Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines

Eunjae Lee, *UNIST;* Junghyun Kim, *TmaxOS;* Keunhak Lim, *Nexon;*
Sam H. Noh, *UNIST;* Jiwon Seo, *Hanyang University*

https://www.usenix.org/conference/atc19/presentation/lee-eunjae

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines

Eunjae Lee
UNIST
kckjn97@unist.ac.kr

Junghyun Kim[*]
TmaxOS
junghyun_kim3@tmax.co.kr

Keunhak Lim[*]
NEXON Korea
limkeunhak@nexon.co.kr

Sam H. Noh
UNIST
next.unist.ac.kr

Jiwon Seo[†]
Hanyang Univ.
seojiwon@hanyang.ac.kr

Abstract

Many important graph algorithms are based on the breadth first search (BFS) approach, which builds itself on recursive vertex traversal. We classify algorithms that share this characteristic into what we call a BFS-like algorithm. In this work, we first analyze and study the I/O request patterns of BFS-like algorithms executed on disk-based graph engines. Our analysis exposes two shortcomings in executing BFS-like algorithms. First, we find that the use of the cache is ineffective. To make use of the cache more effectively, we propose an in-memory static cache, which we call BFS-Aware Static Cache or Basc, for short. Basc is static as its contents, which are edge lists of vertices that are pre-selected before algorithm execution, do not change throughout the execution of the algorithm. Second, we find that the state-of-the-art ordering method for graphs on disks is ineffective with BFS-like algorithms. Thus, based on an I/O cost model that estimates the performance based on the ordering of graphs, we develop an efficient graph ordering called *Neighborhood Ordering* or *Norder*. We provide extensive evaluations of Basc and Norder on two well-known graph engines using five real-world graphs including Twitter that has 1.9 billion edges. Our experimental results show that Basc and Norder, collectively have substantial performance impact.

1 Introduction

Algorithms such as breadth first search (BFS) [26], shortest paths (SP) [15], all pairs shortest path (APSP) [35], diameter computation (DIAM) [1], finding weakly connected components (WCC) [15], and betweenness centrality (BC) [4] are popular graph algorithms widely used in many domains including bioinformatics, social science, and economics. These algorithms share a commonality that they start from a given set of vertices and then recursively traverse their neighboring vertices. Together, we call algorithms with these characteristics BFS-like algorithms.

In BFS-like algorithms, only a subset of vertices are active at any given time. Furthermore, which of the vertices are to be activated among all the vertices is difficult to predict. Due to this reason, the locality of memory access in BFS-like algorithms is generally worse than that of other graph algorithms such as PageRank, where all vertices are active and regularly accessed [27].

Due to their poor locality of memory access, it is difficult to optimize the performance of BFS-like algorithms, particularly on disk-based graph engines that store the input graph on external storage such as SSDs. Although several optimization techniques have been suggested for disk-based graph systems, their impact on BFS-like algorithms is limited. Existing optimizations such as overlapping I/O and CPU operations [12,19] or merging small I/O requests into a single larger request [40] do not consider the characteristics of BFS-like algorithms hence, have substantial room for improvement [2, 27].

The focus of this paper is on BFS-like algorithms, and our contribution can be summarized as follows. First, we present a thorough analysis of BFS-like algorithms running on disk-based graph engines. We observe and report characteristics not previously revealed such as the fact that the number of I/O requests for each vertex is similar among the vertices, regardless of their degrees or relative positions in graphs.

Second, based on our observations, we propose a new form of a cache, which we call Basc (an acronym for BFS-Aware Static Cache). Basc has three distinct characteristics as a cache: 1) it is separate space set aside from the typical page cache[1], 2) it holds edge lists of

[*]Participated in this research as graduate students at UNIST
[†]Corresponding author and principal investigator

[1]Without loss of generality, we will use the term 'page cache' to refer to typical caches that are deployed to improve I/O performance in graph engines or within operating systems.

certain pre-selected vertices, and 3) it is static, that is, the contents of the cache do not change throughout the execution of the algorithm. We show that by judiciously making use of Basc, performance of BFS-like algorithms can be substantially improved.

Finally, our observation shows that the performance of BFS-like algorithms is highly sensitive to the layout of the graphs. Based on this observation, we devise a simple model that estimates the I/O costs of BFS-like algorithms based on the layout of the graph on disk. We experimentally validate that the model is fairly accurate in estimating performance. Moreover, guided by the cost model, we develop a simple, yet efficient graph ordering scheme that we call *Neighborhood Ordering* or *Norder* for short, which substantially improves the performance of BFS-like algorithms, even while the time to compute the ordering takes substantially less than existing ordering schemes.

The methodologies that we propose are for disk-based graph systems adopting the vertex-centric computation model. As this model is widely adopted in large-scale graph analytics, our work is applicable to many existing graph processing systems [12, 21, 28, 34, 40]. For fair comparison with previous schemes we implement our methods in FlashGraph and Graphene, two recent graph engines [21, 40]. Note that all discussions hereafter are done in the context of disk-based graph systems.

The rest of the paper is organized as follows. Section 2 describes our analysis of BFS-like algorithms running on disk-based graph systems. Section 3 introduces Basc, our BFS-aware static cache, as well as the vertex selection algorithm that we propose. In Section 4, we develop our I/O cost model based on graph orderings, then propose an optimized graph ordering that we call Neighborhood Ordering. We evaluate our proposed techniques in Section 5 and discuss related work in Section 6. Finally, we end with conclusions in Section 7.

2 Characteristics of BFS-like Algorithms

In this section, we first discuss the basic workings of disk-based graph engines and BFS-like algorithms. We focus on semi-external graph engines that store vertex attributes in memory, as main memory of commodity computers today is typically large enough to hold vertex attributes in their entirety. Then, through Sections 2.2∼2.4, we discuss the characteristics of BFS-like algorithms that we observe in our analysis.

2.1 Basics of Disk-based Graph Engines

In vertex-centric computations, the entire set or a subset of vertices are activated as they receive messages in each iteration. Then the edge lists of the activated vertices are accessed when necessary to send messages to the neighboring vertices [25]. As they are accessed, these edge lists are retrieved from disk to memory in page granularity, whose size typically ranges from 1KB to a few MBs [7, 12, 34, 40]. These pages are then stored in the page cache, which is either controlled by the graph engine or the file system [7, 12, 19, 40].

The edge lists are generally sorted and indexed by the owner vertex ID and stored sequentially on disk. Some graph engines apply optimized partitioning schemes such as hybrid-cut partitioning, instead of the more general vertex-cut or edge-cut partitioning, to reduce I/O [6, 39, 43]. Also, when I/O requests are issued to retrieve the pages, requests for adjacent pages may be merged for higher throughput [21, 40].

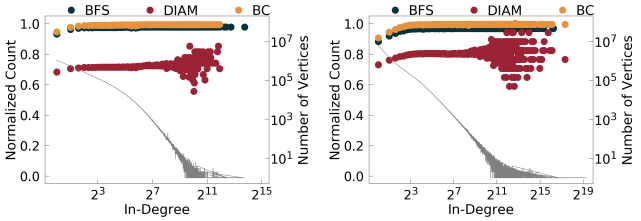
Performance of graph algorithms on disk-based graph engines depends largely on the efficiency of accessing the input graph [7, 19]. In disk-based engines, both the vertex attributes and input graphs are stored on disk that are randomly accessed by the graph algorithms. However, the overhead of accessing the input graph is generally higher than that of accessing vertex attributes as a large portion of vertex attributes are typically cached in memory. In particular, as mentioned previously, semi-external graph engines store all the vertex attributes in main memory.

Graph algorithms written in the vertex-centric model run iteratively, with a varying subset of vertices activated per iteration depending on the algorithm type. In BFS and BFS-like algorithms, only the "frontier" vertices are activated in each iteration. Thus, only the edge lists of these vertices are accessed in a random manner.

As the edge lists of activated vertices are accessed, pages containing these edges are loaded into the page cache. As page units are large, edges of unactivated vertices may also be in the retrieved page and hence, needlessly loaded to memory. To achieve high cache utilization, and consequently, high performance, we want the page cache to contain edges of as many activated vertices as possible. This requires the vertices in the input graph to be ordered such that the edges of activated vertices in the same iteration are stored in proximity.

2.2 Uniform Edge List Reference

Retrieving edge lists on disk has a significant effect on performance [19, 43]. To alleviate this burden, it would be desirable to cache the frequently requested edge lists of the vertices. To this end, we observe the edge list request pattern in representative BFS-like algorithms. Common logic tells us that for a vertex with a large edge list, that is, a large number of neighbor vertices, more requests will be targeted to that vertex and its edge list. However, interestingly and contrary to this



(a) LiveJournal (b) Wikipedia

Figure 1: Distribution of edge list reference counts: reference counts for vertices of the same in-degree are averaged and plotted relative to the vertex with the largest reference count for each algorithm. The gray line shows the number of vertices corresponding to each in-degree.

logic, we find that there are no substantial differences in the number of edge list accesses among the vertices, even for vertices that have widely varying degrees.

Figure 1, which are representative results, shows the distributions of the number of edge list requests for the BFS, DIAM, and BC algorithms with the LiveJournal and Wikipedia dataset [2]. The graphs show the average number of requests for the vertices with the same in-degrees, that is, having the same number of in-bound edges, normalized to the maximum average request count.

Initially, our conjecture was that the number of requests for a vertex (and thus, to its edge list) would be roughly proportional to its in-degree because messages (hence requests) are sent over edges in a vertex-centric model. However, Figure 1 shows that there is only a small difference in the number of edge requests between high and low in-degree vertices. The average counts are nearly constant and the variance (not shown) are low.

The reason for such uniform reference count is that in the vertex computation model, the edge list of a vertex is accessed only once per iteration as multiple requests to a vertex are merged into a single request if they are issued in the same iteration. Thus, for a high in-degree vertex whose neighbor vertices are densely connected to each other in real-world graphs [10, 20], the majority of requests to the vertex are sent as a single request. Moreover, in some BFS-like algorithms such as BFS, each vertex is processed only once in the entire running of the algorithm, with the exception of unreachable vertices. So the edge lists are also requested only once for each vertex. Thus, the number of edge list references is independent of the in-degrees and close to a constant.

The observation that there is no substantial difference in the number of references to the edge list tells us that strategies such as simply storing frequently accessed edge lists in memory is not an effective approach for improving performance. We take this observation to propose a different method for caching the edge lists of vertices, which we discuss in Section 3.

[2]The full dataset descriptions are provided in Table 1.

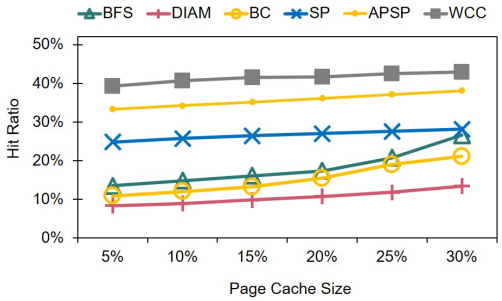


Figure 2: Page cache hit ratios for BFS-like algorithms. The page cache size is varied from 5% to 30% of the input graph size.

2.3 Ineffectiveness of the Page Cache

Locality per I/O access is known to be poor for BFS-like algorithms running on disk-based systems. This is because pages are brought into the page cache by active vertices and these active vertices are determined in a rather random manner at each iteration [23, 27]. To quantify this, we run experiments using the BFS-like algorithms provided in FlashGraph and observe the hit ratio of the page cache. Figure 2 depicts the results for page cache sizes ranging from 5% to 30% of the input graph run on the Twitter dataset. We see that all algorithms show low hit ratios. WCC, which shows the highest hit ratio, operates differently as the algorithm starts out with all the vertices, and as the component ID of each vertex converges, the number of active vertices decreases[3]. However, a small number of vertices linger around in later iterations, and those vertices fit in the page cache resulting in the higher hit ratio. More importantly, though, we find that for all algorithms, increasing the page cache has little impact on the hit ratio, improving only by 5 to 10% even with a six factor increase in page cache size.

Our conclusion here is that the page cache in disk-based graph engines does not play a major role in regards to performance for BFS-like algorithms. Simply increasing the page cache cannot be a solution, and there needs to be a different approach to resolve this ineffectiveness, for which Basc in Section 3 is proposed.

2.4 Impact of Graph Layout on Disk

Both SSDs and HDDs show faster performance with sequential reads than random reads [31]. Thus, how a graph is stored and accessed by the running algorithm has substantial impact on performance. In this section,

[3]Section 5 describes in detail how the algorithms are implemented.

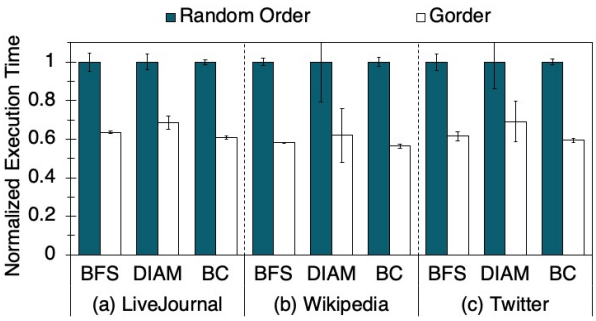


Figure 3: Performance of BFS, DIAM, and BC for Gorder normalized to random ordering for three datasets.

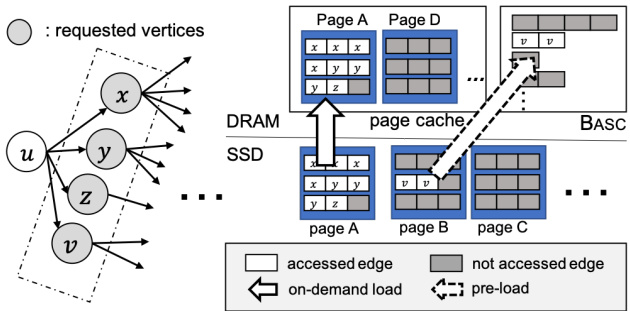


Figure 4: BASC and page cache interaction.

we perform experiments to help us understand the performance impact of graph layouts.

For the experiments, we restructure a graph in two different orderings and measure the performance of the graph algorithms. In the first, we randomly assign the vertex IDs, and in the second, we use Gorder [38], which was proposed to improve the locality of access to vertices and their edge lists for main memory graph systems. In particular, Gorder computes the ordering of vertices and their edge lists by optimizing its locality score, which is defined based on whether densely connected vertices are ordered closely within a given distance. Then, the graph is stored in CSR (Compressed Sparse Row) format, where the edge lists of vertices are ordered by their vertex IDs and stored sequentially.

Figure 3 compares the performance of three BFS-like algorithms with Gorder and random ordering on three datasets. We can see that the algorithms perform consistently better with Gorder than with random ordering. Clearly, ordering strongly affects the performance of BFS-like algorithms. In Section 4, we propose a novel ordering scheme that benefits BFS-like algorithms.

3 BFS-Aware Static Cache

In Section 2.3, we discussed how the page cache is ineffective and that simply growing its size does not help BFS-like algorithms. In this section, we propose a different caching scheme to help improve the performance of BFS-like algorithms.

Aside from the typical cache that a graph engine or the system software manages, we propose to have a separate static cache, which we call the BFS-Aware Static Cache or BASC. BASC is loaded with the edge lists of some pre-selected vertices before the algorithm starts. Hence, there is overhead involved with the initial selection process, which we describe later in this section. Also, unlike a typical page cache that dynamically stores and replaces edge lists as they are accessed, BASC is static, that is, the cache contents do not change throughout its execution and no replacement is involved.

As the edge list of only some selected vertices are statically stored in BASC, it is important to identify the vertices that are likely to improve performance so that they can be stored in BASC. Note that naively storing the edge lists of frequently accessed vertices does not suffice. This is because, as was discussed in Section 2.2 and shown with Figure 1, the number of accesses to the edge lists of each vertex is similar for all vertices. Thus, our approach is to consider the interaction between BASC and the page cache, which we elaborate below.

The key optimization point with BASC is memory utilization. While BASC is separate cache space, we do not make use of extra space, but rather take space from the page cache, that is, reduce the page cache size, and use this space for BASC. Thus, when selecting the vertices for BASC, our goal is to utilize the space for BASC much more efficiently than when used as a page cache. For example, consider a case in Figure 4, where u's neighbor vertices (i.e., their edge lists) are about to be retrieved. (Note that in real graphs, an edge list can be composed of in-bound and/or out-bound edges. Here, we show an illustration using out-bound edges.) Notice that while the edge lists of u's neighbor vertices x, y, and z are stored in page A, the edge list of vertex v is stored in page B. Thus, while a cached page A would be well used, page B, on the other, would be retrieved only to access the edge list of v with the rest of the page being brought in for naught. In such a case, v would be pre-selected and it's edge list stored in BASC, so that the entire page containing the data (page B, in this case) need not be retrieved to the page cache during execution. Selecting vertices for BASC in this manner to improve overall memory utilization is formally discussed in the next section.

3.1 The Vertex Pre-Selection Problem

We formulate the problem of pre-selecting vertices for BASC as a problem of minimizing the overall weighted I/O requests for accessing edge lists. To define the problem, we first make the following assumptions:

Assumption 1 The neighbor vertices of each vertex are accessed simultaneously. Thus, their edge lists are retrieved at the same time.

Assumption 2 The number of edge list requests for each vertex is equivalent among all the vertices.

Assumption 3 Each edge (u,v) probabilistically issues a request to access the edge list of target vertex v. Due to Assumption 2, the probability of issuing the request is inversely proportional to v's in-degree.

We now define the problem of selecting vertices for Basc as a problem of minimizing the overall weighted requests:

$$\underset{C}{\text{minimize}}\ F(C)=\sum_{v\in V}\sum_{\substack{(v,u_1)\in E\\ u_1\notin C}}\left(\frac{1}{r(u_1)d_i(u_1)}\frac{1}{\sum_{\substack{(v,u_2)\in E\\ u_2\in P(u_1)\\ u_2\notin C}} d_o(u_2)}\right)$$

$$\text{subject to}\ \sum_{v\in C} deg(v)\le M,\ \sum_{v\in C} deg(v)\ge M-\epsilon,\ \text{where}$$

- ϵ is a small positive number
- C represents the set of cached vertices
- E is the set of all edges in the given graph
- $P(u)$ is the set of vertices whose edge lists are stored in the same page as vertex u
- $r(u)$ is the expected number of requests to the page where u is stored, which we assume to be proportional to the number of vertices whose edge lists are stored in the page
- $d_i(u)$, $d_o(u)$ are the in- and out-degree of vertex u, respectively
- M represents the size of available memory for Basc.

Function F represents the penalty accrued by misuse of pages in the cache. If the cache is fully utilized, there is no penalty. F increases as the cache is more and more underutilized. Hence, our goal is to minimized F.

More concretely, F iterates over all the edges in a graph; for each vertex, we take every edge $((v,u_1)\in E)$ that is not in the cache $(u_1\notin C)$ and adds up the penalty, which is represented by the terms in parenthesis. Within the left term in the parenthesis, $1/d_i(u_1)$ represents the probability of issuing the request for a given edge. If the page is already in the page cache, we do not need to issue the request (realized by $u_1\notin C$). To consider only those pages not in the cache, we include the other term $1/r(u_1)$, as we assume that the probability of the page holding u_1 to be in the cache is proportional to the number of expected requests to the page. Thus, the left term in the parenthesis represents the probability of issuing a request over the edge (v,u_1) for u_1 not in the cache. The right term in the parenthesis represents the actual penalty incurred, which is inversely proportional to the utilization of the page. Page utilization is computed as the summation of $d_o(u_2)$ because under Assumption 1, the neighbor vertices of each vertex are accessed simultaneously. The right term, $(1/\sum d_o(u_2))$, is minimized for (v,u_1) if all the other neighbor vertices (u_2) of v is stored in the same page as u_1 and they tightly fit in a single page.

Algorithm 1: Greedy Vertex Selection (GVS) for Basc

Input: $G=(V,E)$, M: Basc size, K: iteration number
Output: A set of selected vertices C

```
1  Function SelectVertices (G=(V,E), K, M )
2      C = ∅,   m = 0
3      for k := 1 to K do
4          for v ∈ V do
5              T[v] = 0
6          for v ∈ V do
7              for u1 ∈ neighbor(v) \ C do
8                  t = 0
9                  for u2 ∈ neighbor(v) ∩ P(u1) \C do
10                     t ← t + deg_o(u2)
11                 T[u1] ← T[u1] + 1/r(u1) · 1/t · 1/deg_i(u1)
12         while T ≠ ∅ do
13             u ← n ∈ T with minimum T[n]/deg_o(n)
14             if m + deg_o(u) ≥ k/K M then
15                 break;
16             C ← C ∪ {u},   m ← m + deg_o(u)
17     return C
```

Our intent here is to minimize the overall *weighted*, that is, penalized, requests. If we were to simply minimize the number of requests, we just need to cache the vertices in descending order of their degrees. In the evaluation, we demonstrate that our approach results in better performance than simply caching the vertices in degree order.

The above optimization problem is an integer programming problem. As the objective function $F(C)$ is nonlinear and non-convex, the problem is NP-hard [13]. Thus, a fast algorithm that provides an optimal solution does not exist. We propose a heuristic algorithm to solve this problem in the next section.

3.2 Vertex Selection for Basc

We now present a heuristic algorithm for selecting vertices for Basc. Our algorithm, called Greedy Vertex Selection (GVS), takes a greedy approach based on the profits per cost for the vertex. In particular, a vertex is selected to be cached if the overhead, that is, the penalty, of the request for a vertex is high, where the penalty calculations are based on $F(C)$ described in Section 3.1.

Algorithm 1 shows the overall procedure of GVS. It takes as input G, the input graph, M, the memory size available for Basc, and K, the number of iterations, and

at each iteration selects vertices for M/K amount of memory. The outer-most Σ in $F(C)$ is covered by lines 6–11 and the next Σ by the `for` loop in line 7. The `for` loop in line 9 represents the Σ in the denominator of the second term within the parenthesis. Finally, line 11 represents the calculations for the two terms in the parenthesis for $F(C)$. Instead of summing up the penalties, GVS attributes the computed penalty to target vertices of individual edges – $T[u_1]$ in line 11. After each iteration, the penalty of each vertex is normalized by its degree (line 13). GVS selects the vertices with the highest normalized penalty whose degrees amount to $1/K$ of Basc and puts them in Basc. This is repeated for K iterations to completely fill Basc.

As K becomes larger, we select smaller number of vertices at each iteration and more frequently compute the changes in I/O penalty as the result of the selection. Thus, larger K gives more fine-grained and accurate vertex selection, but incurs more computational overhead.

The time complexity of GVS is $O(K(|E|+|V|))$ as the computation of the page utilizations in lines 10–11 can be calculated once and re-used per each vertex v, and the loop in lines 12–16 can be implemented using selection algorithms for finding the k'th smallest number.

We can further optimize the algorithm by computing the page utilization only for those vertices that are affected by the selection in the previous iteration. The vertices that require re-computation are those that are stored in the same pages as the selected vertices and are in neighbor relations. Let us now consider the complexity of GVS with this optimization. The number of selected vertices in an iteration is $O(M/K)$, if the selected ones have the same degree. The number of vertices for re-computation is proportional to the number of disk pages that the selected vertices are stored in. In one extreme, all the selected vertices may be in a same disk page, while in the other extreme, all may be in separate pages. If the layout of the graph is carefully ordered to improve locality, the selected vertices in an iteration will tend to be grouped and stored in a small number of pages. Thus, we derive the complexity assuming that the number of pages is bounded by the square root of the selected vertices, which is in between the two extremes. Then, the cost for the re-computation in lines 6 through 11 is $O(\sqrt{M/K})$. Furthermore, the sorting and selection of the M/K vertices in lines 12 through 16 can be done incrementally using a heap data structure, thus its complexity is $O(M/K \cdot log(|V|))$. Thus, the complexity of GVS is $O(|E|+|V|+\sqrt{K \cdot M}+M \cdot log(|V|))$, where the first two terms are for the first iteration and the rest are for the remaining $K-1$ iterations. In Section 5, we experimentally show that our complexity analysis for GVS is reasonable and that GVS time is roughly proportional to $\sqrt{K}$ and M.

4 Bringing New Order

In Section 2.4, we showed how ordering affected the performance of BFS-like algorithms and presented the need for effective graph layouts. In this section, we present an ordering scheme that we call Neighborhood Ordering (Norder, for short) that is tailored to BFS-like algorithms. Before so doing, we first present the I/O cost model that forms the basis for the development of Norder.

4.1 Modeling I/O Cost

In all BFS-like algorithms, the vertices that are activated in a particular iteration are the neighbor vertices of the frontiers of the previous iteration. How these activated vertices are ordered on disk substantially affects I/O performance. If these vertices could be stored together, the number of I/O requests could be reduced, leading to improved performance.

From this intuition, we empirically derive the following cost model for BFS-like algorithms, where cost is the edge list access cost, which we want to minimize:

$$Cost = \sum_{v \in V} deg(v) \cdot \sigma^2(nbr(v)) \quad (1)$$

where $deg(v)$ is the in-degree of vertex v and $\sigma^2(nbr(v))$ is the variance of v's neighbor vertex IDs, assuming CSR format. The first term, $deg(v)$, which implies that cost increases with higher in-degree, that is, with more neighbors, comes from empirical and algorithmic analysis. Consider a vertex with high in-degree. Such a vertex is likely to be accessed in an early iteration of BFS traversal. Thus, it is also likely that the majority of its neighbor vertices have not yet been traversed, which, in turn, incurs access to new edge lists, and thus, increases I/O cost. In contrast, a vertex with low in-degree is likely to be traversed in a later iteration. At this point of traversal, it is also likely that the majority of its neighbors would have already been traversed and thus, not incur any more I/O cost. Thus, we conclude that I/O cost is proportional to the degree of the vertex. The second term, $\sigma^2(nbr(v))$ is the overhead of I/O based on the neighbors' vertex ID variance. That is, neighbors whose vertex IDs show large variance are likely to be scattered across the disk, in contrast to those whose IDs are close together. Neighbors widely scattered along the disk will naturally incur more I/Os to have them retrieved.

To assess the model's accuracy, we compare the costs estimated by the model and the actual execution times for three BFS-like algorithms with the datasets YT, FL, and LJ (which we describe in detail in Section 5). For each algorithm and dataset, we generate 20 graph orderings that yield different I/O costs. This is done by, first, applying three orderings – PageRank-sorted, Gorder, and the original ordering, and then, incrementally and partially shuffling the vertex IDs of each ordering.

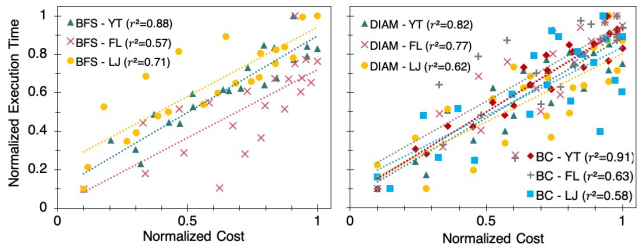


Figure 5: Regression of I/O cost model for three BFS-like algorithms with three datasets YT, FL, and LJ.

Figure 5 shows the results, where each data point represents one of the 20 graph orderings with its position determined by the cost of the model (x-axis) and the execution time (y-axis), both normalized to the maximum value on each axis. The lines are the results of applying linear regression on the data points, with the r^2 values of each dataset shown in parentheses. We see that our cost model results in reasonably high r^2 values, meaning that the estimation is quite accurate.

4.2 Neighborhood Ordering

Based on the cost model in the previous section, we propose a simple, yet effective graph ordering that we call *Neighborhood Ordering* or Norder. The key component of Norder is to simply assign consecutive IDs to the neighbors of all vertices, in particular, to the neighbors of high in-degree vertices. This simple strategy has the effect of decreasing the variance of the neighbor vertex IDs of high in-degree vertices resulting in increased locality, thus minimizing the overall cost of Equation 1.

To reorder a graph with Norder, we first arrange the vertices in descending order of their in-degrees. Then, starting from the vertex with the highest in-degree going downward, we perform a bounded breadth first search and assign a vertex ID in the traversed order. The depth bound for the traversal is set to two as we empirically found it to be effective for overall performance. Depth beyond two showed minimal performance gains, but incurred high overhead for ordering.

This simple ordering scheme is inexpensive to compute compared to other schemes such as Gorder, yet effective for disk-based graph engines. For example, it takes less than five minutes to compute Norder for the Twitter graph with 1.9 billion edges, while Gorder takes more than three hours. We quantify this and other performance issues in the next section.

5 Evaluation

We evaluate the effect of Basc and Norder with six BFS-like algorithms – breadth first search (BFS), measuring diameter of graph (DIAM), betweenness centrality (BC), shortest paths (SP), all-pair shortest paths (APSP), and weakly connected components (WCC). These algorithms represent, to the best of our knowledge, all the BFS-like algorithms in the field, except for Influence Maximization (IM). IM, however, is simply a repetitive execution of BFS, hence omitted from our evaluation [16].

Table 1: Datasets used in evaluation.

Graph	V	E	Graph	V	E
Youtube (YT)	3.2M	9.4M	LiveJournal (LJ)	4.8M	68M
Flickr (FL)	2.3M	33M	Twitter (TW)	53M	1.9B
Wikipedia (WK)	18M	172M			

BFS and SP are written as described in Pregel [25]. DIAM runs BFS multiple times, first from a random vertex and then from the vertices with the maximum distances in previous runs. For APSP, we sample 128 source vertices and compute the distances from those sources using a modified SP that computes the distances from a group of source vertices, the same as it is written in Graphene [21]. BC implements the algorithm proposed by Brandes [4]. It runs SP from each source vertex and counts the number of paths passed for each vertex. This is repeated for all the source vertices. As computation is intense, an approximate approach is taken by computing the centrality scores with 128 randomly sampled source vertices [5]. WCC is implemented in the typical manner of propagating component IDs for each vertex and then computing the minimum IDs.

The experiments are conducted with five real-world networks that are publicly available from KONECT [36] and are shown in Table 1. The datasets include one large network, Twitter (TW) and two relatively small graphs, Youtube (YT) and Flickr (FL), that are used to understand the scalability of our techniques.

We implement and evaluate our optimizations mainly on FlashGraph, a semi-external graph engine optimized for SSDs. We choose FlashGraph because 1) it is a representative semi-external graph engine, 2) it is recently developed thus, most known I/O optimizations are provided, and 3) it is actively maintained and core graph algorithms are already implemented in the system. Of the six algorithms, SP and APSP are our own implementations that we added, while the other four are those provided by FlashGraph. Additionally, we test our optimizations on Graphene [21], an SSD-based graph engine optimized with fine-grained I/O management. Details of these experiments are discussed in Section 5.4.

For all experiments, we run the algorithms with eight computation threads and a separate single I/O thread. We run the experiments ten times and report the average execution times as well as the standard deviation. All the experiments are performed on a machine with Intel Xeon E5-2683 v4 having 128GB physical memory running Ubuntu 16.04. Of the 128GB memory the system is configured to use only a portion of the memory (maximum 10GB) for caching. The actual cache size

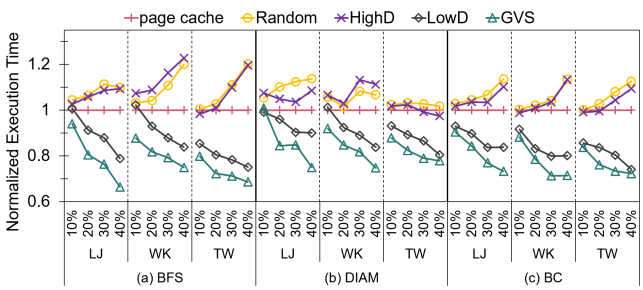


Figure 6: Execution times of graph algorithms on SSD with Basc using different vertex selection schemes normalized to the page cache only case. We vary the cache size to be 10% to 40% of the input graph size.

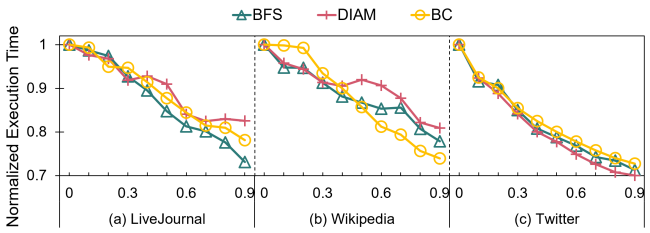


Figure 7: Execution times with varying Basc and page cache size ratios normalized to the page cache only case. The x-axis represents the Basc usage portion.

used varies for the algorithm and dataset, hence, for each experiment, we specify the cache size relative to the input graph size. Note that our goal is to show the effectiveness of our techniques on disk-based graph engines. Hence, we intentionally control memory to exercise the disks. The memory used is controlled by fixing the memory size that each part of the graph system uses and monitoring the total amount of memory used.

An Intel 400GB SSD connected via the SATA 6.0Gb/s interface is used as external storage as this is a common setting. We also run the same experiments on a typical HDD setting, but do not present the results in the interest of space and as the results do not deviate much from the results for the SSD. For each experiment, only a single disk is used. Unless otherwise stated, we set the page size to be 8KB, which is the typical access unit for NAND flash-based SSDs [8]. We use the page cache implemented in the graph system and hence, no OS kernel modification is involved. We completely clear the page cache for each experiment, and the OS page cache is turned off by setting the *O_DIRECT* flag for I/O operations so as to remove the system effect.

Note that, unless otherwise mentioned, we present results only for three datasets, LJ, WK, and TW for the BFS, DIAM, and BC algorithms only. This is done in the interest of space and as the results for the other datasets and algorithms show similar trend. However, we present statistical numbers for all algorithms and datasets and, where there are unique points of interest, we explicitly elaborate on those points separately.

5.1 Evaluation of Basc

In this section, we evaluate Basc and the Greedy Vertex Selection (GVS) algorithm in various settings. In evaluating Basc, we compare the 'page cache plus Basc' setting with the 'page cache only' setting, where both settings use the same amount of memory. To compare with GVS, we test three other selection schemes for Basc: selecting vertices with high degrees (HighD), selecting vertices with low degrees (LowD), and selecting vertices randomly (Rand), of which the last is used as the baseline. For GVS, we set the total number of iterations (K in Algorithm 1) to be 1,000 for YT, FL, LJ, WK, and 200 for TW. (Even though we do not show results for YT and FL, the settings for all experiments are presented for reproducibility.) As was previously discussed, with large K values, a smaller number of vertices are selected per iteration. This results in more frequent computation of I/O penalty changes, thus increased computation overhead. Hence, for feasibility reasons we choose a smaller K value for the large input graph TW. For the experiments in this section we do not apply any reordering algorithm and use the input graphs as they are given in KONECT [36]. We evaluate the combined effect of ordering and Basc separately in Section 5.3.

Evaluation Result. Figure 6 shows the results for experiments where the page cache size is set to 5% of the graph size and an additional 10% to 40% of the graph size is provided for Basc. As mentioned earlier, for the 'page cache only' setting (denoted 'page cache'), the page cache size is equally set, that is, 15% to 45% of the graph size is used. All results are shown normalized to that of 'page cache only'. The results presented here exclude the time to load Basc, which must be done before the algorithms are executed. We discuss this and other forms of overhead in the last part of this section.

Overall, Basc with GVS shows substantially better performance than the other schemes. We see that low-degree selection generally is a sound choice performing, in most cases, better than page cache, though, we do observe cases where it does worse. Overall, for all algorithms and datasets (including those not shown here), Basc with GVS is the clear winner, being 28.87% faster on harmonic average than page cache for all the cases.

To compare the efficiency of 'page cache' and Basc with GVS, we vary the ratio of the page cache and Basc sizes and measure their performance. Figure 7 shows the results as the total size of the two caches is set to 20% of the graph size and as the size of Basc varies from 0% to 90% of the total cache size. We see that the performance consistently improves as Basc size is increased. This is because BFS-like algorithms show

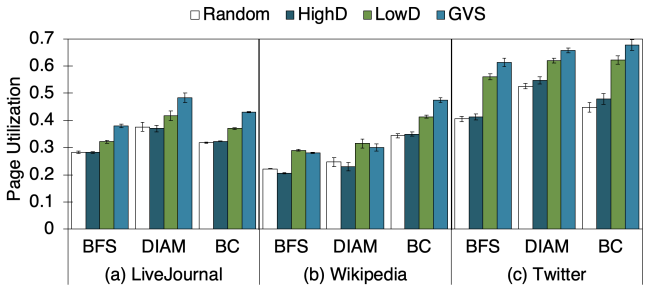


Figure 8: Mean and standard deviation of page cache utilization when executed with Basc with Random, HighD, LowD, and GVS algorithms.

poor temporal locality and using the space as Basc is more effective than using the space as a page cache.

Effect on page cache utilization. To verify that Basc with GVS improves page cache utilization, we evaluate BFS-like algorithms with Basc and measure the page cache utilizations for the different vertex selection algorithms (GVS, HighD, LowD, and Rand).

Page cache utilization is measured as follows. For the entire execution of the algorithms, for each page retrieved to the page cache, individual 64 byte granularity units of the page are monitored throughout while the page resides in the page cache. Utilization of the page, calculated when the page is evicted, is the fraction of the total accessed units within the page size. Page cache utilization that is reported is the average of all the pages evicted as well as those still residing in the page cache at the end of the algorithm execution. Note that the contents in Basc are not considered in these calculations. Essentially, this value tells us how efficiently contents brought in to the page cache are being used.

Figure 8 compares the page cache utilizations for different vertex selection algorithms when the page cache and Basc size is 5% and 20%, respectively, of the input graph size. For all the graphs across the three algorithms, GVS shows highest page cache utilization for most cases. For all algorithms and datasets experimented with (for the same page cache and Basc sizes as above), GVS shows 33.8% higher utilization than random selection in harmonic mean, and in the best case, is almost twice that of random selection (FL with BFS, not shown), with the worst case, being equivalent (YT with DIAM, also not shown). Compared to LowD, GVS shows up to 22.5% higher utilization (FL with APSP). However, there are also occasions where low-degree selection shows slightly higher utilization (WK with BFS and DIAM).

Vertex Selection and Loading Overhead. There are two sources of overhead in deploying Basc. One is the cost for running GVS to select the vertices to load and the other is the overhead to actually load the selected

Table 2: Execution times of running GVS and loading the selected vertices to Basc (unit: seconds).

	K	YT	FL	LJ	WK	TW
GVS	1	4.6	4.9	7.8	19.8	132
	10	5.9	7.6	11.3	29.2	321
	100	16.7	24.7	26.3	76.4	1581
	1000	94.8	103	146	449	5612
Loading		2.5	3.2	4.3	6.2	44.8

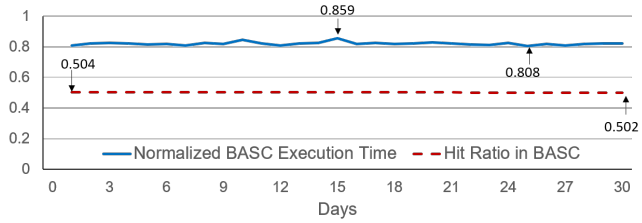
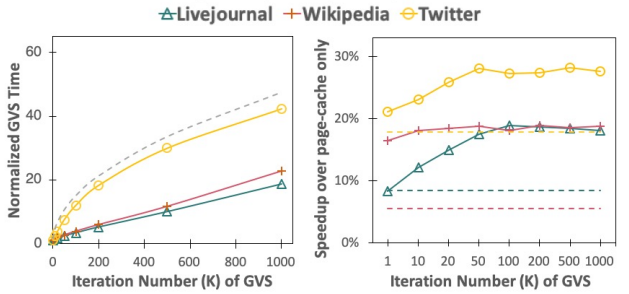


Figure 9: Efficiency of Basc over a graph (LJ) that changes 0.2% each day. Execution time of BFS with Basc is normalized to that of page cache only each day.

vertices into Basc. Both must be executed before the algorithm is executed. We quantify these overheads by measuring the running time of Algorithm 1 separately with varying K from 1 to 1000, as well as the time to load to Basc. Table 2 shows the results when Basc size is 20% of the dataset. We observe that vertex loading overhead is generally lower compared to the vertex selection overhead. For vertex selection, as we increase the value of K, the execution time increases sublinearly. The results also show that the selection time increases proportionally to the graph size, or more accurately, Basc size. (Refer to Table 1 for the characteristics of the dataset and note that we are setting Basc size (M) to be 20% of input graph size, which is proportional to the number of edges.)

Although the selection and loading times are not negligible, once loaded, the algorithm of choice is executed 100s, if not 1000s or even more times [11,42], more than compensating for the overhead for selection and loading. For example, running of DIAM a hundred times on LJ or BC just ten times on TW with Basc improves the overall running time even with the selection and loading overhead. Furthermore, this selection process can be run in the background independent of and without influencing the execution of the graph algorithms.

More importantly, these actions need to run only sparingly. Reports have shown that the social graph in Facebook changes by 0.2% a day [9], which means that in a one month period the graph would only differ by 6%. To quantify how much effect small changes in the input graph have on Basc, we conduct a series of experiments where we change the graph (LJ) by 0.2% over 30 times to simulate changes in a one month period [9].



(a) GVS execution times (b) BFS performance improvements

Figure 10: The sensitivity of GVS on iteration number K. (a) Execution time normalized by that with $K = 1$, where the dotted gray line represents $1.5 \times \sqrt{K}$. (b) Speedup over page cache only where the solid lines are for GVS and the dotted lines are for LowD.

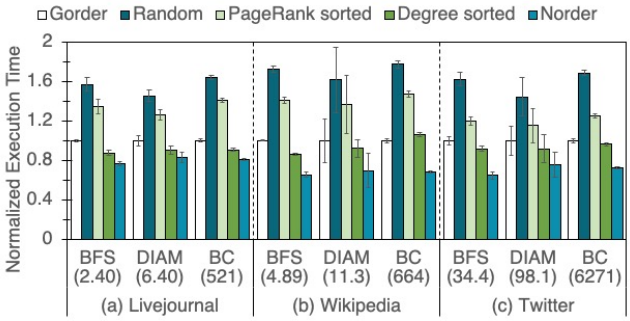


Figure 11: Mean and standard deviation of execution times of five graph orderings normalized to that of Gorder. The numbers in parenthesis below each algorithm are the absolute execution time, in seconds, of the reference, in this case, Gorder. (Note that we use similar presentation format in subsequent figures.)

We randomly add or remove edges with preferential attachment [3] where the ratio to add and remove is 8 to 2. Then, we run BFS with Basc, set to 20% of the data size, for which the selection and loading is done for the initial graph. Figure 9 shows the results of the experiments where the execution times with Basc is normalized to those of page cache only for each day. We observe that the relative performance of Basc over page cache only is nearly constant with only a 5% difference – maximum 19.2% and minimum 14.1%. In addition, the hit ratio of Basc, which is measured as the number of edge list access in Basc over the total number of edge list requests, declines slowly each day from 50.4% to 50.2%. This tells us that selection and loading can be performed over long periods, for example, once every month incurring only minimal overhead.

Sensitivity of GVS on iteration number K. As GVS execution time and its selection outcome rely on the iteration number K, we evaluate their sensitivity on K. We run GVS with varying values of K, starting from 1 up to 1000, and measure the GVS execution time. The size of Basc is set to be 20% of the input graph size. Figure 10(a) plots the results normalized to the execution times with $K = 1$. We observe that execution time increases by less than 45×, even for $K = 1000$, and that the plots for the three input graphs are all below the $1.5 \times \sqrt{K}$ line represented by the gray dotted line.

We also measure the performance of Basc with GVS as K is varied. Figure 10(b) shows the performance improvements over the page cache only settings. The solid lines are for Basc with GVS and the dotted lines are for Basc with LowD (low-degree vertex selection). We observe that Basc with GVS performs consistently better than LowD even when $K = 1$. Moreover, as K increases, the performance improvement by GVS quickly saturates, that is, the performance improvement with $K = 50$ is almost the same as that with $K = 1000$. Hence, for extremely large graphs whose GVS overhead can be potentially quite high, we can reduce the value of K to trade off the performance improvements of running the graph algorithms and GVS running time.

5.2 Evaluation of Neighborhood Ordering

We now consider the performance impact of Norder. For comparison, we use Gorder, the state-of-the-art graph ordering scheme for in-memory graph analysis. Also, we evaluate three other ordering schemes: PageRank sorted ordering, degree sorted ordering, and random ordering. For Gorder we set its parameter w (window size) to be the average number of edge lists in a single page for all the algorithms. Note that only the page cache is used and Basc is not deployed in these experiments. The size of the page cache is set to 25% of the input graph size.

Figure 11 shows the performance of the algorithms on an SSD normalized to Gorder, with the absolute execution times (in seconds) of Gorder also presented in the parentheses below each algorithm name for reference. We use this format of presentation for subsequent results as well. For all three algorithms, graph ordering has strong influence, with Norder performing the best. For all algorithms and datasets, Norder is 31.3% and 68.5% faster in harmonic mean than Gorder and PageRank sorted ordering, respectively. For all algorithms and datasets Norder showed fastest performance except for WCC with TW and FL, for which Degree sorted order was fastest. This is due to the characteristics of WCC. In the later iterations of the algorithm, a small number of vertices linger on; these vertices typically have small in-degrees, hence storing them closely on disk improves the performance of the page cache for WCC.

Cost of ordering: The cost of applying Norder is much lower than that of Gorder. Table 3 compares the computation times of the two ordering schemes. As Gorder stores the input graph in main memory to compute the ordering, all data is loaded to the 128GB memory in our

Table 3: Computation times (unit: seconds)

	YT	FL	LJ	WK	TW
Gorder	12.5	39.6	45.6	169.3	11687.1
Norder	2.0	2.7	7.2	16.9	243.5

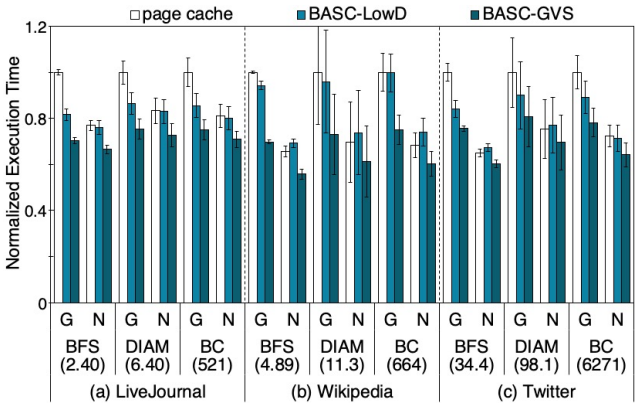


Figure 12: Mean and standard deviation of execution times of three caching schemes with Gorder and Norder normalized to that of page cache with Gorder.

system for these measurements. All orderings are computed using a single thread as this is how it is provided with the open-sourced Gorder. We observe that Norder is 6 to 14 times faster than Gorder for small graphs, while for the large graph TW, it is close to fifty times faster. If we consider the ordering time with the execution time of the algorithms, we can see that BFS-like algorithms benefit even more with Norder.

5.3 Combining Basc and Norder

Now we evaluate the overall performance gain by applying the two optimizations together. For comparison we also perform experiments with Gorder and two caching schemes, page cache and Basc with LowD. For caching, the default setting of 5% of the input graph size is used for the page cache and an additional 20% is added on as Basc or the page cache.

Figure 12 shows the performance results for all combinations of ordering and caching. The two optimizations together noticeably improve the performance of all the algorithms. Overall, for all algorithms and datasets (including those not shown here) Basc with Norder is 1.54 times faster than page cache with Gorder in harmonic mean. In the best case, Basc with Norder is 2.56 times faster for APSP with YT and in the worst case, it is 1.37 times faster for DIAM with LJ.

More importantly, however, the results demonstrate that the two optimizations can be synergistic. For example, low-degree vertex selection sometimes brings about performance degradation compared to page cache (WK with BFS and DIAM), implying that the two optimizations (LowD and Norder) do not interact well. However, GVS consistently and substantially improves performance in all cases with the two orderings, especially with Norder.

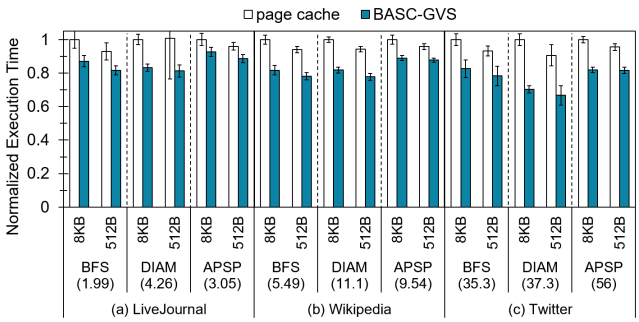


Figure 13: Mean and standard deviation of execution times of BFS-like algorithms in Graphene, with and without Basc, normalized to that of page cache only case with 8KB page size.

5.4 BASC with Graphene

The problem of low page cache utilization for disk-based graph systems was studied by Liu and Huang [21]. In their proposed graph system, Graphene, they address this problem by supporting finer-grained I/O. Specifically, Graphene stores input graphs in 512-byte pages instead of 8KB and applies bitmap-based request management to reorder and merge I/Os.

In this section, we incorporate Basc on Graphene and observe its effect. To do so, we simply modify the page cache mechanism within Graphene to accommodate Basc. Here we evaluate the effect of Basc with three algorithms – BFS, APSP, and DIAM. The first two are provided in Graphene and we implement DIAM for our experiments. We were unable to implement BC due to the complexity of Graphene's interface. Note that in Graphene, APSP is implemented to run BFS from 32 random sources and stores the result in a 4-byte attribute, of which each bit indicates if the traversal from the corresponding source vertex is reached. We compare performance with and without Basc in Graphene with the typical 8KB and the 512-byte page sizes. The page cache is set to 30% of the input graph without Basc and with Basc, the page cache is set to 10%, plus 20% space set for Basc, along with the default thread setting. Norder is not considered in these experiments as this is an optimization independent of Graphene.

Figure 13 shows the evaluation results. The results are normalized to the 8KB 'page cache only' results for every algorithm for each dataset. For most of the results the average performance of fine-grained management is better than the coarse-grained 8K page size, though for some, the variance for fine-grained management is larger. We observe that overall, Basc provides similar

improvements with Graphene showing that Basc is orthogonal to Graphene's fine-grained I/O optimizations.

6 Related Work

Disk-Based Graph Engines. GraphChi is the first disk-based graph engine [19]. Its *Parallel Sliding Windows* helps it run efficiently on HDDs. TurboGraph is a disk-based graph engine for SSDs [12]. Its *pin-and-slide* technique overlaps random I/O with CPU computation. TurboGraph and other graph systems such as GTS and GraphZ [17, 41] that use the page cache for random I/O can take advantage of Basc or Norder.

While the vertex-centric computation model is widely used, an alternative *edge-centric* computation model was recently proposed for disk-based graph systems; this model sequentially streams edges into memory to eliminate random disk access [24, 32, 43]. While the edge-centric model shows good performance for algorithms accessing the entire graph repeatedly, its performance for BFS-like algorithms is not as efficient.

In semi-external graph engines, vertex attributes are stored in main memory for fast updates [18, 30, 34, 40]. Pearce et al. proposed asynchronous optimization techniques for graph traversal algorithms for semi-external graph processing [30]. FlashGraph implements several I/O optimizations for SSDs and SSD arrays such as merging I/O requests and overlapping I/O and computation [40]. Building on top of these optimizations, our methods improve BFS-like algorithms having poor I/O locality even with those previous optimizations.

Several other I/O optimizations have recently been proposed. Vora et al. employs a dynamic partitioning scheme that prevents loading unnecessary edges [37]. GridGraph supports 2D edge partitioning [43]. In Graphene, a bitmap based I/O optimization is applied to merge small I/O requests [21]. Our proposed optimization techniques are applicable on top of these I/O optimizations as we have shown with Graphene.

Main Memory Graph Processing. For large-scale graph processing, the vertex-centric computation model was first proposed in Pregel [25], a distributed in-memory graph system. GraphLab and its successor PowerGraph is a distributed machine learning and graph analysis system with the vertex-centric model [11, 22]. SociaLite is a Datalog-based query language for distributed graph analysis [33]. Green-Marl is a domain-specific language for writing parallel graph algorithms for shared-memory [14]. Galois supports an implicitly parallel vertex iterator for graph processing [29].

Wei et al. studied graph ordering for main memory graph processing [38]. They proposed Gorder that optimizes the locality of accessing vertex attributes. While Gorder is designed for main memory systems, Norder is for disk-based graph engines. Norder is based on an I/O cost model and its optimization that we derive for BFS-like algorithms on disk-based graph engines.

7 Conclusion

In this paper, we conducted an analysis of BFS-like algorithms running on disk-based graph systems. We showed that BFS-like algorithms have poor I/O performance and the page cache in existing systems is not effective. To supplement the page cache, we proposed a BFS-Aware Static Cache or Basc that stores edge lists of a select set of vertices in memory aside from the page cache. We formulate the problem of selecting the optimal set of such vertices as a problem of maximizing overall I/O efficiency of BFS-like algorithms. As this problem is NP-hard, an approximate algorithm, called Greedy Vertex Selection (GVS), is developed. Also, based on our analysis of BFS-like algorithms, we proposed an I/O cost model upon which we develop an efficient graph ordering scheme called *Neighborhood Ordering* (abbreviated Norder) that stores neighboring vertices closely on disk.

We implemented our methodologies in two well-known graph engines and evaluated them using five real-world graphs for six BFS-like algorithms. Through a vast set of experiments, we show that the execution of BFS-like algorithms can be improved with Basc and GVS compared to simply using the page cache. We also show that Norder is less costly to compute than Gorder, yet achieves considerable performance improvements over Gorder. Our experimental results show that the two optimizations collectively and synergistically provide substantial performance gains for BFS-like algorithms.

Acknowledgement

This work is supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B1016114), by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (NRF-2016M3C4A7952635), by Basic Research Laboratory Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT Future Planning (MSIP) (No. 2017R1A4A1015498), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C2009476). The corresponding author is Jiwon Seo.

References

[1] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.

[2] Sam Ainsworth and Timothy M Jones. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing (SC 16)*, pages 1–11. ACM, 2016.

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Ulrik Brandes. A faster algorithm for Betweenness Centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[5] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys 15)*, pages 1–15, 2015.

[7] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C.S. Lui, and Cheng He. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE 15)*, pages 1131–1142, 2015.

[8] Michael Cornwell. Anatomy of a solid-state drive. *Communications of the ACM*, 55(12):59–63, 2012.

[9] Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. Walking in Facebook: A case study of unbiased sampling of OSNs. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM 10)*, pages 1–9, 2010.

[10] Debra S. Goldberg and Frederick P. Roth. Assessing experimentally derived interactions in a small world. *Proceedings of the National Academy of Sciences*, 100(8):4372–4376, 2003.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[12] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD 13)*, pages 77–85, 2013.

[13] Raymond Hemmecke, Matthias Köppe, Jon Lee, and Robert Weismantel. Nonlinear integer programming. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, pages 561–618. Springer-Verlag, 2010.

[14] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 12)*, pages 349–362, 2012.

[15] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[16] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD 03)*, pages 137–146, 2003.

[17] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 16)*, pages 447–461, 2016.

[18] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*, pages 830–841, 2016.

[19] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[20] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM 03)*, pages 556–559, 2003.

[21] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, 2017.

[22] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 2007.

[24] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the European Conference on Computer Systems (EuroSys 17)*, pages 527–543, 2017.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 10)*, pages 135–146, 2010.

[26] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Switching Theory, 1959*, pages 285–292, 1959.

[27] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. In *Proceedings of the International Workshop on Architecture for Graph Processing (AGP 17)*, 2017.

[28] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *International Journal of Multimedia & Ubiquitous Engineering*, 9(2):199–212, 2014.

[29] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 13)*, pages 456–471, 2013.

[30] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*, pages 1–11, 2010.

[31] Milo Polte, Jiri Simsa, and Garth Gibson. Comparing performance of solid state devices and mechanical disks. In *Proceedings of the Annual Workshop on Petascale Data Storage (PDSW 08)*, pages 1–7. IEEE, 2008.

[32] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 13)*, pages 472–488, 2013.

[33] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed SociaLite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

[34] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. Fog: A fast out-of-core graph processing framework. *International Journal of Parallel Programming*, pages 1–14, 2016.

[35] Alfonso Shimbel. Structural parameters of communication networks. *The bulletin of mathematical biophysics*, 15(4):501–507, 1953.

[36] The koblenz network collection. http://konect.uni-koblenz.de/.

[37] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of the USENIX Annual Technical Conference (ATC 16)*, pages 507–522, 2016.

[38] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 16)*, pages 1813–1828, 2016.

[39] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, 2016.

[40] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[41] Zhixuan Zhou and Henry Hoffmann. Graphz: Improving the performance of large-scale graph analytics on small-scale machines. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE 18)*, pages 1368–1371, 2018.

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, 2016.

[43] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC 15)*, pages 375–386, 2015.