GRAPH SUMMARIZATION: ALGORITHMS, TRAINED HEURISTICS, AND PRACTICAL STORAGE APPLICATION

GEORGE HODULIK

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

Advisor: Dr. Z. Meral Ozsoyoglu

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May 2017

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

George Hodulik

for the degree of Master of Science *.

Committee Chair

Dr. Z. Meral Ozsoyoglu

Committee Member

Dr. Harold Connamacher

Committee Member

Dr. Mehmet Koyuturk

Date of Defense

December 14, 2016

*We also certify that written approval has been obtained for any proprietary material contained therein.

Table of Contents

List of Tables

List of Figures

Graph Summarization: Algorithms, Trained Heuristics, and Practical Storage Application

GEORGE HODULIK

1 Abstract

The problem of graph summarization has practical applications involving visualization and graph compression. As graph-structured databases become popular and large, summarizing and compressing graph-structured databases can become more and more useful. We explore the use of a particular family of graph summarization algorithms we call Summaries with Supernodes, Superedges, and Corrections (SSSC) and the feasibility of using SSSC algorithms when summarizing large Resource Description Framework (RDF) graph datasets. We also propose optimizations to the Uniform Randomized SSSC algorithm by using trained heuristics to pick seed nodes. We also show how SSSC summaries may be stored in a similar manner as RDF triple stores, and we discuss possibilities for future work involving localized SSSC algorithms.

2 Introduction

As use of the Semantic Web through Resource Description Framework (RDF) is becoming more popular, data size is also dramatically increasing. Google Freebase[1] (now deprecated), for instance, has billions of RDF subject-predicate-object triples, which may be represented in several hundred gigabytes. Since database sizes are becoming so large, there may be a benefit to meaningfully compress the data. Since triple stores are often represented as graphs, where each triple can be seen as a directed edge from the subject to object, we can apply concepts of graph compression or summarization to RDF triple stores. In this paper, we specifically explore how we can apply a family of graph summarization methods to RDF triple stores. We will refer to this family of algorithms as Summaries with Supernodes, Superedges, and Corrections, or SSSC.

The authors from [1] first explored what we refer to as SSSC algorithms, proposing a greedy and randomized algorithm. The authors proposed the SSSC summarization model we discuss, and justify the model based on Minimum Description Length principle, which roughly states the best theory to infer from a set of data is the theory which minimizes the sum of the size of the theory and the size of the data when encoded using the theory[1]. By defining the size of the theory to be a graph summary, and data given the summary as a set of corrections, there is a cost of a model that we can minimize to find the “best” summary or compression. However, since the document’s publication in 2008, adoption of the Semantic Web has grown dramatically and little work, if any, has been done to show if SSSC summarization may be useful or even feasible for the Semantic Web. We explore the feasibility of SSSC summarization with real and synthetic RDF datasets,

[1] Google, Freebase Data Dumps, https://developers.google.com/freebase/data, Dec. 8, 2016.

identifying points of conflict, optimization, and discussing possibilities for datasets using summarized data as a primary method of storage.

2.1 Our contributions

We provide the following contributions:

- We provide a general, abstract algorithm to describe most SSSC algorithms, and discuss the differences between different SSSC algorithms with respect to the abstract algorithm. Having an abstract algorithm makes it easier to identify how changes between algorithms will affect performance and where one could work to optimize SSSC algorithms. A general algorithm also provides an easy introduction to SSSC algorithms to those who are not familiar. The general SSSC algorithm is discussed in section 5.
- We test various SSSC on different real and synthetic RDF data sets. Our datasets range from about 1 to 7 million RDF triples, although we had to simplify the data, which resulted in a range of about 0.2 to 1.8 million triples. Our tests use some of the largest datasets used to test SSSC algorithms. We show that for many data sets, compression ratios of about 0.20-0.70 can be achieved. We overview our experimental process, including RDF graph simplification, and discuss resulting summaries in section 6.
- We show that we can use trained heuristics based on a relatively small sample of nodes to improve the selection of seed nodes, improving the quality of summaries that may have a limited runtime or a specific desired compression ratio. This is significant, as summarization is costly, and achieving a desired compression ratio even 10% faster could mean hours or days of saved computation time, depending on the original dataset size. We discuss this in section 7.

- We store graph summaries in Microsoft SQL tables, and compare runtimes of simple connection queries on the summarized data as well as the original triple stores. We find that querying the summary tables, in many cases, is not significantly slower than querying the original tables, averaging at only about a 5% slowdown. Though our implementation is rudimentary, we hope it inspires future research on the possibility of using summaries as a primary storage method for RDF data. As far as we know, ours is the first attempt at using SSSC summaries for this purpose. We also discuss how such a framework may work for updating data. This is discussed in section 8.
- We discuss in section 9 how summary properties may imply natural distributed, parallelized, localized, and/or main-memory optimized SSSC algorithms.

Lastly, we discuss areas of future work in section 10.

3 Background

In this section, we provide background information for SSSC representations of graphs.

3.1 Graph and summary representation and definitions

A graph, $G = (V,E)$, is a set of vertices, V, and a set of edges, E. Edges are represented as (u,v), where $u,v \in V$. If we were to represent RDF data as a graph, subjects and objects would be nodes, and there would be a directed edge from subject to object for each triple. The edge would be annotated with the predicate value. Notably, vertices could also have multiple edges between each other. For our purposes, however, we only consider undirected, unannotated graphs, and so we simplify the RDF data to fit this kind of graph.

A graph summary of graph $G = (V,E)$, is denoted by $G_S = (S, C)$. The summary graph, $S = (V_S, E_S,)$ is a collection of supernodes, V_S, which contain original nodes in V, a collection of superedges, E_S, and a collection of corrections, C. A superedge in a summary connects two supernodes. A superedge represents that an edge between every original node contained by the two connected supernodes is also connected in the original graph, *G*. However, a summary may have superedges that represent some edges between original nodes that do not actually exist in the original data. If this is the case, we must record that such edges do not actually exist in the original data by adding appropriate corrections to *C*. For instance, suppose $s,t \in V$, $S,T \in V_S$, $s \in S$, $t \in T$, $(S,T) \in E_S$, and $(s,t) \neg\in E$. Thus, according to the summary, *s* and *t* are connected originally, but this is actually not true. We would fix this by adding the correction (-, *s*, *t*) to *C*. Corrections are negative when representing the subtraction of an edge represented by the summary which is not actually in the original data. Corrections may also be positive when an edge is not represented by the summary that exists in the original data.

In Figure 1, there is an example of a SSSC representation of a small graph. The original nodes *b* and *c* are merged into supernode *w*. Notice how *w* has a self-loop because *b* and *c* are connected in the original graph. Similarly, original nodes *h* and *g* are merged into supernode *y*, and original nodes *d*, *e*, and *f* are merged into supernode *z*. Original node *a* is simply assigned supernode *x*, which only contains *a*. Notice how superedges can represent the majority of the original edges, but the edge (*a*, *e*) must be added as a correction because no superedge represents it. Also, the edge (*g*, *d*) is represented by the superedge (*y*, *z*), but (*g*, *d*) is not present in the original graph, so a negative correction must be added to the correction set.

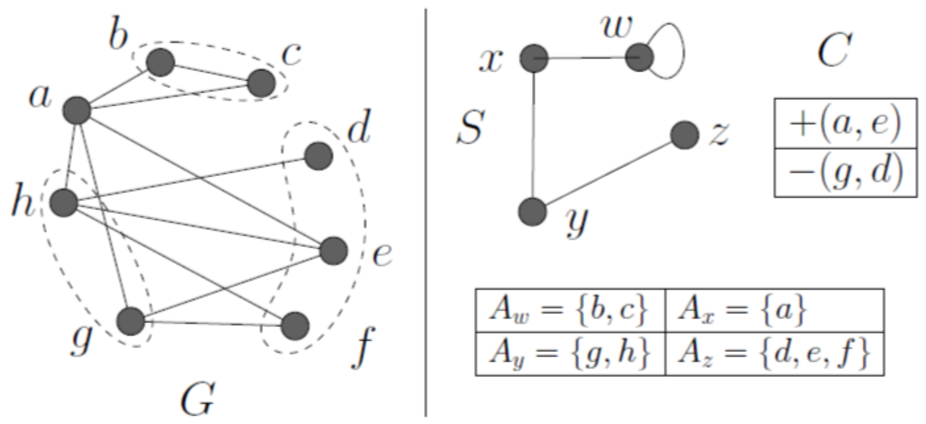


Figure 1 [1] A summary produced by merging nodes with the same neighbors, allowing for corrections.

3.2 Summary cost, compression ratio, and MDL

The cost of a summary is defined in [1], and uses Minimum Description Length (MDL) principle for its justification. MDL roughly states that the best theory to infer from a set of data is the one which minimizes the sum of the size of the theory and the size of the data when encoded using the theory[1]. In our graph summarization, we can conceptualize summary graph *S* to be our theory, and the corrections *C* to be our data encoded in terms of the theory. Therefore, [1] declares the best summary *S* to be the one which minimizes the storage cost of *S* and *C*. This cost includes the cost of storing superedges, corrections, and mappings from original nodes to supernodes. Note that the mappings from original nodes from *V* to the supernodes in V_S, should cost roughly the same regardless of the summary, so we can ignore the cost of the mappings in the summary cost we try to minimize. Assuming the cost of storing a superedge and the cost of a correction is roughly the same, we are left with a relative cost of a summary to be the sum of the number of superedges and the number of corrections.

$$\text{Cost}(G_S) = |E_S| + |C|$$

[1] also states that we can ignore the cost of the mappings because they will generally be small compared to the storage costs E_S and *C*. However, for our use of summarizing RDF data, the

original vertices are typically URIs stored as strings, which do not have negligible storage costs, especially when supernodes can be represented integers, which are much cheaper than long URI strings. Therefore, in this paper, we make a distinction between the theoretical cost presented in [1], or the storage cost without the cost of mappings, and the implemented cost, or the storage cost in our implementation which includes mappings.

Finally, notice that the original graph can be represented as a summary where every supernode just contains one original node, and there are no corrections. Then, the theoretical cost of a graph G is just the number of edges $|E|$. Therefore, we can calculate a theoretical compression ratio of a summary to be

$$\text{CompressionRatio}(G_S,G) = \text{Cost}(G_S) \,/\, |E|$$

Since the goal is to minimize Cost(G_S), it is important to note that, given a V_S, there is exactly one E_S and exactly one C which minimizes Cost(G_S). This is because, if we look at pairs of supernodes one at a time, we can easily determine whether or not a superedge should exist between those two nodes, using the cost of a superedge as our decision criteria. If $u,v \in V_S$, let Π_{uv} be the set of all pairs (a,b) where $a \in u$, and $b \in v$ (Note that $|\Pi_{uv}| = |u| * |v|$), and let $A_{uv} \subseteq \Pi_{uv}$ be the set of edges which actually exist in the original graph. Then, the cost of having a superedge between u and v would be $|\Pi_{uv}| - |A_{uv}| + 1$ because storing the superedge adds a cost of 1, and we would need a negative correction for all $|\Pi_{uv}| - |A_{uv}|$ edges that do not exist in the original graph. On the other hand, the cost of not having a superedge between u and v would be $|A_{uv}|$ because we would need a positive correction for each edge in the original graph. Finally, we can decide whether or not (u,v) should exist in E_S based on which option has the lower cost.

3.3 Edge representation Cost, Supernode Cost, and Reduced Cost

As discussed previously, the cost of representing the original edges between two supernodes, *u* and *v*, can be calculated deterministically based on $|\Pi_{uv}|$, the number of possible edges represented, and $|A_{uv}|$, the number of edges between nodes in *u* and *v* which are present in the original graph. Since we will be choosing whether or not to include a superedge between *u* and *v* based on which option has a lower cost, we can denote the cost of the representing the original edges between *u* and *v* as

$$\text{Cost}(u,v) = c_{uv} = \min(|\Pi_{uv}| - |A_{uv}| + 1, |A_{uv}|)$$

Furthermore, we can denote the cost of a supernode, *u*, to be the sum of the cost of all its related edges

$$\text{Cost(u)} = c_u = \sum_{i \in V_S} c_{ui}$$

Note that the calculation of this cost in implementation need not actually iterate over all supernodes if we already know the set of supernodes for which there is at least one original connection to *u*; all nodes not in this set would simply add zero to the cost of *u*.

Finally, with the cost of a node defined, we can calculate the reduced cost of merging nodes. If we wanted to merge two supernodes, *u* and *v*, into one supernode, *w*, we can easily calculate how much this merge would contribute to the lowering the overall summary cost: it would be the cost of the original supernodes minus the cost of the new supernode, $c_u + c_v - c_w$. As long as this sum is positive, merging *u* and *v* would reduce the overall summary cost. In our definition of reduced cost, we also normalize this value. The purpose behind this is to try to avoid suboptimal local minima in summary costs, particularly if a greedy approach is being used. For instance, if we are trying to find the “best” merge for supernode *u*, we may greedily pick the node which has the highest absolute reduced cost. However, this approach favors merges which may be inefficient despite a high absolute cost, when a more efficient (higher normalized reduced cost) merge may be present.

Therefore, it is important to use the normalized reduced cost, which we refer to as reduced cost from now on, when making decisions greedily. So, we define a reduced cost value of

$$s(u, v) = \frac{c_u + c_v - c_w}{c_u + c_v}$$

Note that this metric has a maximum value of 0.5, which would be the case if u and v had identical sets of neighbors and no corrections. This metric does not have a general lower bound, but any pair of nodes which have a non-positive reduced cost will not benefit the overall summary cost by being merged.

4 Related work

Outside of SSSC summaries, there has been research in many other methods of graph compression and summarization. In this section, we discuss other summarization and compression techniques and their purposes.

4.1 Supernodes based on node attributes

For graphs with annotated nodes, a straightforward approach is to merge nodes with the same attribute values all into a supernode, then add superedges to the summary, annotated with the support of the superedge. Algorithms that do so include the SNAP and k-SNAP algorithm[2]. The k-SNAP operation takes a parameter, k, specifying the number of supernodes desired in an output. By increasing or decreasing k, a user can vary the level of detail of summaries.

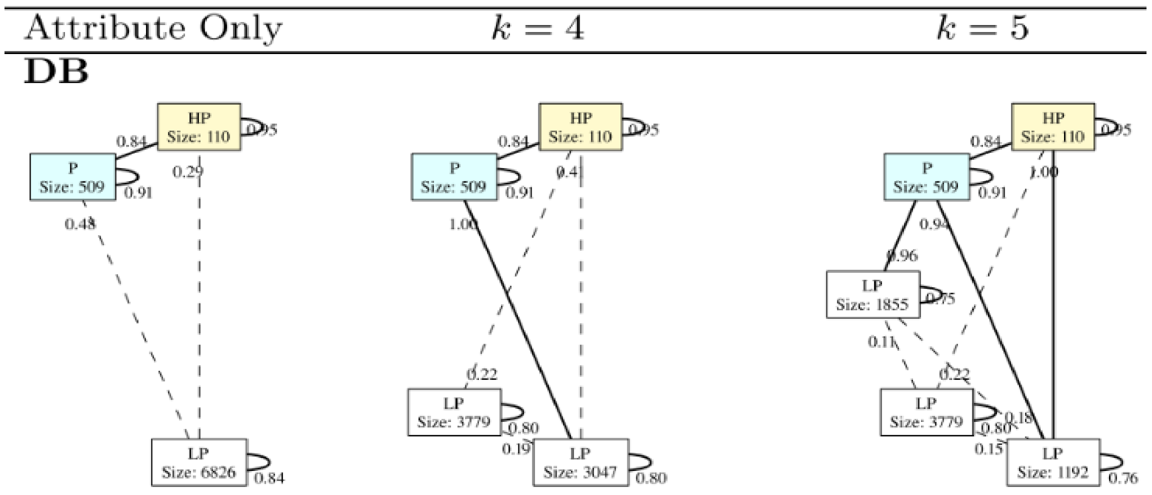


Figure 2 The k-SNAP algorithm in action. The original graph contains authors in the DBLP dataset as nodes, and an edge between two authors indicates that the authors co-authored a document. Authors are annotated with low prolific (LP), prolific (P), and high prolific (HP) based on how many papers they have published. Notice that, as k increases, we learn that there are subgroups of the LP group - notably, when k=5, there is an LP group which very often co-authors with P authors, and a group which very often co-authors with HP and P authors, and there is a third LP group which co-authors often with neither P nor HP authors. Using these results to drive analysis, it was found that LP authors in the group which strongly co-authors with P and HP authors publish more frequently than the other LP groups, on average, and an LP group which strongly co-authors with P authors published more frequently than an LP group which does not strongly co-author with P nor HP authors[2].

From Figure 2, notice that such a summary can reveal some interesting information about the graph. However, the summary does not encapsulate the structure of the underlying graph. Here, we highlight the difference between summaries which are meant to discover interesting trends (like data-mining), versus summaries which are meant to summarize the overall structure of a graph, making it easier to comprehend, explore, or store the graph. For a discovery driven summary, one desires an algorithm which automatically discovers interesting trends, rather than requiring input; the CANAL algorithm aims to improve upon the k-SNAP operation by automatically finding the most “interesting” value of k[3].

4.2 Modular Decomposition

Modular Decomposition is a general problem that is not limited to graph summarization[4]. In the context of graphs, a module, *M*, of a graph, *G*=(*V*,*E*), is a set of nodes such that for any $n \in V/M$, *n* is either 1) connected to every node in *M*, or 2) connected to no node in *M*. From this definition, we see that a module is also a set of nodes which all share exactly the same neighbors.

Some modules are subsets of other modules, so a modular decomposition is a tree of modules which makes clear the decomposition of modules into other modules. Figure 3 shows a graph and its modular decomposition tree. For example, {2,3,4} is a module because the nodes 2, 3, and 4 each have the neighbors {1,5,6,7} exactly. The set {2,3} is also a module, which is a subset of {2,3,4}. The modular decomposition of a graph offers a means of summarization with a natural way of "zooming in" by breaking up modules. However, at best, modular decomposition only merges nodes with exactly the same neighbors, which is different than SSSC summaries, which allow corrections. Still, there exist linear time (with respect to number of nodes and edges) algorithms for finding modular decomposition trees[4][5], which begs the question of whether such algorithms can be tweaked to allow corrections or not.

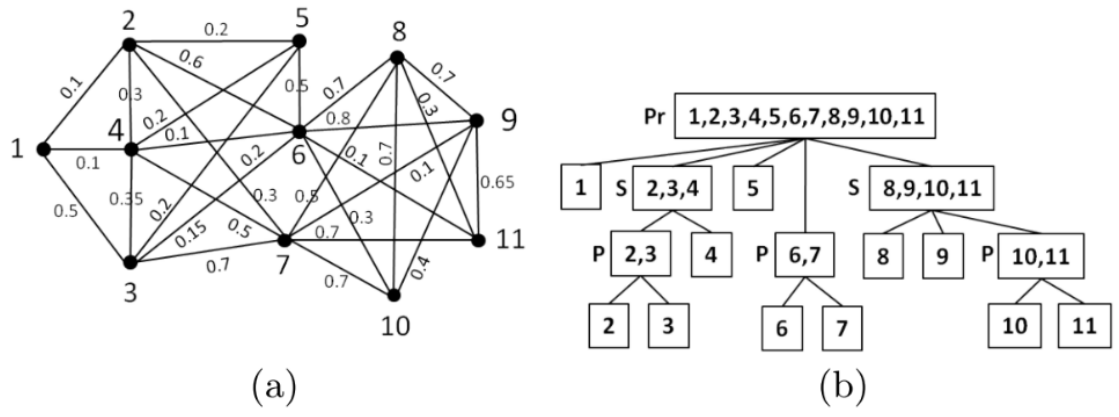


Figure 3 (a) A graph and (b) the graph's modular decomposition tree[6].

4.3 Compression with k^2-trees

An approach that does not fit our model of supernodes and superedges is to compress the adjacency matrix of a graph using k^2-trees. A k^2-tree utilizes the fact that most graphs based on real data are very sparse, so an adjacency matrix can be efficiently represented by a tree, where each branch represents a quadrant of the adjacency matrix: a 0 represents that all entries in that quadrant are 0, and a 1 indicates that at least one entry in that quadrant is a 1. This approach can be quite efficient, and can also be efficiently queried. Figure 4 shows an adjacency matrix and its k^2-tree representation. In [7], an extension to k^2-trees is proposed which efficiently represents annotated nodes and edges. This method can be efficiently queried as well.

However, while k^2-trees efficiently store graphs in a useful manner, they do not summarize the graph in any way. A user cannot look at a k^2-tree and learn about the structure of the graph, notice any interesting trends, or explore the graph.

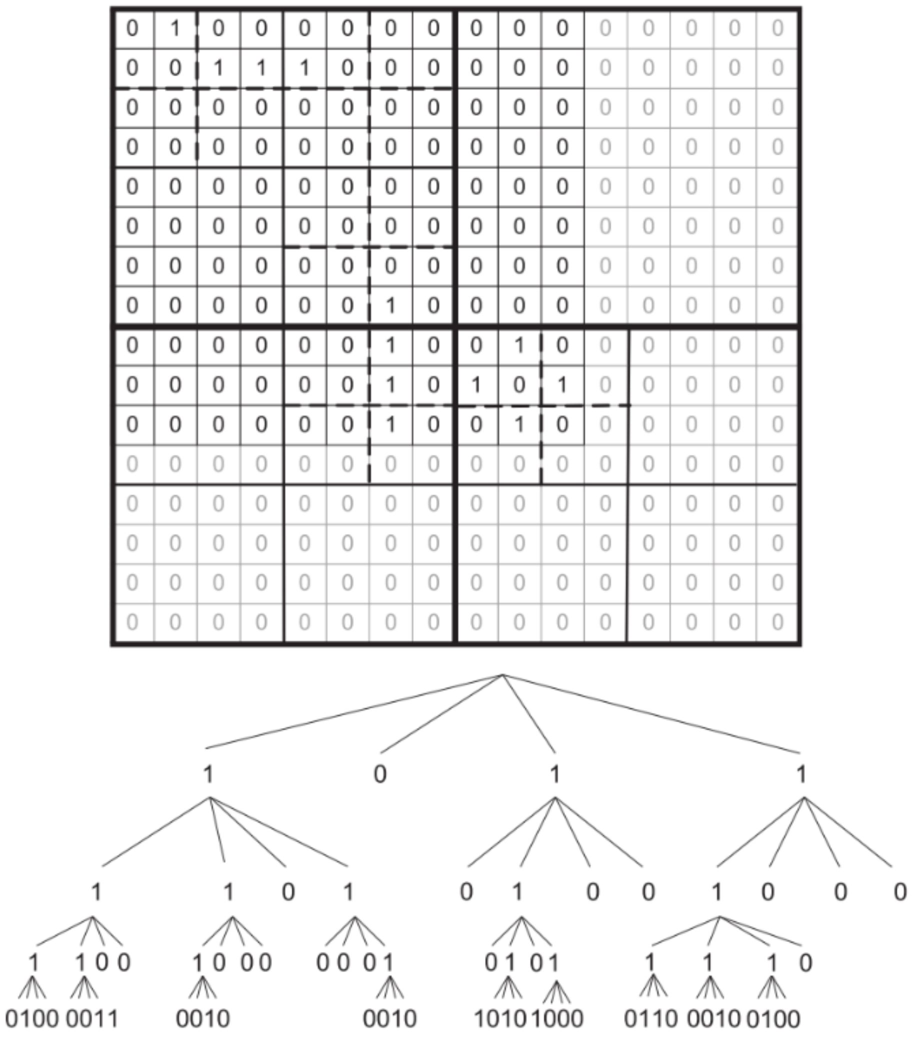


Figure 4 An adjacency matrix of a graph (upper), and its k^2-tree representation (lower)[7].

4.4 Supernodes based on motifs

We define a motif to be a pattern which appears frequently in a graph. For instance, a star pattern, or a fully connected subgraph are patterns that appear frequently in some databases. Figure 5 offers some more examples of motifs.

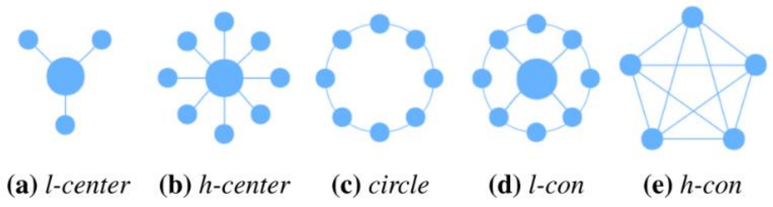


Figure 5 Examples of motifs[8].

Some approaches summarize by merging these motifs into a supernode. For instance, *ModulGraph*[8] defines the motifs in Figure 5, and summarizes based on those motifs. Figure 6 gives an example of a graph summarized by *ModulGraph*. Note that *ModulGraph* refers to these motifs as modules, but these are not the same as the modules discussed in section 4.2. We use the term motif to avoid confusion. [9] also aims to summarize using frequent subgraphs.

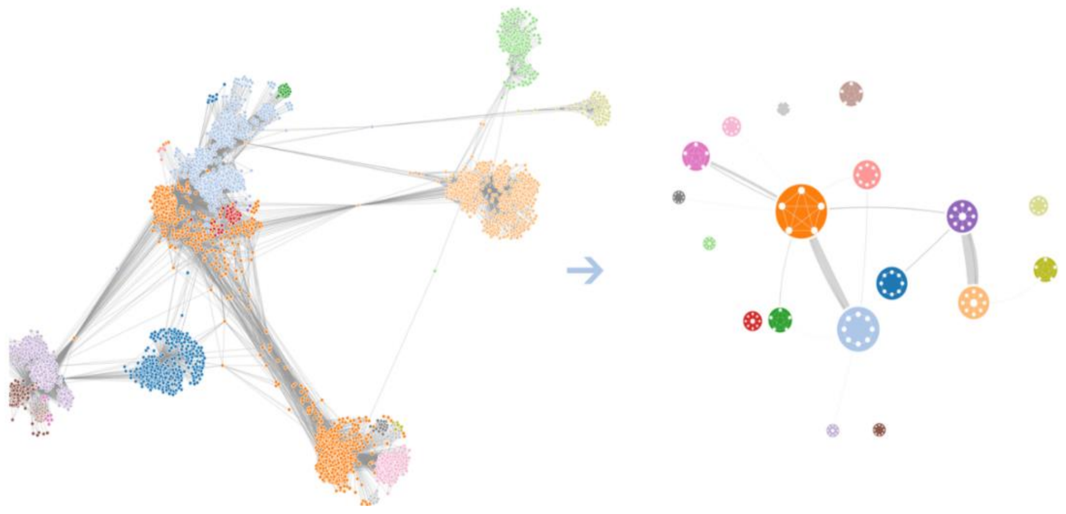


Figure 6 A summary produced by *ModulGraph*. Notice how the supernodes in the summary are annotated with which kind of motif they represent[8].

4.5 Approximate SSSC summaries

In addition to proposing SSSC summaries, [1] also proposes approximate SSSC summaries where some information is lost in summarization in order to achieve greater compression. Essentially, the set of corrections is incomplete, but in such a manner that one may have bounded error. Such compression is said to be *lossy*, as opposed to *lossless*. Though we do not explore approximate summaries in this paper, the topic is applicable in scenarios where high compression is more important than summary accuracy.

5 Overview of SSSC algorithms

We will discuss the differences between and optimizations of many proposed SSSC algorithms. We mostly do this with respect to the generalized algorithm we propose. We believe our abstract algorithm will make it easier to conceptualize the performance of such algorithms, as well as imagine optimizations. The general algorithm may also provide a gentle introduction to this family of algorithms for those who are unfamiliar.

5.1 Generalized algorithm for MDL summarization with corrections

The summarization algorithms we discuss can be generalized in the following abstract algorithm. In short, we pick seed supernodes from a set of unvisited supernodes until the set of unvisited supernodes is empty. For each seed supernode, we find a supernode to merge with the seed supernode (or possibly do nothing if there is no candidate merge node which reduces summary cost), merge the two supernodes, then update the unvisited set accordingly. Many of the functions in this algorithm are intentionally vague, as implementation can vary greatly depending on the specific algorithm. This general algorithm is only meant to be loosely followed, and we hope it provides an overall conceptual understanding of this family of algorithms that makes thinking about the differences and optimizations of specific algorithms easy to digest. Notice that we separate the generalized algorithm into Initialization, Merge, and Output phases, similar to how the authors in [1] did with the Uniform Greedy algorithm. In the Initialization phase, the starting states of the summary and the unvisited collection of nodes is initialized. In the Merge phase, merges of supernodes are performed. After the Merge phase, assignments of original nodes to supernodes are completed, and the Output phase adds superedges and corrections to the summary.

Below the algorithm, we will discuss the purpose of each function in more detail, and discuss how some algorithms have optimized certain functions.

```
Summarize(G)
    /* Initialization Phase */
    $G_S$ ← initialize_summary(G)
    unvisited ← initialize_unvisited($G_S$)

    /*Merge Phase */
    while unvisited is not empty
        seed ← pick_seed_node(unvisited)
        merge_candidates ← get_merge_candidates(seed)
        to_merge_with_seed ← get_best_merge(merge_candidates, seed )
        new_supernode ←merge_supernodes(seed, to_merge_with_seed )
        update_unvisited(unvisited, seed, to_merge_with_seed, new_supernode )

    /* Output Phase */
    put_edges_in_summary(G, $G_S$ )
    return $G_S$
```

Function: initialize_summary(G)

Returns an initial summary of *G* which acts as a starting point for the summarization algorithm. In most cases, this is simply a summary G_S with which every supernode contains exactly one original node, and there are no corrections, as every original edge can be represented with a superedge. However, one may perform some pre-processing in this step, such as merging all nodes which have exactly the same neighbors (such pre-processing was found in our research to be significant in some datasets).

Function: initialize_unvisited(G_S)

Returns a collection of supernodes in G_S which will be the initial collection of unvisited nodes at the start of the algorithm. This would usually just be a set of all supernodes in G_S. However, another possibility could be perhaps returning a subgraph of G_S if the algorithm is optimized to summaries only sections of the graph at a time. This possibility is explored further in section 9. Note that we use the term collection intentionally abstractly, as, depending on the algorithm, *unvisited* could be a set, a priority queue, or something else altogether. However, in most previous work, *unvisited* is a set from which we will pick nodes from uniformly at random.

Function: pick_seed_node(*unvisited*)

Returns some supernode that is in *unvisited*. Many algorithms pick a supernode from *unvisited* uniformly at random. However, one could instead pick based on metrics that suggest some supernode would make a good seed. We discuss this further, as well as show the results of some of our trained heuristics for this step, later in section 7.

Function: get_merge_candidates(*seed*)

Based on the seed node *seed*, we return a set of supernodes that we think may reduce the overall cost of the summary when merged with the seed node. Often, this is simply the set of 2-hop neighbors of *seed*, meaning those supernodes which contain at least one node that has an original 2-hop connection to some node in *seed*. However, others[10] have taken a different approach, instead using Locality Sensitive Hashing to find candidate merge nodes. In cases where getting the 2-hop neighbors of *seed* may be costly, perhaps because the graph is very dense, or any other case where there are simply too many 2-hop neighbors for *seed*, this approach may be more optimal than getting 2-hop neighbors.

Function: get_best_merge(*merge_candidates, seed*)

Returns the best supernode, or set of supernodes, which makes the best merge with *seed*. Note that get_best_merge and get_merge_candidates need not necessarily be two separate functions because they both contribute to the end goal of simply finding the best merge based on *seed*.

However, in most SSSC algorithms, we see a trend of first getting merge candidates, for instance getting all 2-hop neighbors or using LSH, and then pruning those candidates afterward. Such pruning would take place in get_best_merge.

For finding the best merge from *merge_candidates*, several methods have been proposed. The most straightforward is simply to return the candidate which has the highest reduced cost[1]. Other algorithms filter based on other metrics, like node degree[10], percentage of shared neighbors with *seed*[11], or reduced cost of the group[12]. Of these algorithms, one distinction is very important: those algorithms that may return multiple merge candidates, resulting in a group merge, and those which only return at most one merge candidate, resulting in a pair-wise merge. With both group and pair-wise merge algorithms, it should be noted that the number of supernodes decreases monotonically. However, group merge algorithms have the potential to greatly reduce the number of iterations and algorithm makes in the Merge phase, especially in the case of dense graphs or graphs with several dense subgraphs or cliques. In our experiments, we tested a group merge algorithm which continuously merges *seed* with nodes in *merge_candidates* (in order of decreasing reduced cost with *seed*) until merging no longer resulted in a positive reduced cost. We saw that, in the databases we tested, this did not lead to a significant difference in algorithm performance. Furthermore, we also discovered when training heuristics to improve pick_seed_node that supernodes which only contained one original node were more likely to lead to good merges, suggesting that most supernodes will not make a good merge immediately after merging already.

We looked into some other optimizations. First, when considering each merge candidate, one could terminate get_best_merge early if a perfect merge with a reduced cost of 0.5 is found, since there cannot be any higher reduced cost. One may also early terminate with a lower lower-bound, such as 0.4, to reduce the runtime of this step while still ensuring a good merge (0.4, in most cases, is still a good reduced cost). However, there is a risk that the returned merge will not be the best. We also considered a strategy in which only merges with a reduced cost having some lower bound would be considered, and if no merge candidates fit the lower-bound, that *seed* node is skipped for later evaluation. If a set number of skips occur consecutively, we lower the lower-bound, until we eventually get to a lower-bound of zero, and the algorithm acts the same as usual. The idea is that we would somewhat be simulating the greedy algorithm proposed in [1], without the maintenance of a heap to record all potential merges. Instead, we just skip seed nodes if we suspect there are still better seed nodes out there. However, we found that such a strategy did not lead to significantly better algorithm performance in our tests.

Function: merge_supernodes(*seed*, *to_merge_with_seed*)

This function simply merges *seed* with the node or nodes in *to_merge_with_seed*. Optimizations in this function are primarily implementation dependent rather than algorithm dependent.

Function: update_unvisited(*unvisited*, *seed*, *to_merge_with_seed*, *new_supernode*)

This function simply updates *unvisited* accordingly based on the merge that just occurred, or didn't occur. For example, if *to_merge_with_seed* contains a supernode or supernodes that were then merged with *seed* to create *new_supernode*, the proper action would be to remove *seed* and the supernode(s) in *to_merge_with_seed* from *unvisited*, then add *new_supernode* to unvisited. If no merge candidates were found to lead to a positive reduced cost, *seed* would simply be removed from *unvisited*. This ensures the monotonic decrease in the size of *unvisited* when *unvisited* is initially a set of all supernodes.

However, this need not be the only possible way to implement this function. For instance, if our algorithm were only summarizing sections of a graph at a time, we might need to reinitialize *unvisited* in a different area of the graph. This is further explored in section 9.

Function: put_edges_in_summary(G, G_S)

In this function, we assume that G_S has node assignments to V_S. Then, as discussed previously, there is exactly one optimal way to build E_S and C accordingly, and we do so in this step. If one's implementation already has E_S and C completed, this step can be skipped, but it is usually much easier and cheaper to do the earlier parts of algorithms without explicitly keeping track of E_S and C, and then building E_S and C at the end. This is because every merge may require many updates to E_S and C, and these updates, of course, will not be free.

5.2 Recognizing points of optimization in SSSC algorithms

We hope our abstract algorithm makes it very easy to recognize points of optimization for SSSC algorithms, as well as recognizing which points currently proposed algorithms aim to optimize. For instance, it is somewhat difficult to figure out what optimizations are occurring in [10], but, when viewing the proposed algorithm with the lens of the abstract algorithm, it is clear that the authors in [10] are aiming to optimize how one finds the best merge from a seed node using Locality Sensitive Hashing and pruning, which optimizes the functions **get_merge_candidates** and **get_best_merge**.

The abstract algorithm also pinpoints the step where group-merge algorithms, such as those proposed in [11][12], aim to reduce the number of iterations by returning multiple supernodes from **get_best_merge**. However, the algorithm is abstract enough to conceptualize the optimizations of both algorithms without needing to look into the specifics of their implementations.

This abstraction also points out a function that there has been little to no attempts to optimize: **pick_seed_node**. We believe that the order in which seed nodes are picked can affect summary quality, especially in the beginning iterations of SSSC algorithms. As such, picking seed nodes uniformly at random, which most proposed algorithms do, has the potential to be very suboptimal. We trained heuristics based on a small sample size of nodes to influence how we pick seed nodes, and we discuss our findings in section 6.

5.3 Algorithms used in this paper

In section 5.2, we discuss the optimizations of some iterations of SSSC algorithms. However, in our experiments and the rest of this paper, we mainly focus on just a couple algorithms, which we will give an overview of here.

The Uniform Randomized algorithm, originally proposed in [1], initializes *unvisited* to a set of all supernodes, then iteratively picks seed nodes uniformly at random, gets two-hop neighbors of the seed node as merge candidates, then merges the seed node with the highest positive reduced cost candidate.

The Non-Uniform Randomized algorithm, or trained heuristic algorithm, is like the Uniform Randomized algorithm, but instead seed nodes are picked non-uniformly randomly based on a trained heuristic that favors nodes that are likely to have a good best merge.

The Greedy algorithm, originally proposed in [1], is fundamentally different from the Uniform Randomized Algorithm in that it stores a max-heap of all potential merges in the graph, and then always performs the best merge. We did not test this algorithm because it was significantly slower than other methods, but we do discuss optimization involving localization of summarization as future work in section 10.

The output phases of each of these algorithms are all the same: after the merge phase, we have assignments of original nodes to supernodes. From these assignments, the optimal superedge and correction set is generated by iterating over each supernode, considering all its neighbors that which it is connected to by an original edge, determining whether there should be a superedge between the two supernodes, then adding (or not adding) to the set of superedges and adding (or not adding) corrections for each original edge between the two supernodes.

6 Overview of performance and scalability of SSSC

We tested the Uniform Randomized SSSC algorithm on a range of real and synthetic datasets, and we found that we can consistently get theoretical compression ratios of about 0.30-0.70, as well as implemented compression ratios of about 0.20-0.57. We discuss the complications involved with summarizing RDF data. We also explore the scalability of SSSC summarization with respect to runtime and compression ratios.

6.1 Experimental overview: complications with RDF data

Recall that RDF datasets, or sets of subject-predicate-object triples, can be conceptually represented as a multi-edged, edge-annotated, directed graph. For our purposes, we simplify this graph to be an undirected, single-edged, un-annotated graph. This made experiments much simpler, but it should be noted that most SSSC algorithms should easily adapt to directed graphs. Representing the multi-edged, edge-annotated nature of RDF datasets is a problem that needs to be addressed by SSSC algorithms more completely.

Also, RDF datasets contain attribute nodes which describe information about a node. For instance, an author, Author1, node may represent an author whose name is ‘John Smith’, and this may be represented with the triple (‘Author1’, ‘#has_name’, ‘John Smith’). For SSSC algorithms, such attribute nodes are problematic because any node may have several attributes that are unique to that node, making merges with that node quite difficult. For instance, consider a case in a DBLP database where two authors have all the same edge connections – they co-author all the same papers, for instance – but because they have different first, middle, and last names, and perhaps other differing attribute nodes like address, etc., the reduced cost of merging these two authors is non-positive. The attributes of these authors overshadow their graphical similarities. In other words, just because node attributes can be well-represented in a graph, it does not mean that node attributes are representative of the graph structure we are trying to summarize. For this reason, we remove such attribute nodes altogether. We also define attribute nodes more broadly as any one-degree node. The justifications are twofold: 1) including attribute nodes greatly hinders the ability of SSSC summaries because they are not part of the core graph structure we aim to summarize, and 2) attribute nodes, being a special case, should be able to be efficiently stored in ways outside of summaries anyway, perhaps with property tables, vertical partitions[15], or other methods.

In addition to attribute nodes, which we generalize as one-degree nodes, RDF data also has some very high degree descriptive nodes. For instance, RDF data usually has type nodes, for which most nodes might have a connection to a type node representing the node is of that type. If an RDF graph contains millions of nodes, but much fewer types of nodes, perhaps hundreds or thousands, it is obvious that these type nodes have a very high degree. Since the SSSC algorithms we test involve checking a node's two-hop neighbors for merge potential, having such high degree nodes hugely slows down our algorithms, so we remove type nodes. Again, we can justify this because we want to fully test these algorithms, and, since nearly every node has a type, this information can be stored in a more efficient manner than in summaries. Additionally, type nodes are descriptive of nodes and do not represent graph relationships between other nodes.

So, we perform the removal of one-degree nodes and type nodes before beginning summarization. This generally leaves us with a much smaller set of edges than the initial dataset, but what is leftover is really the core of the graph represented in the triples, which is what we are aiming to summarize. Lastly, before starting our SSSC algorithms, we preprocess the graph by merging all nodes which have exactly the same set of neighbors into a supernode. We call such nodes “identical” nodes. Note that such nodes have different labels and are not truly identical. We merge these nodes because we can do so rather efficiently just by iterating through each node and checking its neighbors, and not doing so would obviously lead to worse summaries – this step ensures that all merges with a 0.5 reduced cost, the best possible reduced cost, are made before less optimal merges are made. In our tests, merging these “identical” nodes often compressed the data by 5-30%, rendering the process significant. As such, our plots of compression ratio over time (or percent completion) will not usually start at 1, but a lower value such as 0.95 or 0.70.

6.2 Data sets

We ran tests on the real datasets DBLP, Internet Movie Database (IMDB), and WordNet, as well as the synthetic data set Berlin SPARQL Benchmark (BSBM). We will describe each dataset, and then show a table of stats about the data in table 1.

6.2.1 DBLP

The DBLP dataset contains bibliography data about Computer Science data. Nodes may be authors, publications, conferences, etc. while edge’s may represent relationships such as cited_by or author_of. We use four versions of the DBLP dataset from [13]: DBLP-1, DBLP-2, DBLP-3, DBLP-4. Each DBLP-*n* dataset contains approximately *n* million triples before simplification.

6.2.2 IMDB

The Internet Movie Database[2] contains information about movies. Nodes may be actors, movies, etc. while edges may represent a relationship like acted_in. We use a subset of the complete IMDB dataset.

6.2.3 WordNet

WordNet[3] contains lexical information on words, such as nouns, verbs, adjective, etc.

6.2.4 BSBM

Berlin SPARQL Benchmark is a synthetic dataset created for SPARQL benchmarking. The benchmark is built around an e-commerce use case, including products, vendors, consumers, and reviews of products. We generated three datasets, BSBM-3, BSBM-5, and BSBM-7 from the generator[4] [14]. Each BSBM-*n* has roughly *n* million triples before simplification.

Dataset	Original number of triples (K)	Number of triples after GS (K)	Number of nodes after GS (K)	Density after GS ($\times 10^{-5}$)	Average node degree after GS	Theoretical compression ratio after merging identical nodes
DBLP-1	1614	199	118	2.9	3.372881	0.462
DBLP-2	2241	431	241	1.5	3.576763	0.526
DBLP-3	2995	597	323	1.1	3.696594	0.559
DBLP-4	3966	816	428	0.9	3.813084	0.586
IMDB	3267	1268	169	8.6	15.00592	0.738
WordNet	1043	163	339	0.8	0.961652	0.909
BSBM-3	3565	939	326	1.8	5.760736	0.945
BSBM-5	5319	1402	484	1.2	5.793388	0.963
BSBM-7	7074	1866	642	0.9	5.813084	0.972

Table 1 The properties of each dataset after data simplification and merging identical nodes. "GS" stands for graph simplification in this table. Notice how the graph simplification significantly reduces the number of triples in the data set, indicating that several triples represented edges to one-degree nodes or were representing multiple edges between two nodes (there were reduced to just one edge in simplification). Also

[2] http://www.imdb.com
[3] Princeton University "About WordNet." WordNet. Princeton University. 2010. Available at http://wordnet.princeton.edu
[4] Generator available at http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

notice IMDB has a particularly high density, and that, in many of the data sets, simply merging identical nodes, or nodes with exactly the same neighbors, produced a significant compression ratio.

6.3 Resulting summaries

In this section, we discuss the results of running the Uniform Randomized algorithms on these data sets.

In Figure 7, we can see that, even though merging identical nodes alone sometimes produced a low compression ratio, running the algorithm still significantly decreased the compression ratio. DBLP-1, for instance, could achieve a compression ratio of 0.462 simply from merging identical nodes, and summarization further reduced the compression ratio to 0.289.

In Figure 8, we can see the theoretical compression ratios compared with the compression ratios in our implemented storage. Recall that the theoretical compression ratio ignores node to supernode mappings, and assumes that the cost of storing a superedge is equal to the cost of a correction. However, in our implementation, our node to supernode mappings took a significant amount of space the nodes are identified by URIs, which can be lengthy strings. Also, each of our edge corrections stores both URIs that the correction pertains to, while superedges are just represented with the two supernodes, represented as ints. So, superedges cost significantly less to store than corrections. For more on storage implementation details, see section 8. What we can conclude from this is that there is no simple relationship between theoretical compression ratio and our implementation compression ratio, because our implementation compression ratio depends on many additional factors that the theoretical compression ratio does not take into account, including the cost of storing original nodes vs supernodes, the number of nodes in the original data set, and the difference in cost for storing corrections vs superedges. Figure 8 reflects this discrepancy, as some implementation compression ratios are higher than their corresponding theoretical compression ratios, and some are not. Still, the implementation compression ratios provide significant compression, ranging from 0.204 for IMDB to 0.563 in DBLP-4.

In Figure 9 we can see the breakdown of each summary. We notice that, consistently, the number of supernodes in the summaries are much fewer than the number of nodes in the original graph. We see a similar trend for superedge and edges. Finally, we see that the number of corrections, of which the strong majority are additions, is significantly more than the number of superedges in the summaries. In table 2, we see that the proportion of original edges which are represented as additions range from 0.189 in DBLP-1 to 0.484 in BSBM-7. In a way, a set of additions forms its own triple store, while the superedges with a very small set of subtractions represent the parts of the original data which easily lent itself to summarization.

Dataset	# Additions / # original edges
DBLP-1	0.188939
DBLP-2	0.217688
DBLP-3	0.232631
DBLP-4	0.248148
IMDB	0.228539
BSBM-3	0.457129
BSBM-5	0.478898
BSBM-7	0.484246
Wordnet	0.42714

Table 2 The proportion of the number of additions in a summary over the number of edges in the original data, or the proportion of original edges which are represented by additions in the summary.

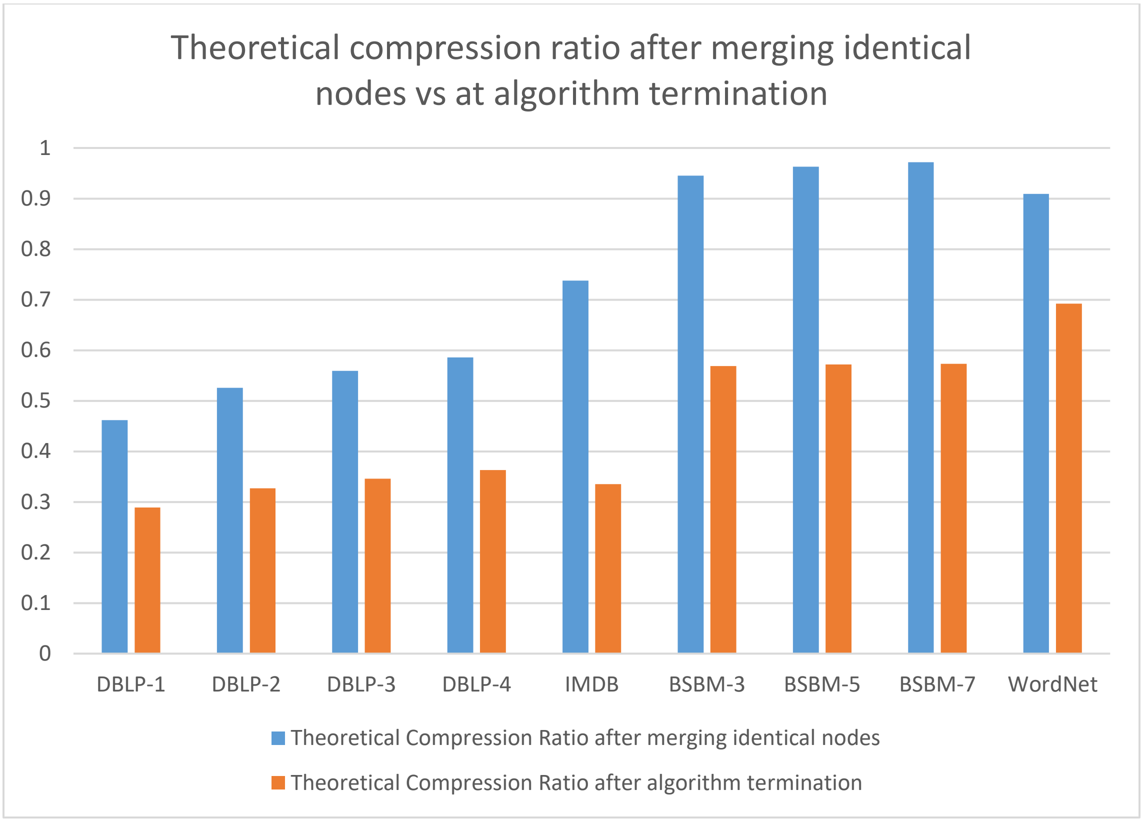


Figure 7 Theoretical compression ratio after merging identical nodes vs at algorithm termination.

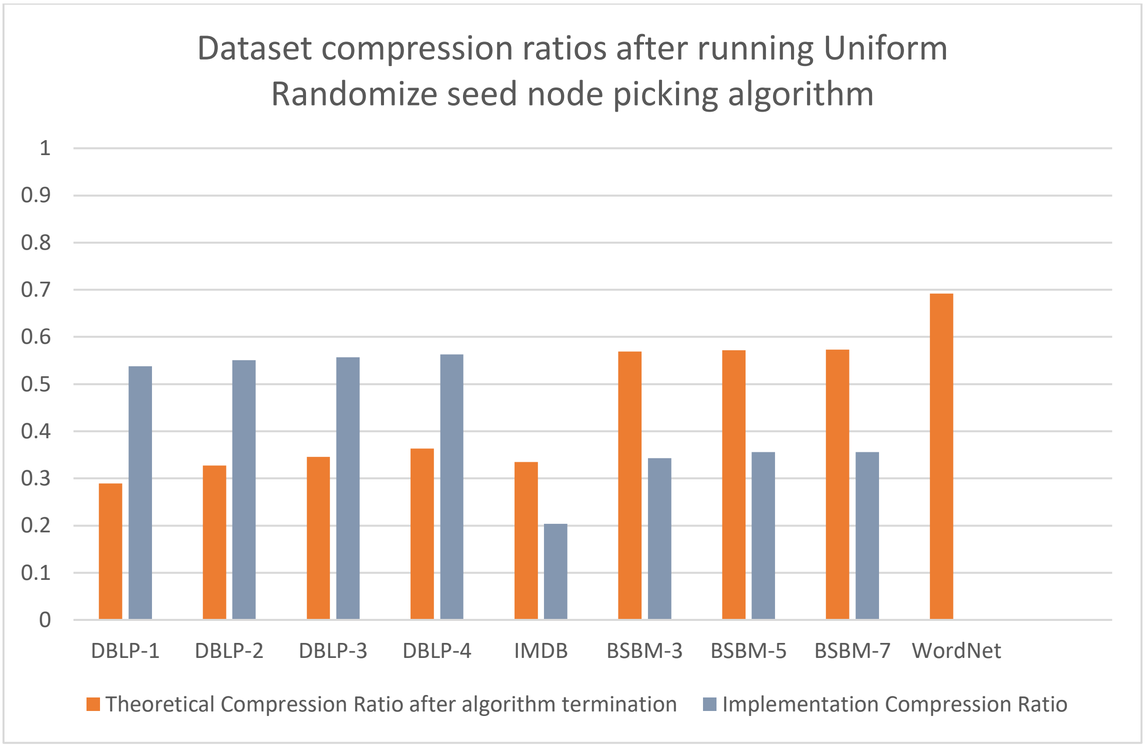


Figure 8 Dataset compression ratios after running Uniform Random seed node picking algorithm.

The summary for WordNet was not exported, so it does not have an implementation compression ratio.

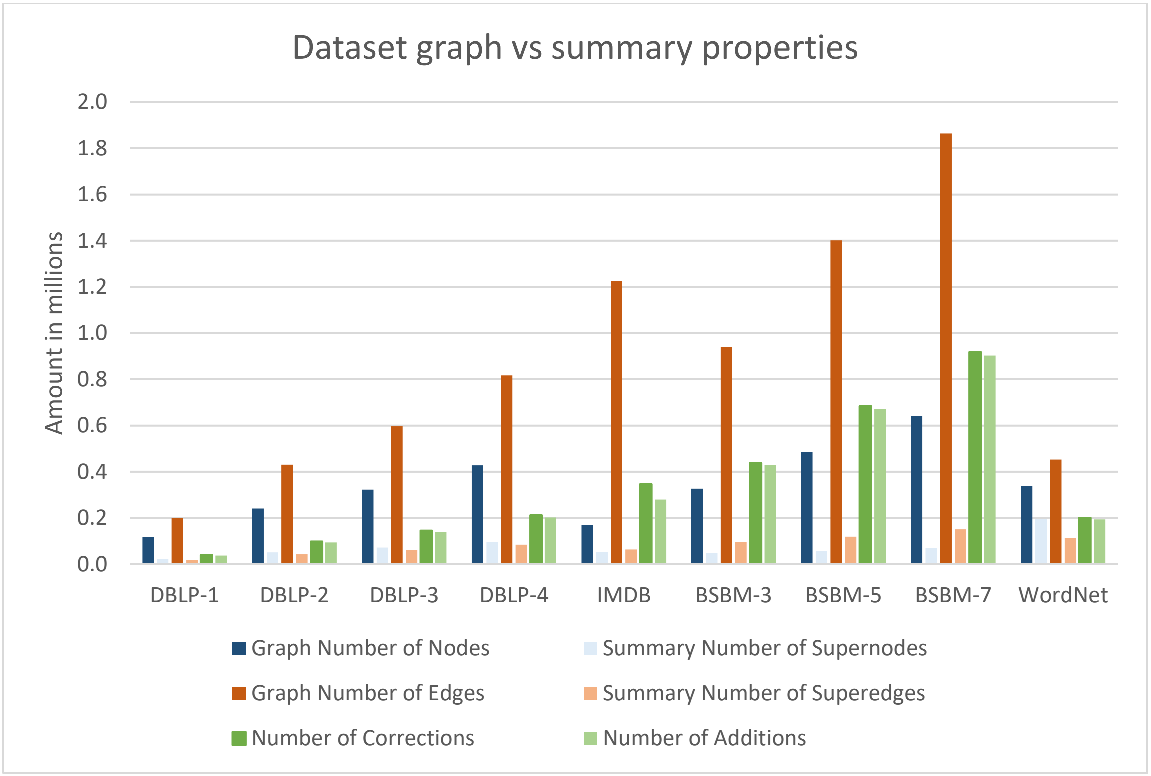


Figure 9 Dataset graph vs summary properties. For each dataset, we breakdown the dataset's number of nodes and edges, compared to the resulting summary's number of supernodes and superedges. We also include the number of corrections and number of additions in each summary. Number of subractions is not included becasuse it is simply the number of corrections minus the number of additions, and the number of subtractions was always insignificant compared to the number of additions.

6.4 Runtime

In Table 3, we see the breakdown of the runtime of summarizing the data sets. We can see that the runtime of merging the identical nodes is negligible compared to the merging and output phases of the algorithm, taking mere seconds as opposed to hours. Notice also that the runtime of the output phase is significant: in WordNet tests, for example, the output phase runtime is more than 10 times longer than the merge phase runtime. However, we omit the output phase runtime from most of our analyses for two reasons: we believe the relatively long runtime is due to inefficiencies in our implementation that we could not fix due to time constraints, and the output phase was simply not the focus of our tests as there is little variation in the output phase among SSSC algorithms.

Dataset	Merge Identical Runtime (s)	Merging Phase (h)	Output Phase (h)	Total Runtime (h)
DBLP-1	25	0.130	0.072	0.208
DBLP-2	62	0.481	0.459	0.956
DBLP-3	90	0.835	1.063	1.924
DBLP-4	129	1.374	2.287	3.697
IMDB	43	1.831	1.620	3.462
BSBM-3	93	39.058	7.163	46.247
BSBM-5	102	69.453	16.189	85.671
BSBM-7	113	106.633	35.441	142.106
WordNet	6	0.232	3.100	3.334

Table 3 The runtimes of the three summarization phases for each data set.

Since the DBLP and BSBM datasets are related in that the structure of the datasets are similar, we show these runtimes over the different database sizes in Figures 10 and 11. We omit the time to merge identical nodes, since it is negligible, and the output phase runtime for reasons mentioned previously. We compare the sizes of the databases by the number of triples (after RDF simplification) by dividing by the number of triples in the smallest datasets (DBLP-1 and BSBM-3, respectively. This provides a simple ratio for representing the growth in the size of the datasets. Similarly, we divide the runtime of the merge phases of each test by the runtimes of the merge phases in the smallest dataset. As a result, we see that the merge phase runtimes is almost linearly related to the increase in the number of triples. In Figure 11, we see the BSBM runtime ratios fall almost directly on a linear trendline, with BSBM-5 being slightly below the trendline and BSBM-3 and BSBM-7 being slightly above the trendline. We see a similar trend in Figure 10 with the DBLP datasets: the slope from DBLP-3 to DBLP-4 is slightly higher than the slope from DBLP-2 to DBLP-3, which is in turn slightly higher than the slope from DBLP-1 to DBLP-2. This can be noticed by the fact that the points for DBLP-2 and DBLP-3 lay below the trendline while the points for DBLP-1 and DBLP-4 lay above the trendline. When we take into account the runtime complexity of the merge phase, we can explain this trend. The runtime complexity of the merge phase for the Uniform Randomized algorithm is

$O(|V| * d^3)$ [1] $= O(|V| * (|E| / |V|)^3) = O((|V| * (|E| / |V|) * (|E| / |V|)^2) = O(|E| * d^2)$,

where d is the average node degree of the graph. The number of edges, $|E|$, is just the number of triples, and we can find the average node degree in Table 1. When we look at the average node degrees for the DBLP datasets, we see that they are 3.37, 3.58, 3.70, 3.81 (increase of 0.54) in order from DBLP-1 to DBLP-4. The BSBM datasets' average node degree also increases as the dataset size increase, but only from 5.76 to 5.81 (increase of 0.05). Since the runtime complexity is related to both number of triples and average node degree, this explains why the datasets' merge phase runtimes increase slightly nonlinearly with number of triples, as the average node degrees were not constant. Also, since the change in average node degree is much less in the BSBM datasets than it is for the DBLP datasets, the nonlinearity is less pronounced in the BSBM datasets. It is also worth noting that the relatively higher average node degree of BSBM datasets compared to DBLP contributes to BSBM's much longer merge phase runtime, as seen in Table 3.

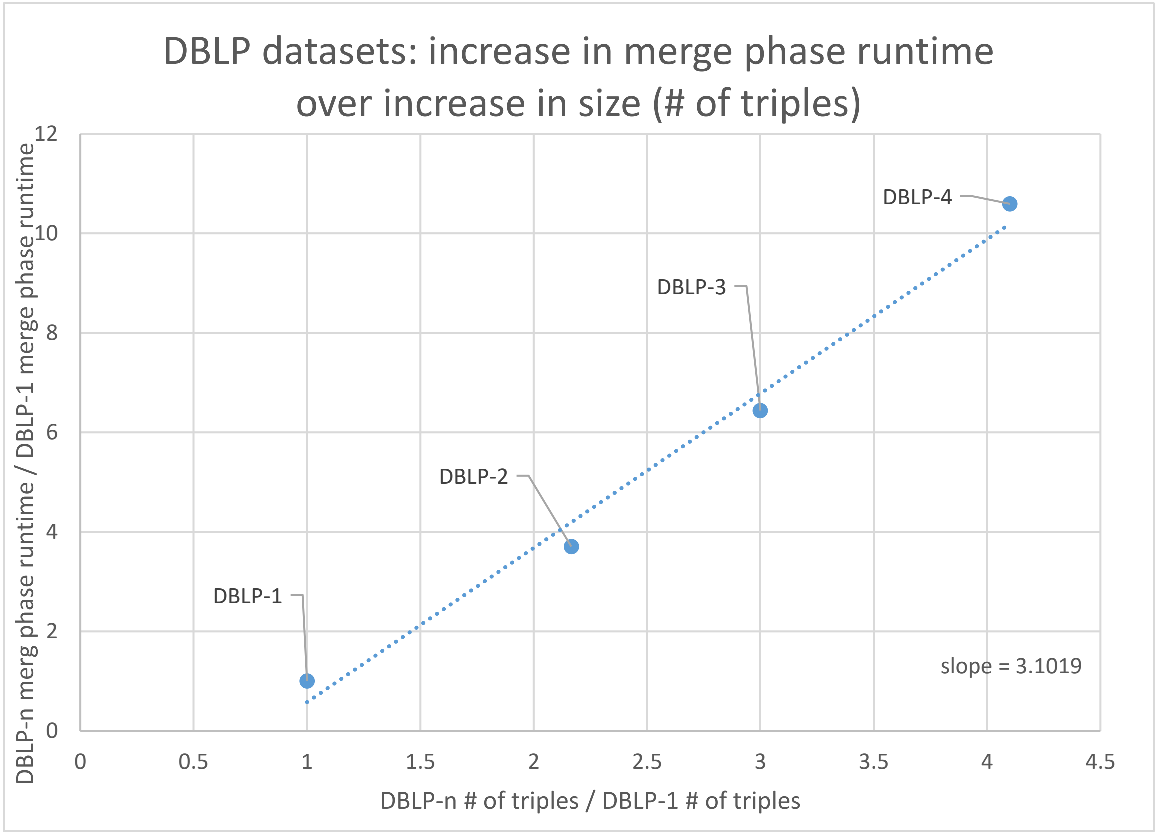


Figure 10 DBLP datasets: increase in merge phase runtime over increase in size (# of triple). This plot compares the sizes (# of triples) and merge phase runtimes of summarizing the different DBLP datasets. We use DBLP-1 as a baseline, so we divide each DBLP-*n* value for # of triples and runtime by the respective value from DBLP-1. The result is a ratio with DBLP-1. We do the same for the runtimes. We do notice that the runtime increase does not appear to be completely linearly related to # of triples for DBLP-3 and DBLP-4, and this can be attributed to the fact that merge phase runtime is also related to average node degree of the datasets, and the average node degrees in each DBLP-*n* increase as *n* increases.

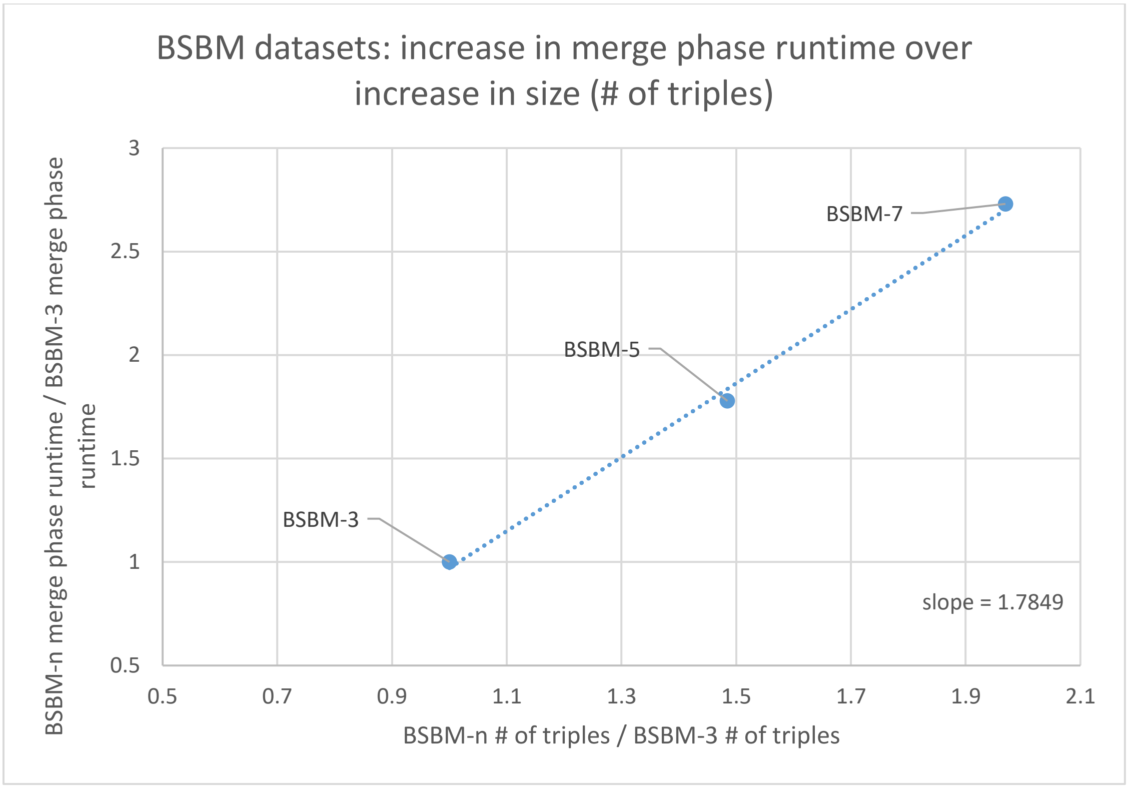


Figure 11 BSBM datasets: increase in merge phase runtime over increase in size (# of triples). This plot compares the sizes (# of triples) and merge phase runtimes of summarizing the different BSBM datasets. We use BSBM-3 as a baseline, so we divide each BSBM-*n* value for # of triples and runtime by the respective value from BSBM-3. The result is a ratio with BSBM-3. We do the same for the runtimes. We do notice that the runtime increase does not appear to be completely linearly related to # of triples for DBLP-3 and DBLP-4, and this can be attributed to the fact that merge phase runtime is also related to average node degree of the datasets, and the average node degrees in each DBLP-*n* increase as *n* increases.

6.5 Scalability

In terms of compression quality measured by compression ratio, we can see that the Uniform Randomized SSSC algorithm scales well because the compression ratio of similarly structured graphs does not significantly increase as the data set grows. In Figure 8, we can see that the BSBM datasets maintain a theoretical compression ratio of about 0.57 for all three datasets. In Figure 9, we see that the DBLP datasets maintain an implementation compression ratio of about 0.55, although the theoretical compression ratio does increase gradually. Also in Figure 9, the BSBM datasets maintain an implementation compression ratio of about 0.35.

In terms of runtime, the Uniform Randomized algorithm merging phase scales nearly linearly with respect to number of triples for similarly structured graphs when average node degree is constant. The reasons for this are quite clear: in the merging phase, the unvisited set size begins as the number of original nodes and decreases by one at each iteration, so the number of iterations is roughly equal to the number of nodes. For each iteration, the three-hop neighborhood of the seed node is calculated in order to find the seed node's best merge. So, the merge phase runs O(|*V*|**3hop*) where *3hop* is average cost of getting the three-hop neighborhood of a node. We could also write $O(|V| * d^3)$ where *d* is the average degree of a node. The |*V*| explains why the merging phases increase almost linearly in the datasets in Figures 10 and 11. The d^3 helps explain why the BSBM datasets take dramatically longer to summarize that the DBLP datasets: BSBM-3 actually has less nodes than DBLP-4 after simplification, but the average degree of a node in BSBM-3 is about 5.76 while DBLP-4's average node degree is 3.37. BSBM-3's merge phase runtime is more than 20x slower than DBLP-4's runtime because of this difference in average node degree. We also found that the last few percent of the iterations of the merge phases tended to take longer than earlier iterations, and this may have affected larger datasets more dramatically, which may also contribute to BSBM-3's longer runtime.

6.6 A note on summary visualization

In past research, summaries have been used in order to simplify visualizations of large graphs. While we do not dismiss the application, we did not focus on visualization in our experiments. Even though the supernodes and superedges in summaries do produce much smaller graphs than their corresponding original graphs, plotting said summaries is still very costly, and would still produce a large graph that is hard to visualize. We did perform some small-scale visualization tests and, consistent with [1], we did find that summaries did well represent bipartite subgraphs. However, it was difficult to determine if the summaries were otherwise visually meaningful, as such determination is subjective and would require intimate knowledge of the original datasets.

6.7 Conclusions

We verify that the Uniform Randomized SSSC algorithm produces good compressions, with theoretical and implemented compression ratios ranging from 0.2-0.7, in real and synthetic data. We establish the scalability of the algorithm in terms of compression ratio and linear runtime with respect to number of triples, with average node degree also being a factor in runtime. We show that merging "identical" nodes, or nodes with exactly the same set of neighbors, can alone summarize real data sets. Our tests were also performed on some of the largest data sets we know of to test SSSC algorithms.

7 Picking seed nodes heuristically

We came up with a couple of hypotheses about the effect of seed node order in the iterations of SSSC algorithms. First, we hypothesized that seed nodes that have at least one good merge candidate will often have easily-calculable characteristics in common with other seed nodes that have at least one good merge candidate (We will define goodness and easily-calculable in section 7.1). Therefore, if we use a sample of nodes to train a heuristic to score the likelihood of a node having a good merge candidate based on some characteristics of the node, picking seed nodes based on such a heuristic should outperform picking a seed node uniformly at random. Using node characteristics such as supernode size, degree, number of 2-hop neighbors, cluster coefficient, and proximity to articulation points, we created such heuristics, and have experimental evidence to support this hypothesis.

We also hypothesized that suboptimal (i.e. uniformly at random most of the time) order of picking seed nodes would lead to an overall worse quality summary upon algorithm termination, as opposed to a greedier order where higher reduced cost merges are made before lower reduced cost merges. However, our data is inconclusive with regards to this hypothesis. Picking seed nodes using our trained heuristic does not appear to produce a summary with a better final compression ratio in all cases.

7.1 Experiment setup

For our tests, we needed to produce a heuristic based on easily calculable characteristics of a node. For our purposes, we consider "easily-calculable" to be $O(|V| + |E|)$ and $O(|V| * 2hop)$ where *2hop* is the average cost of getting the two-hop neighbors of a node. In other words, we considered metrics that could be calculated in a traversal of the graph, or which could be calculated with a traversal to a node's two-hop neighbors given a node. For calculating these metrics, we used the python[5] igraph library. It should be noted that the runtime of calculating these values using the python igraph library was not found to be significant compared to the overall summarization runtime in our implementation.

Note that our metrics are based on the characteristics of the original graph nodes, but our heuristic must be applied to supernodes, which may contain several original nodes. To address this, we found the graph metrics for each node a sample supernode contained, and recorded the average, median, minimum, maximum, and standard deviation of the metrics. These aggregate functions were the features we used in our heuristic. An overview of the node characteristics used as training data is given in section 7.2.

Our heuristics used node characteristics to assign an integer score to nodes (with a minimum score of 1), with a higher score indicating a higher likelihood that the node has a good best merge. We

[5] Pre-compiled windows binary available at http://www.lfd.uci.edu/~gohlke/pythonlibs/#python-igraph

produced training data for our heuristics by first performing the Uniform Randomized algorithm, which picks seed nodes uniformly at random. At increments of every 5% of the algorithm completed, we sampled 350 nodes uniformly at random, recorded each supernode's characteristics, and recorded highest possible reduced cost that could be produced by merging a sample node with one of its two-hop neighbors. We then labeled each sample node as follows: if the best reduced cost for this sample node is non-positive, label 'bad'; if the best reduced cost for this sample node is in the 75th percentile of best positive reduced costs in this sample, label 'good'; else, label 'okay.' Later, we trained decision trees[6] on this data (a separate decision tree at every 5% of this algorithm) to classify nodes as 'good,' 'okay,' or 'bad' using the characteristics of each node. From these decision trees, we chose heuristics that would favor nodes with behaviors that would classify the nodes as 'good.' An example of how we produced heuristics based on the decision trees is on Figure 12.

Once the heuristics were produced, we summarized the original data again, this time picking seed nodes non-uniformly at random based on the node scores generated by the trained heuristic. We can compare the performance of using the trained heuristic against the performance of a normal run of Uniform Randomized. It should be emphasized that the generation of training data from and the training of decision trees is performed separately from our summarization which uses the outputted heuristic. This effectively removes the overhead in generating training data and training from our analyses. Ideally, the training data and the decision trees would be generated and trained online as the algorithm ran, so that the overhead of generating the heuristic would be included in our analyses. However, the cost of taking a small sample of nodes (in our case, only 350), generating

[6] To produce the decision trees, we used the R Programming Language[16] and the rpart package [17].

classification data, and training a decision tree with only 350 samples is relatively cheap – about the cost of 350 iterations of the algorithm, with some overhead to generate the classification data and train a decision tree – so we are confident our tests are still very suggestive of how such an online heuristic algorithm would run.

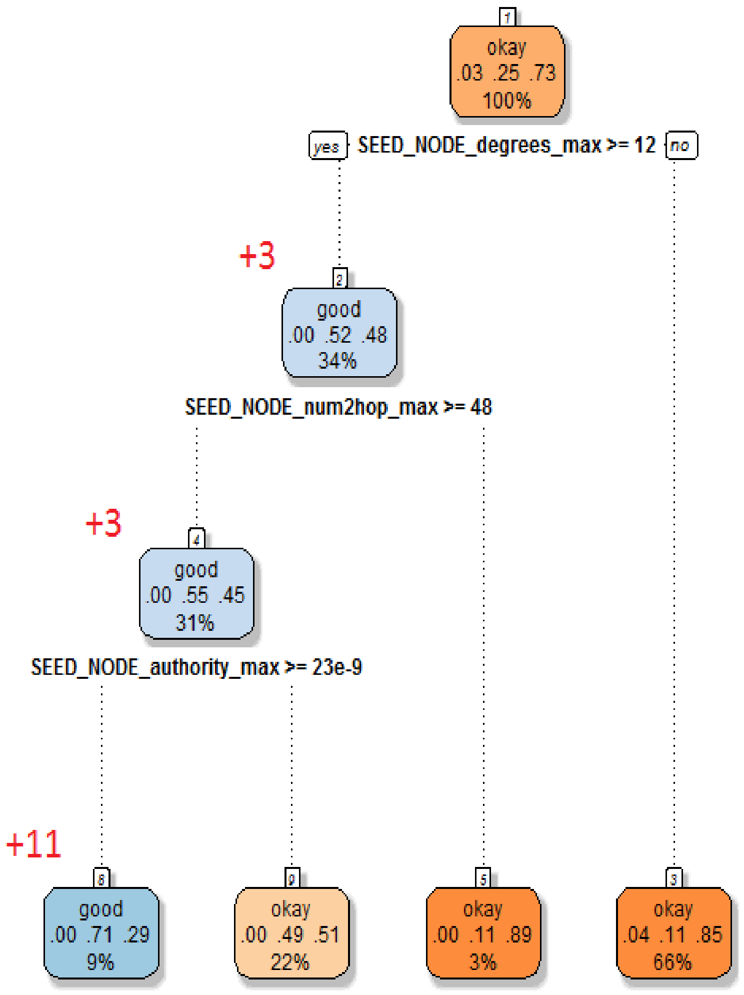


Figure 12[7] An example of a scoring mechanism used for our heuristic scoring. For example, if a supernode has a maximum node degree (out of all the original nodes it contains) greater than or equal to 12, we move left down the decision tree. This node's score is increased by 3. If the supernode also has a maximum number of two-hop neighbors greater than or equal to 48, we move left down the tree again, and the supernode's score

[7] Decision tree graphic produced using Rattle R package[18].

increases by 3 again (note that scores increases are cumulative as we move down the tree). Finally, if the supernode also has a maximum authority score greater than or equal to 23e-9, the supernode's score increases by 11.

To pick a seed node based on our heuristic, we used the heuristic to give each supernode an integer score, with a minimum score being one. Since the set of possible scores was rather small, we stored a list of supernodes for each score. From this, we could generate a random integer between 0 and the sum of all supernode scores – 1, and use this number to quickly find the corresponding supernode. When a node was removed, we added it to a set of removed nodes rather than altering our lists, and we checked whether a node was removed before returning it from a pop of the unvisited set in the algorithm. If the size of the removed set reached half the size of the unvisited set, we rebalanced our lists, removing those items which were in the removed set.

7.2 Overview of heuristics

In this section, we overview the node characteristics we used as features when training decision trees in order to produce a heuristic. We will also discuss how we calculated these characteristics in our implementation.

The following characteristics were used as features when training the decision trees. The resulting trained heuristics also used these features for their node scoring. Recall that for characteristics based on original nodes, the minimum, maximum, average, median, and standard deviation is calculated for the original nodes contained by the sample supernode, unless otherwise stated:

7.2.1 Supernode size

The size of the supernode is just the number of original nodes the supernode contains. This is a single value for each supernode.

7.2.2 Original node degrees

The degrees of the original nodes contained by a supernodes.

7.2.3 Original node number of two-hop neighbors

The number of two hop neighbors of original nodes contained by a supernode.

7.2.4 Original node ratio of number of two-hop neighbors over degree

The ratio of the number of two-hop neighbors of the original node over the degree of the original node in a super node.

7.2.5 Original node minimum distance to articulation point

The distance from an original node in a supernode to an articulation point. Only the minimum distance over all the original nodes in a supernode is recorded, and a distance greater than 2 is recorded as '>2'. A node is an articulation point if its removal increases the number of connected components in the graph. A connected component of a graph is a subgraph from which there is a path between every two nodes in the subgraph.

7.2.6 Original node clustering coefficient

The clustering coefficient of an original node is a measure of the probability that two neighbors of the original node are connected.

7.2.7 Original node authority score

For an original node's authority score, the Kleinberg authority score is used from the python igraph library. The authority and hub scores come from an idea that nodes in a network (or a graph) can be recognized as either authorities or hubs of information. A node has a high authority score if it is linked by many hubs, and a node has a high hub score if it is linked to many authority nodes. Hub and authority scores are calculated and normalized iteratively until convergence.

7.2.8 Original node hub score

See 7.2.7 for information about hub scores.

7.2.9 Methodology

The original node characteristics were calculated by loading the datasets into a python igraph graph object, and we used library functions to calculate the characteristics, finally creating a dictionary of URIs to dictionaries containing the node characteristics for each original node. When calculating the values for supernodes, the values for each original node are found, and then the aggregate operations are performed.

We did not find consistent characteristics that worked as a good heuristic across datasets. However, all of our chosen characteristics, with the exception of hub score, were found to be used in some decision trees, and so also our heuristics.

7.3 Results of trained heuristic algorithm for DBLP, IMDB, and WordNet datasets

We tested or trained heuristics on the DBLP-4, IMDB, and WordNet datasets. In Figures 13 – 18, we can see the performance of the merge phase of the trained heuristic vs uniform performance with respect to runtime and percent completion. Note that percent completion is equal to one minus the size of the unvisited set divided by the initial size of the unvisited set – when the size of the unvisited set is equal to zero, the merge phase is complete. Note that the output phase does not concern this analysis, since our heuristic only affects the merge phase. Also note that the compression ratios used in this section are theoretical compression ratios, not implementation compression ratios; this is because we could not export summaries "live" as we ran the algorithm, whereas keeping track of the theoretical compression ratios "live" was very simple.

We notice in Figures 13 - 18 that the using the trained heuristics always performs better than the uniform algorithm with respect to runtime initially, but the compression ratios tend to eventually converge or intersect toward algorithm completion. This suggests that the picking seed nodes heuristically based on a small sample set can outperform picking seed nodes uniformly at random in the initial iterations of the algorithm. We also notice in Figure 17 that the heuristic algorithm in WordNet takes longer than the uniform algorithm to terminate. Additionally, we see in Figures 13 and 14 that for IMDB, the gap between the heuristic and uniform algorithms is much smaller with respect to runtime than with respect to percent completed. This suggests that the overhead of selecting nodes non-uniformly at random with our heuristic is not negligible – if it were, the runtime graphs would be identical to the percent completed graphs.

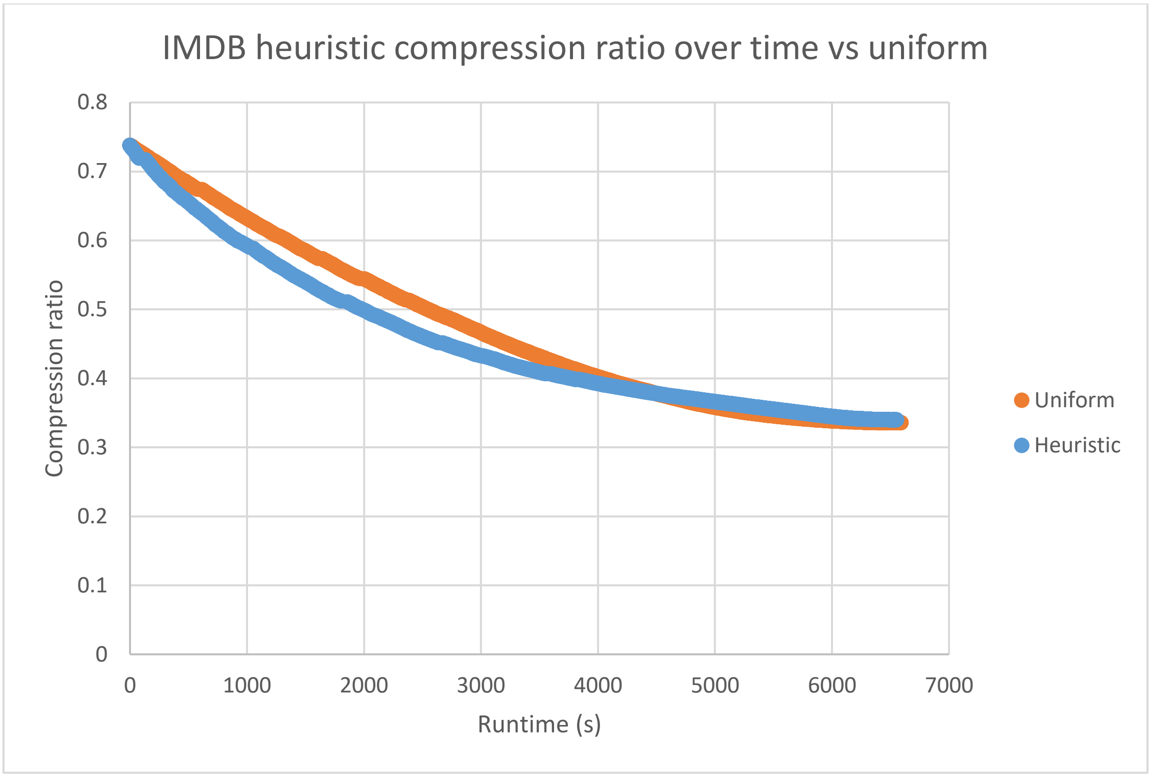
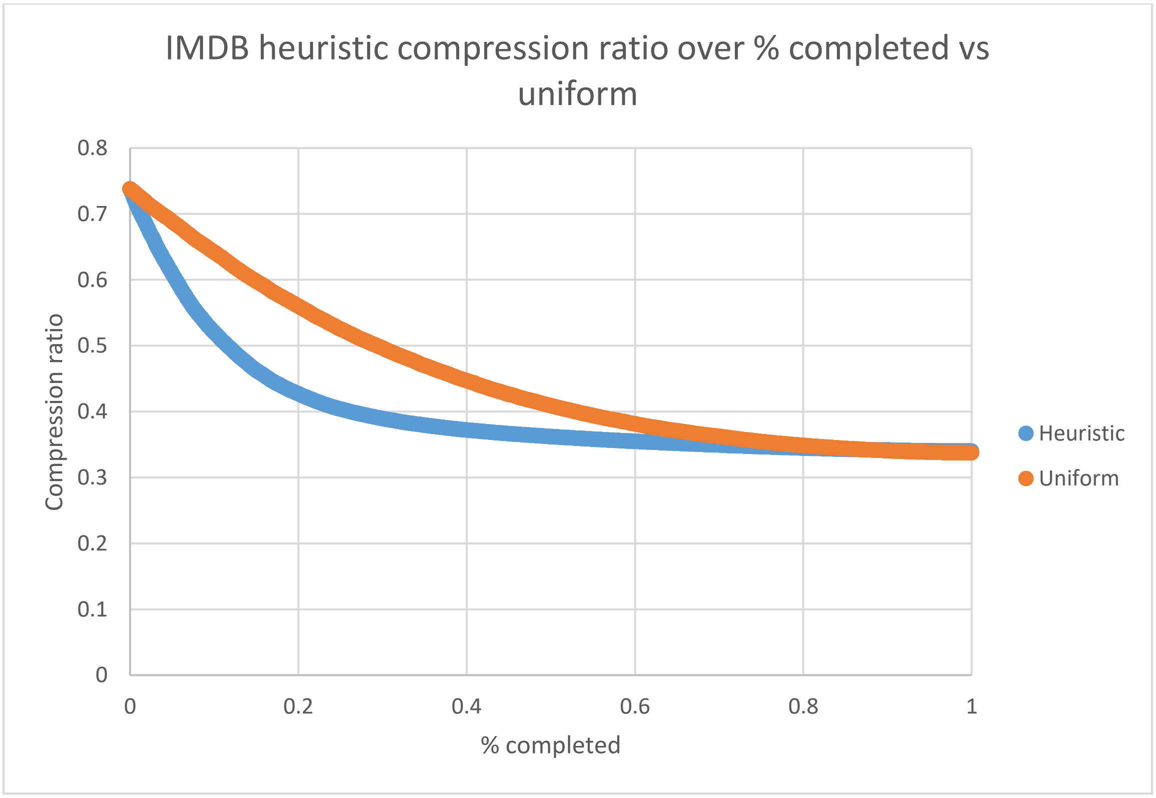


Figure 13 IMDB heuristic compression ratio over time vs uniform

Figure 14 IMDB heuristic compression ratio over % completed vs uniform



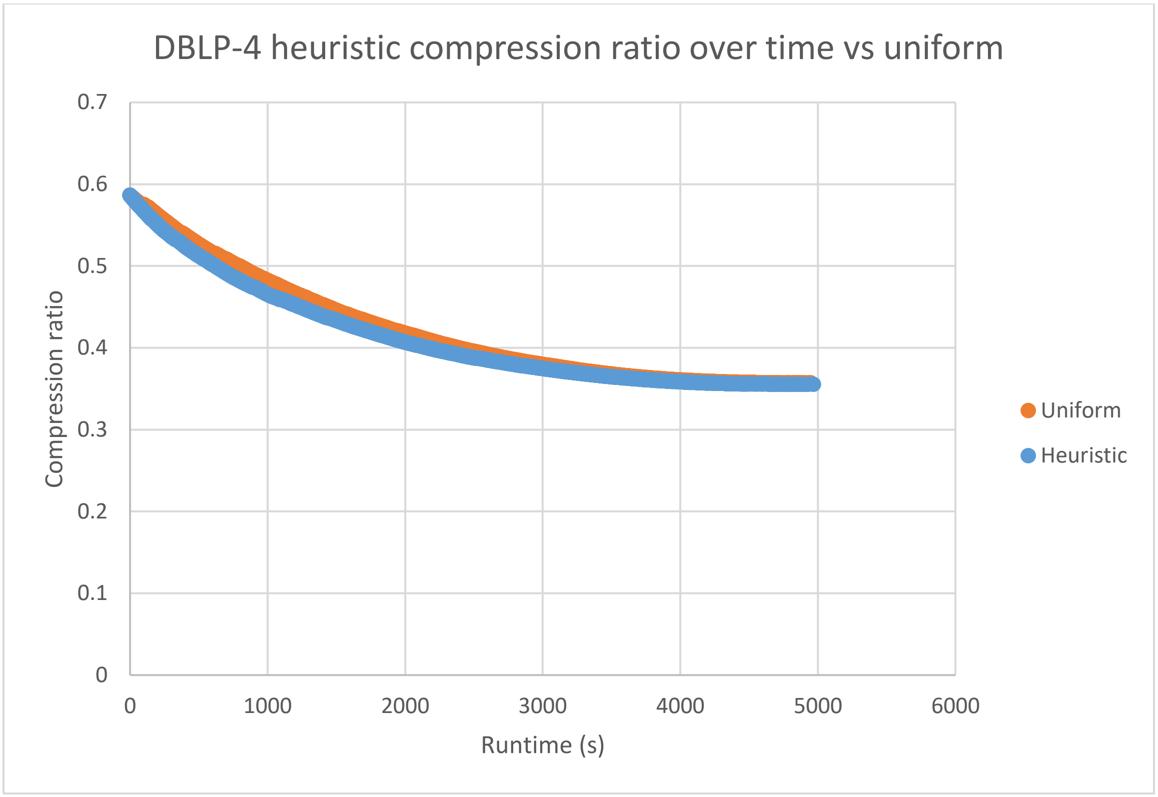
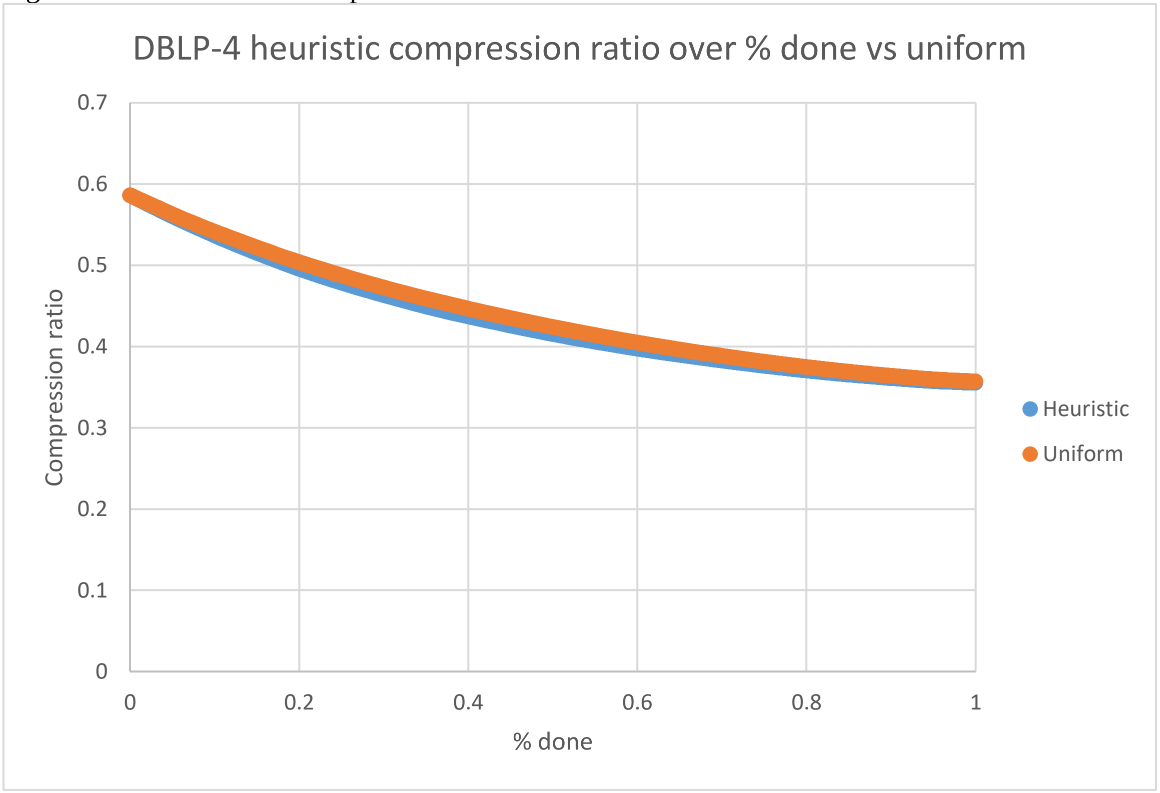


Figure 15 DBLP-4 heuristic compression ratio over time vs uniform

Figure 16 DBLP-4 heuristic compression ratio over % done vs uniform



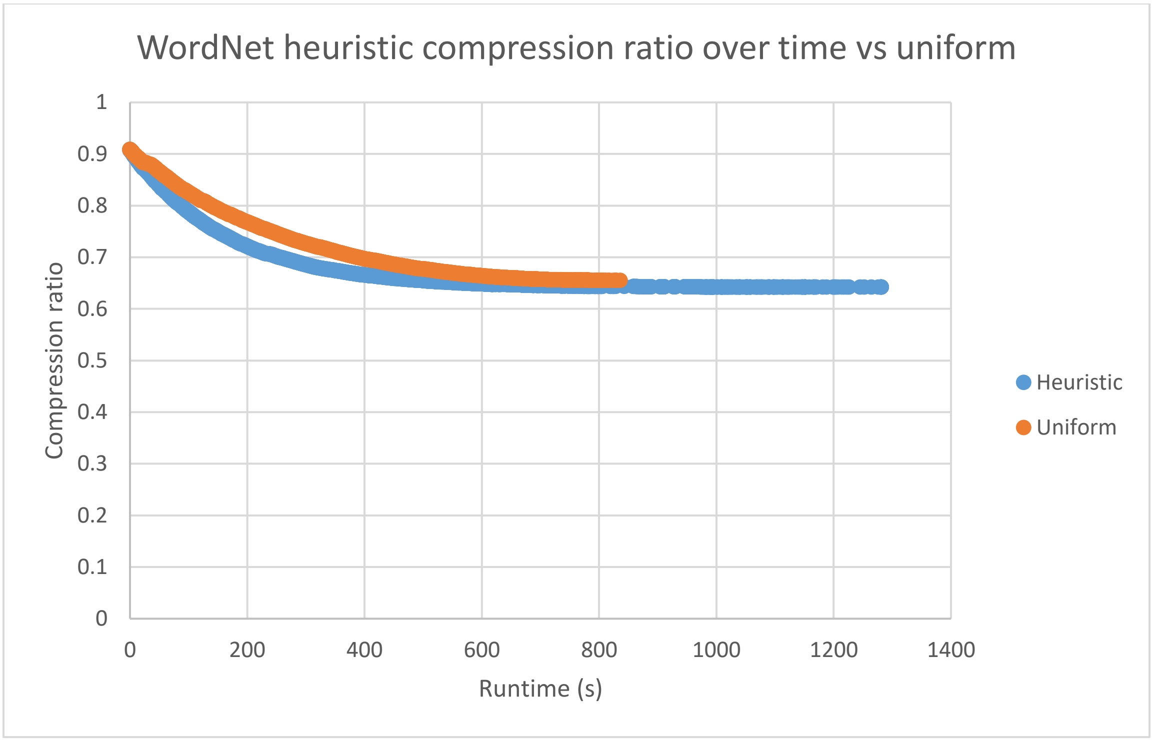


Figure 17 WordNet heuristic compression ratio over time vs uniform

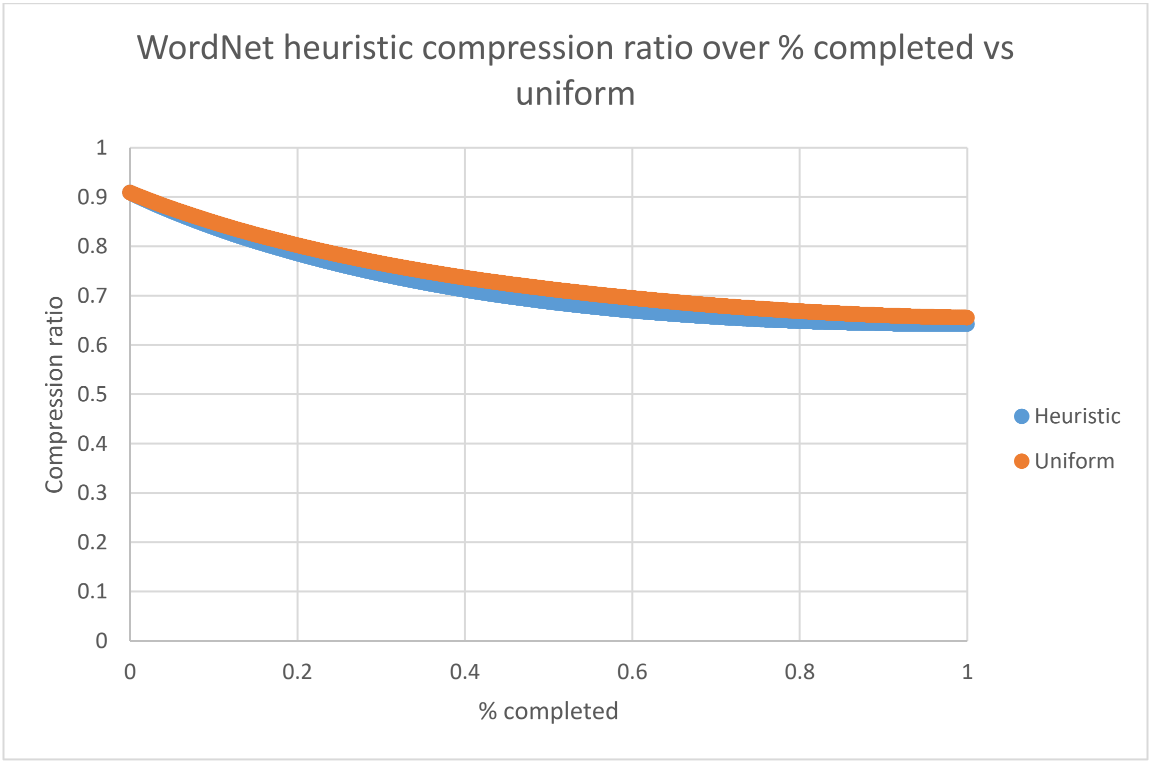


Figure 18 WordNet heuristic compression ratio over % completed vs uniform

7.4 Scalability of heuristic algorithm

We also tested applied our trained heuristic method to the other DBLP datasets for purposes of testing scalability. From figures 19-24 and 15-16, we can see that our trained heuristic process appears to scale with the increasingly larger DBLP datasets: with respect to runtime, the heuristic always outperforms the uniform algorithm initially, although, as seen with other tests, the compression ratios tend to converge or intersect as time goes on. We see in Figure 19 that the heuristic algorithm takes longer to terminate than the uniform algorithm, due to the overhead of picking seed nodes non-uniformly at random. However, we see that as the datasets increase, the time to termination for the heuristic and uniform algorithms appear to converge, suggesting that the overhead caused by picking seed nodes non-uniformly at random is eventually dominated by the overhead of dealing with large graphs.

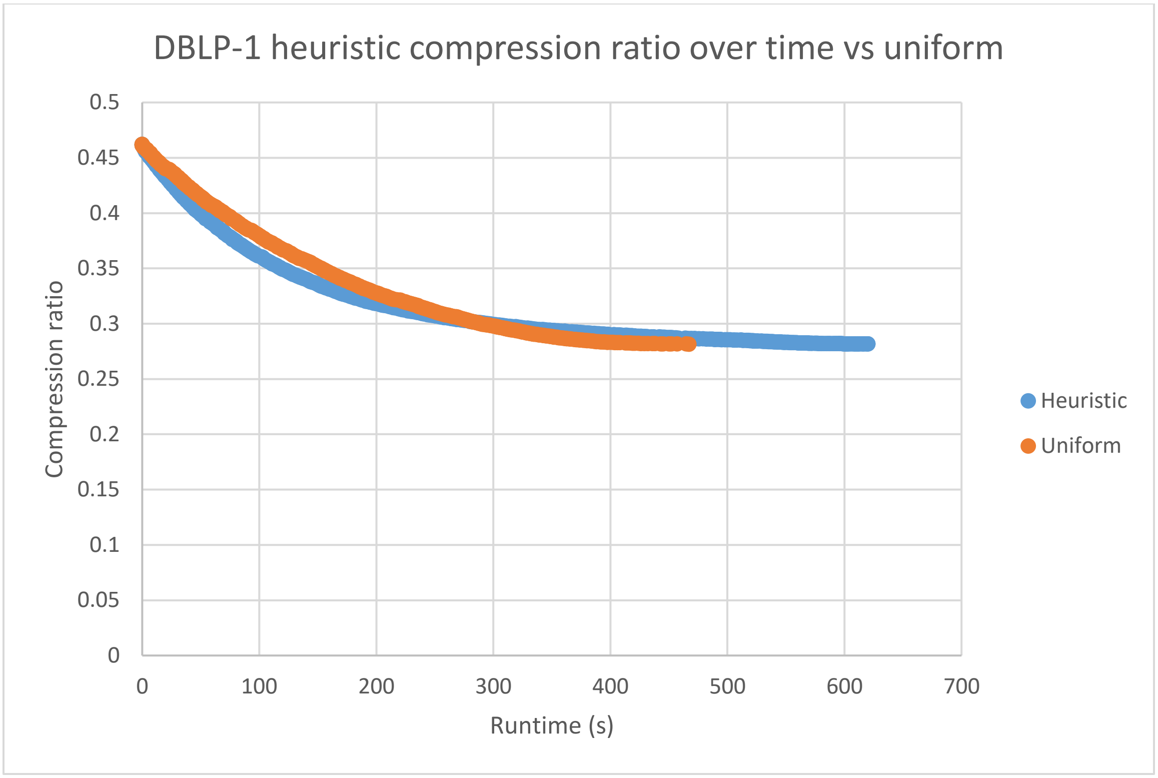
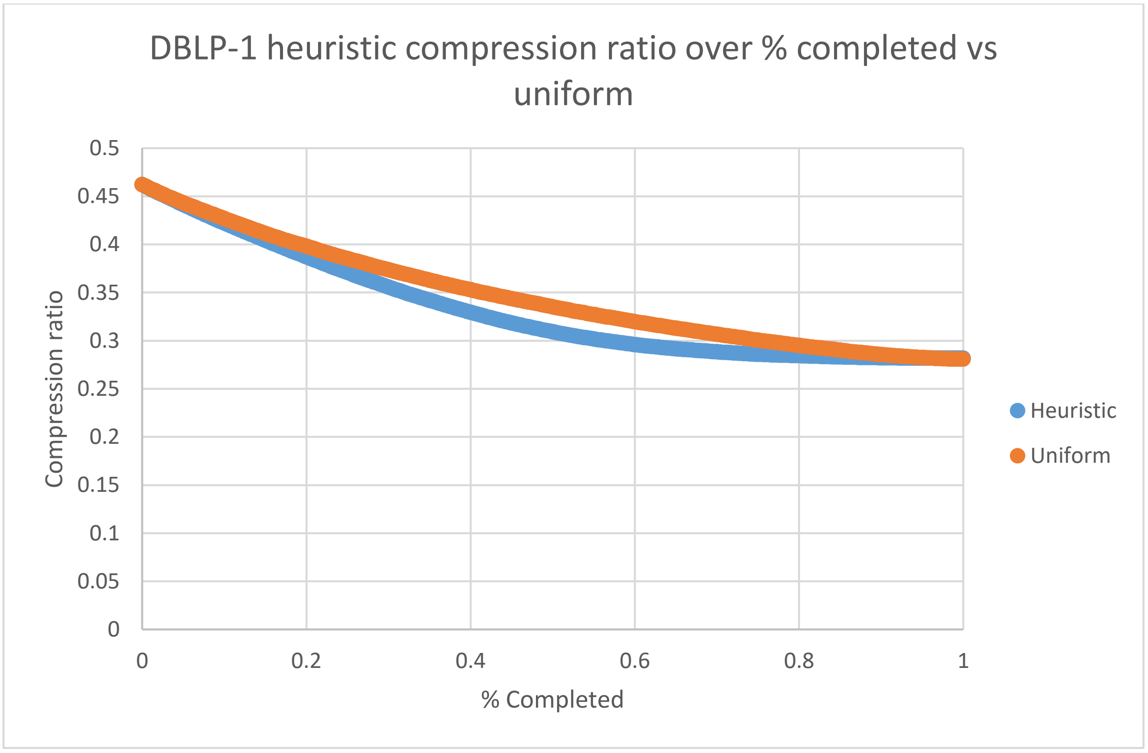


Figure 19 DBLP-1 heuristic compression ratio over time vs uniform

Figure 20 DBLP-1 heuristic compression ratio over % completed vs uniform



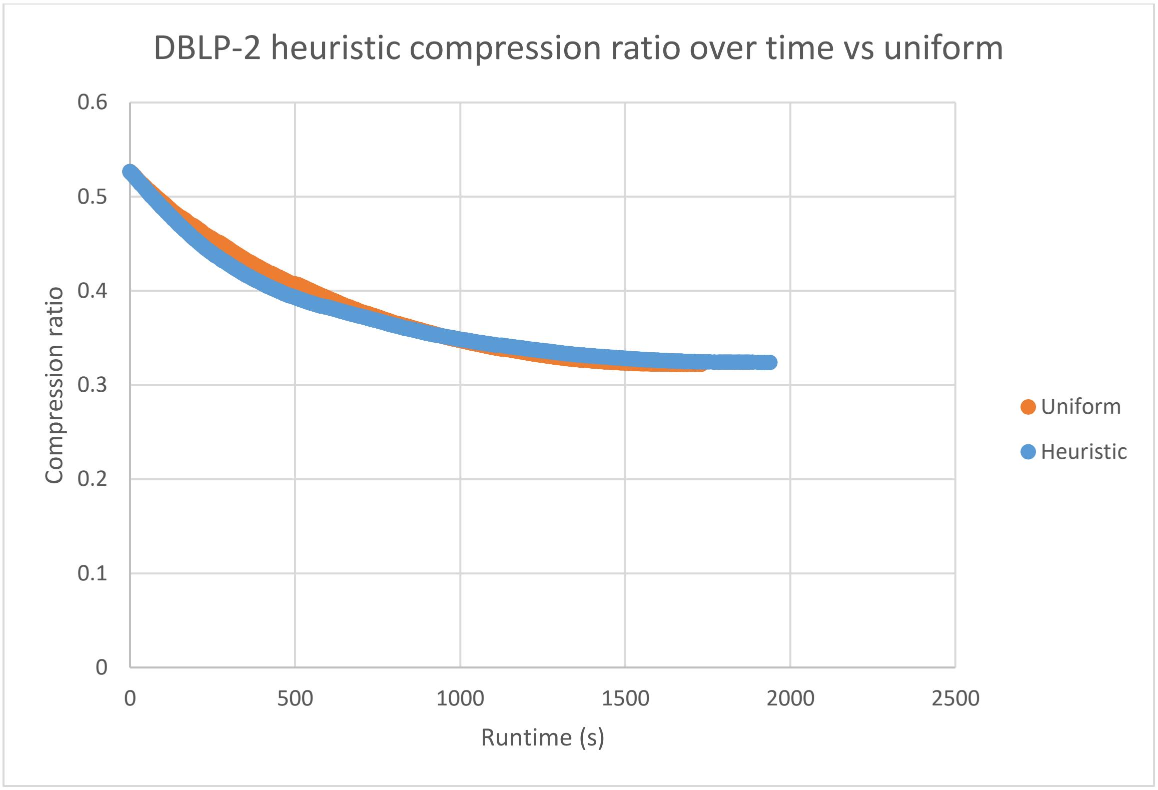
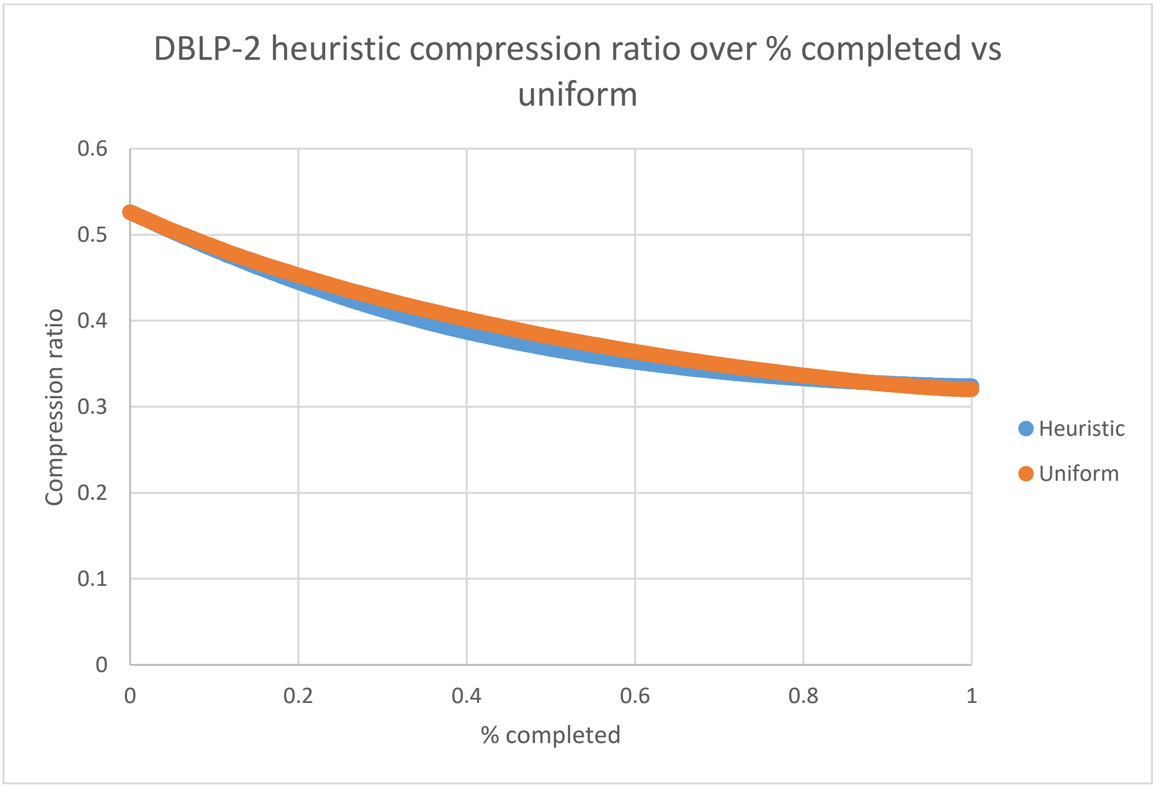


Figure 21 DBLP-2 heuristic compression ratio over time vs uniform

Figure 22 DBLP-2 heuristic compression ratio over % completed vs uniform



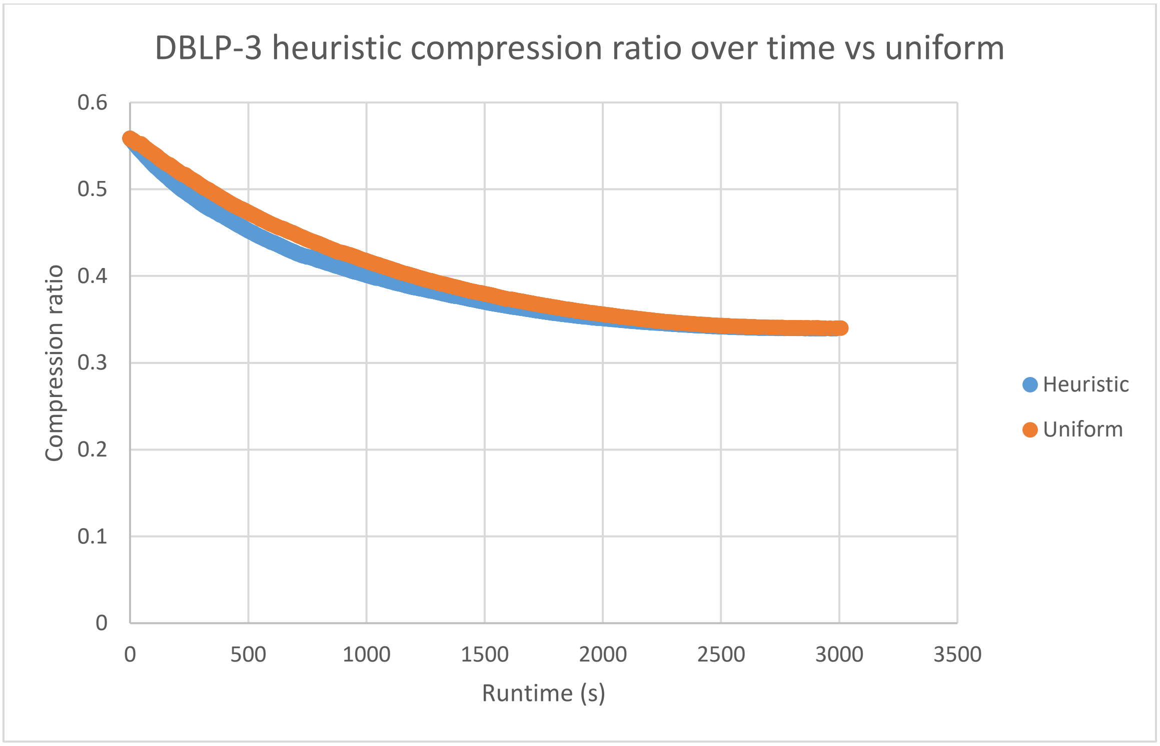
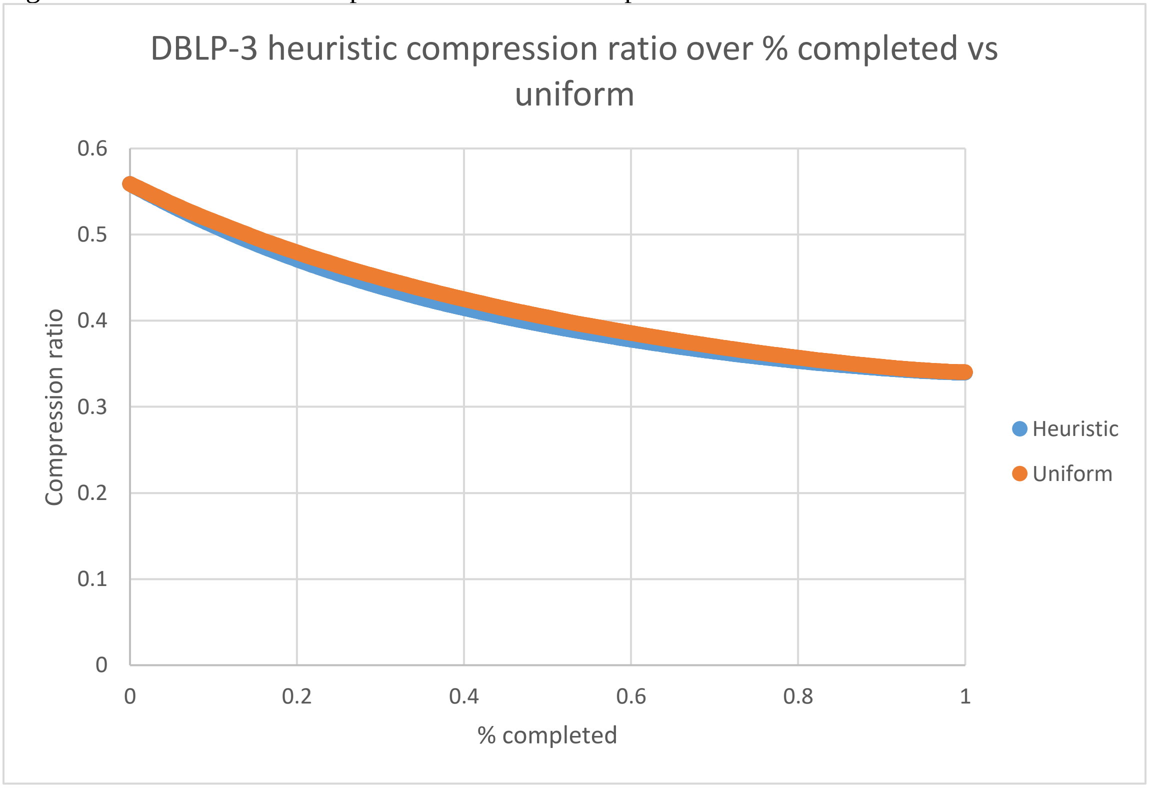


Figure 23 DBLP-3 heuristic compression ratio over time vs uniform

Figure 24 DBLP-3 heuristic compression ratio over % completed vs uniform



7.5 Conclusions

Our data suggests that using trained heuristics can significantly affect summarization initially, in that a certain compression ratio may be reached much quicker using a heuristic. These heuristics can be calculated using simple properties of nodes in the original graph. This supports our hypothesis that nodes that have at least one good merge do have easily calculable characteristics in common with other nodes that have at least one good merge. In practical application, this suggests that picking seed nodes non-uniformly at random using trained heuristics can likely result in a better compression ratio if there is limited runtime, provided that the limited runtime is a fraction of the runtime that would be needed to run the uniform algorithm to completion. This also suggests that the heuristic algorithm could produce a summary quicker if there is a specific desired theoretical compression ratio, provided that the desired theoretical compression ratio is higher that the compression ratio that could be achieved by running the uniform algorithm to completion.

We also see, however, that heuristics have little effect on the overall resulting compression ratios of summaries when the algorithms run to completion – we see this in the fact that the difference between the resulting compression ratios is usually less than one percent, and neither algorithm produces a consistently lower compression ratio across datasets. While this does support the hypothesis that order of picking nodes can affect the overall compression ratio of a summary (since there was a minor difference between the algorithms), this does not support the premise that a greedier approach will result in a better compression ratio. This does not necessarily mean that the greediest approach will not generally produce better compression ratios than the uniform algorithm – as was the finding in [1] – but certainly a greedier approach (the heuristic) does not always produce a better compression ratio than uniformly at random.

In future work, a more "online" trained heuristic approach should be tested – that is, the algorithm should do the sampling and the decision tree generation in an online manner, perhaps resampling every 5-10% of algorithm completion. Additionally, more research should be done to determine which node characteristics tend to be more decisive in finding good seed nodes.

8 Storing and querying summaries

After summarizing the graphs produced from pre-processing RDF data (described in section 6.1), we stored the summaries. From this, we can demonstrate and actual compression ratio, rather than theoretical based on number of corrections and superedges. We also ran queries on the summarized and unsummarized data to test the overhead of running a query on a summary. Here, we describe the storage scheme of the graph and the summary, and our approach to queries, and the result time of running queries.

8.1 Original RDF storage

Our RDF data was originally stored as a triple store in a Microsoft SQL Server. Each dataset was a column of three NVARCHAR(255) items, named 'Subject', 'Predicate', and 'Object'. Because we simplified the RDF data by removing one degree nodes and only used unannotated, undirected edges, we created a new dataset which reflects these changes. This new dataset was represented as a two-column table of NVARCHAR(255) items, and contained 'Subject', 'Object' pairs representing an edge in the graph. The 'Predicate' Column is omitted because we consider edges to be unannotated. We consider an entry in the table to be undirected – we do not contain a reverse pair for any pair in the table. The RDF tables were not indexed.

8.2 Summary storage

Our summary is stored as three Microsoft SQL Server tables. First, the mappings table maps each original node to a supernode. The original nodes are represented as their NVARCHAR(255) URIs, and the supernodes are represented as ints. Then, we have the superedges table, which contains two int columns, where a row in the edges table represents an undirected superedge between the two supernodes in that row. The corrections table has three columns: the positive-or-negative column is a bit which represents whether or not the correction is an addition or subtraction, and then the two NVARCHAR(255) items which are the URIs of the original nodes for which this correction pertains to. None of the summary tables were indexed.

8.3 Querying

Here, we describe our process for querying the original and summarized data.

8.3.1 Query Limitations

Because of the simplification of the RDF that we performed before summarization, we are limited in the RDF queries we can run. The only queries we performed were connection queries, which simply ask “Is node *x* connected to node *y* in *n* hops or less, and what is the path between them, if any, with *n* hops or less?” Connection queries, though only a small subset of the possible queries one might make on an RDF dataset, are still useful for finding the connections, and therefore how closely related, two concepts in an RDF dataset are.

8.3.2 Query evaluation

In order to run queries, we essentially did a breadth first search from one of the query nodes, terminating when the other query node is found or the connection limit is surpassed. In order to perform a breadth first search, we needed methods to get the neighbors of an original node from the original data or the summary.

Getting a node’s neighbors from the original (‘Subject’, ‘Object’) table was simple – we simply retrieve all the rows where the node is either a subject or object, and return the other node. This is easy to write as a SQL query.

Getting a node's neighbors from the summary is a bit more complicated. We have two sources of neighbors. First, positive corrections containing the node as a subject or object. Then, we also have the edges represented by a superedge. To retrieve these edges, we must find the supernode which maps to the node, get the mapped supernode's neighbors, then get all the nodes to which the supernode neighbors map. Finally, we must check if any of these neighbors need to be removed because of negative corrections. Though more complicated, this process is still fairly simple to write as a SQL query.

Lastly, we use the get node neighbor SQL queries while we ran the breadth first search in a python script.

8.4 Results

We tested running queries on the datasets IMDB and DBLP-4. As the primary difference between querying the original data and querying the summary is that corrections must be considered in addition to superedges, we would expect that there is a relationship between the number of corrections involved in a query and the runtime of evaluating the query on the summary. Figures 26 and 28 support this hypothesis, as the ratio of the runtime of running the query on the summary over the original data does increase as the average number of corrections involved in each hop of the path between the two query nodes increases.

We also see in figures 25 and 27 that the runtime ratio appears to begin to converge as the number of hops between the two query nodes increases. This trend is much more pronounced in IMDB, and we can see that the runtime ratio is converging at around 1.025 when the path length is 4. For DBLP we see ratios converging roughly around 0.95-1.1 as the path length increases up to 7. This is suggestive that the overhead cost of checking corrections when getting a node's neighbors is, over increasing path lengths, made up for by the lesser number of edges in the summary.

Seeing that most of these ratios are between 0.9 and 1.5, and that they appear to converge not far from 1 as path length increases, these results are suggestive that, at least in some cases, storing SSSC summaries could be a viable means of storing the RDF data.

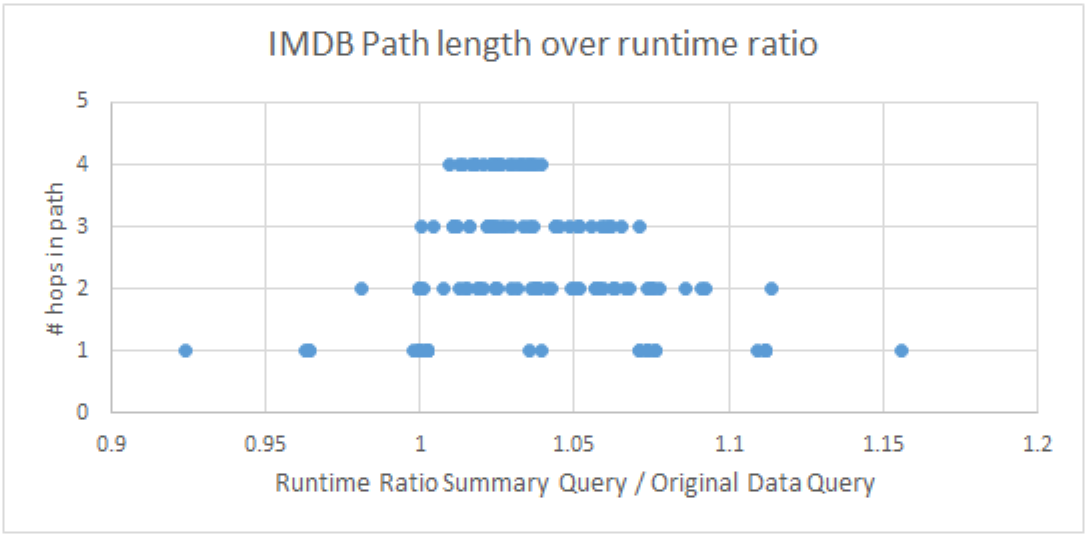


Figure 25 IMDB Path legnth over runtme ratio.

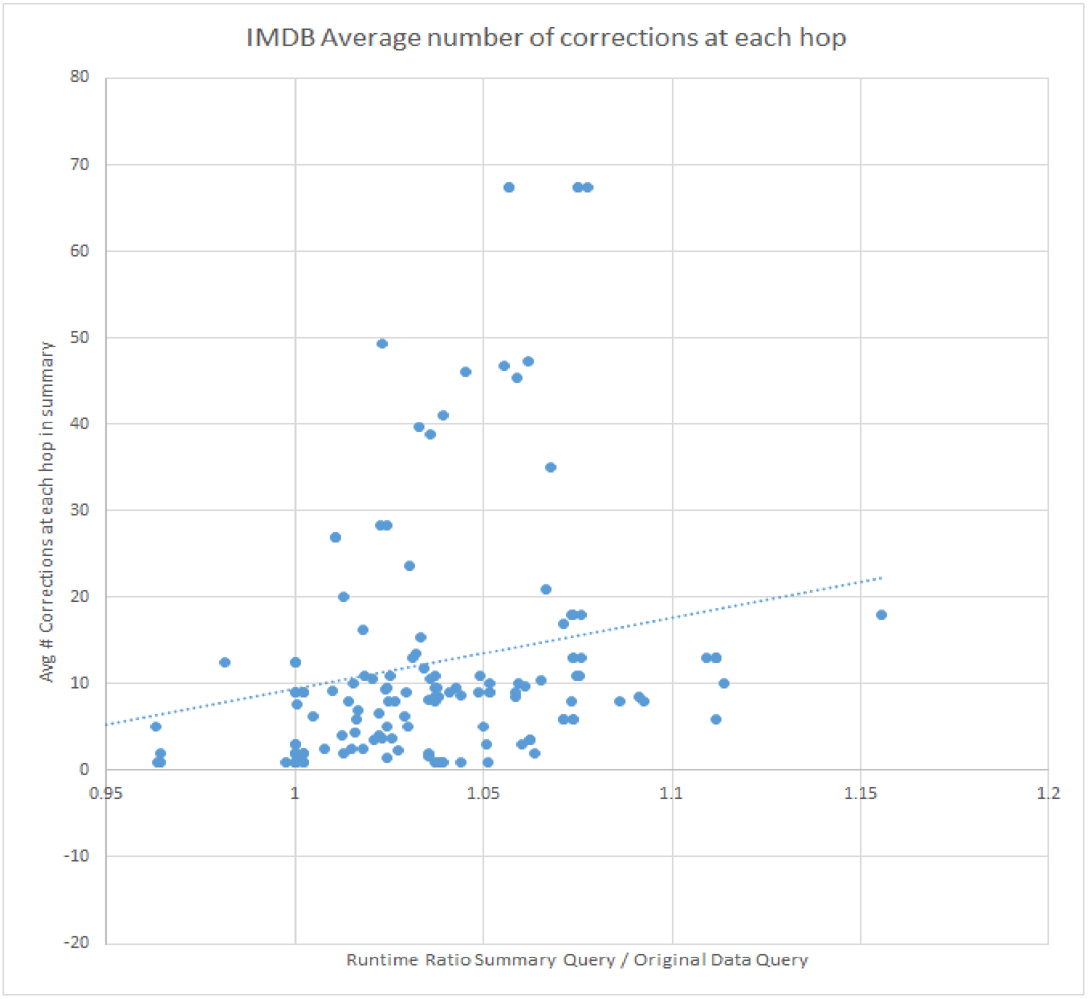


Figure 26 IMDB Average number of corrections at each hop.

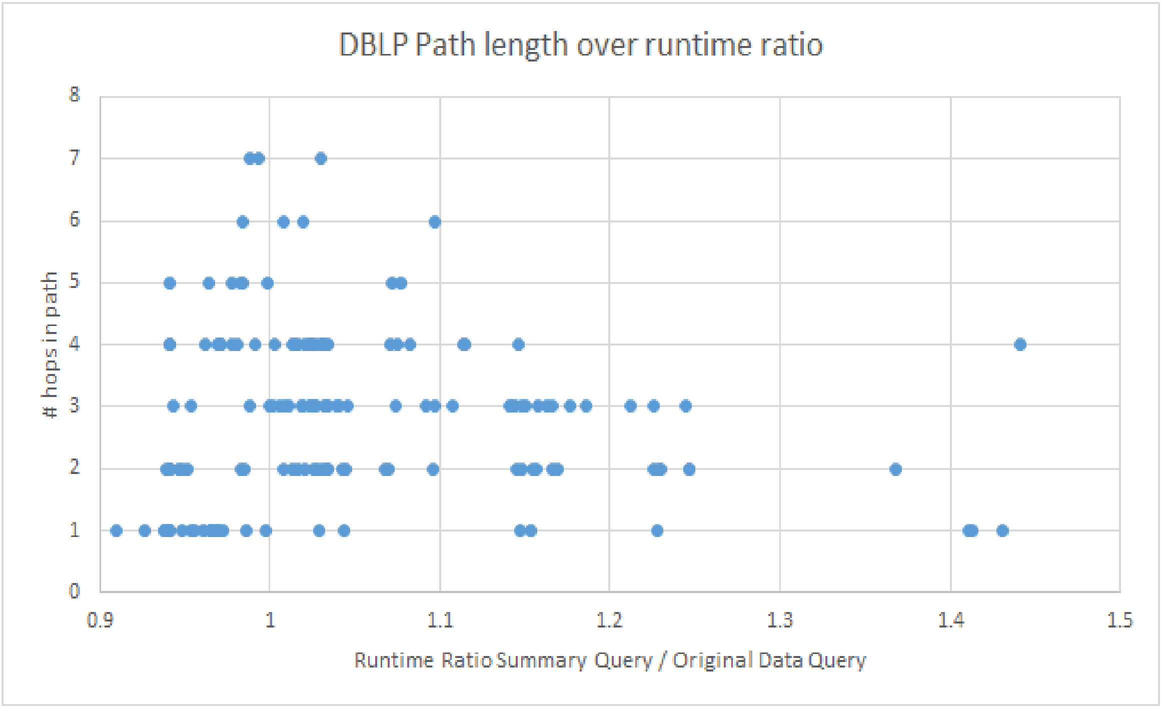


Figure 27 DBLP Path length over runtime ratio.

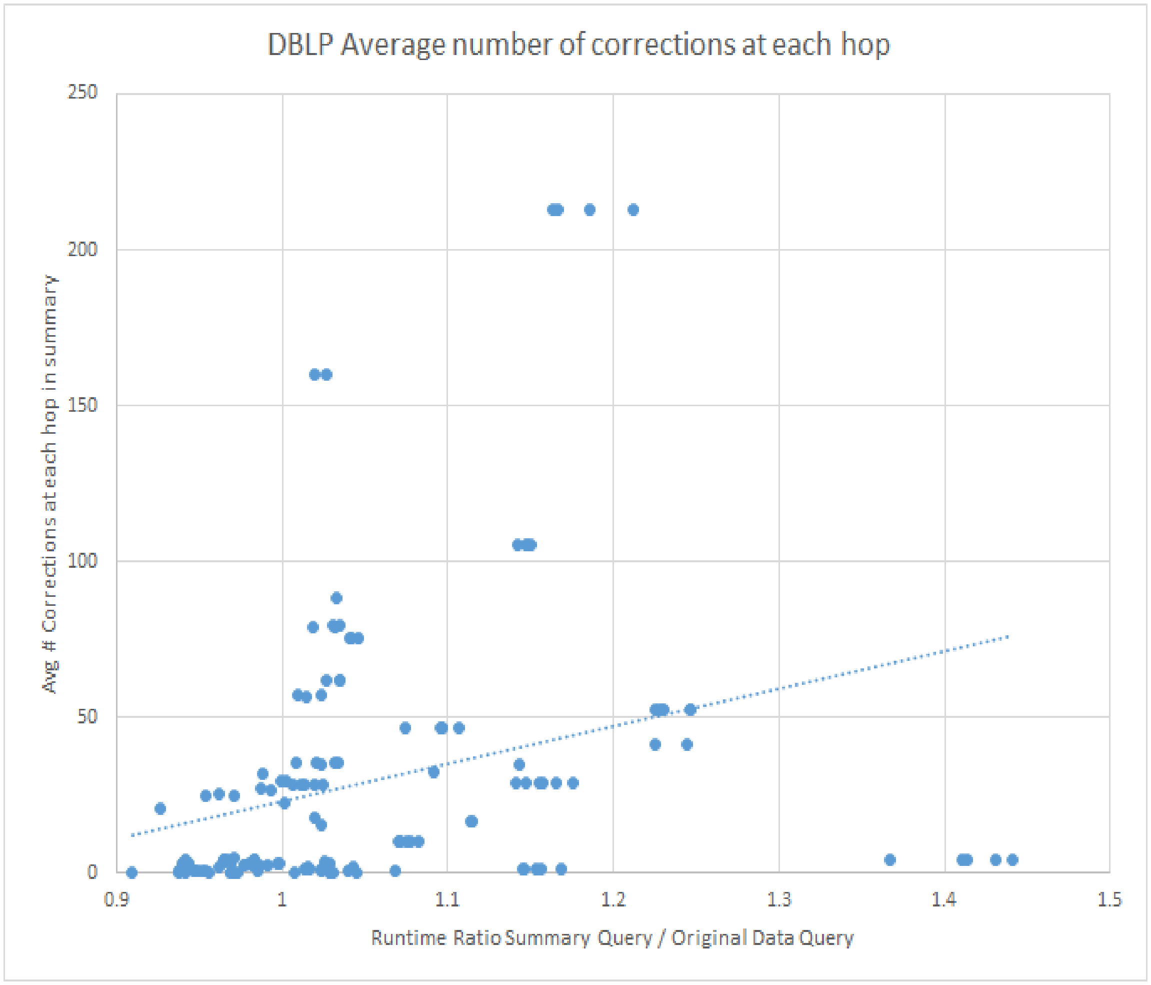


Figure 28 DBLP Average number of corrections at each hop.

8.5 Discussion of live summary storage

Our summarization occurs entirely offline, and we make no updates to the summary upon completion. However, we would like to note in this section some way our summary storage implementation could work as a live database – that is, with insertions and deletions.

First, note that any insertion or deletion of edges between pre-existing nodes can be easily added to a summary by updating the corrections table. An insert can easily be made by adding a positive correction for the edge, or removing a negative correction for the edge if a negative correction already existed. Similarly, a deletion can be made by adding a negative correction for the edge, or removing a positive correction if one existed. If an insertion involved new nodes, an entry for said new nodes must be added to the mappings table as well; new nodes could simply be mapped to a new supernode, then we can merge said new supernode with its best merge candidate as if we were running an iteration of an SSSC algorithm with the new supernode as the seed node. This method of updating is simple, but could eventually lead to poor summaries if enough corrections are inserted or deleted so that a superedge should be removed or added.

In order to re-optimize such a summary after many updates, subgraphs of the graph could be re-optimized in a localized manner (see section 9) by decompressing the subgraphs then recompressing that subgraph. Such an operation would be costly, but could be scheduled during off-time of the database.

8.6 Conclusions

We show that a real industry-useable SSSC-based summary stored in Microsoft SQL tables can still attain a compression ratio of around 0.20-0.70. We also show that such a summary can be queried, often with slowdown rates of around 1-1.3. We also describe methods for which a summary could be used in a live database. Overall, our data suggests that summaries may have the potential to be a primary method of storing RDF data. This is additionally meaningful as there is not a significant structural difference between a summary store and an RDF triple store, and significant research has been done in the area of efficient or alternate means of storing triple stores. Such methods may potentially also be applied to summaries. However, given our simplification of RDF data, our rudimentary method of storing summaries, and small datasets (by RDF benchmarking standards), we recognize that much more research is needed to support this conclusion before SSSC summaries should be considered a viable storage method of RDF data. We hope that our results inspire such research.

9 Discussion of localization of summaries

We limited our implementations to those with which we could load our datasets entirely into memory. However, we would like to discuss how loading the entire dataset into memory may not be necessary for all summarization algorithms.

9.1 Neighborhood requirements for merge

Notice how, in most algorithms, the merge candidates for a seed node is its two-hop neighbors. In order to calculate the reduced cost of merging a seed node and its two-hop neighbor, we must consider the cost of each superedge for such a merged supernode. Therefore, when considering a seed node and a merge candidate, the only information we need in order to calculate the reduced cost of merging the seed node and merge candidate is all the neighbors of the seed node and all the neighbors of the merge candidate. Now, if we want to be able to find the best merge out of a seed node and all its merge candidates, we must be able to calculate the reduced cost for the seed node with each of its merge candidates. Finally, notice that all this information is contained in the three-hop neighborhood of a seed node. In summary:

Requirements for calculating reduced cost of seed node, u, and merge candidate, v, or $s(u,v)$: we must know the neighbors of both u and v.

Requirements for finding the merge with the highest reduced cost given seed node, u, and list of all merge candidates, m: we must know all the requirements for calculating $s(u,v)$ for all v in m, which is contained by the three-hop neighborhood of u.

These requirements are further demonstrated in Figure 29. This is significant because, even in very large databases, a node’s three-hop neighborhood, or the one-hop neighbors of a seed node and iteratively each of its merge candidates, should be able to fit in main memory – the exception possibly being extremely dense graphs or graphs with very high-degree nodes. Of course, constantly loading and writing merges would have a very costly I/O overhead, but these properties are very noteworthy when we want to consider summarizing larger-than-memory databases, distributed or parallel summarization, or localized algorithms.

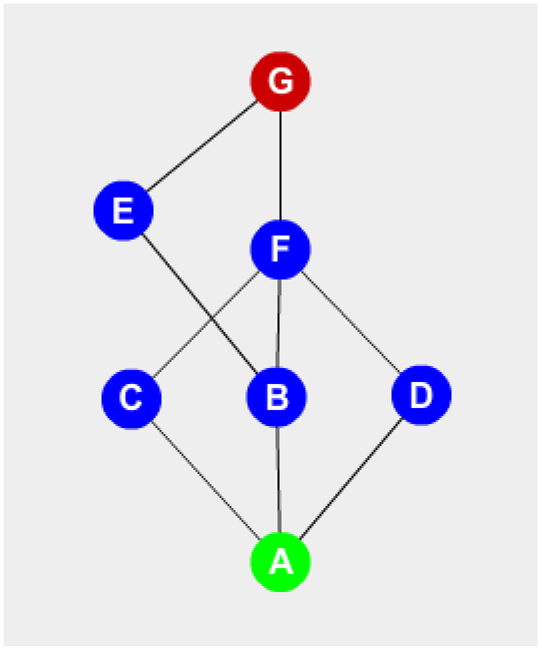


Figure 29[8] An example subgraph of a larger graph datababase. Blue and green nodes have all edges loaded into memory, whereas red nodes may not have all edges loaded into main memory, meaning that G may have neighbors that are not included in this subgraph. However, blue and green nodes have all of their edges loaded. Then, we can distinguish these nodes based on what we can do with the information contained in this subgraph. For each of the blue or green nodes, we have enough information to calculate its reduced cost with any other blue or green node, since we know all the neighbors of each blue or green node. However, the green node A can be further distinguished because we have enough information in this subgraph to calculate its best merge, because A's three-hop neighborhood is contained in this subgraph. Notice how A's merge candidates would be E and F, and from this we would find that F is the best merge for A, and, because we have the requirements fulfilled, we can be confident that merging A and F is the best merge with A as the seed node. Also notice how, even though we do not have the full three-hop neighborhoods of C and D, we can still be confident that merging C and D is a good merge simply because it is a perfect merge (C and D have exactly the same neighbors).

9.2 Distributed, parallel, and localized algorithms

Suppose we had a graph dataset with multiple non-overlapping subgraphs such as the one in Figure 29. If we loaded said subgraphs in a distributed, parallel, or main-memory-optimized manner,

[8] Figure generated from online tool available at https://illuminations.nctm.org/Activity.aspx?id=3550

notice that we could summarize each of these subgraphs separately, so long as only “green” nodes, or those for which its three-hop neighborhood is contained in the subgraph, are the only seed nodes considered. Notice that when comparing such an algorithm to Uniform Randomized, the only fundamental difference (other than obviously managing the distributed or parallel loads and writes) would be the probability distribution from which seed nodes are picked. This lends itself to the idea that distributed, parallel, or main-memory-optimized summarization algorithms are certainly within reach for graphs that have such subgraphs, such has very high diameter graphs.

10 Future work

In this section, we define what we consider to be the primary areas of future work.

10.1 Localized greedy algorithm

Recall from section 9 that SSSC algorithms easily lend themselves to localization. Also recall that, in the greedy algorithm proposed in [1], the main performance bottleneck involved heap updates. This was because all potential merges, or at least the merges which a reduced cost merge above a certain parameter, would be stored in a max-heap, so that the highest reduced cost merge would always be made. An area of future work includes proposing and testing a greedy algorithm which uses the localization properties of SSSC algorithms to reduce the size of the heap, perhaps by limiting the merges in the heap to a certain *n*-hop neighborhood.

10.2 Online trained heuristic node picking algorithm

We proposed a method of generating training data, training decision trees, creating heuristic node scores based on decision trees, then summarizing by picking seed nodes non-uniformly at random using the trained heuristics. The next step would be to evaluate an algorithm which does the data generation and training in an online manner, rather than offline before the heuristic summarization begins.

10.3 More research on SSSC storage application

We proposed a method of storing SSSC summaries in a manner similar to RDF triple stores. Future work includes further research into methods of storing SSSC summaries, potentially applying methods of RDF storage to SSSC summaries. Since we ignore attribute and type nodes in our work, future work would consider SSSC stores which also store attribute and type nodes, possibly in a manner which is entirely different than how the graph structure is stored. Additionally, future work includes evaluating SSSC summary stores as a live storage alternative, include triple updates and summary rebalancing. Finally, more work needs to be done on storing all aspects of RDF data, including annotated, directed, multi-edged properties, as well as algorithms that take into account the discrepancies between theoretical compression ratio and implementation compression ratio.

11 Conclusion

We have shown that running SSSC algorithms on RDF databases involves some points of conflict due to attribute nodes and type nodes. We can justify ignoring attribute nodes because they are not involved the core graph structure of datasets, and there are other means to storing such attributes like property tables or vertical partitions[15]. When we ignore attribute nodes, we show that SSSC algorithms can achieve compression ratios ranging roughly from 0.2-0.7 theoretically and in implementation. We also show how preprocessing datasets by merging "identical" nodes, or nodes with exactly the same set of neighbors, can easily and quickly be done.

We show that using trained heuristics to pick seed nodes non-uniformly at random does improve summary quality initially, which can be applicable to scenarios where runtime is costly. However, we also show that using said heuristics does not necessarily lead to a compression ratio which is better than the Uniform Randomized algorithm at termination.

We show that SSSC summaries can be stored in SQL tables similarly to an RDF triple store, and the overhead in connection query runtimes on summaries may not be significant.

We discuss how SSSC summaries may lend themselves to localization, distribution, or parallelization

Lastly, we describe several areas of future work for SSSC summarization.

12 References

[1] S. Navlakha et al. 2008. Graph Summarization with Bounded Error. ACM SIGMOD Conference (2008).

[2] Y. Tian et al. 2008. Efficient Aggregation for Graph Summarization. ACM SIGMOD Conference (2008).

[3] N. Zhang et al. 2010. Discovery-Driven Graph Summarization. IEEE ICDE (2010).

[4] M. Habib and C. Paul. A survey of the algorithmic aspects of modular decomposition. Elsevier Inc, Amsterdam, Netherlands, 2010.

[5] M. Tedder et al. 2008. Simple, Linear-time Modular Decomposition (Extended Abstract)∗. arXiv archive, Paper 0710.3901v2.

[6] P. Serafino. 2013. Speeding up Graph Clustering via Modular Decomposition Based Compression. ACM SAC (2013).

[7] S. Alvarez et al. 2010. A Compact Representation of Graph Databases. MLG Proceedings of the Eighth Workshop on Mining and Learning with Graphs (2010).

[8] C. Li et al. 2015. ModulGraph: modularity-based visualization of massive graphs. ACM SIGGRAPH (2015).

[9] Z. Liu et al. 2013. Frequent subgraph summarization with error control. WAIM (2013).

[10] K. Khan. 2015. Set-based Approach for Lossless Graph Summarization using Locality Sensitive Hashing. IEEE ICDEW (2015).

[11] K. Khan et al. 2015. Lossless graph summarization using dense subgraphs discovery. ACM IMCOM (2015).

[12] K. Khan et al. 2014. An Efficient Algorithm for MDL Based Graph Summarization for Dense Graphs. HIKARI Ltd.

[13] S. Qiao, "Querying graph structured RDF data," Ph.D. dissertation, Dept. Electrical Engineering and Computer Science, Case Western Reserve Univ., Cleveland, OH, 2016.

[14] C. Bizer and A Schultz. 2009. The Berlin SPARQL Benchmark.

[15] D. J. Abadi et. al. 2007. Scalable semantic web data management using vertical partitioning, Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria

[16] R Development Core Team (2008). R: A language and environment for statistical computing. R Foundation for Statistical Computing,Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

[17] B. Ripley, et. al. May 29, 2016. Package 'rpart'. Available: https://cran.r-project.org/web/packages/rpart

[18] G. J. Williams. Dec. 2, 2009. Rattle: a data mining GUI for R. The R Journal Vol. 1.