Graph Reordering for Cache-Efficient Near Neighbor Search: Supplementary

Benjamin Coleman
ECE Department
Rice University
Houston, TX 77005
brc7@rice.edu

Santiago Segarra
ECE Department
Rice University
Houston, TX 77005
segarra@rice.edu

Alex Smola
Amazon Web Services
alex@smola.org

Anshumali Shrivastava
Department of Computer Science
Rice University
Houston, TX 77005
anshumali@rice.edu

1 Graph Indices: A Systems Perspective

In this section, we argue that graph reordering is an orthogonal improvement to most graph-based search indices. To explain why, we consider the key ingredients behind graph-based search indexing. As will become apparent, popular graph-based indices (HNSW, PANNG, and ONNG) share the same core components. Thus, although we focus on HNSW for our experiments, we expect our findings to generalize to most graph-based indices used in practice. From an implementation perspective, the core components of a graph index are *diversification / pruning*, *search initialization*, and *beam search*. Although major search algorithms differ in nontrivial ways, these differences critically do not substantially affect the node access pattern of the application. The properties of the graph that are relevant to graph reordering are the same for the vast majority of indexing algorithms.

1.1 Constructing Near Neighbor Graphs

The degree distribution and maximum out-degree are among the most important graph properties for reordering Wei et al. (2016); Balaji & Lucia (2018). In this section, we argue that although popular index construction algorithms differ in important ways, they produce graphs with similar degree distributions.

We begin by briefly reviewing the methods to construct a near neighbor graph. In a near neighbor graph index, each node corresponds to an element from the dataset. Two nodes are connected if there is a small distance between the corresponding elements. To the best of our knowledge, all existing theoretical models of near neighbor graphs adopt one of the following two construction processes Laarhoven (2017); Sebastian & Kimia (2002), where we assume the existence of a metric $d : V \to V$ between every pair of nodes.

Definition 1. A graph is an *α-near neighbor (α-NN)* graph if there is an edge between all nodes v_1 and v_2 such that $d(v_1, v_2) < \alpha$.

Definition 2. A graph is a *k-nearest neighbor (k-NN)* graph if each node v is connected to its k nearest nodes.

The two graph models differ by degree distribution, connectivity and directedness. In the k-NN graph, each node has an out-degree equal to k, while nodes in the α-NN graph may be connected to a variable number of neighbors. However, it should be noted that the maximum out-degree of an

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

α-NN graph is bounded with high probability, and the out-degree distribution is sharply concentrated about the mean for uniform embedding search Prokhorenkova & Shekhovtsov (2020). Graphs may be disconnected under either model, with nodes in sparsely populated regions of the dataset forming separate connected components. Finally, k-NN graphs are directed, while α-NN graphs are undirected because the distance function d is symmetric.

In practice, one cannot efficiently construct the k-NN or α-NN graphs because $O(N^2)$ computations are required to find the nearest neighbors of each point in the dataset. Nearly all modern algorithms solve this chicken-and-egg problem via "bootstrapping," where we iteratively add nodes to a partially constructed graph. To add a node q to the graph, we query the current version of the graph to find the (approximate) neighbors of q, whose edges are then updated to include q. The result is an approximate k-NN or α-NN graph, which we may *refine* using the same procedure, by querying and updating each node in the graph. Another process – referred to as *NN-Descent* – performs essentially the same refinement procedure, but begins with a fully random initial k-NN graph Dong et al. (2011).

However, high-performing indices do not directly use the raw k-NN or α-NN graphs. The final search index is obtained by removing unnecessary edges (or *pruning*) and by adding edges (or *diversifying*) in order to improve the navigability of the graph. Each graph index implements slightly different heuristics to accomplish this improved navigability, but they are all driven by the same principles. Below, we describe three examples.

HNSW. HNSW was described in the main text, but we deferred a detailed description of the pruning heuristic to the appendix. The pruning heuristic used by HNSW is as follows. Suppose that node B is one of the k nearest neighbors of node A. If node B is closer to one of the other neighbors of A than it is to A, then prune the link between A and B. This heuristic was originally proposed by Arya & Mount (1993) and has been used by many graph algorithms. It should be noted that HNSW applies this heuristic incrementally, *during* the graph construction. When a node is added to the graph, the connections of the node are immediately pruned. This permits the use of the efficient pruned graph to locate the neighbors of future additions to the graph, which reduces construction time.

PANNG. This index is obtained by pruning a k-NN graph to remove medium-distance edges Iwasaki (2016). The intuition behind PANNG is that some edges in a k-NN graph are redundant because there are likely alternative paths that connect the two nodes. We may safely remove these edges, since graph search will simply take the other path. The PANNG algorithm first adds edges to the k-NN graph so that the graph is fully connected and undirected. Then, the algorithm removes edges according to the following pruning heuristic. Given two nodes A and B, if B is not one of the top k_r neighbors of A and there exists a different path between A and B of length $\leq p$, then prune the edge. The values of k_r and p are hyperparameters. When $p = 2$ and $k_r = 0$, the PANNG heuristic prunes the same links as the HNSW heuristic, except under a corner case that is rare in practice[1].

ONNG. This index can be seen as an extension to PANNG that applies an additional diversification step to the k-NN graph Iwasaki & Miyazaki (2018). The intuition behind ONNG is that nodes with a small in-degree are difficult to find in the graph, while nodes with a large out-degree are expensive to visit. First, ONNG diversifies the k-NN graph by adding edges so that each node has in-degree at least k_I. Once the graph is constructed, ONNG applies the same link pruning heuristic as PANNG using $p = 2$. In other words, ONNG adds edges to ensure that all nodes are reachable, then removes the longest edge from all triangles in the graph.

Similarities Among Graph Indices. From an implementation perspective, all of these algorithms construct pruned regular graphs with a limit on the maximum node out-degree. HNSW, ONNG and PANNG all begin by adding "long distance" edges between nodes and then pruning edges when a more desirable alternative path can be found. Under common hyperparameter choices, the pruning heuristics all behave similarly. Also, most implementations use similar tricks (such as packing the nodes of the finished index into a static, contiguous block of memory). It should come as no surprise that the resulting graphs have approximately the same performance on real-world search tasks Aumüller et al. (2020). As previously mentioned, we expect these similarities to drive the generalization of our results (here illustrated for HNSW) to PANNG and ONNG, as well as to other popular graph indices that are based on the same principles.

---

[1] This occurs when the intermediary node C on the path from A to B is not a neighbor of B. This is rare when B and C are near neighbors of A because $d(B, C) \leq d(A, B) + d(A, C)$.

1.2 Searching Near Neighbor Graphs

In this section, we argue that common near neighbor graph algorithms have a similar node access pattern and will therefore benefit from reordering in similar ways. Graph-based near neighbor search consists of a walk along the edges of the graph. At each node x encountered during the walk, we record the distance $d(q, x)$ between the query q and the current node. We explore the graph until we arrive at a node which is closer to the query than any candidate or neighbor of a candidate seen so far.

Greedy Search and Beam Search. There are two search algorithms in widespread use: greedy search and beam search. In greedy search, we simply choose the nearest node to the query from the neighbors of our current node. We stop the search process once none of the outgoing edges lead to a node that is closer than the current node. Beam search is conceptually similar to greedy search, but we maintain a dynamic list of M candidates to investigate. Beam search allows us to explore outbound paths from any of the best M nodes seen so far. If none of the outgoing edges of the i^{th} candidate lead to a closer node, then we consider the neighbors of candidate $(i + 1)$. Once none of the M nodes have edges that lead to closer neighbors, we return the best points from the list of candidates as our search results. Note that when $M = 1$, beam search and greedy search are equivalent.

Graph Search Initialization. Many recent innovations in graph k-NN reduce to *initialization methods* that produce a good starting position for beam search. For example, HNSW uses greedy search over the hierarchical portion of the graph to find a seed node in the lowest layer, which is used to initialize beam search. PANNG and ONNG traverse a space-partitioning tree to extract seed nodes. Alternative options include initialization via locality-sensitive hashing, node centrality measures, and even random selection. Recent experiments suggest that the type of initialization is immaterial to performance Lin & Zhao (2019a,b). Prokhorenkova & Shekhovtsov (2020) provide theoretical support for this idea by proving that beam search over an α-NN graph solves the approximate near neighbor problem given a "sufficiently close" initialization that can be found by repeated random sampling.

Initialization is also fairly inexpensive. To illustrate this, we constructed an HNSW graph on the SIFT1M dataset with $k = 8$ edges per node. We used beam search with $M = 500$ and found that the hierarchical part of the search process was responsible for only 2.9% $(\pm 1.9\%)$ of the query time. We then replaced the hierarchy with a process where we randomly select 50 nodes and use the closest option as the initialization, to reproduce the experiments by Lin & Zhao (2019a). We found no statistically significant difference in terms of recall or query time over 10k queries. The practical takeaway is that under typical query conditions, initialization is an important but computationally insignificant part of the search process. The most time consuming part of graph search is the final stage, where we walk along the edges to locate the neighbors of a query. It is this stage that is targeted by reordering algorithms.

1.3 Why should our results generalize?

Most graph indices rely on edge walks to locate neighbors. No matter how the search is initialized or how the original k-NN graph may be modified, the most expensive part of the query is graph exploration via beam search or greedy search. This process is inherently cache-unfriendly, even when all of the nodes are placed in a contiguous block of memory Wei et al. (2016). Graph reordering is a general technique to shuffle the order of nodes so that connected nodes are nearby, and thus that graph walks are more cache efficient. Unlike traditional applications of graph reordering, nodes in near neighbor graphs are often larger than the cache line size. However, this simply means that performance gains are driven by prefetching rather than by co-loading nodes from the same cache line.

2 Graph Reordering Techniques

In this section, we provide a detailed review of existing graph reordering methods. Formally, graph reordering can be seen as constructing a labeling or indexing function $P : V \rightarrow \{1, \ldots, N\}$ that assigns each node $v \in V$ in a graph to a unique integer index (or label) between 1 and N following a pre-specified rule or in order to maximize some objective. Many formulations are possible, but generally we require P to map connected nodes to similar (nearby) labels. The function P is then used as the memory layout for the graph, with node v assigned to memory location $P(v)$.

2.1 Techniques

There are three broad categories of reordering algorithms. The first category contains algorithms that optimize P to maximize an objective function. The optimization problem is typically NP-hard, so we are usually forced to accept an approximate solution for P. The second category assigns labels to nodes based on their degree or other local graph features. Finally, graph partitioning algorithms can be repurposed to work as reordering algorithms by assigning a contiguous memory range to each partition. Below, we review several recent algorithms from each category. These are considered in the experiments to assess their ability to accelerate near neighbor search.

Gorder. Gorder (graph-order) is a graph reordering algorithm that seeks to maximize the overlap between the neighborhoods of nodes with consecutive labels Wei et al. (2016). Indeed, Gorder finds P by maximizing the number of shared edges among size-w blocks of consecutive nodes. This is a good proxy for cache efficiency because a block that contains many overlapping nodes is likely to avoid a cache miss, as each node's neighbors are stored no further than w memory locations away. Formally, the Gorder labeling function P_{GO} can be found through the following maximization

$$P_{\mathrm{GO}} = \arg\max_{P} \sum_{\substack{u,v \in V \text{s.t.} \\ |P(u)-P(v)|<w}} S_s(u,v) + S_n(u,v), \tag{1}$$

where $S_s(u,v)$ indicates whether u and v are directly connected and $S_n(u,v)$ counts how many common in-neighbors they have. In other words, Gorder maximizes the *average neighborhood overlap* between nodes that are within w positions of each other under the labeling function. Intuitively, two nodes will be placed together if they share a direct edge or, even if that is not the case, if they share many common neighbors. Maximizing the objective in (1) is NP-hard, but the Gorder algorithm in Wei et al. (2016) provides a $1/(2w)$-approximate solution. Algorithm 1 shows pseudocode for the Gorder algorithm. We also show psueodcode for the Porder algorithm, which is very similar.

Reverse Cuthill Mckee. The *bandwidth* of a matrix is defined as the maximum distance of a nonzero element from the main diagonal. The Reverse Cuthill Mckee (RCM) algorithm Cuthill & McKee (1969); George (1971) is a reordering method originally introduced to minimize the bandwidth of a sparse symmetric matrix. If we apply this method to an adjacency matrix, then the corresponding optimization problem has an immediate interpretation in terms of graph reordering Auroux et al. (2015)

$$P_{\mathrm{RCM}} = \arg\min_{P} \max_{(u,v)\in E} |P(u) - P(v)|, \tag{2}$$

where E is the edge set of the graph of interest. From (2) it follows that the RCM objective is to minimize the *maximum label difference* between connected nodes. Similar to Gorder, the problem is difficult: the exact solution is NP-hard. The RCM algorithm is a heuristic method based on breadth-first search to find a function P with low (but not necessarily optimal) bandwidth. Algorithm 3 presents pseudocode for RCM.

MLOGA and MLINA. The objective in (2) motivated extensions that focus on an aggregated measure (as opposed to the maximum) of the discrepancies between labels of connected nodes. Both the minimum logarithmic arrangement (MLOGA) and the minimum linear arrangement (MLINA), which originally arose in the context of social network compression, are examples of these approaches Chierichetti et al. (2009). More precisely, MLINA seeks to minimize the sum of label discrepancies

$$P_{\mathrm{MLN}} = \arg\min_{P} \sum_{(u,v)\in E} |P(u) - P(v)|, \tag{3}$$

whereas MLOGA first applies a logarithmic transformation to these discrepancies

$$P_{\mathrm{MLG}} = \arg\min_{P} \sum_{(u,v)\in E} \log(|P(u) - P(v)|), \tag{4}$$

where, again, E is the edge set of the graph of interest. As expected, both problems are NP-hard. However, specialized heuristics have been developed to approximate the solution of both MLOGA Chierichetti et al. (2009); Safro & Temkin (2011) and MLINA Wei et al. (2016).

Degree Sorting. Degree sorting is a lightweight reordering algorithm based on the idea that high-degree nodes are likely to share many edges. Indeed, many practical graphs obey a power-law degree

Algorithm 1 Gorder Algorithm

```
Input: A graph $G = (V, E)$ with vertices in an original order $V = \{v_1, ...v_N\}$ and window size $w$.
Output: A permutation vector $P$ where $P[i]$ contains the new location for vertex $i$.
$Q$ = empty priority queue.
$P$ = array of $N$ zeros.
for $i = 1$ to $N$ do
    Insert $v_i$ into $Q$ with priority 0.
Increment $Q[V[0]]$ by 1.
$P[0] = Q$.pop()
for $i = 1$ to $N$ do
    $v_e = P[i - 1]$ new node in window.
    for Node $u$ in out-edges of $v_e$ do
        if $u \in Q$ (i.e. node is not yet assigned) then
            Increment $Q[u]$ by 1.
    for Node $u$ in in-edges of $v_e$ do
        if $u \in Q$ then
            Increment $Q[u]$ by 1.
            for Node $v$ in out-edges of $u$ do
                if $u \in Q$ then
                    Increment $Q[v]$ by 1.
    if $i > w + 1$ (Remove tailing node) then
        $v_b = P[i - w - 1]$
        for Node $u$ in out-edges of $v_b$ do
            if $u \in Q$ (i.e. node is not yet assigned) then
                Decrement $Q[u]$ by 1.
        for Node $u$ in in-edges of $v_b$ do
            if $u \in Q$ then
                Decrement $Q[u]$ by 1.
                for Node $v$ in out-edges of $u$ do
                    if $u \in Q$ then
                        Decrement $Q[v]$ by 1.
    $P[i] = Q$.pop()
    Increment $i = i + 1$
```

Algorithm 2 Porder Algorithm

```
Input: A graph $G = (V, E)$ with vertices in an original order $V = \{v_1, ...v_N\}$, a set of queries $Q$, and window size $w$.
Output: A permutation vector $P$ where $P[i]$ contains the new location for vertex $i$.
Construct a weighted graph $G_w = (V, E, W)$, identical to $G$ with all weights = 1.
for Query $q$ in $Q$ do
    Run query $q$ through the $G$.
    For every edge traverse by $q$, increment weight by 1.
Run Algorithm 1 on $G_w$ to get $P$ with the following modification: Instead of incrementing/decrementing by 1, increment/decrement by the edge weight.
```

distribution, where a small number of nodes form a densely connected sub-graph. For undirected graphs with a power law degree distribution, degree sorting will create a group of neighboring high-degree nodes in the first contiguous memory block. To obtain P, we simply sort the nodes in descending degree order and let $P(v)$ be the sorted rank of node v. Since degree sorting only requires local node information, it is orders of magnitude faster than the optimization-based methods in (1)-(4).

Hub Sorting. Hub sorting Zhang et al. (2017) is similar to degree sorting, but with the caveat that we only sort hubs (nodes with many connections). The algorithm first splits the nodes into two groups (hubs and non-hubs) based on a degree threshold. The threshold is set as the average degree, so that hubs are defined as nodes with greater-than-average degree. To find P, the hubs are sorted by degree

Algorithm 3 RCM Algorithm

```
Input: A graph $G = (V, E)$ with vertices in an original order $V = \{v_1, ...v_N\}$ and window size $w$.
Output: A permutation vector $P$ where $P[i]$ contains the new location for vertex $i$.
$V' = V$ sorted in ascending order of degree.
$P =$ empty array.
for $i = 1$ to $N$ do
    $v = V'[i]$
    if $v \notin P$ then
        Append $v$ to $P$.
    Push the out-neighbors of $v$ onto $Q$ in order of ascending degree.
    while $Q$ is non-empty. do
        $u = Q$.pop()
        if $u \notin P$ then
            Append $u$ to $P$.
            Push the out-neighbors of $u$ onto $Q$ in order of ascending degree.
$P' =$ reverse($P$)
for $i = 1$ to $N$ do
    $P[P'[i]] = i$
```

and the non-hubs keep their original ordering. Experiments have show that degree and hub sorting are not always beneficial: they may even slow down graph processing if the original ordering of the graph had good locality properties Balaji & Lucia (2018). *Selective* hub sorting was introduced to address this issue. In selective hub sorting, we only reorder a graph if the hubs are densely connected to each other. To decide whether to sort the graph, we compute the packing factor, a computationally inexpensive diagnostic value that predicts whether hub sorting is likely to be effective Balaji & Lucia (2018).

Hub Clustering. Hub clustering is the same as hub sorting, but without sorting the hubs after separating high and low degree nodes Balaji & Lucia (2018). The primary advantage of hub clustering is that it is an incredibly lightweight reordering algorithm that can be accomplished in a single pass through the graph.

Degree-Based Grouping. Degree-Based Grouping (DBG) Faldu et al. (2019) is an extension of hub clustering to multiple groups. Nodes are divided into w groups based on degree ranges associated with each group. The groups are sorted in descending degree order, but the order of the nodes within each group is not changed. The authors of Faldu et al. (2019) use logarithmically spaced thresholds to evenly distribute nodes among groups for power-law graphs. Since k-NN graphs do not have power-law degree distributions, our implementation uses the w quantiles of the degree distribution instead.

RADAR. RADAR is a recent graph ordering method that improves processing times by duplicating commonly-accessed nodes Balaji & Lucia (2019). The duplication allows important nodes to be physically located near more connections by providing multiple copies of the node in the data structure representing the graph. The authors treat high-degree nodes as important, though one could also use other measures of network centrality to identify candidates for duplication.

Unfortunately, this process is difficult to apply to near neighbor graphs for two reasons. First, neighbor graphs do not have the power-law structure that results in a small number of central nodes. Many nodes in a near neighbor graph would require duplication. Second, each node in a search index contains approximately 1 kB of vector attributes. RADAR assumes a much lower amount of meta-data (approximately 4 bytes), so the duplication would lead to GB of overhead.

RECALL. RECALL is a graph ordering method that incorporates both spatial and temporal locality to improve the speedup obtained by graph ordering methods Lakhotia et al. (2017). RECALL targets graph algorithms that iterate through every vertex multiple times. The critical observation is that graph processing times can improve if the iterative node processing order is scheduled to prefer recently-seen nodes (or their spatial neighbors). Unfortunately, this does not work well in the context of near neighbor indices. A search query visits each node in the graph at most one time, ultimately

Table 1: Comparison of objective-driven graph order methods and graph partitioning (LDG).

Query time (99% recall)	Original Order	Gorder	RCM	LDG
SIFT1M	1.4069 ms (0%)	1.253 ms (11%)	1.2671 ms (10%)	1.3816 ms (2%)

processing a very small subset of the graph. This violates the stationarity assumption made by methods like RECALL.

Graph Partitioning. Graph partitioning is a very well-studied area with a large number of established methods. One can create a graph reordering algorithm from any graph partitioning method by using the graph partitioning method to assign nodes to groups and using the same labeling scheme as in DBG.

However, there is substantial evidence that graph partitioning methods do not work well as objective-driven reorderings. While graph partitioning algorithms are suitable to accelerate computation by sharding a graph among several different processors, they do not produce a sufficiently fine-grained ordering to accelerate single-core operations via cache mechanisms. For example, Wei et al. (2016) and Lecuyer & Tabourier (2021) both demonstrate that graph partitioning does not consistently result in orderings that are better than random.

We reproduce this result in Table 1, where we compare objective-driven methods (Gorder and RCM) with LDG, a leading graph partitioning method Stanton & Kliot (2012). We chose LDG because it works well on directed graphs and is efficient enough to partition graphs with millions of nodes into hundreds of thousands of partitions. It should be noted that LDG is a highly competitive baseline. For example, Stanton & Kliot (2012) evaluate 17 different methods and find that LDG is second only to METIS Karypis & Kumar (1997). Tsourakakis et al. (2014) demonstrate similar results, placing LDG in the top three methods. Because methods like METIS are very expensive to run in the context of reordering (due to the large number of partitions Wei et al. (2016)), LDG is likely one of the best options available from graph partitioning.

We also observe that graph partitioning is not an effective reordering method. For this reason, we exclude it from our experimental comparison.

3 Proofs

In this section, we provide proofs for the theorems, as well as a more complete description of our theory. This section is an independent presentation of the material, and we re-state the theorems as needed. Recall from the main text that we consider the number of cache misses for one step of breadth-first search, for the reasons described in the main text. That is, we consider the cost to first access a node, and then to access each of its out-neighbors in sequence. We also consider the number of cache hits, or the cache efficiency CE_i, associated with visiting node v_i.

$$C_i = 1 + \sum_{\substack{v_j \in E(v_i) \\ j=1\ldots\deg(v_i)}} \mathbb{1}_{\{v_j \notin l(v_i)\}} \prod_{k<j} \mathbb{1}_{\{v_j \notin l(v_k)\}}$$

$$\mathrm{CE}_i = \sum_{\substack{v_j \in E(v_i) \\ j=1\ldots\deg(v_i)}} \mathbb{1}_{\{v_j \in l(v_i) \bigcup_{k<j} l(v_k)\}}$$

Here, $l(v)$ is the cache line containing v and $E(v)$ is the set of outbound edges of v. The 1 represents the cost to load v_i in $l(v_i)$, and the sum represents the cost to load the lines containing neighbors of v_i.

Minimizing the cost is equivalent to maximizing the efficiency score because a cache hit occurs if and only if there is not a cache miss.

Alternative Expression for the Efficiency Score. A critical part of our theory is to rewrite the cache efficiency score in terms of the $S_s(u, v)$ and $S_n(u, v)$ terms of the gorder objective. This will allow us to connect the notion of cache efficiency with the quality of the graph layout.

Theorem 1. *Given a graph $G = (V, E)$, let*

$$\mathrm{UB} = \sum_{i=1}^{N} \left(\sum_{j=\lfloor i/B \rfloor+1}^{i-1} S_s(v_i, v_j) + S_n(v_i, v_j) - \max\left\{ \sum_{j=\lfloor i/B \rfloor+1}^{\lceil i/B \rceil} \mathbb{1}_{\{(v_i,v_j)\in E\}} - 1, 0 \right\} \right)$$

Then the cache efficiency score $\mathrm{CE}_{\mathrm{DFS}}(P)$ obeys the following inequalities.

$$\mathrm{UB} - \sum_{i=1}^{N} \mathrm{Triplets}(v_i) \leq \mathrm{CE}_{\mathrm{DFS}}(P) \leq \mathrm{UB}$$

where $S_s(v_i, v_j)$ and $S_n(v_i, v_j)$ are as defined in Equation 1 and $\mathrm{Triplets}(v_i)$ is the number of triplet combinations of v_i's out-neighbors that fall into the same cache line.

Proof. We write the cache efficiency as follows.

$$\mathrm{CE}_{\mathrm{DFS}} = \sum_{i=1}^{N} \sum_{\substack{j:\lfloor j/B \rfloor = \lfloor i/B \rfloor \\ j \neq i}} \mathbb{1}_{\{(v_j,v_i)\in E\}} + (1/2) \sum_{v_k = \mathrm{Parents}(v_j)} \mathbb{1}_{\{\cup_{j<i} \text{siblings } (v_i,v_j) \text{ from } v_k\}} \mathbb{1}_{\{\lfloor k/B \rfloor \neq \lfloor i/B \rfloor\}}$$

The sum is over all nodes in each cache line, and the first indicator counts the number of times that v_j is a parent of v_i. The second sum is over all parents of v_j, and counts the number of sibling relationships between v_i and v_j where the parent v_k is not in the cache line. It is necessary to exclude situations where v_k is already in the cache line, since those cache hits are already counted by the first term. Also, we must exclude extra counts that arise when v_i has multiple siblings in the cache line, since only the first one matters for the purposes of a cache hit. To count once if a sibling is present (rather than once for each sibling that is present), we take the union $\cup_{j<i}$ over siblings inside the second indicator. Also, note that the $\frac{1}{2}$ factor is needed to avoid double-counting a sibling cache hit, since the sibling relationship is symmetric. Finally, observe that nodes at permutation locations i and j are in the same cache line if $\lfloor i/B \rfloor = \lfloor j/B \rfloor$.

The next step is to change the bound on the internal sum from $j \neq i : \lfloor j/B \rfloor = \lfloor i/B \rfloor$ to $j = [\lfloor j/B \rfloor, i-1]$. For the first term, we simply need to add an indicator to count when v_i is a parent of v_j in addition to the existing indicator that counts when v_j is a parent of v_i. For the second term, when we change the sum from being over $\lfloor j/B \rfloor = \lfloor i/B \rfloor$, we pick up two indicator functions: one for when v_i and v_j are siblings and one for when v_j and v_i are siblings. For this reason, we may remove the $1/2$ term when summing over the items in the line up to item $i-1$.

$$\mathrm{CE}_{\mathrm{DFS}} = \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor}^{i-1} \mathbb{1}_{\{(v_j,v_i)\in E\}} + \mathbb{1}_{\{v_i,v_j\}\in E} + \sum_{v_k = \mathrm{Parents}(v_j)} \mathbb{1}_{\{\cup_{j<i} \text{siblings } (v_i,v_j) \text{ from } v_k\}} \mathbb{1}_{\{\lfloor k/B \rfloor \neq \lfloor i/B \rfloor\}}$$

Upper Bound. First, we will prove the upper bound. We will return to this point in the analysis later, when we prove the lower bound.

To get the upper bound, we start by applying the Bonferroni inequality (with just one expansion of the inclusion-exclusion principle) to the union $U_{j<i}$ over siblings in the cache line.

$$\mathbb{1}_{\{\cup_{j<i} \text{siblings } (v_i,v_j) \text{ from } v_k\}} \leq \sum_{\substack{j<i \\ \lfloor j/B \rfloor = \lfloor i/B \rfloor}} \mathbb{1}_{\{\text{siblings } (v_i,v_j) \text{ from } v_k\}}$$

Using the upper bound on the indicator, we can express the second sum as *the number of shared parents of v_i and v_j that do not appear in the cache line*, which we denote as $|\,(\mathrm{Parents}(v_i) \cap \mathrm{Parents}(v_j)) \setminus l(v_i)|$ (where $l(v_i)$ is the set of nodes in the cache line of v_i).

$$\mathrm{CE}_{\mathrm{DFS}} \leq \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} \left(\mathbb{1}_{\{(v_j,v_i)\in E\}} + \mathbb{1}_{\{(v_i,v_j)\in E\}} + |\,(\mathrm{Parents}(v_i) \cap \mathrm{Parents}(v_j)) \setminus l(v_i)| \right)$$

Observe that $\mathbb{1}_{\{(v_j,v_i)\in E\}} = S_s(v_i, v_j)$ from the GO objective. Also note that the intersection of parents is equivalent to

$$|\text{Parents}(v_i) \cap \text{Parents}(v_j)| - |\text{Parents}(v_i) \cap \text{Parents}(v_j) \cap l(v_i)|$$

This allows us to write the cache efficiency as follows:

$$\text{CE}_{\text{DFS}} \leq \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} S_s(v_i, v_j) + S_n(v_i, v_j) - \sum_{j=\lfloor i/B \rfloor}^{i-1} |\text{Parents}(v_i) \cap \text{Parents}(v_j) \cap l(v_i)|$$

Finally, examine the sum of $|\text{Parents}(v_i) \cap \text{Parents}(v_j) \cap l(v_i)|$. This sum accumulates a $+1$ every time a child of a node in $l(v_i)$ is also in $l(v_i)$, with the exception of the first child. This can be written as

$$\sum_{j=\lfloor i/B \rfloor}^{\lceil i/B \rceil} \mathbb{1}_{\{(v_i,v_j)\in E\}} - 1$$

Of course, if there are no children, the cost is 0 rather than -1, so the term is actually

$$\max \left\{ \sum_{j=\lfloor i/B \rfloor+1}^{\lceil i/B \rceil} \mathbb{1}_{\{(v_i,v_j)\in E\}} - 1, 0 \right\}$$

which completes the expression for the upper bound.

Lower Bound. To get the lower bound, this time we will apply the Bonferroni inequality with two expansions of the inclusion-exclusion principle.

$$\begin{aligned} \mathbb{1}_{\{\cup_{j<i}\text{siblings } (v_i,v_j) \text{ from } v_k\}} \geq & \sum_{\substack{j<i \\ \lfloor j/B \rfloor = \lfloor i/B \rfloor}} \mathbb{1}_{\{\text{siblings } (v_i,v_j) \text{ from } v_k\}} \\ & - \sum_{\substack{j<l<i \\ \lfloor j/B \rfloor = \lfloor i/B \rfloor = \lfloor l/B \rfloor}} \mathbb{1}_{\{\text{siblings } (v_i,v_j) \text{ from } v_k\}} \mathbb{1}_{\{\text{siblings } (v_i,v_l) \text{ from } v_k\}} \end{aligned}$$

The analysis proceeds as before, by grouping the first part of the expression with the other terms to create $S_s(v_i, v_j)$ and $S_n(v_i, v_j)$. However, now we have to deal with a second term that contributes

$$\sum_{i=1}^{N} \sum_{j<l<i} |\text{Parents}(v_j) \cap \text{Parents}(v_l) \cap \text{Parents}(v_i) \setminus l(v_i)|$$

where nodes (v_j, v_l, v_i) are constrained to be in the same cache line (i.e. $\lfloor j/B \rfloor = \lfloor l/B \rfloor = \lfloor i/B \rfloor$). Since we are interested in a lower bound, we may disregard the intersection $\setminus l(v_i)$, as $-|A \setminus B| \geq -|A|$ for any sets A and B. Upon closer examination, the remaining terms pick up a $+1$ increment for each triplet combination of children in the same cache line. This results in the following expression.

$$\sum_{i=1}^{N} \text{Triplets}(v_i)$$

where $\text{Triplets}(v_i)$ is defined as the number of triplet combinations of children of v_i that are in the same cache line.

□

Using the expression for CE_{DFS} that we derived in Theorem 1, one can show that the graph reordering problem is an instance of the NP-hard maximum perfect matching problem on hypergraphs.

Lemma 1. *Given a graph $G = (V, E)$, the problem of finding the layout with an optimal DFS cache cost is an NP-hard instance of maximum weight perfect hypergraph matching.*

Proof. The maximum weight perfect hypergraph matching problem is as follows. Given a B-regular hypergraph $G = (V, E)$, associate a weight w_e with each edge $e \in E$. The task is to select a set of edges S that maximize the sum $\sum_{e \in S} w_e$, subject to the constraint that each vertex v appear in only one edge $e \in S$. This corresponds to the problem of Parekh & Pritchard (2012) where $d_e = 1$, $b_v = 1$ and $k = B$.

The graph reordering task is to select groups of B vertices $e = \{v_i, v_j, ...v_k\}$ with $|e| = B$ to place into the same cache line with the objective of maximizing the cache efficiency score $\mathrm{CE}_{\mathrm{DFS}}$. Each possible cache line contributes a term (or weight) w_e to the $\mathrm{CE}_{\mathrm{DFS}}$ sum. This is an instance of the maximum weight perfect hypergraph matching problem, with G as the hypergraph with the edges e being all possible combination of B nodes. The maximum weight perfect hypergraph matching problem is NP-hard Parekh & Pritchard (2012), and the special case considered here does not improve the complexity. □

Now, we will show that the gorder algorithm with window size $w = B$ is an approximation algorithm for the DFS cache layout problem. Note that we must add an additional $O(NB)$ step to the end of the algorithm to cover an edge case in the proof, but the graph ordering remains essentially unchanged.

Our proof strategy is to prove bounds on the cache efficiency score $\mathrm{CE}_{\mathrm{DFS}}$ that are in terms of the gorder objective F_{GO}. We will then use these bounds, together with the fact that gorder provides a $\frac{1}{2B}$-approximation to $F^{\star}_{\mathrm{GO}}$ (the optimal value of the gorder objective), to show that the gorder layout also provides an approximation to $\mathrm{CE}^{\star}_{\mathrm{DFS}}$ (the optimal cache efficiency). The upper bound (Lemma 2) is straightforward.

Lemma 2. *Upper bound:*

$$\mathrm{CE}_{\mathrm{DFS}}(P) \leq F_{\mathrm{GO}}(P)$$

Proof. This follows from Theorem 1 by removing the $\max$ operation. □

The lower bound is both more complicated to show and looser than the upper bound.

Lemma 3. *Lower bound: Given a permutation P with gorder objective $F_{\mathrm{GO}}(P)$, we can shift the permutation to obtain a permutation P' which satisfies*

$$\mathrm{CE}_{\mathrm{DFS}}(P') \geq \frac{1}{B} F_{\mathrm{GO}}(P) - N(B-1) - \lfloor M/B \rfloor \binom{B}{3}$$

Proof. Recall that the cache efficiency score is

$$\mathrm{CE}_{\mathrm{DFS}} \geq \sum_{i=1}^{N} \Bigg(\sum_{j=\lfloor i/B \rfloor + 1}^{i-1} S_s(v_i, v_j) + S_n(v_i, v_j) \\ - \max \left\{ \sum_{j=\lfloor i/B \rfloor + 1}^{\lceil i/B \rceil} \mathbb{1}_{\{(v_i, v_j) \in E\}} - 1, 0 \right\} - \mathrm{Triplets}(v_i) \Bigg)$$

Since we require a lower bound for $\mathrm{CE}_{\mathrm{DFS}}$, we may replace the maximum operation to get the following inequality:

$$\mathrm{CE}_{\mathrm{DFS}} \geq \sum_{i=1}^{N} \Bigg(\sum_{j=\lfloor i/B \rfloor + 1}^{i-1} S_s(v_i, v_j) + S_n(v_i, v_j) - \sum_{j=\lfloor i/B \rfloor}^{\lceil i/B \rceil - 1} (\mathbb{1}_{\{(v_i, v_j) \in E\}} - 1) - \mathrm{Triplets}(v_i) \Bigg)$$

Remark. If $B = 2$ (i.e. we consider only pairs of nodes) then both $\mathrm{Triplets}(v_i)$ and $\max\{\mathbb{1}_{(v_i, v_j) \in l(v_i)} - 1, 0\}$ are zero. In this case, a tighter lower bound is $\frac{1}{B} F_{\mathrm{GO}}$.

We can use the same trick used by Gorder to rewrite the second sum in terms of the bounds of the first sum. Specifically, we perform the sum over half of the terms, but include a second indicator for the case when v_j is a parent of v_i.

$$= \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} \Big(S_s(v_i, v_j) + S_n(v_i, v_j) - \mathbb{1}_{\{(v_i,v_j)\in E\}} - \mathbb{1}_{\{(v_j,v_i)\in E\}} - \text{Triplets}(v_i) \Big)$$

$$\geq \quad \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} \Big(S_s(v_i, v_j) \quad + \quad S_n(v_i, v_j) \quad - \quad 2 \Big) \quad - \quad \text{Triplets}(v_i)$$

If every node has a maximum of M children and there are B locations available in the cache line, then the maximum value $\text{Triplets}(v_i)$ can take is $\lfloor M/B \rfloor \binom{B}{3}$.

$$\geq \quad \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} \Big(S_s(v_i, v_j) \quad + \quad S_n(v_i, v_j) \quad - \quad 2 \Big) \quad - \quad N\lfloor M/B \rfloor \binom{B}{3}$$

$$= \quad \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} \Big(S_s(v_i, v_j) \; + \; S_n(v_i, v_j) \Big) \; - \; 2\frac{N}{B}\frac{(B-1)B}{2} \; - \; N\lfloor M/B \rfloor \binom{B}{3}$$

where the final equality comes from the fact that the sum over j adds the -2 constant factor $1, 2, ...B-1$ times, and we repeat this process N/B times. It remains to show that

$$\sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} S_s(v_i, v_j) \; + \; S_n(v_i, v_j) \quad \leq \quad \sum_{i=1}^{N} \sum_{j=i-B}^{i-1} S_s(v_i, v_j) \; + \; S_n(v_i, v_j) \quad = \quad F_{\text{GO}}$$

To show this, suppose the gorder objective returns a permutation P_1. Every B^{th} term of the outer sum is identical for F_{GO} and the efficiency score, since we have a sum from $j = k(B-1)$ to $j = kB-1$ for $k \in \mathbb{Z}$. However, the other terms of the efficiency score are smaller. For example, the term corresponding to the first element in a new cache line is *zero* because it is a sum from $j = kB$ to $j = kB - 1$.

To relate the two sums, we will use the following fact. Given a set of numbers $x_1, x_2, ...x_N$ with mean $\bar{x}$, there exists an $x_i \geq \bar{x}$. We take the mean of the terms in the cache efficiency score over the first B shifted permutations $P_1, ...P_{B-1}$.

Cache Alignment (Shifting) Algorithm: When we say *shifted* permutation, we mean that the order of nodes remains the same, but the cache boundaries have been shifted. For example, suppose that under a permutation P_1, the first cache line consists of nodes $[v_1, v_2, ...v_B]$. The second cache line begins at node v_{B+1}, and contains an additional B nodes. We define the *shifted* permutation P_2 as the layout having the same node ordering, but with cache line boundaries beginning one slot to the *left*. That is, the first cache line begins one space to the left of v_1. Since there is no node to the left of v_1, this line contains the first $B-1$ nodes $[v_1, v_2, ...v_{B-1}]$, and the second line begins at v_B. It is easy to see that there are a total of B shifted permutations (since shifting the locations by B results in the same line boundaries). Also, note that shifting does not change the gorder score because it maintains the order of the nodes.

Observe that, because of the shifting, the permutations together contain *exactly one* full copy of the terms in F_{GO}. They also contain a number of other, partial terms. Let $x(P)$ be the following quantity, under the permutation P:

$$x(P) = \sum_{i=1}^{N} \sum_{j=\lfloor i/B \rfloor+1}^{i-1} S_s(v_i, v_j) + S_n(v_i, v_j)$$

Because shifting does not affect the gorder score, we have the following inequality:

$$\bar{x} = \frac{1}{B}\sum_{i=1}^{B} x(P_i) \leq \frac{1}{B}F_{\text{GO}}(P_1)$$

The fact presented earlier implies that at least one of the permutations satisfies

$$x(P) \leq \frac{1}{B}F_{\text{GO}}(P)$$

It is easy to find the permutation of the gorder layout that satisfies this inequality. We can simply evaluate the cache efficiency score for the first B shifted permutations and select the permutation with the largest score. Given a permutation, one can compute the gorder score in $O(N)$ time, so this algorithm runs in time $O(BN)$. Let P' be the permutation found in this way. Then we have

$$\text{CE}_{\text{DFS}}(P') \geq \frac{1}{B}F_{\text{GO}}(P) - N(B-1) - N\lfloor M/B \rfloor \binom{B}{3}$$

□

We are finally ready to prove Theorem 2 using a combination of Lemma 2 and Lemma 3.

Theorem 2. *Let $G = (V, E)$ be a graph with a maximum out-degree M, P be a layout with objective score $F_{\text{GO}}(P)$, and P' be the layout obtained by running Gorder plus a simple cache boundary alignment procedure on P. P' satisfies the following inequalities:*

$$\text{CE}_{\text{DFS}}(P') \geq \frac{1}{B}F_{\text{GO}}(P) - N(B-1) - N\lfloor M/B \rfloor \binom{B}{3} \qquad \text{CE}_{\text{DFS}}(P') \leq F_{\text{GO}}(P)$$

We use the convention $\binom{a}{b} = 0$ when $a < b$.

Proof. The proof of this statement is straightforward. The lower bound comes directly from Lemma 3. Lemma 2 does not directly give the result in the theorem, instead yielding

$$\text{CE}_{\text{DFS}}(P') \leq F_{\text{GO}}(P')$$

However, recall from the proof of Lemma 3 that the shifting process does not modify the gorder score. Therefore, we have that

$$F_{\text{GO}}(P') = F_{\text{GO}}(P)$$

□

Finally, we use Theorem 2 to prove that any algorithm which makes a large enough improvement to the gorder objective will also improve the cache efficiency. Theorem 3 essentially states that “better” layouts (under the gorder objective) have higher cache efficiency scores.

Theorem 3. *Given two node orderings P_1 and P_2 on a graph with maximum out-degree M, $\text{CE}_{\text{DFS}}(P_2) \geq \text{CE}_{\text{DFS}}(P_1)$ if $F_{\text{GO}}(P_2) \geq BF_{\text{GO}}(P_1) + B(B-1)N + NM\binom{B}{3}$.*

Proof. For convenience, let $F_2 = F_{\text{GO}}(P_2)$, $F_1 = F_{\text{GO}}(P_1)$, $\text{CE}_1 = \text{CE}_{\text{DFS}}(P_1)$ and $\text{CE}_2 = \text{CE}_{\text{DFS}}(P_2)$. Suppose

$$F_2 = F_1 + \Delta$$

We wish to find the smallest value of Δ that guarantees $\text{CE}_2 \geq \text{CE}_1$. First, note that

$$C_2 \geq \frac{1}{B}F_2 - N(B-1) - N\lfloor M/B \rfloor \binom{B}{3}$$

Since $F_2 = F_1 + \Delta$,

$$C_2 \geq \frac{1}{B}F_1 + \frac{1}{B}\Delta - N(B-1)$$

Suppose we have the following value of Δ

$$\Delta = F_1(B-1) + N(B-1)B + BN\lfloor M/B \rfloor \binom{B}{3}$$

Then the previous inequality becomes

$$C_2 \geq \frac{1}{B}F_1 + \frac{B-1}{B}F_1 + \frac{N(B-1)B}{B} - N(B-1) + \frac{BN\lfloor M/B \rfloor}{B}\binom{B}{3} - N\lfloor M/B \rfloor \binom{B}{3}$$

or, simply

$$C_2 \geq F_1$$

From Lemma 2, $F_1 \geq C_1$, so we have that $C_2 \geq C_1$ when

$$\Delta \geq F_1(B-1) + N(B-1)B + BN\lfloor M/B \rfloor \binom{B}{3}$$

Since $\lfloor M/B \rfloor \leq M/B$, any Δ satisfying the following (slightly looser) condition will also work.

$$\Delta \geq F_1(B-1) + N(B-1)B + NM\binom{B}{3}$$

Or, in other words,

$$F_2 \geq BF_1 + B(B-1)N + NM\binom{B}{3}$$

□

4 Supplementary Experiments

In this section, we provide more details about our experiments, as well as additional experiments on different hardware and with various cache ablations.

4.1 Datasets

We obtained our smaller datasets and queries from the ANN-benchmarks repository. These datasets are released under the MIT license. For the large DEEP100M and SIFT100M experiments, we used the original DEEP1B and SIFT1B datasets. DEEP1B is hosted by Yandex under the CC BY 4.0 license: `https://research.yandex.com/datasets/biganns`. SIFT1B is available in the public domain, here: `http://corpus-texmex.irisa.fr`. We used the DEEP100M ground truth results available here: `https://github.com/matsui528/deep1b_gt`, which are also released under the MIT license.

4.2 Methods

Objective-Based Reordering. Since our k-NN search benchmarks involve datasets with millions of nodes, we avoid algorithms that are known to scale poorly to large graphs. We implemented Gorder and RCM, but we did not implement MLOGA or MLINA because these algorithms have very high runtime without tangible improvements over Gorder or RCM Wei et al. (2016). We exclude several other baselines that are also not well-suited to near-neighbor graphs, including RADAR, RECALL and Graph Partitioning. RADAR requires a substantial memory overhead, while RECALL requires a different type of graph workload to attain acceleration. Graph partitioning methods are not fine-grained enough to produce a cache-based speedup for graphs with large node sizes.

Degree-Based Reordering. At first glance, degree-based methods seem inappropriate because near-neighbor graphs may have a constant degree distribution. However, this is not actually the case (Figure 4), so it is reasonable to consider lightweight reordering. We follow the suggestion of Balaji & Lucia (2018) to use the *in-degree* as a local feature for hub clustering and degree-based methods because beam search is a so-called push application.[2] We use the average degree as the threshold for hub clustering and sorting, and we use 8 groups given by the 8 quantiles of the degree distribution for DBG.

---

[2] See Section 2 of Balaji & Lucia (2018) for the definition of push/pull applications.

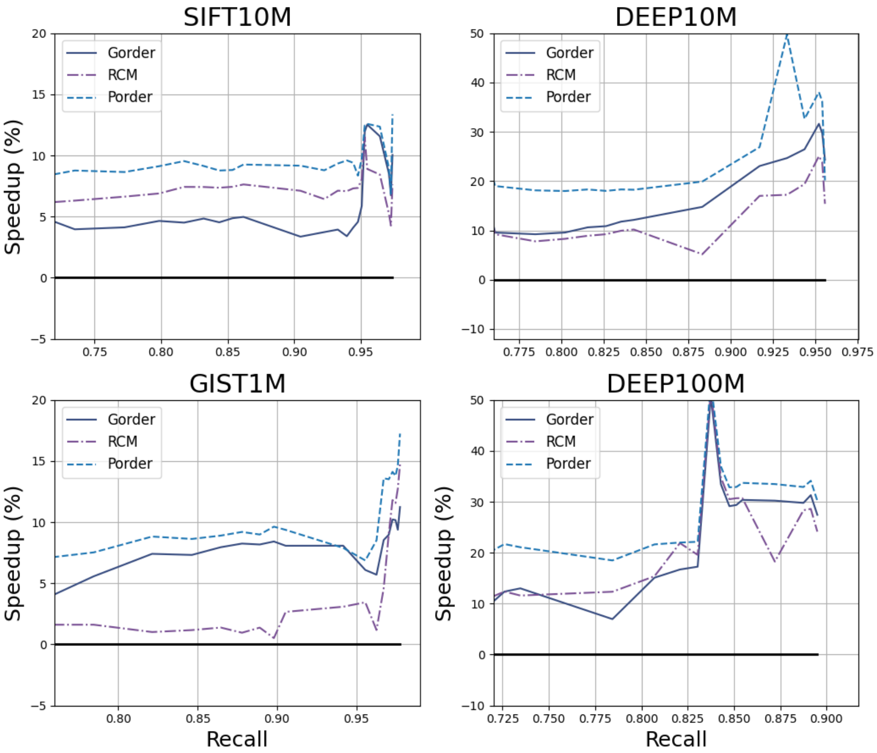

Figure 1: Graph reordering experiments, reproduced on a different CPU. We observe similar query accelerations even on a machine with substantially different cache dimensions.

4.3 Implementation Details

We wrote our code to use the same base memory layout as the nmslib implementation of HNSW. In the lowest search level (the focus of our study), HNSW allocates node data in a contiguous block of memory. Node data is stored alongside the edge list and metadata for each node. We use a similar layout, where our nodes are stored in a pre-allocated contiguous block of memory. Each node is laid out as follows: [integer array of edges][node data][node metadata (label)]. We directly use the query algorithm from nmslib/hnswlib in our implementation. We considered both the hash-based visited list of the NGT library and the array-based visited list of the nmslib library. We chose the array-based list because our performance tests revealed it to be faster.

4.4 Additional Experiments

In this section, we present several ablations and additional experiments that support our conclusions.

Reproducibility Experiment. We re-ran our top performing methods for SIFT10M, DEEP10M, GIST1M and DEEP100M on a different CPU with a different cache profile. This experiment was run on a machine with 96 Intel Xeon Gold 5220R CPUs, each having 1.5 MB of L1 cache, 48 MB of L2 cache and a 71.5 MB shared L3 cache. We rebuilt all the indices for Gorder, Porder and RCM on this machine to understand how our results can be expected to translate to other hardware platforms. These results are presented in Figure 1. We find that our results are reproducible and that Porder consistently outperforms Gorder (albeit by a small margin on uniformly-distributed datasets). We expect an even better performance difference for real-world problems because many search problems have non-uniform, highly skewed query distributions (e.g. in product recommendation systems, some queries are issued much more frequently than others).

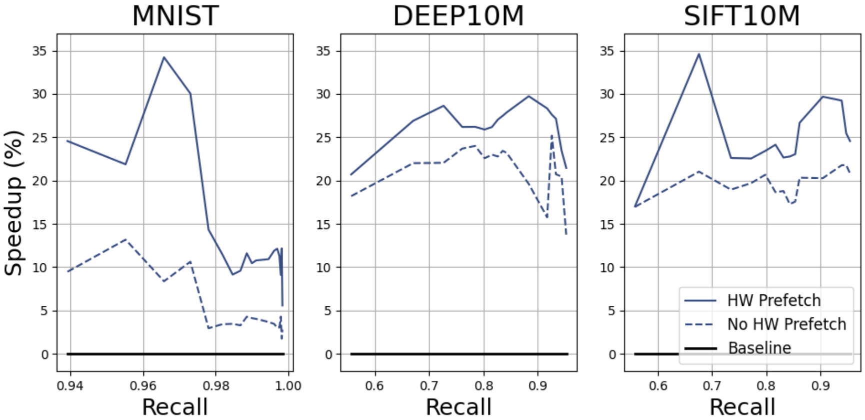

Figure 2: Prefetcher ablation, where we disable the L1 and L2 prefetcher. The average speedups with/without prefetching for MNIST, DEEP10M and SIFT10M are as follows: (15%, 5%), (26%, 21%), (25%, 20%).

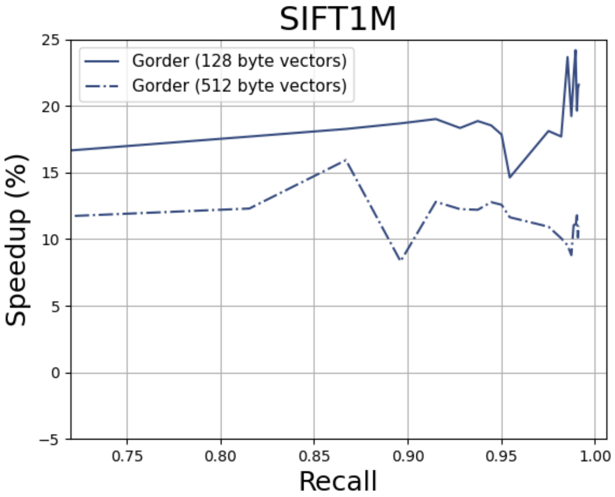

Figure 3: Node size ablation, where we consider two equivalent versions of SIFT1M. In the 128-byte version, we represent each dimension as an 8-bit integer. In the 512-byte version, we represent each dimension as a 32-bit float. The speedup is larger for smaller node sizes.

Prefetcher Ablation. To isolate the effect of cache coherency from that of hardware prefetching, we turned off the L1/L2 hardware prefetcher. We issued 10K queries and measured the average speedup (Figure 3). We ran the prefetch experiments on an Intel Broadwell Core i5 and we turned off the prefetchers via the MSR registers (specifically, register 0x1A4). This disables the L1 data prefetcher and the next-line prefetchers for the L2 data and instruction caches. It also disables the L2 streaming prefetcher, which can prefetch into either the L2 or L3 cache. However, the BIOS settings expose an LLC prefetch setting for the L3 cache and the TLB cache has its own prefetcher. We were unable to completely disable these for the "No HW Prefetch" results in Figure 3 without unrestricted access to the BIOS / UEFI.

Because we see a drop in performance when hardware prefetching is turned off, L1/L2 hardware prefetching is responsible for a fairly large amount of performance, especially for datasets with large vector sizes (e.g. MNIST). The remaining speedup is likely due to cache coherence for auxiliary data structures such as the list of node statuses and L3 / TLB prefetching (which we were unable to disable).

Node Size Ablation. To understand how data vector size affects reordered performance, we constructed two equivalent versions of the SIFT1M dataset - one where vectors are represented as 32-bit

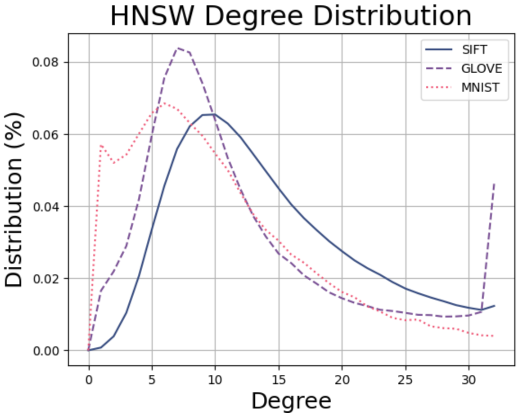

Figure 4: Empirical out-degree distribution of HNSW with a maximum of $M = 32$ outbound edges. Although we begin with $k = 32$ outbound edges per node, the pruning heuristic removes over half of the edges from the network. The result is a graph that is far from k-regular while also not belonging to the common class of power-law graphs.

floats (512 bytes / vector) and one where vectors are represented as 8-bit integers (128 bytes / vector). Because the datasets are the same, we use the same graph connections for both indices and modify only the on-disk representation of each vector. To remove speedup effects due to faster distance functions, we compare speedups relative to the non-reordered graph. We observe that larger vectors result in a smaller speedup (average of 11% vs 19%) (Figure 3).

References

Arya, S. and Mount, D. M. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, volume 93, pp. 271–280. Citeseer, 1993.

Aumüller, M., Bernhardsson, E., and Faithfull, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

Auroux, L., Burelle, M., and Erra, R. Reordering very large graphs for fun & profit. In *International Symposium on Web AlGorithms*, 2015.

Balaji, V. and Lucia, B. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 203–214. IEEE, 2018.

Balaji, V. and Lucia, B. Combining data duplication and graph reordering to accelerate parallel graph processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 133–144, 2019.

Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., and Raghavan, P. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 219–228, 2009.

Cuthill, E. and McKee, J. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pp. 157–172, 1969.

Dong, W., Moses, C., and Li, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pp. 577–586, 2011.

Faldu, P., Diamond, J., and Grot, B. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–13. IEEE, 2019.

George, J. A. Computer implementation of the finite element method. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1971.

Iwasaki, M. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*, pp. 20–33. Springer, 2016.

Iwasaki, M. and Miyazaki, D. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018.

Karypis, G. and Kumar, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

Laarhoven, T. Graph-based time-space trade-offs for approximate near neighbors. *arXiv preprint arXiv:1712.03158*, 2017.

Lakhotia, K., Singapura, S., Kannan, R., and Prasanna, V. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pp. 273–282. IEEE, 2017.

Lecuyer, Fabrice, M. D. and Tabourier, L. [re] speedup graph processing by graph ordering. *The ReScience Journal*, 7, 2021.

Lin, P.-C. and Zhao, W.-L. A comparative study on hierarchical navigable small world graphs. *arXiv preprint arXiv:1904.02077*, 2019a.

Lin, P.-C. and Zhao, W.-L. Graph based nearest neighbor search: Promises and failures. *arXiv preprint arXiv:1904.02077*, 2019b.

Parekh, O. D. and Pritchard, D. Approximation algorithms for generalized hypergraph matching problems. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2012.

Prokhorenkova, L. and Shekhovtsov, A. Graph-based nearest neighbor search: From practice to theory. In *International Conference on Machine Learning*, pp. 7803–7813. PMLR, 2020.

Safro, I. and Temkin, B. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 9(2):190–202, 2011.

Sebastian, T. B. and Kimia, B. B. Metric-based shape retrieval in large databases. In *Object recognition supported by user interaction for service robots*, volume 3, pp. 291–296. IEEE, 2002.

Stanton, I. and Kliot, G. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1222–1230, 2012.

Tsourakakis, C., Gkantsidis, C., Radunovic, B., and Vojnovic, M. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pp. 333–342, 2014.

Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1813–1828, 2016.

Zhang, Y., Kiriansky, V., Mendis, C., Amarasinghe, S., and Zaharia, M. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 293–302. IEEE, 2017.