of cancer. This is helpful for all of humanity. Using the information for selfish purposes requires more investigation to determine if it is acceptable. Retrieving personal data may be fine in some cases, but in many other cases should be strictly forbidden. In general, if everyone benefits from the information from the NF, it is fine to use.

6.2 Future work

DANF is currently not a fully dynamic algorithm as deletions in the graph are not fully supported. As this is a part of the problem description, the next natural step is to implement deletions. There are two missing parts to achieve deletions: removing edges from a graph file and decreasing HyperLogLog counters affected by a deletion. Deleting edges from a graph file can be performed by re-storing the graph, but a better solution may be to temporarily keep track of which edges are deleted and ignore these until a store is performed. Decreasing HyperLogLog counters can be implemented by a two step algorithm. First, perform a BFS in the transpose of the graph to find all nodes that are possibly affected by the deletion. Then, each possibly affected node can be treated as a new node which will recalculate the node counters. This can be implemented in $O(m^2)$ time, where m is the number of edges. Another way to decrease the counters is to probabilistically decrease the counters of all affected nodes. This would speed up the algorithm but can affect the precision of the counters.

DANF also have to be further optimized. Due to time limitations there exist bottlenecks. An improvement is to make the program execute more in parallel. With the exception of the MS-BFS, the current program is run sequentially. Another improvement is to separate responsibilities among machines. Currently, a single machine will update and save the graph, calculate node counters and keep track of top counters. These tasks can be divided among several machines, hence speeding up the algorithm and making it more scalable. Also, the main part of the algorithm, the MS-BFS, can be distributed. Making the MS-BFS scale with the number of machines greatly increases the potential of the algorithm. There are already implementations of standard BFS on several machines. Adapting those implementations to handle MS-BFS should be possible. One tool that can be used is Giraph [19]. Giraph is an open-source implementation of a vertex-centric distributed graph processing system.

Bibliography

[1] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, “A fast approximation of the neighbourhood function for massive graphs,” tech. rep., f, 2001.

[2] P. Boldi, M. Rosa, and S. Vigna, “Hyperanf: Approximating the neighbourhood function of very large graphs on a budget,” *CoRR*, vol. abs/1011.5599, 2010.

[3] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, “Anf: A fast and scalable tool for data mining in massive graphs,” in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’02, (New York, NY, USA), pp. 81–90, ACM, 2002.

[4] P. Flajolet, Éric Fusy, O. Gandouet, and et al., “Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm,” in *IN AOFA ’07: Proceedings of the 2007 international conference on analysis of algorithms*, 2007.

[5] R. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations* (R. Miller and J. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[6] P. Boldi and S. Vigna, “In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond,” in *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, (Dallas, TX, USA), pp. 621–628, IEEE, 2013.

[7] D. M. Kane, J. Nelson, and D. P. Woodruff, “An optimal algorithm for the distinct elements problem,” in *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS ’10, (New York, NY, USA), pp. 41–52, ACM, 2010.

[8] S. Heule, M. Nunkesser, and A. Hall, “Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm,” in *Proceedings of the EDBT 2013 Conference*, (Genoa, Italy), 2013.

[9] P. Chassaing and L. Gérin, “Efficient estimation of the cardinality of large data sets,” in *Proceedings of the 4th Colloquium on Mathematics and Computer Science*, pp. 419–422, 2006.

[10] S. Vigna, “Webgraph.” [Online]. Available: http://webgraph.di.unimi.it/. [Accessed: 2016-03-29].

[11] P. Boldi and S. Vigna, “The webgraph framework i: Compression techniques,” in *Proceedings of the 13th International Conference on World Wide Web*, WWW ’04, (New York, NY, USA), pp. 595–602, ACM, 2004.

[12] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th International Conference on World Wide Web*, WWW ’11, (New York, NY, USA), pp. 587–596, ACM, 2011.

[13] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo, “The more the merrier: Efficient multi-source graph traversal,” *Proc. VLDB Endow.*, vol. 8, pp. 449–460, Dec. 2014.

[14] S. Khot and O. Regev, “Vertex cover might be hard to approximate to within 2-ϵ,” *J. Comput. Syst. Sci.*, vol. 74, pp. 335–349, May 2008.

[15] Z. Ivkovic and E. L. Lloyd, “Fully dynamic maintenance of vertex cover,” in *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG ’93, (London, UK, UK), pp. 99–111, Springer-Verlag, 1994.

[16] S. Baswana, M. Gupta, and S. Sen, “Fully dynamic maximal matching in o(log n) update time,” *CoRR*, vol. abs/1103.1109, 2011.

[17] S. Bhattacharya, M. Henzinger, and G. F. Italiano, “Deterministic fully dynamic data structures for vertex cover and matching,” in *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’15, pp. 785–804, SIAM, 2015.

[18] D. W. Harder, “Array resizing.” [Online]. Available: https://ece.uwaterloo.ca/∼dwharder/aads/Algorithms/Array_resizing/. [Accessed: 2016-03-04].

[19] “Apache giraph.” `http://giraph.apache.org/`. Accessed: 2016-05-23.

[20] “It-2004 graph.” `http://law.di.unimi.it/webdata/it-2004/`. Accessed: 2016-04-12.

[21] “In-2004 graph.” `http://law.di.unimi.it/webdata/in-2004/`. Accessed: 2016-04-12.

[22] “uk-2007-05@100000 graph.” `http://law.di.unimi.it/webdata/uk-2007-05@100000/`. Accessed: 2016-05-16.

A

Appendix A

A.1 Benchmark equipment

The benchmarks have all been performed on a MSI GE70 laptop. The CPU used is an Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz. The RAM used are 2x Kingston SODIMM DDR3 Synchronous 1600 MHz MSI16D3LS1KFG/8G, total 16GB RAM. The disk used is a Hitachi HGST HTS721010A9, 1TB 7200RPM.

A.2 Graphs

Graph	#Nodes	#Edges	Avg. Degree	Source	Thanks
it-2004	41,291,594	1,150,725,436	28	[20]	[11, 12]
in-2004	1,382,908	16,917,053	12	[21]	[11, 12]
uk-2007-05@100000	100,000	3,050,615	30.506	[22]	[11, 12]