Efficient Single-Source Shortest Path and Distance Queries on Large Graphs

Andy Diwen Zhu　Xiaokui Xiao　Sibo Wang　Wenqing Lin
School of Computer Engineering
Nanyang Technological University
Singapore
{dwzhu, xkxiao, swang, wlin}@ntu.edu.sg

ABSTRACT

This paper investigates two types of graph queries: *single source distance (SSD)* queries and *single source shortest path (SSSP)* queries. Given a node v in a graph G, an SSD query from v asks for the distance from v to any other node in G, while an SSSP query retrieves the shortest path from v to any other node. These two types of queries find important applications in graph analysis, especially in the computation of graph measures. Most of the existing solutions for SSD and SSSP queries, however, require that the input graph fits in the main memory, which renders them inapplicable for the massive disk-resident graphs commonly used in web and social applications. There are several techniques that are designed to be I/O efficient, but they all focus on undirected and/or unweighted graphs, and they only offer sub-optimal query efficiency.

To address the deficiency of existing work, this paper presents *Highways-on-Disk (HoD)*, a disk-based index that supports both SSD and SSSP queries on directed and weighted graphs. The key idea of HoD is to augment the input graph with a set of auxiliary edges, and exploit them during query processing to reduce I/O and computation costs. We experimentally evaluate HoD on both directed and undirected real-world graphs with up to billions of nodes and edges, and we demonstrate that HoD significantly outperforms alternative solutions in terms of query efficiency.

Categories and Subject Descriptors

D.2.2 [**Graph theory**]: Graph algorithms

Keywords

Shortest path queries; Distance queries; Graph; Algorithm

1. INTRODUCTION

Given a graph G, a *single source distance (SSD)* query from a node $v \in G$ asks for the distance from v to any other node in G. Meanwhile, a *single source shortest path (SSSP)* query retrieves the shortest path from v to any other node. These two types of queries find important applications in graph analysis [8], especially in the computation of graph measures [5, 7, 11, 24]. For example, the estimation of *closeness* measures [11] on a graph G requires performing SSD queries from a large number of nodes in G, while the approximation of *betweenness* measures [7] requires executing numerous SSSP queries.

The classic solution for SSD and SSSP queries is Dijkstra's algorithm [10]. Given a SSD or SSSP query from a node s, Dijkstra's algorithm traverses the graph starting from s, such that the nodes in G are visited in ascending order of their distances from s. Once a node v is visited, the algorithm returns the distance from s to v based on the information maintained during the traversal; the shortest path from s to v can also be efficiently derived if required.

A plethora of techniques have been proposed to improve over Dijkstra's algorithm for higher query efficiency (see [9, 23] for surveys). Although those techniques all require pre-processing the given graph (which incurs extra overhead compared with Dijkstra's algorithm), the pre-computation pays off when the number of queries to be processed is large, as is often the case in graph analysis. Nevertheless, most of the existing techniques assume that the given graph fits in the main memory (for pre-computation and/or query processing), which renders them inapplicable for the massive disk-resident graphs commonly used in web and social applications. There are a few methods [15, 17–20] that address this issue by incorporating Dijkstra's algorithm with I/O-efficient data structures, but the performance of those methods are shown to be insufficient for practical applications [8]. The main reason is that, when Dijkstra's algorithm traverses the graph, the order in which it visits nodes can be drastically different from the order in which the nodes are arranged on the disk. This leads to a significant number of random disk accesses, which results in poor query performance.

In contrast to the aforementioned techniques, Cheng et al. [8] propose the first practically efficient index (named *VC-Index*) for SSD and SSSP queries on disk-resident graphs. The basic idea of VC-Index is to pre-compute a number of *reduced* versions of the input graph G. Each reduced graph contains some relatively important nodes in G, as well as the distances between some pairs of those nodes. During query processing, VC-Index scans a selected subset of reduced graphs, and then derives query results based on the pre-computed distances. Compared with those methods based on Dijkstra's algorithm [15, 17–20], VC-Index is more efficient as it only performs sequential reads on disk-resident data.

Motivation and Contribution. All existing disk-based solutions for SSD and SSSP queries [15, 17–20] require that the input graph is undirected, which renders them inapplicable for any application built upon directed graphs. This is rather restrictive as numerous important types of graphs (e.g., road networks, web graphs, social graphs) are directed in nature. Furthermore, even when the input graph is undirected, the query efficiency of the existing solutions is less than satisfactory. In particular, our experiments (in Section 7)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD'13, August 11–14, 2013, Chicago, Illinois, USA.
Copyright 2013 ACM 978-1-4503-2174-7/13/08 ...$15.00.

show that VC-Index, albeit being the state of the art, requires tens of seconds to answer a single SSD query on a graph with less than 100 million edges, and needs more than two days to estimate the closeness measures on the same graph.

To address the deficiency of existing work, this paper proposes *Highways-on-Disk (HoD)*, a disk-based index that supports both SSD and SSSP queries on directed and weighted graphs. The key idea of HoD is to augment the input graph with a set of auxiliary edges (referred to as *shortcuts* [22]), and exploit them during query processing to reduce I/O and computation costs. For example, Figure 1a illustrates a graph G, and Figure 1e shows an augmented graph G^* constructed from G. G^* contains three shortcuts: $\langle v_8, v_9\rangle$, $\langle v_9, v_7\rangle$, and $\langle v_9, v_{10}\rangle$. Each shortcut has the same length with the shortest path connecting the endpoints of the shortcut. For example, the length of $\langle v_8, v_9\rangle$ equals 2, which is identical to the length of the shortest path from v_8 to v_9. Intuitively, the shortcuts in G^* enable HoD to efficiently traverse from one node to another (in a manner similar to how highways facilitate traversal between distant locations). For instance, if we are to traverse from v_1 to v_{10} in G^*, we may follow the path $\langle v_1, v_9, v_{10}\rangle$, which consists of only three nodes; in contrast, a traversal from v_1 to v_{10} in G would require visiting five nodes: v_1, v_9, v_6, v_7, and v_{10}.

In general, when HoD answers an SSD or SSSP query, it often traverses the augmented graph via shortcuts (instead of the original edges in G). We show that, with proper shortcut construction and index organization, the query algorithm of HoD always traverses nodes in the same order as they are arranged in the index file. Consequently, HoD can answer any SSD or SSSP query with a linear scan of the index file, and its CPU cost is linear to the number of edges in the augmented graph. We experimentally evaluate HoD on a variety of real-world graphs with up to 100 million nodes and 3 billion edges, and we demonstrate that HoD significantly outperforms VC-Index in terms of query efficiency. In particular, the query time of HoD is smaller than that of VC-Index by up to two orders of magnitude. Furthermore, HoD requires a smaller space and pre-computation time than VC-Index in most cases.

2. PROBLEM DEFINITION

Let G be a weighted and directed graph with a set V of nodes and a set E of edges. Each edge e in E is associated with a positive weight $l(e)$, which is referred to as the *length* of e. A path P in G is a sequence of nodes $\langle v_1, v_2, \ldots, v_k\rangle$, such that $\langle v_i, v_{i+1}\rangle$ ($i \in [1, k-1]$) is a directed edge in G. The length of P is defined as the sum of the length of each edge on P. We use $l(e)$ and $l(P)$ to denote the length of an edge e and a path P, respectively.

For any two nodes s and t in G, we define the *distance* from s to t, denoted as $dist(s, t)$, as the length of the shortest path from s to t. Given a *source node* s in G, a *single-source distance (SSD)* query asks for the distance from s to any other node in G. Meanwhile, a *single-source shortest path (SSSP)* query from s retrieves not only the distance from s to any other node v, but also the *predecessor* of v, i.e., the node that immediately precedes v in the shortest path from s to v. Note that, given the predecessor of each node, we can easily reconstruct the shortest path from s to any node v by backtracking from v following the predecessors. One may also consider an alternative formulation of SSD (resp. SSSP) query that, given only a *destination node* t, asks for the distance (resp. shortest path) from any other node to t. For simplicity, we will focus on SSD and SSSP queries from a source node s, but our solution can be easily extended to handle queries under the alternative formulation.

Let M be the size of the main memory available, and B be the size of a disk block, both measured in the number of words. We assume that $B \leq |V| \leq M \leq |E|$, i.e., the main memory can accommodate all nodes but not all edges in G. This is a realistic assumption since modern machines (even the commodity ones) have gigabytes of main memory, which is sufficient to store the node set of a graph with up to a few billion nodes. On the other hand, the number of edges in a real graph is often over an order of magnitude larger than the number of nodes, due to which E can be enormous in size and does not fit in the main memory.

Our objective is to devise an index structure on G that answers any SSD or SSSP query with small I/O and CPU costs, such that the index requires at most M main memory in pre-computation and query processing. In what follows, we will first focus on SSD queries in Sections 3-5, and will extend our solution for SSSP queries in Section 6.

3. SOLUTION OVERVIEW

As mentioned in Section 1, the main structure of HoD is a graph G^* that augments the input graph G with shortcuts. In this section, we present the overall idea of how the shortcuts in G^* are constructed and how they can be utilized for query processing, so as to form a basis for the detailed discussions in subsequent sections.

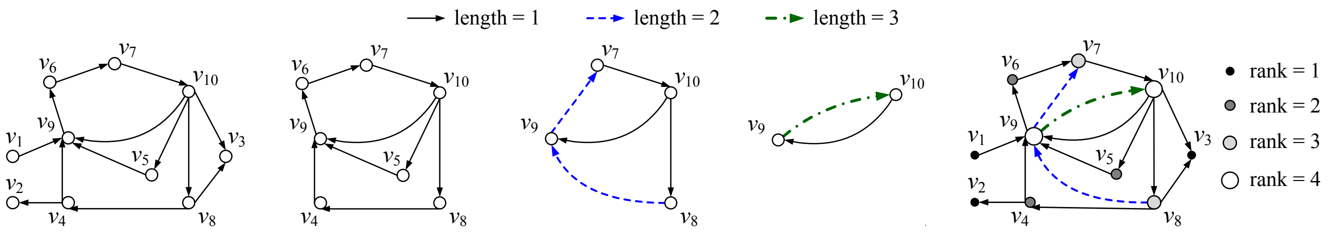
3.1 Shortcut Construction

In a nut shell, HoD constructs shortcuts with an iterative procedure, which takes as input a copy of the graph G (denoted as G_0). In the i-th ($i \geq 1$) iteration of the procedure, HoD first *reduces* G_{i-1} by removing a selected set of *less important* nodes in G_{i-1}, and then, it constructs shortcuts in the reduced graph to ensure that the distance between any two remaining nodes is not affected by the node removal. The resulting graph (with shortcuts added) is denoted as G_i, and it is fed as the input of the $(i+1)$-th iteration of procedure. This procedure terminates only when the reduced graph G_i is *sufficiently small*. All shortcuts created during the procedure are inserted into the original graph G, leading to an augmented graph G^* that would be used by HoD for query processing. We illustrate the iterative procedure with an example as follows.

EXAMPLE 1. Assume that the input to the iterative procedure is the graph G_0 in Figure 1a. Further assume that the reduced graph is sufficiently small if it contains at most two nodes and two edges. In the first iteration of the procedure, HoD inspects G_0 and identifies v_1, v_2, and v_3 as less important nodes. To explain, observe that the node v_1 in G_0 does not have any incoming edge, while v_2 and v_3 have no outgoing edges. As a consequence, v_1, v_2, and v_3 do not lie on the shortest path between any two other nodes. That is, even if we remove v_1, v_2, and v_3 from G_0, the distance between any two remaining nodes is not affected. Intuitively, this indicates that v_1, v_2, v_3 are of little importance for SSD queries. Therefore, HoD eliminates v_1, v_2, and v_3 from G_0, which results in the reduced graph G_1 in Figure 1b.

In the second iteration, HoD selects v_4, v_5, and v_6 as the less important nodes in G_1, and removes them from G_1. The removal of v_4 changes the distance from v_8 to v_9 to $+\infty$, since $\langle v_8, v_4, v_9\rangle$ is the only path (in G_1) that starts at v_8 and ends at v_9. To mitigate this change, HoD inserts into G_1 a shortcut $\langle v_8, v_9\rangle$ that has the same length with $\langle v_8, v_4, v_9\rangle$, as illustrated in Figure 1c. As such, the distance between any two nodes in G_1 remains unchanged after v_4 is removed. Similarly, when HoD eliminates v_6, it constructs a shortcut $\langle v_9, v_7\rangle$ with a length 2 to reconnect the two neighbors of v_6. Meanwhile, v_5 is removed without creating any shortcut, since deleting v_5 does not change the distance between its two neighbors. Figure 1c illustrates the resulting reduced graph G_2.

To explain why HoD chooses to remove v_4, v_5, and v_6 from G_1, observe that each of those nodes has only two neighbors. For any



(a) Graph G. (equivalently, G_0)
(b) Reduced graph G_1.
(c) Reduced graph G_2.
(d) Reduced graph G_3 (i.e., the core graph G_c).
(e) Augmented Graph G^*.

Figure 1: Graph reduction and shortcut construction.

of such nodes, even if the removal of the node changes the distance between its neighbors, HoD only needs to construct one shortcut to reconnect its neighbors. In other words, the number of shortcuts required is minimum, which helps reduce the space consumption of HoD. In contrast, if HoD chooses to remove v_9 from G_1 (which has a larger number of neighbors than v_4, v_5, and v_6), then much more shortcuts would need to be constructed.

Finally, in the third iteration, HoD removes v_7 and v_8 from G_2 as they are considered unimportant. The removal of v_7 leads to a new shortcut $\langle v_9, v_{10}\rangle$ with a length 3, since $\langle v_9, v_7, v_{10}\rangle$ is the only path connecting v_9 to v_{10}, and the length of the path equals 3. On the other hand, v_8 is directly eliminated as it is not on the shortest path between its only two neighbors v_9 and v_{10}. Figure 1d shows the reduced graph G_3 after the removal of v_7 and v_8.

Assume that the reduced graph G_3 is considered sufficiently small by HoD. Then, the iterative procedure of HoD would terminate. The three shortcuts created during the procedure (i.e., $\langle v_8, v_9\rangle$, $\langle v_9, v_7\rangle$, and $\langle v_9, v_{10}\rangle$) are added into the original graph G, which leads to the augmented graph G^* in Figure 1d. □

The above discussion leaves several issues open, i.e., (i) the specific criterion for identifying less important nodes in the reduced graph, (ii) the detailed algorithm for generating shortcuts after node removal, and (iii) the exact termination condition of the reduction procedure. We will clarify these issues in Section 4 by presenting the detailed preprocessing algorithm of HoD. For the discussions in the rest of this section, it suffices to know that when HoD terminates the reduction procedure, the reduced graph must fit in the main memory. We use G_c to denote this memory-resident reduced graph, and we refer to it as the *core graph*. (Note that G_c is a subgraph of the augmented graph G^*.) In addition, we define the *rank* $r(v)$ of each node v in G as follows:

1. If v is removed in the i-th iteration of the iterative procedure, then $r(v) = i$;
2. If v is not removed in any iteration (i.e., v is retained in the core graph G_c), then $r(v) = 1 + \max_{v \notin G_c} r(v)$, i.e., $r(v)$ is larger than the maximum rank of any node not in G_c.

For instance, in Example 1, the ranks of v_1, v_2, and v_3 equal 1, since they are removed from G in the first iteration of the reduction procedure. Similarly, $r(v_4) = r(v_5) = r(v_6) = 2$, and $r(v_7) = r(v_8) = 3$. The ranks of v_9 and v_{10} equal 4, since they are in the core graph G_c. The ranks of the nodes are utilized in the query processing algorithms of HoD, as will be illustrated shortly. Unless otherwise specified, we use the term *edge* to refer to both a shortcut and an original edge in G^*.

3.2 Query Processing

Given an SSD query from a node s, HoD answers the query with two traversals of the augmented graph G^*. The first traversal starts from s, and it follows only the *outgoing* edges of each node, ignoring any edge *whose starting point ranks higher than the ending point*. For instance, if HoD traverses from the node v_9 in Figure 1e, it would ignore the outgoing edge $\langle v_9, v_7\rangle$, since $r(v_9) = 4 > r(v_7) = 3$. As such, the first traversal of HoD never moves from a high-rank node to a low-rank node, and it terminates only when no higher-rank nodes can be reached. For each node v visited, HoD maintains the distance from s to v along the paths that have been seen during the traversal, denoted as $dist(s, v)$.

Let V' be the set of nodes that are not in the core graph of G^*. The second traversal of HoD is performed as a linear scan of the nodes in V', *in descending order of their ranks*. For each node $v' \in V'$ scanned, HoD inspects each *incoming* edge e of v', and then checks the starting point u of the edge. For any such u, HoD calculates $dist(s, u) + l(e)$ as an upperbound of the distance from s to v'. (Our solution guarantees that u should have been visited by HoD before v'.) Once all incoming edges of v' are inspected, HoD derives the distance from s to v' based on the upperbounds, and then it moves on to the next node in V'. This process terminates when all nodes in V' are examined.

We illustrate the above query algorithm of HoD with an example.

EXAMPLE 2. Consider an SSD query from node v_1 in Figure 1a. Given the augmented graph G^* in Figure 1e, HoD first traverses G^* starting from v_1, following the outgoing edges whose ending points rank higher than the starting points. Since v_1 has only one outgoing edge $\langle v_1, v_9\rangle$, and since v_9 ranks higher than v_1, HoD moves from v_1 to v_9. v_9 has three outgoing edges: $\langle v_9, v_6\rangle$, $\langle v_9, v_7\rangle$, and $\langle v_9, v_{10}\rangle$. Among them, only $\langle v_9, v_{10}\rangle$ has an ending point that ranks higher than the starting point. Therefore, HoD moves from v_9 to v_{10}. v_{10} has outgoing edges to three unvisited nodes, i.e., v_3, v_5, and v_8. Nevertheless, all of those nodes rank lower than v_{10}, and hence, they are ignored. As none of the remaining nodes can be reached without violating the constraints on node ranks, the first traversal of HoD ends. Based on the edges visited, HoD calculates $dist(v_1, v_9) = 1$ and $dist(v_1, v_{10}) = 4$.

The second traversal of HoD examines the nodes *not* in the core graph in descending order of their ranks, i.e., it first examines v_7 and v_8 (whose ranks equal 3), followed by v_4, v_5, and v_6 (whose ranks equal 2), and finally v_2 and v_3 (whose ranks equal 1), ignoring v_1 (as it is the source node of the query). v_7 has two incoming edges, $\langle v_6, v_7\rangle$ and $\langle v_9, v_7\rangle$. Among v_6 and v_9, only v_9 has been visited by HoD. Therefore, HoD calculates $dist(v_1, v_7) = dist(v_1, v_9) + l(\langle v_9, v_7\rangle) = 3$. Similarly, after inspecting v_8's only incoming edge $\langle v_{10}, v_8\rangle$, HoD computes $dist(v_1, v_8) = dist(v_1, v_{10}) + l(\langle v_{10}, v_8\rangle) = 5$. The remaining nodes are processed in the same manner, resulting in

$$\begin{aligned}
dist(v_1, v_4) &= dist(v_1, v_8) + l(\langle v_8, v_4\rangle) &= 6\\
dist(v_1, v_5) &= dist(v_1, v_{10}) + l(\langle v_{10}, v_5\rangle) &= 5\\
dist(v_1, v_6) &= dist(v_1, v_9) + l(\langle v_9, v_6\rangle) &= 2\\
dist(v_1, v_2) &= dist(v_1, v_4) + l(\langle v_4, v_1\rangle) &= 7.
\end{aligned}$$

Observe that all the above distances computed from G^* are identical with those from the original graph in Figure 1a. □

The query algorithm of HoD has an interesting property: the first traversal of the algorithm always visits nodes in ascending order of their ranks (as it never follows an edge that connects a high-rank node to low-rank node), while the second phase always visits nodes in descending rank order. Intuitively, if we maintain two copies of the augmented graph, such that the first (resp. second) copy stores nodes in ascending (resp. descending) order of their ranks, then HoD can answer any SSD query with a linear scan of the two copies. This leads to high query efficiency as it avoids random disk accesses. In Section 4, we will elaborate how such two copies of the augmented graph can be constructed.

4. INDEX CONSTRUCTION

As discussed in Section 3.1, the preprocessing algorithm of HoD takes as input a copy G_0 of the graph G, and it iteratively reduces G_0 into smaller graphs $G_1, G_2, \ldots$, during which it creates shortcuts to augment G. More specifically, the $(i+1)$-th ($i \geq 0$) iteration of the algorithm has four steps:

1. Select a set R_i of nodes to be removed from G_i.
2. For each node $v \in R_i$, construct shortcuts in G_i to ensure that the removal of v does not change the distance between any two remaining nodes.
3. Remove the nodes in R_i from G_i to obtain a further reduced graph G_{i+1}. Store information about the removed nodes in the index file of HoD.
4. Pass the G_{i+1} to the $(i+2)$-th iteration as input.

In the following, we first elaborate Steps 2 and 3, and then clarify Step 1. After that, we will discuss the termination condition of the preprocessing algorithm, as well as its space and time complexities.

For ease of exposition, we represent each edge $e = \langle u, v\rangle$ as a triplet $\langle u, v, l(e)\rangle$ or $\langle v, u, -l(e)\rangle$, where $l(e)$ is the length of e. For example, the edge $\langle v_9, v_7\rangle$ in Figure 1a can be represented as either $\langle v_9, v_7, 2\rangle$ or $\langle v_7, v_9, -2\rangle$. That is, a negative length in the triplet indicates that the second node in the triplet is the starting point of the edge. In addition, we assume that the input graph G is stored on the disk as adjacency lists, such that (i) for any two nodes v_i and v_j, the adjacency list of v_i precedes that of v_j if $i < j$, and (ii) each edge $\langle v_i, v_j\rangle$ with length l is stored twice: once in the adjacency list of v_i (as a triplet $\langle v_i, v_j, l\rangle$), and another in the adjacency list of v_j (as a triplet $\langle v_j, v_i, -l\rangle$).

4.1 Node Removal and Shortcut Generation

Let v^* be a node to be removed from G_i. We define an *outgoing neighbor* of v^* as a node u to which v^* has an outgoing edge. Similarly, an *incoming neighbor* of v^* is a node w from which v^* has an incoming edge. We have the following observation:

OBSERVATION 1. For any two nodes v_j and v_k in G_i, the distance from v_j to v_k changes after v^* is removed, if and only if the shortest path from v_j to v_k contains a sub-path $\langle u, v^*, w\rangle$, such that u (resp. w) is an incoming (resp. outgoing) neighbor of v^*. □

By Observation 1, we can preserve the distance between any two nodes in G_i after removing v^*, as long as we ensure that the distance between any incoming neighbor and any outgoing neighbor of v^* remains unchanged. This can be achieved by connecting the incoming and outgoing neighbors of v^* with shortcuts, as demonstrated in Section 3.1. Towards this end, a straightforward approach is to generate a shortcut $\langle u, w\rangle$ for any incoming neighbor u and any outgoing neighbor w. The shortcuts thus generated, however, are often *redundant*. For example, consider the graph G_i in Figure 2a. Suppose that we are to remove v_2, which has an incoming neighbor v_1 and an outgoing neighbor v_3. If we construct a shortcut from v_1 to v_3, it is useless since (i) v_1 already has an outgoing edge to v_3, and (ii) the edge $\langle v_1, v_3\rangle$ is even shorter than the path from v_1 to v_3 via v_2. As another example, assume that v_4 in Figure 2a is also to be removed. v_4 has an incoming neighbor v_1 and an outgoing neighbor v_5, but the path $\langle v_1, v_4, v_5\rangle$ is no shorter than another path from v_1 to v_5, i.e., $\langle v_1, v_3, v_5\rangle$, which does not go through v_4. As a consequence, even if we remove v_4 from G_i, the distance from v_1 to v_5 is still retained, and hence, it is unnecessary to insert a shortcut from v_1 to v_5.

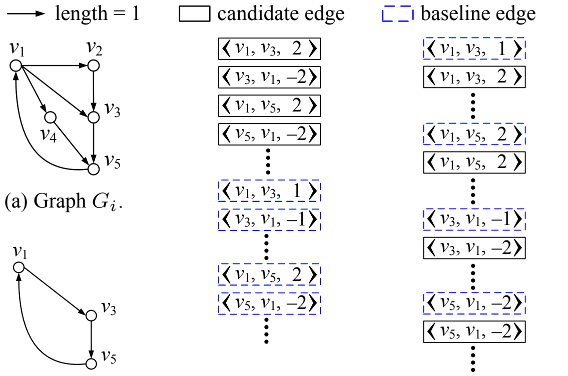
In general, for any incoming neighbor u and outgoing neighbor w of v^*, a shortcut from u to w is unnecessary if there is a path P from u to v, such that (i) P does not go through v^*, and (ii) P is no longer than $\langle u, v^*, w\rangle$. To check whether such a path P exists, one may apply Dijkstra's algorithm to traverse G_i from u (or w), ignoring v^* during the traversal. However, when G_i does not fit in main memory (as is often the case in the pre-computation process of HoD), this approach incurs significant overhead, due to the inefficiency of Dijkstra's algorithm for disk-resident graphs (as discussed in Section 1). To address this issue, we adopt a heuristic approach that is not as effective (in avoiding redundant shortcuts) but much more efficient. Specifically, for each v^* in the node set R_i to be removed from G_i, we generate a *candidate edge* $e_c = \langle u, w\rangle$ from each incoming neighbor u of v^* to each outgoing neighbor w of v^*, setting the length of the shortcut to $l(\langle u, v^*, w\rangle)$. For any such candidate edge e_c, we insert it into a temporary file T as two triplets: $\langle u, w, l(e_c)\rangle$ and $\langle w, u, -l(e_c)\rangle$.

In addition to the candidate edges, we also insert two additional groups of edges (referred to as *baseline edges*) into the temporary file T as triplets. The first group consists of any edge e in G_i connecting two nodes not in R_i, i.e., the two endpoints of e are not to be removed. The second group is generated as follows: for each node v not in R_i, we select v's certain incoming neighbor u' and outgoing neighbor w', and we construct a baseline edge $\langle u', w'\rangle$, setting its length to $l(\langle u', v, w'\rangle)$.

The purpose of inserting a baseline edge e into the temporary file T is to help eliminate any redundant candidate edge that (i) shares the same endpoints with e but (ii) is not shorter than e. Towards this end, once all baseline edges are added into T, we sort the triplets in T using a standard algorithm for external sort, such that a triplet $t_1 = \langle v_a, v_b, l_1\rangle$ precedes another triplet $t_2 = \langle v_\alpha, v_\beta, l_2\rangle$, if any of the following conditions hold:

1. $a < \alpha$, or $a = \alpha$ but $b < \beta$.
2. $a = \alpha$, $b = \beta$, and $l_1 > 0 > l_2$. That is, any outgoing edge of a node precedes its incoming edges.
3. $a = \alpha$, $b = \beta$, $l_1 \cdot l_2 > 0$ (i.e., t_1 and t_2 are both incoming edges or both outgoing edges), and $|l_1| < |l_2|$. That is, t_1 and t_2 share the same starting and ending points, but t_1 is shorter than t_2.
4. $a = \alpha$, $b = \beta$, $l_1 \cdot l_2 > 0$, $|l_1| = |l_2|$, and t_1 is a baseline edge while t_2 is a candidate edge.

Once T is sorted, the outgoing (resp. incoming) edges with the same endpoints are grouped together, and the first edge in each group should have the smallest length within the group. If the first edge e in a group is a candidate edge, then we retain e as it is shorter than any other baseline or candidate edges that we have generated. On the other hand, if e is a baseline edge, then the distance between



(a) Graph G_i.

(b) v_2, v_4 removed.

(c) Before sorting.

(d) After sorting.

Figure 2: Node removal and shortcut generation.

the endpoints of e must not be affected by the removal of any nodes in R_i. In that case, all candidate edges in the group can be omitted. With one linear scan of the sorted T and the adjacency lists of G_i, we can remove the information of any node in R_i, and merge the retained candidate edges into the adjacency lists of the remaining nodes. We illustrate the above algorithm with an example.

EXAMPLE 3. Suppose that, given the graph G_i in Figure 2a, we are to remove a node set $R_i = \{v_2, v_4\}$ from G_i. v_2 has only one incoming neighbor v_1 and one outgoing neighbor v_3, and $l(\langle v_1, v_2, v_3\rangle) = 2$. Accordingly, HoD generates a candidate edge $\langle v_1, v_3\rangle$ by inserting into the temporary file T two triplets, $\langle v_1, v_3, 2\rangle$ and $\langle v_3, v_1, -2\rangle$. Similarly, for v_4, HoD creates a candidate edge $\langle v_1, v_5\rangle$, which is represented as two triplets in T: $\langle v_1, v_5, 2\rangle$ and $\langle v_5, v_1, -2\rangle$.

Meanwhile, the edge $\langle v_1, v_3\rangle$ in G_i is selected as a baseline edge and is inserted into T, since neither v_1 nor v_3 is in R_i. In addition, HoD also generates a baseline edge $\langle v_1, v_5\rangle$ from the neighbors of v_3. This is because that (i) v_1, v_3, v_5 are not in R_i, (ii) v_1 is an incoming neighbor of v_3, and (iii) v_5 is an outgoing neighbor of v_3. Figure 2c illustrates the temporary file T after all candidate and baseline edges are inserted, with some triplets omitted for simplicity. Figure 2d shows the file T after it is sorted. The baseline edge $\langle v_1, v_3, 1\rangle$ precedes the candidate edge $\langle v_1, v_3, 2\rangle$, which indicates that we do not need to add a shortcut from v_1 to v_3. Similarly, the baseline edge $\langle v_3, v_1, -1\rangle$ precedes the candidate edge $\langle v_3, v_1, -2\rangle$, in which case the latter is omitted. Overall, each of the candidate edges in T is preceded by a baseline edge, and hence, no shortcut will be created. Consequently, HoD removes from G_i the adjacency lists of v_2 and v_4, as well as all edges to and from v_2 and v_4 in any other adjacency lists. This results in the reduced graph illustrated in Figure 2b. □

In summary, HoD decides whether a candidate edge e should be retained, by comparing it with all edges in G_i as well as some two-hop paths in G_i. This heuristic approach may retain unnecessary candidate edges, but it does not affect the correctness of SSD queries. To understand this, recall that each candidate edge $e = \langle u, w\rangle$ has the same length with a certain path $\langle u, v^*, w\rangle$ that *exists* in G_i, where v^* is the node whose removal leads to the creation of e. In other words, the length of e is at least larger than or equal to the distance from u to w. Adding such an edge into G_i would not decrease the distance between any two nodes in G_i, and hence, retaining e does not change the results of any SSD queries.

The above discussions assume that HoD has selected a set R_i of nodes to be removed from G_i, and has decided which baseline edges are to be generated from the neighbors of the nodes not in R_i. We will clarify these two issues in Section 4.2 and 4.3.

4.2 Selecting Nodes for Removal

Consider any node v in G_i. Intuitively, if the removal of v requires us to insert a large number of shortcuts into G_i, then v may lie on the shortest paths between many pair of nodes, in which case v should be considered important. Let B_{in} and B_{out} be the set of incoming and outgoing neighbors of v, respectively. The maximum number of shortcuts induced by v's removal is:

$$s(v) = |B_{in}| \cdot |B_{out} \setminus B_{in}| + |B_{out}| \cdot |B_{in} \setminus B_{out}|. \quad (1)$$

We refer to $s(v)$ as the *score* of v in G_i, and we consider v *unimportant* if $s(v)$ is no more than the median score in G_i. (For practical efficiency, we use an approximated value of the median score computed from a sample set of the nodes.)

Ideally, we would like to remove all unimportant nodes from G_i, but this is not always feasible. To explain, consider that we are given the reduced graph G_1 in Figure 1b, and we aim to eliminate both v_6 and v_7. v_6 has only one incoming neighbor v_9 and one outgoing neighbor v_7, and hence, HoD creates one candidate edge $\langle v_9, v_7\rangle$, setting its length to 2 (i.e., the length of the path $\langle v_9, v_6, v_7\rangle$). Similarly, for v_7, HoD generates a candidate edge $\langle v_6, v_{10}\rangle$. These two candidate edges are intended to preserve the distance between any two nodes in G_i after v_6 and v_7 are removed. However, none of the two candidate edges is valid if both v_6 and v_7 are eliminated. In particular, $\langle v_9, v_7\rangle$ points from v_9 to v_7, i.e., it connects v_9 to a node that no longer exists. To avoid the above error, whenever HoD chooses to delete a node v from G_i, it will retain all neighbors of v in G_i, even if some neighbor might be unimportant. For example, in Figure 1b, if HoD decides to remove v_6, then it will prevent v_7 from being removed at the same time, and vice versa.

4.3 Generation of Baseline Edges

As mentioned in Section 4.1, a baseline edge generated by the preprocessing algorithm of HoD is either (i) an edge in G_i whose endpoints are not to be removed, or (ii) an artificial edge $\langle u, w\rangle$ that corresponds to certain two-hop path $\langle u, v, w\rangle$ in G_i, such that none of u, v, w is to be removed. The construction of baseline edges from two-hop paths is worth discussing. First, given that there exists an enormous number of two-hop paths in G_i, it is prohibitive to convert each two-hop path into a baseline edge. Therefore, we only select a subset of the two-hop paths in G_i for baseline edge generation. In particular, the total number of two-hop paths selected is set to $c \cdot \sum_{v \in R_i} s(v)$, where c is a small constant, $s(v)$ is as defined in Equation 1, and $\sum_{v \in R_i} s(v)$ is the total number of candidate edges induced by the removal of nodes in R_i. In other words, we require that the number of baseline edges generated from two-hop paths is at most c times the number of candidate edges. In our implementation of HoD, we set $c = 5$.

Those $c \cdot \sum_{v \in R_i} s(v)$ baseline edges are generated as follows. First, we randomly choose an edge in G_i, and arbitrarily select an endpoint v of the edge that is not in R_i. (Note that such an endpoint always exists.) After that, from the incoming (resp. outgoing) neighbors of v, we randomly select a node u (resp. w), and we generate a baseline edge $\langle u, w\rangle$, setting its length to $l(\langle u, v, w\rangle)$. As such, if a node v is adjacent to a large number of edges, then it has a high chance of being selected to produce baseline edges. This is intuitive since such a node v tends to lie on the shortest paths between many pairs of nodes, and hence, the baseline edges generated from v may be more effective in eliminating redundant shortcuts.

4.4 Termination Condition

As mentioned in Section 3, HoD requires that the core graph G_c fits in the main memory, where G_c is the reduced graph obtained in the last iteration of HoD's preprocessing algorithm. Accordingly, we do not allow the pre-computation procedure of HoD to terminate before the reduced graph G_i has a size no more than M. In addition, even after G_i fits in the main memory, we will still continue the preprocessing procedure, until the size of G_i is reduced by less than 5% in an iteration of the processing algorithm. This is intended to reduce the size of the core graph G_c to improve query efficiency, as will be explained in Section 5.

4.5 Index File Organization

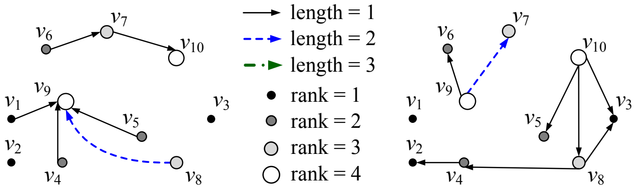
Once the preprocessing procedure completes, the core graph G_c is written to the disk in the form of adjacency lists. Meanwhile, the adjacency list of each node not in G_c is separated into two parts that are stored in two different files, F_f and F_b. These two files are created at the beginning of HoD's preprocessing algorithm, and they are initially empty. Whenever a node v is removed from the reduced graph G_i, we inspect the adjacency list of v in G_i, and we append all of v's outgoing (resp. incoming) edges to F_f (resp. F_b). Upon termination of the preprocessing procedure, we reverse the order of nodes in F_b, but retain the order of nodes in F_f. We refer to the graph represented by F_f as the *forward graph*, denoted as G_f. Meanwhile, we refer to the graph represented by F_b as the *backward graph*, denoted as G_b. When combined, G_c, G_f, and G_b form the augmented graph that is used by HoD for query processing. For example, for the augmented graph in Figure 1a, its core graph is as illustrated in Figure 1e, while its forward and backward graphs are as shown in Figure 3. For ease of exposition, we will abuse notation and use G_f (resp. G_b) to refer to both G_f (resp. G_b) and its underlying file structure F_f (resp. F_b).

G_f and G_b have two interesting properties. First, all nodes in G_f (resp. G_b) are stored in ascending (resp. descending) order of their ranks. To explain this, recall that any node v removed in the i-th iteration of the preprocessing algorithm has a rank $r(v) = i$. Consequently, if a node u is stored in G_f before another node w, then $r(u) \leq r(w)$. As for G_b, since we reverse the order of all edges in G_b upon termination of the preprocessing produce, we have $r(u) \geq r(w)$ for any node u that precedes another node w in G_b. Second, for any node v, its edges in G_f and G_b only connect it to the nodes whose ranks are strictly higher than v. This is because, by the time v is removed from the reduced graph, all nodes that rank lower than v must have been eliminated from the reduced graph, and hence, any edge in v's adjacency list only links v to the nodes whose rank is at least $r(v)$. Meanwhile, any neighbor u of v in the reduced graph should have a rank higher than $r(v)$. Otherwise, we have $r(u) = r(v)$, which, by the definition of node ranks, indicates that u and v are removed in the same iteration of the preprocessing algorithm. This is impossible as the pre-computation procedure of HoD never eliminates two adjacent nodes in the same iteration, as explained in Section 4.1. In Section 5, we will show how HoD exploits the above two properties of G_f and G_b to efficiently process SSD queries.

4.6 Cost Analysis

The preprocessing algorithm of HoD requires $O(n)$ main memory, where n is the number of nodes in G. This is because (i) when removing a node v from the reduced graph, HoD needs to record the neighbors of v and exclude them from the node removal process, and (ii) v may have $O(n)$ neighbors. Other parts of the preprocessing algorithm do not have a significant memory requirement.

The major I/O and CPU costs of the preprocessing algorithm are incurred by sorting the edge triplets in each iteration. In the worst case when the input graph G is a complete graph, there are $O(n^2)$ triplets generated in each iteration, leading to a prohibitive I/O and CPU overhead. Fortunately, real-world graphs are seldom complete graphs, and they tend to contain a large number of nodes with small degrees. In that case, each iteration of HoD's preprocessing procedure would only generates a moderate number of edge triplets, leading to a relatively small overhead.



(a) G_f.

(b) G_b.

Figure 3: Forward graph G_f and backward graph G_b.

Lastly, the space consumption of HoD's index is $O(n^2)$ since, in the worst case, HoD may construct a shortcut from each node v to every node that ranks higher than v. This space complexity is unfavorable, but it is comparable to the space complexity of VC-Index [8]. In addition, as shown in our experiments, the space requirement of HoD in practice is significantly smaller than the worst-case bound.

5. ALGORITHM FOR SSD QUERIES

Given an SSD query from a node s that is not in the core graph G_c, HoD processes the query in three steps:

1. **Forward Search:** HoD traverses the forward graph G_f starting from s, and for each node v visited, computes the distance from s to v in G_f.

2. **Core Search:** HoD reads the core graph G_c into the main memory, and continues the forward search by following the outgoing edges of each node in G_c.

3. **Backward Search:** HoD linearly scans the backward graph G_b to derive the exact distance from s to any node not in G_c.

On the other hand, if s is in G_c, then HoD would answer the query with a core search followed by a backward search, skipping the forward search. In what follows, we will present the details of three searches performed by HoD. For convenience, we define an *index* value $\theta(v)$ for each node *not* in the core graph G_c, such that $\theta(v) = i$ only if the i-th adjacency list in G_f belongs to v. By the way G_f is constructed (see Section 4.5), for any two nodes u and v with $\theta(u) < \theta(v)$, the rank of u is no larger than the rank of v.

5.1 Forward Search

The forward search of HoD maintains a hash table H_f and a min-heap Q_f. In particular, H_f maps each node v to a key $\kappa_f(v)$, which equals the length of the shortest path from s to v that is seen so far. Initially, $\kappa_f(s) = 0$, and $\kappa_f(v) = +\infty$ for any node $v \neq s$. On the other hand, each entry in Q_f corresponds to a node v, and the key of the entry equals $\theta(v)$, i.e., the index of v. As will become evident shortly, Q_f ensures that the forward search visits nodes in ascending order of their indices, and hence, it scans the file structure of G_f only once, without the need to backtrack to any disk block that it has visited before.

HoD starts the forward search by inspecting each edge $e = \langle s, v \rangle$ adjacent to s in G_f, and then inserting v into H_f with a key $\kappa(v) = l(e)$. (Note that G_f contains only outgoing edges.) In addition, HoD also inserts v into Q_f. After that, HoD iteratively removes the top entry u in Q_f, and processes u as follows: for each edge $e = \langle u, v \rangle$ adjacent to u, if $\kappa_f(v) = +\infty$ in the hash table H_f, HoD sets $\kappa_f(v) = \kappa_f(u) + l(e)$ and inserts v into Q_f; otherwise,

Table 1: Datasets.

Name	|V|	|E|	Weighted?	Directed?	Size
USRN	24.9M	28.9M	yes	no	0.86GB
FB	58.8M	92.2M	no	no	2.42GB
u-BTC	16.3M	95.7M	no	no	1.79GB
u-UKWeb	6.9M	56.5M	yes	no	1.02GB
BTC	16.3M	99.4M	no	yes	1.98GB
Meme	53.6M	117.9M	no	yes	3.17GB
UKWeb	104M	3708M	no	yes	61.8GB

HoD sets $\kappa_f(v) = \min\{\kappa_f(v), \kappa_f(u)+l(e)\}$. When Q_f becomes empty, HoD terminates the forward search, and retains the hash table H_f for the second step of the algorithm, i.e., the core search.

5.2 Core Search

The core search of HoD is a continuation of the forward search, and it inherits the hash table H_f created during the forward search. In addition to H_f, HoD creates a min-heap Q_c, such that Q_c stores entries of the form $\langle v, \kappa_f(v)\rangle$, where v is a node and $\kappa_f(v)$ is the key of v in H_f. For any node u with $\kappa_f(u) \neq +\infty$ (i.e., u is visited by the forward search), HoD inserts u into Q_c.

Given H_f and Q_c, HoD performs the core search in iterations. In each iteration, HoD extracts the top entry v from Q_c, and examines each outgoing edge e of v. For every such edge, HoD inspects its ending point w, and sets $\kappa_f(w) = \min\{\kappa_f(w), \kappa_f(v) + l(e)\}$. Then, HoD adds w into Q_c if w is currently not in Q_c. This iterative procedure is repeated until Q_c becomes empty. After that, the hash table H_f is sent to the last step (i.e., the backward search) for further processing.

5.3 Backward Search

Given the hash table H_f obtained from the core search, the reversed search of HoD is performed as a sequential scan of the backward graph G_b, which stores nodes in descending order of their index values. For each node v visited during the sequential scan, HoD checks each edge $e = \langle u, v\rangle$ adjacent to v. (Note that G_b contains only incoming edges). If $\kappa_f(u) \neq +\infty$ and $\kappa_f(u)+l(e) < \kappa_f(v)$, then HoD sets $\kappa_f(v) = \kappa_f(u) + l(e)$. Once all nodes in G_b are scanned, HoD terminates the backward search and, for each node v, returns $\kappa_f(v)$ as the distance from s to v.

One interesting fact about the backward search is that it does not require a heap to decide the order in which the nodes are visited. This leads to a much smaller CPU cost compared with Dijkstra's algorithm, as it avoids all expensive heap operations.

5.4 Correctness and Complexities

Compared with Dijkstra's algorithm, the main difference of HoD's query algorithm is that it visits nodes in a pre-defined order based on their ranks. The correctness of this approach is ensured by the shortcuts constructed by the preprocessing algorithm of HoD. In particular, for any two nodes s and t in G, it can be proved that the augmented graph G^* always contains a path P from s to t, such that (i) P's length equals the distance from s to t in G, and (ii) P can be identified by HoD with a forward search from s, followed by a core search and a backward search. More formally, we have the following theorem.

THEOREM 1. *Given a source node s, the SSD query algorithm of HoD returns $dist(s, v)$ for each node $v \in G$.*

Interested readers are referred to our technical report [27] for the proof of Theorem 1.

The query algorithm of HoD requires $O(n + m_c)$ main memory, where m_c is the size of the core graph. This is due to the fact that (i) the forward, core, and back searches of HoD all maintain a hash table that takes $O(n)$ space, and (ii) the core search requires reading the core graph G_c into the memory. The time complexity of the algorithm is $O(n \log n + m')$, where m' is the total number of edges in G_c, G_f, and G_b. The reason is that, when processing an SSD query, HoD may need to scan G_f, G_c, and G_b once, and it may need to put $O(n)$ nodes into a min-heap. Finally, the I/O costs of the algorithm is $O((n + m')/B)$, since it requires at most one scan of G_f, G_c, and G_b.

6. EXTENSION FOR SSSP QUERIES

Given a source node s, an SSSP query differs from an SSD query only in that it asks for not only (i) the distance from s to any other node v, but also (ii) the predecessor of v, i.e., the node that immediately precedes v on the shortest path from s to v. To extend HoD for SSSP queries, we associate each edge $\langle u, w\rangle$ in the augment graph G^* with a node v, such that v immediately precedes w on the shortest path from u to w in G. For example, given the augmented graph in Figure 1e, we would associate the edge $\langle v_9, v_7\rangle$ with v_6, since (i) the shortest path from v_9 to v_7 in G is $\langle v_9, v_6, v_7\rangle$, and (ii) v_6 immediately precedes v_7 in the path.

With the above extension, HoD processes any SSSP query from a node s using the algorithm for SSD query with one modification: Whenever HoD traverses an edge $\langle u, w\rangle$ and finds that $dist(s, u) + l(\langle u, w\rangle) < dist(s, w)$, HoD would not only update $dist(s, w)$ but also record the node associated with $\langle u, w\rangle$. That is, for each node w visited, HoD keeps track of the predecessor of w in the shortest path from s to w that have been seen so far. As such, when the SSD query algorithm terminates, HoD can immediately return $dist(s, w)$ as well as the predecessor of w.

Finally, we clarify how the preprocessing algorithm of HoD can be extended to derive the node associated with each edge. First, for each edge e in the original graph, HoD associates e with the starting point of e. After that, whenever HoD generates a candidate edge $\langle u, w\rangle$ during the removal of a node v, HoD would associate $\langle u, w\rangle$ with the node that is associated with the edge $\langle v, w\rangle$. For example, in Figure 1c, when HoD removes v_7 and creates a candidate edge $\langle v_9, v_{10}\rangle$, it associates the edge with v_7, which is the node associated with $\langle v_7, v_{10}\rangle$.

7. EXPERIMENTS

This section experimentally compares HoD with three methods: (i) VC-Index [8], the state-of-the-art approach for SSD and SSSP queries on disk-resident graphs; (ii) EM-BFS [6], an I/O efficient method for breadth first search; and (iii) EM-Dijk [18], an I/O efficient version of Dijkstra's algorithm. We include EM-BFS since, on unweighted graphs, any SSD query can be answered using breadth first search, which is generally more efficient than Dijkstra's algorithm. We obtain the C++ implementations of VC-Index, EM-BFS, and EM-Dijk from their inventors, and we implement HoD with C++. As the implementation of VC-Index only supports SSD queries, we will focus on SSD queries instead of SSSP queries. All of our experiments are conducted on a machine with a 2.4GHz CPU and 32GB memory.

7.1 Datasets

We use five real graph datasets as follows: (i) USRN [1], which represents the road network in the US; (ii) FB [14], a subgraph of the Facebook friendship graph; (iii) BTC [2], a semantic graph; (iv) Meme [16] and UKWeb [3], which are two web graphs. Among them, only USRN and FB are undirected. Since VC-Index, EM-BFS, and EM-Dijk are all designed for undirected graphs only, we

Table 2: Preprocessing time (in minutes).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	4.0	22.4	34.4	105.5
VC-Index	20.3	281.8	78.1	768.2

Table 3: Space Consumption (in GB).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	2.5	5.1	3.8	3.3
VC-Index	4.3	8.3	1.2	14.0

Table 4: Average running time for SSD queries (in seconds).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	1.8	3.2	1.6	1.4
VC-Index	27.2	94.9	10.1	70.0
EM-BFS	–	465.3	395.4	–
EM-Dijk	430.7	1597.4	844.1	553.8

are indeed of more undirected datasets for experiments. For this purpose, we transform BTC and UKWeb into undirected graphs, using the same approach as in previous work (see [8] for details). After that, for each undirected (resp. directed) graph G, we compute its largest connected component (resp. weakly connected component) C, and we use C for experiments. Table 1 illustrates the details of the largest component obtained from each graph. In particular, u-BTC and u-UKWeb are obtained from the undirected versions of BTC and UKWeb, respectively.

Remark. The previous experimental study on VC-Index [8] uses USRN, u-BTC, and u-UKWeb instead of their largest connected components (CC) for experiments. We do not follow this approach as it leads to less meaningful results. To explain, consider a massive undirected graph G where each CC is small enough to fit in the main-memory. On such a graph, even if a disk-based method can efficiently answer SSD queries, it does not necessarily mean that it is more scalable than a main-memory algorithm. In particular, one can easily answer an SSD query from any node s in G, by first reading into memory the CC that contains s, and then running a main-memory SSD algorithm on the CC. In general, given any graph G, one can use an I/O efficient algorithm [21] to pre-compute the (weakly) connected components in G, and then handle SSD queries on each component separately.

7.2 Results on Undirected Graphs

In the first sets of experiments, we evaluate the performance of each method on four undirected graphs: USRN, FB, u-BTC, and u-UKWeb. For HoD, EM-BFS, and EM-Dijk, we limit the amount of memory available to them to 1GB, which is smaller than the sizes of all datasets. For VC-Index, we test it with 2GB memory as it cannot handle any of our datasets under a smaller memory size.

Table 2 shows the preprocessing time of HoD and VC-Index on the four graphs. (EM-BFS and EM-Dijk are omitted as they do not require any pre-computation.) In all cases, HoD incurs a significantly smaller overhead than VC-Index does. In particular, on FB, the preprocessing time of HoD is more than ten times smaller than that of VC-Index. Table 3 compares the space consumptions of HoD and VC-Index. Except for the case of u-BTC, the space required by VC-Index is consistently larger than that by HoD.

To evaluate the query efficiency of each method, we generate 100 SSD queries for each dataset, such that the source node of each query is randomly selected. Table 4 shows the average running time of each approach in answering an SSD query. The query time of HoD is at least an order of magnitude smaller than that of VC-Index. Meanwhile, VC-Index always outperforms EM-BFS, which is consistent with the experimental results reported in previous work [8]. We omit EM-BFS on USRN and u-UKWeb, since those two graphs are weighted, for which EM-BFS cannot be used to answer SSD queries. Finally, EM-Dijk incurs a larger query overhead than all other methods.

Table 5: Estimated time for *closeness* computation (in hours).

Method	USRN	FB	u-BTC	u-UKWeb
HoD	0.9	2.0	1.3	2.4
VC-Index	13.2	51.8	6.1	43.1
EM-BFS	–	231.1	182.2	–
EM-Dijk	203.2	793.3	389.0	240.0

Table 6: Performance of HoD on directed graphs.

Dataset	Preprocessing	Index Size	SSD Query Time
BTC	11.4 minutes	2.1 GB	2.6 sec
Meme	1.2 minutes	2.3 GB	1.8 sec
UKWeb	9.2 hours	72.6 GB	53.7 sec

In the next experiment, we demonstrate an application of HoD for efficient graph analysis. In particular, we consider the task of approximating the *closeness* for all nodes in a graph G, using the algorithm by Eppstein and Wang [11]. The algorithm requires executing $k = \ln n/\epsilon^2$ SSD queries from randomly selected source nodes, where n is the number of nodes in G and ϵ is a parameter that controls the approximation error. Following previous work [8], we set $\epsilon = 0.1$.

Table 5 shows an estimation of the time required by each method to complete the approximation task. Specifically, we estimate the total processing time of each method as (i) its query time in Table 4 multiplied by k, plus (ii) its preprocessing time (if any). Observe that both EM-BFS and EM-Dijk incur prohibitive overheads – they require more than a week to finish the approximation task. In contrast, HoD takes at most 2.4 hours to complete the task, despite that it pays an initial cost for pre-computation. Meanwhile, VC-Index is significantly outperformed by HoD, and it needs around two days to accomplish the task on FB and u-UKWeb.

7.3 Results on Directed Graphs

Our last experiments focus on the three directed graphs: BTC, Meme, and UKWeb. We run HoD on BTC and Meme with 1GB memory, and on UKWeb with 16GB memory, as the enormous size of UKWeb leads to a higher memory requirement. Table 6 shows the preprocessing and space overheads of HoD, as well as its average query time in answering 100 randomly generated SSD queries on each dataset. (We omit VC-Index, EM-BFS, and EM-Dijk as they do not support directed graphs.) On BTC and Meme, HoD only incurs small pre-computation costs and moderate space consumptions. On UKWeb, the preprocessing, space, and query overheads of HoD are considerably higher, but are still reasonable given that UKWeb contains 30 times more edges than BTC and Meme do. To our knowledge, this is the first result in the literature that demonstrates practical support for SSD queries on a billion-edge graph.

8. RELATED WORK

As mentioned in Section 1, the existing techniques for I/O-efficient SSD and SSSP queries include VC-Index [8] and a few methods that adopt Dijkstra's algorithm [15, 17–20]. All of those techniques are exclusively designed for undirected graphs, and they incur significant query overheads, as is shown in our experiments. In contrast, HoD supports both directed and undirected graphs, and it offers high query efficiency along with small costs of pre-computation and space.

Other than the aforementioned work, there exists large body of literature on in-memory algorithms for shortest path and distance queries (see [9, 23, 25, 26] for surveys). The majority of those algorithms focus on two types of queries: (i) *point-to-point shortest path (PPSP)* queries, which ask for the shortest path from one node to another, and (ii) *point-to-point distance (PPD)* queries, which ask for the length of the shortest path between two given nodes. These two types of queries are closely related to SSD and SSSP queries, in the sense that any SSD (resp. SSSP) query can be answered using the results of n PPD (resp. PPSP) queries, where n is the number of nodes in the graph. Therefore, it is possible to adopt a solution for PPD (resp. PPSP) queries to handle SSD (resp. SSSP) queries. Such an adoption, however, incurs significant overheads, especially when n is large. For example, the state-of-the-art solution [4] for PPD queries requires 266ns to answer a PPD query on the USRN dataset in Section 7. (Note: the solution is not I/O efficient and it requires 25.4GB memory to handle USRN.) If we use this solution to answer an SSD query on USRN, then we need to execute 24.5 million PPD queries, which takes roughly $266\text{ns} \times 24.5 \times 10^6 = 6.52\text{s}$. In contrast, HoD requires only 1.8s to process an SSD query on USRN, using only 1GB memory.

Furthermore, almost all existing solutions for PPD and PPSP queries require that the dataset fits in the main memory during pre-computation and/or query processing. This renders them inapplicable for the massive disk-resident graphs considered in this paper. The only exception that we are aware of is a concurrent work by Fu et al. [12], who propose an I/O-efficient method called *IS-Label*. HoD and IS-Label's preprocessing algorithms are similar in spirit, but their index structures and query algorithms are drastically different, as they are designed for different types of queries. In particular, IS-Label focuses on PPD and PPSP queries, and does not efficiently support SSD or SSSP queries.

Finally, we note that previous work [9, 13, 22] has exploited the idea of augmenting graphs with shortcuts to accelerate PPD and PPSP queries. Our adoption of shortcuts is inspired by previous work [9, 13, 22], but it is rather non-trivial due to the facts that (i) we address I/O efficiency under memory-constrained environments, while previous work [9, 13, 22] focuses on main memory algorithms; (ii) we tackle SSD and SSSP queries instead of PPD and PPSP queries; (iii) we focus on general graphs, while pervious work [9, 13, 22] considers only road networks (where each node is degree-bounded).

9. CONCLUSIONS

This paper presents HoD, a practically efficient index structure for distance queries on massive disk-resident graphs. In particular, HoD supports both directed and undirected graphs, and it efficiently handles *single-source shortest path (SSSP) queries* and *single-source distance (SSD) queries* under memory-constrained environments. This contrasts the existing methods, which either (i) require that the dataset fits in the main memory during pre-computation and/or query processing, or (ii) support only undirected graphs. With extensive experiments on a variety of real-world graphs, we demonstrate that HoD significantly outperforms the state of the art in terms of query efficiency, space consumption, and pre-computation costs. For future work, we plan to investigate how HoD can be extended to (i) support point-to-point shortest path and distance queries and (ii) handle dynamic graphs that change with time.

Acknowledgement

This work was supported by the Nanyang Technological University under SUG Grant M58020016.

10. REFERENCES

[1] `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[2] `http://vmlion25.deri.ie/`.

[3] `http://barcelona.research.yahoo.net/webspam/datasets/uk2007/links/`.

[4] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.

[5] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999.

[6] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory bfs algorithms. In *SODA*, pages 601–610, 2006.

[7] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, pages 124–137, 2007.

[8] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD*, pages 457–468, 2012.

[9] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139, 2009.

[10] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[11] D. Eppstein and J. Wang. Fast approximation of centrality. *J. Graph Algorithms Appl.*, 8:39–45, 2004.

[12] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying on large graphs. *PVLDB*.

[13] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[14] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *INFOCOM*, pages 2498–2506, 2010.

[15] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP*, pages 169–176, 1996.

[16] J. Leskovec, L. Backstrom, J. M. Kleinberg, and J. M. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD*, pages 497–506, 2009.

[17] U. Meyer. Via detours to i/o-efficient shortest paths. In *Efficient Algorithms*, pages 219–232, 2009.

[18] U. Meyer and V. Osipov. Design and implementation of a practical i/o-efficient shortest paths algorithm. In *ALENEX*, pages 85–96, 2009.

[19] U. Meyer and N. Zeh. I/o-efficient undirected shortest paths. In *ESA*, pages 434–445, 2003.

[20] U. Meyer and N. Zeh. I/o-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths. *ACM Transactions on Algorithms*, 8(3):22, 2012.

[21] K. Munagala and A. G. Ranade. I/o-complexity of graph algorithms. In *SODA*, pages 687–694, 1999.

[22] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579, 2005.

[23] C. Sommer. Shortest-path queries in static networks, 2012. `http://www.shortestpaths.org/spq-survey.pdf`.

[24] F. W. Takes and W. A. Kosters. Determining the diameter of small world networks. In *CIKM*, pages 1191–1196, 2011.

[25] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*, pages 421–432, 2011.

[26] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: Towards bridging theory and practice. In *SIGMOD*, 2013.

[27] A. D. Zhu, X. Xiao, S. Wang, and W. Lin. Efficient single-source shortest path and distance queries on large graphs. *CoRR*, abs/1306.1153, 2013.