Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach

James Cheng
Nanyang Technological University, Singapore
j.cheng@acm.org

Yiping Ke
Institute of High Performance Computing, Singapore
keyp@ihpc.a-star.edu.sg

Shumo Chu
Nanyang Technological University, Singapore
shumo.chu@acm.org

Carter Cheng
Nanyang Technological University, Singapore
carter.cheng.w@acm.org

ABSTRACT

We propose a novel disk-based index for processing *single-source shortest path or distance queries*. The index is useful in a wide range of important applications (e.g., network analysis, routing planning, etc.). Our index is a tree-structured index constructed based on the concept of *vertex cover*. We propose an I/O-efficient algorithm to construct the index when the input graph is too large to fit in main memory. We give detailed analysis of I/O and CPU complexity for both index construction and query processing, and verify the efficiency of our index for query processing in massive real-world graphs.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; E.1 [**Data structures**]: Graphs and Networks, Trees

General Terms

Algorithms, Performance, Experimentation

Keywords

Single-source shortest paths, distance queries, disk-based shortest-path index, vertex cover, distance graph

1. INTRODUCTION

We study the problem of indexing a graph for answering *single-source shortest path or distance* (**SSSP** or **SSdist**) queries. Given a graph G and a source vertex s, an SSSP or SSdist query finds the shortest path or distance from s to every vertex in G. We consider both weighted and un-weighted graphs. In an un-weighted graph, SSSP from s is simply *breadth-first search* (**BFS**) from s.

SSSP and BFS are two classic algorithmic problems whose solutions have numerous applications. However, in situations where *the input graphs are large* which is becoming more common [19, 7] and where *repeated queries are made over the same data*, classic solutions become impractical. To process large graphs that cannot fit in memory, many *external-memory* (**EM**) algorithms have been proposed for SSSP [17, 21, 22, 19, 20] and BFS [24, 6, 18, 4, 5]. However, the EM algorithms, though having polynomial I/O complexity, are not efficient enough in many practical applications. As reported in our experiments, the EM BFS algorithm [18] takes up to 7,500 seconds to process just one single query.

We propose a novel disk-based index for processing SSSP and BFS queries. Most existing shortest-path or distance indexes focus on answering *source-to-target* queries [1], which have applications mainly in road networks. However, the index for SSSP and BFS also has many important applications such as urban planning, routing in telecommunication networks, crawling and analyzing the Web [25], finding the k-neighbors or k-hops of a vertex in a network/graph, etc. In addition, the index is useful for a large number of problems that need to process SSSP or BFS repeatedly with different source vertices. We list a few as follows.

- In network analysis, we may build an index and then efficiently compute important measures such as closeness [12], stress [26], betweenness [15], global efficiency [27], characteristic path length [8], integration and radiality [29], etc., which all require to compute SSSP or BFS repeatedly from all vertices or from selected vertices (for approximation).
- In estimating the diameter of a graph, a key step is to compute SSSP or BFS from a chosen subset of vertices [3], which is very costly for large graphs but can be efficiently processed with an index. The index can also be used for efficient estimation of the radius, girth, and circumference of a graph.
- For a number of flow problems, the index can be used repeatedly for any input source vertices (i.e., queries), instead of re-computing the flow from scratch each time. For example, Ford-Fulkerson's maximum flow algorithm uses BFS to find all paths from a source vertex [14], and the successive shortest paths algorithm for min-cost flow computes SSSP repeatedly from the source vertices [11].
- For several maximum matching algorithms [16, 23], we can use the index to speed up a key step which finds the BFS level of each vertex from the free vertices (i.e., source vertices).
- We may also apply the index in the approximation of the

Travel Salesman problem, and the Chinese Postman problem, the shortest common superstring problem [28], etc.

Our index, named as **VC-index**, applies the concept of *vertex cover* (**VC**) to construct a tree-structured index. The construction of the index is based on the observation that all vertices in a given graph are within 1 hop of the VC and this can be related to Dijkstra's observation [10] that the greedy strategy to grow the shortest paths is optimal via an inductive argument.

By employing the property of VC, we show that, once the distance from the source vertex s to the vertices in a VC $\mathbb{C}$ of G is known, the distance from s to each remaining vertex in G can be obtained at a small cost. However, it is inefficient to compute the distance from s to the vertices in $\mathbb{C}$ online. Moreover, it is also impractical to pre-compute all the distances offline and retrieve them online, because the storage size is prohibitively large. Our solution is to compress the distance information in G step by step until it becomes affordable to do online distance computation with limited memory and short response time. To this end, we formulate the concept of *distance graph*, which is smaller than G but fully retains the distance information between the vertices in $\mathbb{C}$. After obtaining the distance graph D_0 of G, we recursively construct a distance graph D_1 from D_0, and so on until we obtain a distance graph D_i that is small enough for in-memory distance computation. In this way, as long as s is in D_i, the distance from s to all the vertices in G can be obtained by unfolding the distance information in these distance graphs, which form a path in VC-index.

We summarize the main contributions of the paper as follows.

- We propose a novel disk-based index for processing SSSP and BFS queries. In addition to many traditional applications of SSSP and BFS, our index can also be applied to devise efficient solutions for a large number of important problems [12, 26, 15, 27, 8, 29, 3, 14, 11, 23, 16, 28], especially in the case when the input graph is disk resident.

- Our work is the first to apply VC to build a tree-structured index. We propose a streaming 2-approximation algorithm for computing the minimum VC in large graphs, which makes efficient application of VC for indexing possible.

- The existing (source-to-target) shortest path indexes for general graphs rely on in-memory construction [31, 30] (even though the constructed index may be disk-resident). However, in-memory index construction algorithms cannot scale to large graphs that cannot fit in memory. In our work, we design I/O-efficient algorithms to construct VC-index.

- Our experimental results show that our algorithm for index construction is efficient in large real graphs with more than 100 million vertices and over 1 billion edges, while other closely related indexes [31, 30] ran out of memory. The results also verify the high efficiency of query processing using VC-index, together with a demonstration on the benefit of VC-index in some important applications.

Organization. Section 2 defines the problem and the basic notations. Section 3 describes a simple VC-based index. Section 4 gives the overall framework. Sections 5 and 6 present the construction of VC-index and query processing. Section 7 extends VC-index to process other types of queries. Section 8 reports the experimental results. Section 9 discusses the update maintenance of VC-index. Sections 10 and 11 give the related work and the conclusion.

Table 1: Frequently-used notations

Notation	Description
$G = (V_G, E_G, \omega_G)$	A weighted, undirected simple graph
$\lvert G\rvert = (\lvert V_G\rvert + \lvert E_G\rvert)$	The size of G
$adj_G(v)$	The set of adjacent vertices of v in G
$deg_G(v)$	The degree of v in G
$\mathbb{C}$	A vertex cover of G
$len(p)$	The length of a path p
$SP_G(s, v)$	The shortest path from s to v in G
$dist_G(s, v)$	The distance from s to v in G
X	A node X in VC-index
$\mathbb{X}$	The VC at the node X
$D_X = (V_X{=}\mathbb{X}, E_X, \omega_X)$	The distance graph at the node X
M	The main memory size
B	The disk block size

2. PROBLEM DEFINITION

We first give the notations used throughout the paper.

Let $G = (V_G, E_G, \omega_G)$ be a weighted, undirected simple graph, where V_G is the set of vertices, E_G is the set of edges, and $\omega_G : E_G \rightarrow \mathbb{N}^*$ is a function that assigns a positive integer to each edge as the weight. If G is unweighted, then $\forall e \in E_G, \omega_G(e) = 1$. The size of G is defined as $|G| = (|V_G| + |E_G|)$.

We define the set of *adjacent* vertices (or *neighbors*) of a vertex v in G as $adj_G(v) = \{u : (u, v) \in E_G\}$, and the *degree* of v in G as $deg_G(v) = |adj_G(v)|$. For simplicity, the weight of an edge (u, v), $\omega_G((u, v))$, is written as $\omega_G(u, v)$.

We assume that a graph is stored in its adjacency list representation (whether in memory or on disk), where each vertex is assigned a unique vertex ID (or vertex label) and vertices are ordered in the ascending order of their vertex IDs.

Given a vertex set $S \subseteq V_G$, we say that an edge (u, v) in G is *covered* by S (or equivalently S *covers* (u, v)) if and only if $\{u, v\} \cap S \neq \emptyset$. A *vertex cover* (**VC**) of G, denoted by $\mathbb{C}$, is a subset of V_G that covers all edges in G, i.e., each edge of G is incident to at least one vertex in $\mathbb{C}$. Given a VC $\mathbb{C}$ of G, any superset of $\mathbb{C}$ is also a VC of G. The vertex set V_G is trivially a VC of G. A *minimum VC* of G is a VC that has the minimum set cardinality among all VCs of G.

Given a path p in G, the *length* of p is defined as $len(p) = \sum_{e \in p} \omega_G(e)$, i.e., the sum of the weight of the edges on p. Given a vertex $s \in V_G$, the *shortest path* from s to a vertex $v \in V_G$, denoted by $SP_G(s, v)$, is a path in G that has the minimum length among all paths from s to v in G. We define the *distance* from s to v in G as $dist_G(s, v) = len(SP_G(s, v))$. We define $dist_G(s, s) = 0$ for any $s \in V_G$.

For the analysis of I/O complexity, we use the standard I/O model [2] with the following parameters: M is the main memory size and B is the disk block size, where $1 \ll B \leq M/2$. Data is read/written in blocks from/to disk. Thus, reading/writing a piece of data of size N from/to disk requires (N/B) I/Os.

Table 1 gives the frequently-used notations in the paper.

Problem Definition. This paper studies the following problem: Given a graph $G = (V_G, E_G, \omega_G)$, construct a *disk-based index* for processing *single-source shortest path or distance* (**SSSP** or **SS-dist**) queries, i.e., for any input source vertex s, find $SP_G(s, v)$ or $dist_G(s, v)$ for all $v \in V_G$.

We propose I/O-efficient algorithms to construct the index, as to tackle the case when the input graph G cannot fit in main memory, i.e., $|G| > M$. We focus on *sparse graphs* since most large and many fast growing real-world networks are sparse. Apart from pro-

Algorithm 1 *A Simple VC-Based Index*

Input: A graph $G = (V_G, E_G, \omega_G)$
Output: A VC-based index, $DIST_G$

```
1. compute a vertex cover, $\mathbb{C}$, of $G$;
2. for each $u \in \mathbb{C}$ do
3.     compute $DIST_G(u, \mathbb{C}) = \{(v, dist_G(u, v)) : v \in \mathbb{C}\}$;
4. return $DIST_G(u, \mathbb{C})$ for all $u \in \mathbb{C}$;
```

Algorithm 2 *Query Processing By VC-Based Index*

Input: A graph $G = (V_G, E_G, \omega_G)$, a VC-based index $DIST_G$, and a source vertex s
Output: $dist_G(s, v)$ for all $v \in V_G$

```
1. if($s \in \mathbb{C}$)
2.     read $DIST_G(s, \mathbb{C})$;
3.     return GetDistance($s$, $DIST_G(s, \mathbb{C}), G$);
4. else  $/* s \notin \mathbb{C} */$
5.     $\mathbb{C}' \leftarrow \{s\} \cup \mathbb{C}$;
6.     compute $DIST_G(s, \mathbb{C}') = \{(v, dist_G(s, v)) : v \in \mathbb{C}'\}$, where:
         if $v = s$, $dist_G(s, v) = 0$;
         else, $dist_G(s,v)=min\{\omega_G(s,u)+dist_G(u,v): u \in adj_G(s)\}$;
7.     return GetDistance($s$, $DIST_G(s, \mathbb{C}'), G$);
```

Procedure 3 *GetDistance*(s, $DIST_G(s, \mathbb{C}), G$)

```
1. for each $v \in V_G$ do
2.     if($v \in \mathbb{C}$)
3.         get $(v, dist_G(s, v))$ from $DIST_G(s, \mathbb{C})$;
4.     else  $/* v \notin \mathbb{C} */$
5.         $dist_G(s, v) \leftarrow min\{dist_G(s, u) + \omega_G(u, v): u \in adj_G(v)\}$;
6. return $dist_G(s, v)$ for all $v \in V_G$;
```

cessing SSSP and SSdist queries, we also show that our index can be easily extended to process a few other related types of queries.

3. A SIMPLE VC-BASED INDEX

In this section, we propose a simple *VC-based* index for answering SSdist queries. We then identify the limitations of the index for processing large graphs.

3.1 Index Construction and Query Processing

We sketch how the index is constructed in Algorithm 1. The algorithm first computes a VC, $\mathbb{C}$, of the input graph G. The index is essentially a matrix, $DIST_G$, that keeps the pair-wise distance between the vertices in $\mathbb{C}$. For each $u \in \mathbb{C}$, $DIST_G(u, \mathbb{C})$ stores the distance from u to every vertex $v \in \mathbb{C}$.

We sketch how we process an SSdist query in Algorithm 2. Given a source vertex s, there are two cases: either $s \in \mathbb{C}$ (Steps 1-3 of Algorithm 2), or $s \notin \mathbb{C}$ (Steps 4-7 of Algorithm 2).

If $s \in \mathbb{C}$, we first read $DIST_G(s, \mathbb{C})$ (from disk) and then call Procedure 3 to compute $dist_G(s, v)$ for each $v \in V_G$. If $s \notin \mathbb{C}$, we add s to $\mathbb{C}$ to form a new VC $\mathbb{C}'$ of G and compute a new distance vector $DIST_G(s, \mathbb{C}')$. We then also call Procedure 3 with $DIST_G(s, \mathbb{C}')$ as input to compute $dist_G(s, v)$ for each $v \in V_G$.

We now discuss how Procedure 3 correctly computes the distance. We first examine an important property of a VC as follows.

LEMMA 1. *Given a graph G, a VC $\mathbb{C}$ of G, and a vertex v in G, if $v \notin \mathbb{C}$, then $\forall u \in adj_G(v)$, $u \in \mathbb{C}$.*

PROOF. Suppose on the contrary that $\exists u \in adj_G(v)$ such that $u \notin \mathbb{C}$. Then, the edge (u, v) is not covered by $\mathbb{C}$, which contradicts to the fact that $\mathbb{C}$ is a VC of G. □

We first show how Procedure 3 works and prove its correctness.

LEMMA 2. *Given a graph $G = (V_G, E_G, \omega_G)$, a source vertex s, and a distance vector $DIST_G(s, \mathbb{C})$, Procedure 3 correctly computes $dist_G(s, v)$ for all $v \in V_G$.*

PROOF. Procedure 3 computes $dist_G(s, v)$ for each $v \in V_G$ by considering two cases: (1) $v \in \mathbb{C}$ and (2) $v \notin \mathbb{C}$.

Case (1): $v \in \mathbb{C}$ (Steps 2-3 of Procedure 3). Then, $dist_G(s, v)$ is pre-computed and stored in $DIST_G(s, \mathbb{C})$.

Case (2): $v \notin \mathbb{C}$ (Steps 4-5 of Procedure 3). The shortest path in G from s to v must contain an adjacent vertex u of v. Thus, $SP_G(s, v)$ is 1 edge (i.e., (u, v)) further away from $SP_G(s, u)$ for some $u \in adj_G(v)$. By Lemma 1, if $v \notin \mathbb{C}$, then $\forall u \in adj_G(v)$, $u \in \mathbb{C}$. Thus, $\forall u \in adj_G(v)$, $dist_G(s, u)$ can be obtained directly from $DIST_G(s, \mathbb{C})$. From which, we compute $dist_G(s, v) = min\{dist_G(s, u) + \omega_G(u, v) : u \in adj_G(v)\}$. □

We now prove the correctness of Algorithm 2.

THEOREM 1. *Given a graph $G = (V_G, E_G, \omega_G)$, a VC-based index $DIST_G$ computed by Algorithm 1, and a source vertex s, Algorithm 2 correctly computes $dist_G(s, v)$ for all $v \in V_G$.*

PROOF. There are two cases of s: (1) $s \in \mathbb{C}$ (Steps 1-3 of Algorithm 2); and (2) $s \notin \mathbb{C}$ (Steps 4-7 of Algorithm 2).

Case (1): $s \in \mathbb{C}$. $DIST_G(s, \mathbb{C})$ is pre-computed by Algorithm 1 and the correctness of Procedure 3 is given in Lemma 2.

Case (2): $s \notin \mathbb{C}$. First, $\mathbb{C}' = (\{s\} \cup \mathbb{C})$ is a VC of G since $\mathbb{C}$ is a VC and any superset of a VC is also a VC. We now show that $DIST_G(s, \mathbb{C}')$ is correctly computed in Step 6. By Lemma 1, if $s \notin \mathbb{C}$, then $\forall u \in adj_G(s)$, $u \in \mathbb{C}$. For $v \in (\mathbb{C}' \setminus \{s\})$ (i.e., $v \in \mathbb{C}$), $SP_G(s, v)$ must contain some $u \in adj_G(s)$. Thus, $dist_G(s, v) = min\{\omega_G(s, u) + dist_G(u, v) : u \in adj_G(s)\}$. Since $u, v \in \mathbb{C}$, $dist_G(u, v)$ can be obtained from $DIST_G(u, \mathbb{C})$. Then, Step 7 passes $DIST_G(s, \mathbb{C}')$ to Procedure 3, which correctly computes the distance as proved in Lemma 2. □

We give an example of query processing as follows.

Example 1. Consider a graph G (for simplicity, assume that all edges are of unit weight), as shown in Figure 1. A VC of G is $\mathbb{C} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$. Let the source vertex be a. Since $\mathtt{a} \in \mathbb{C}$, $DIST_G(\mathtt{a}, \mathbb{C}) = \{(\mathtt{a}, 0), (\mathtt{b}, 2), (\mathtt{c}, 1), (\mathtt{d}, 2), (\mathtt{e}, 2), (\mathtt{f}, 3)\}$ is indexed. Procedure 3 is then invoked. It first retrieves $dist_G(\mathtt{a}, v)$ from $DIST_G(\mathtt{a}, \mathbb{C})$ directly for $v \in \mathbb{C}$. For any vertex not in $\mathbb{C}$, for example, h, the distance from a to h is computed from the adjacent vertices of h, i.e., $dist_G(\mathtt{a}, \mathtt{h}) = min\{dist_G(\mathtt{a}, \mathtt{a}) + \omega_G(\mathtt{a}, \mathtt{h}), dist_G(\mathtt{a}, \mathtt{b}) + \omega_G(\mathtt{b}, \mathtt{h})\} = min\{0 + 1, 2 + 1\} = 1$.

If the source vertex is not in $\mathbb{C}$, for example, h, we construct a new distance vector of h in $\mathbb{C}' = \{\mathtt{h}\} \cup \mathbb{C}$. First, $dist_G(\mathtt{h}, \mathtt{h}) = 0$. For each $v \in \mathbb{C}$, we compute $dist_G(\mathtt{h}, v)$ through the adjacent vertices of h, i.e., a and b. Thus, $dist_G(\mathtt{h}, v) = min\{\omega_G(\mathtt{h}, \mathtt{a}) + dist_G(\mathtt{a}, v), \omega_G(\mathtt{h}, \mathtt{b}) + dist_G(\mathtt{b}, v)\}$. For example, $dist_G(\mathtt{h}, \mathtt{e}) = min\{\omega_G(\mathtt{h}, \mathtt{a}) + dist_G(\mathtt{a}, \mathtt{e}), \omega_G(\mathtt{h}, \mathtt{b}) + dist_G(\mathtt{b}, \mathtt{e})\} = min\{1 + 2, 1 + 1\} = 2$. Then, $DIST_G(\mathtt{h}, \mathbb{C}')$ is passed to Procedure 3 and the distances are computed in the same way as discussed above. □

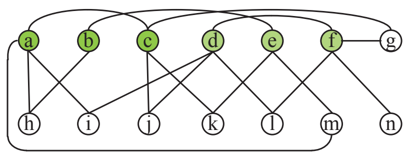

Figure 1: A graph G

3.2 Limitations of the Simple VC-based Index

We first analyze the complexity of both index construction and query processing and then identify the limitations of the index.

For query processing, if $s \in \mathbb{C}$, Algorithm 2 requires $O((|\mathbb{C}| + |G|)/B)$ I/Os, since we only need to read $DIST_G(s, \mathbb{C})$ from disk and scan G once, where $|DIST_G(s, \mathbb{C})| = 2|\mathbb{C}|$. If $s \notin \mathbb{C}$, the algorithm requires $O((deg_G(s) \cdot |\mathbb{C}| + |G|)/B)$ I/Os, since we need to read $DIST_G(u, \mathbb{C})$ for each $u \in adj_G(s)$.

The above analysis shows that query processing using the index is I/O-efficient, as Algorithm 2 reads only relevant distance vector(s) in blocks and sequentially scans G only once. However, if G is large, constructing such a VC-based index becomes impractical for the following reasons.

First, it can be expensive to compute a VC for a disk-resident graph (Step 1 of Algorithm 1). Second, computing $DIST_G(u, \mathbb{C})$ for all $u \in \mathbb{C}$ incurs a huge I/O cost for a large disk-resident graph (Steps 2-3 of Algorithm 1). Third, the space for storing $DIST_G(u, \mathbb{C})$, $\forall u \in \mathbb{C}$, is $O(|\mathbb{C}|^2)$, which is prohibitively large for a large graph. Such a large index is not practical even with the low cost of disk storage today. Finally, the memory space required for query processing is $O(|\mathbb{C}|)$, which is a problem if $|\mathbb{C}| > M$.

4. VC-INDEX: OVERALL FRAMEWORK

In the following sections, we propose a novel tree-structured index, named as *VC-index*, which applies VC for processing SSSP and SSdist queries. The new index addresses the limitations of the simple VC-based index discussed in Section 3.2. In this section, we first give the overall framework of our work.

We given the framework of VC-index as follows.

- **Index structure**:
  - Let G be the input graph. VC-index is a (rooted) tree-structured index of G.
  - A node in the tree is constructed based on and identified by a VC $\mathbb{X}$. We name the node as X (note that X is a node, while $\mathbb{X}$ is a VC at X).
  - A node X keeps a *distance graph*, D_X, where $\mathbb{X}$ is the set of vertices of D_X, and D_X is a graph from which we can compute the distance (in G) between any two vertices in $\mathbb{X}$.
  - The VC at the *root* of the tree is a VC of G.
  - The VC at each node (except the root) of the tree is a VC of the distance graph of its parent.
  - The VC at each internal node of the tree is the union of the VCs at its children.
- **Index construction**:
  - *Initialization*: compute a VC of G and construct the distance graph for the root of the index tree.
  - *Depth-first construction*:
    Let X be a node in the tree, and $\mathbb{X}$ be the VC at X, we use $D_X = (V_X = \mathbb{X}, E_X, \omega_X)$ to denote the distance graph of X.
    - ∗ At each internal node X, compute a VC of D_X for each of X's children, until the union of the VCs computed for X's children is equal to V_X.
    - ∗ For each new VC $\mathbb{Y}$ of D_X computed, create a new child Y of X and construct D_Y for Y, with $V_Y = \mathbb{Y}$. Recursively create the children of Y in a depth-first manner, until the distance graph of the child node is smaller than a pre-defined size threshold.

We give the framework of **query processing** as follows.

(1) If the source vertex s is in the VC of G:

- Locate the smallest leaf node, X, that is closest to the root of VC-index, where D_X contains s.
- Load D_X into memory and compute the distance from s to all other vertices in D_X.
- Move from X towards the root of VC-index. Assume that at the current step we move from Y to Z, where Z is the parent of Y, scan D_Z and compute the distance from s to each vertex in $V_Z \backslash V_Y$. Initially $Y = X$ and this process stops when Y becomes the root of VC-index.
- Scan G to compute the distance from s to each vertex that is not in the VC of G.

(2) If the source vertex s is not in the VC of G:
By Lemma 1, $\forall u \in adj_G(s)$, u is in the VC of G. Thus, we process the query with u as the source vertex, for each $u \in adj_G(s)$, in the same way as in Step (1). Then, the distance from s to each vertex v in G is the smallest $(\omega_G(s,u) + dist_G(u,v))$, where $u \in adj_G(s)$.

The above steps compute only the distances from s to all vertices in G. We discuss how to extend the index to report the shortest paths in Section 7.

5. VC-INDEX: INDEX CONSTRUCTION

In Section 4 we give the structure of VC-index and the framework of index construction. We now present the details.

5.1 Streaming Minimum VC Approximation

In the process of index construction, the distance graph at each node of VC-index is constructed based on a VC of the distance graph at its parent, or a VC of the input graph G if the node is the root. Therefore, it is crucial to ensure that the VC computation is efficient. Given a graph H, it is easy to obtain a trivial VC of H (e.g., V_H itself is a VC of H). However, the larger the size of the VC, the greater is the height of the VC-index tree, which implies higher cost in both index construction and query processing. Therefore, ideally we want every VC computed to be minimum.

Computing the minimum VC, however, is NP-hard. The best constant-factor approximation known is 2-approximation [9], which is an in-memory algorithm that randomly selects an uncovered edge and adds both of its end vertices to the VC, until all edges are covered. Note that the vertices in the 2-approximate minimum VC may not be ordered.

In our index construction, we need to compute VCs from distance graphs, which may not fit in memory. We also want the vertices in the VC to be ordered, so as to allow sequential scan and pre-fetching for I/O-efficient index construction and query processing. To overcome the memory usage barrier and to avoid costly external-memory sorting, we devise a streaming 2-approximation algorithm which outputs an ordered VC.

As shown in Algorithm 4, we scan the input graph H (either the original graph G or a distance graph in VC-index) only once in a streaming fashion and do not keep the graph in memory. We keep a dynamically maintained hashtable T. A vertex u is inserted into T only by an adjacent vertex v of u that is ordered before u (Steps 7-8), meaning that u is selected to be in the VC (with the selection of the edge (v, u)). For each vertex v, we output v either (1) if v is in T, i.e., v was selected to be in the VC (Steps 3-4); or (2)

Algorithm 4 *Streaming Minimum VC Approximation*

Input: A graph $H = (V_H, E_H, \omega_H)$
Output: An ordered, 2-approximate minimum VC of H

```
1. initialize an empty hashtable, $T$;
2. for each $v \in V_H$ do
3.     if($v$ is in $T$)
4.         output $v$ and delete $v$ from $T$;
5.     else   /* $v$ is not in $T$ */
6.         for each $u \in adj_H(v)$ do
7.             if($u$ is ordered after $v$ and $u$ is not in $T$)
8.                 output $v$ and insert $u$ into $T$;
9.                 goto Step 2 to process next $v \in V_H$;
```

there exists some $u \in adj_H(v)$ such that the edge (v, u) is not yet covered, i.e., both v and u are not in T (Steps 6-8).

We give an example of approximating the minimum VC by Algorithm 4 as follows.

Example 2. Assume that the vertices in the graph in Figure 1 are ordered alphabetically. Algorithm 4 first reads a, finds its first adjacent vertex c, then outputs a and inserts c into T. Similarly, it next outputs b and inserts e into T. Next, when it reads c, it outputs c and deletes c from T. Next, it outputs d and inserts f into T. Then, it outputs e and deletes e from T, and similarly for f. Now, for the rest of the vertices, although they are not in T, they do not have any adjacent vertex ordered after them. Thus, the VC computed is $\mathbb{C} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$. We can easily verify that every edge in the graph is incident to at least one vertex in $\mathbb{C}$. □

We now show that the VC computed by Algorithm 4 is indeed an ordered, 2-approximate minimum VC of H.

THEOREM 2. *Given a graph $H = (V_H, E_H, \omega_H)$, let $\mathbb{M}$ be the minimum VC of H and $\mathbb{C}$ be the output of Algorithm 4. Then, $\mathbb{C}$ is a VC of H, $|\mathbb{C}| \leq 2|\mathbb{M}|$, and $\mathbb{C}$ is ordered.*

PROOF. To prove that $\mathbb{C}$ is a VC of H, we show that for each $v \in V_H$ and each $u \in adj_H(v)$, the edge (v, u) is covered by $\mathbb{C}$. If v is in T when we read v, we output v (Steps 3-4), i.e., add v to $\mathbb{C}$, and thus $\forall u \in adj_H(v)$, (v, u) is covered. If v is not in T, there are two cases: (1) there exists some $u \in adj_H(v)$ ordered after v and u is not in T (Step 7); and (2) otherwise. In Case (1), since the edge (v, u) is not covered yet, we add v to $\mathbb{C}$ (Step 8), thus covering (v, u) for all $u \in adj_H(v)$. In Case (2), we do not output v (i.e., v is not added to $\mathbb{C}$). In this case, all $u \in adj_H(v)$ ordered after v (if any) are in T, thus covering (v, u) for all $u \in adj_H(v)$ ordered after v (if any). For each $u \in adj_H(v)$ ordered before v, since v is not in T, u must be in $\mathbb{C}$ because otherwise v must have been inserted into T when we read and process u in Steps 5-8. Thus, $\mathbb{C}$ covers (v, u) for all $u \in adj_H(v)$, no matter u is ordered before or after v.

Now we prove that $|\mathbb{C}| \leq 2|\mathbb{M}|$. It is easy to see that Algorithm 4 always adds pairs of vertices to $\mathbb{C}$. Whenever a vertex v is determined to be added to $\mathbb{C}$ (Steps 7-8), we output v and select one of its adjacent vertices u ordered after v to be in $\mathbb{C}$, though u is to be outputted later when we read and process u by Steps 3-4. For each pair of v and u added to $\mathbb{C}$, either v or u (or both) must be in $\mathbb{M}$ (otherwise the edge (v, u) is not covered by $\mathbb{M}$ and $\mathbb{M}$ is not a VC). Therefore, $|\mathbb{C}|/2 \leq |\mathbb{M}|$.

Finally, $\mathbb{C}$ is ordered because Algorithm 4 outputs a vertex v only at the time when it reads v (note that the vertices are ordered in the adjacency list graph representation, see Section 2). □

Algorithm 4 uses $(|G|/B)$ I/Os since it scans G only once. The memory space required is $|\mathbb{C}|/2$ in the worst case for keeping the hashtable T. Since we delete a vertex v from T as soon as we output v, the memory required in practice is much less than $|\mathbb{C}|/2$.

5.2 Construction of Distance Graph

Each node in VC-index keeps a distance graph. We define the distance graph as follows.

Definition 1 (DISTANCE GRAPH). *Given a graph $H = (V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H, let X be the node in VC-index constructed based on $\mathbb{X}$. The* distance graph *at X, denoted by $D_X = (V_X, E_X, \omega_X)$, is defined as follows:*

- *$V_X = \mathbb{X}$;*
- *E_X is a set of edges that ensures: $\forall u, v \in \mathbb{X}$, $dist_H(u, v) = dist_{D_X}(u, v)$ (i.e., $dist_H(u, v)$ can be computed in D_X).*

The difference between the vertex sets of D_X and H is the set of *non-covering* vertices, i.e., vertices that are in V_H but not in $\mathbb{X}$. Since D_X maintains the distance (in H) between the vertices in $\mathbb{X}$, the construction of D_X from H is essentially to re-establish the connection between the vertices in $\mathbb{X}$ after removing the non-covering vertices and their incident edges. However, the non-covering vertices are scattered in H and removing any edge from H may completely change some shortest paths. Re-establishing the connection between vertices in D_X by simply computing their shortest paths in H is too expensive, especially for a disk-resident graph.

We describe a graph construction method and prove that the graph constructed by this method is a distance graph as defined by Definition 1. We then design an I/O-efficient algorithm to construct the distance graph.

We first categorize the paths in H into two types.

Definition 2 (PATH TYPE). *Given a graph $H=(V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H, for any $u, v \in \mathbb{X}$, where $u \neq v$, we use $P_H(u, v)$ to denote the set of paths from u to v in H. We categorize the paths in $P_H(u, v)$ into two types.*

- *Type 1, denoted by $P_H^1(u, v)$: paths that do not pass through any vertex in $\mathbb{X}$, i.e., all vertices on the path, except u and v, are in $(V_H \setminus \mathbb{X})$;*
- *Type 2, denoted by $P_H^2(u, v)$: paths that pass through some vertex in $\mathbb{X}$, i.e., there exists some vertex w on the path such that $w \neq u$, $w \neq v$ and $w \in \mathbb{X}$.*

As a special case, if u and v are directly connected by an edge in H, then the path $\langle u, v\rangle$ belongs to Type 1. Apparently, $P_H(u, v) = P_H^1(u, v) \cup P_H^2(u, v)$ and $P_H^1(u, v) \cap P_H^2(u, v) = \emptyset$.

We define a graph that is constructed by compressing the Type 1 paths as follows.

Definition 3 (P_H^1-GRAPH). *Given a graph $H=(V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H, we define a P_H^1-*graph*, $D = (V_D, E_D, \omega_D)$, as follows:*

- *$V_D = \mathbb{X}$;*
- *$E_D = \{(u, v) : u, v \in \mathbb{X}$ and $P_H^1(u, v) \neq \emptyset\}$;*
- *For each $(u, v) \in E_D$, $\omega_D(u, v) = min\{len(p) : p \in P_H^1(u, v)\}$.*

In Definition 3, we connect two vertices $u, v \in \mathbb{X}$ in the P_H^1-graph D if and only if there exists some path from u to v in H that does not pass through any vertex in $\mathbb{X}$. Intuitively, the paths in $P_H^1(u, v)$ are in H but not in D since the intermediate vertices on the paths (if any) are not in $V_D = \mathbb{X}$. Thus, we add the edge (u, v) to D to capture the distance information carried by these paths.

We now show that the P_H^1-graph is in fact a distance graph.

THEOREM 3. *Given a graph $H = (V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H, the P^1_H-graph D is a distance graph of H.*

PROOF. To prove $\forall u, v \in \mathbb{X}$, where $u \neq v$, $dist_D(u,v) = dist_H(u,v)$, there are two cases: (1) $SP_H(u,v) \in P^1_H(u,v)$, and (2) $SP_H(u,v) \in P^2_H(u,v)$.

In Case (1), since $P^1_H(u,v) \neq \emptyset$, by Definition 3, the edge (u,v) is in D. Since $SP_H(u,v) \in P^1_H(u,v)$, we have $\omega_D(u,v) = dist_H(u,v)$. But we still need to prove that the path $p = \langle u, v\rangle$ in D, where $len(p) = \omega_D(u,v) = dist_H(u,v)$, is the shortest path from u to v in D. Suppose on the contrary that there exists another path $p' = \langle u = x_0, x_1, \ldots, x_y = v\rangle$ in D, where $y > 1$, such that $len(p') = \sum_{0 \leq i \leq y-1} \omega_D(x_i, x_{i+1}) < len(p)$. By Definition 3, $\omega_D(x_i, x_{i+1})$ is the minimum length of some path $\langle x_i, \ldots, x_{i+1}\rangle$ in $P^1_H(x_i, x_{i+1})$. Concatenating these paths then forms a path $\langle u = x_0, \ldots, x_i, \ldots, x_{i+1}, \ldots, x_y = v\rangle$ in H of length $len(p')$, which is smaller than $len(p) = dist_H(u,v)$. But "$len(p') < len(p) = dist_H(u,v)$" contradicts to the fact that $dist_H(u,v)$ is the (shortest) distance between u and v in H. Thus, the path $p = \langle u, v\rangle$ in D is the shortest path from u to v in D.

In Case (2), we assume that $SP_H(u,v)$ passes through y ($y \geq 1$) vertices in $\mathbb{X}$. Let $p = SP_H(u,v) = \langle u = x_0, \ldots, x_1, \ldots, \ldots, x_y, \ldots, x_{y+1} = v\rangle$, where $x_i \in \mathbb{X}$ for $0 \leq i \leq y+1$. For each subpath $p_i = \langle x_i, \ldots, x_{i+1}\rangle$ of p, where $0 \leq i \leq y$, all vertices between x_i and x_{i+1} are not in $\mathbb{X}$, and p_i is a shortest path from x_i to x_{i+1} in H (otherwise, p cannot be the shortest). Thus, we have $dist_H(u,v) = len(p) = \sum_{0 \leq i \leq y} dist_H(x_i, x_{i+1})$. Since p_i does not pass through any vertex in $\mathbb{X}$, this is reduced to Case (1) and thus we can find a path $p' = \langle u = x_0, x_1, \ldots, x_y, x_{y+1} = v\rangle$ in D with $len(p') = \sum_{0 \leq i \leq y} \omega_D(x_i, x_{i+1}) = \sum_{0 \leq i \leq y} dist_D(x_i, x_{i+1}) = \sum_{0 \leq i \leq y} dist_H(x_i, x_{i+1}) = dist_H(u,v)$. Following a similar proof in Case (1), we can prove that $SP_D(u,v) = p'$ and thus $dist_D(u,v) = len(p') = dist_H(u,v)$. □

Although the P^1_H-graph is indeed a distance graph, its definition does not give a hint on the design of an efficient algorithm. According to Definition 3, it requires the set of paths $P^1_H(u,v)$ for every pair of vertices $u, v \in \mathbb{X}$ in order to construct the edges. However, computing $P^1_H(u,v)$ for every u and v is expensive, especially if H cannot fit in memory.

We investigate the property of $P^1_H(u,v)$ and design an I/O-efficient algorithm to construct the distance graph defined in Definition 3. We first examine an important property of $P^1_H(u,v)$ as follows.

LEMMA 3. *Let $H = (V_H, E_H, \omega_H)$ be a graph and $\mathbb{X}$ be a VC of H. For any $u, v \in \mathbb{X}$ and any path $p = \langle u, \ldots, v\rangle$ in $P^1_H(u,v)$, there is at most one vertex between u and v on p.*

PROOF. Suppose on the contrary that there is more than one vertex between u and v on p. Let $p = \langle u, w, t, \ldots, v\rangle$. Since $p \in P^1_H(u,v)$, we know that $w \notin \mathbb{X}$ and $t \notin \mathbb{X}$. Then the edge (w,t) is not covered by $\mathbb{X}$, which contradicts to the fact that $\mathbb{X}$ is a VC of H. □

Lemma 3 implies that if $p = \langle u, \ldots, v\rangle \in P^1_H(u,v)$, then either $p = \langle u, v\rangle$ or $p = \langle u, w, v\rangle$, where $w \notin \mathbb{X}$. In other words, given $u, v \in \mathbb{X}$, v must be within 2 hops of u if $P^1_H(u,v) \neq \emptyset$. The first hop of u is simply $adj_H(u)$, while the second hop of u is the union of $adj_H(v)$ for each $v \in adj_H(u)$. Therefore, we only need to consider the 2-hop neighborhood of a vertex $u \in \mathbb{X}$ in order to establish the connections between u and other vertices in the distance graph.

Given a vertex u, the first hop is easily accessible by reading the adjacency list of a vertex. However, accessing the second hop requires random access to different locations of the graph. To reduce the I/O cost incurred by random disk access and wasteful data read (i.e., read a block of data but only use a small portion), we formulate the distance graph construction as a *block nested-loop join* to allow sequential disk scans, which is shown in Algorithm 5. Note that if H can fit in memory, the construction of distance graph is much faster since a 2-hop BFS from each vertex suffices.

Algorithm 5 *Distance Graph Construction*

Input: A graph $H = (V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H
Output: The distance graph $D_X = (V_X, E_X, \omega_X)$

```
1. $V_X = \mathbb{X}$;
2. for each block, $B_1$, of $H$ read do
3.     for each block, $B_2$, of $H$ read do
4.         for each vertex $u$ in $B_1$, where $u \in \mathbb{X}$, do
5.             for each vertex $v$ in $B_2$, do
6.                 if($v \in adj_H(u)$)
7.                     if($v \notin \mathbb{X}$)
8.                         for each $w \in adj_H(v)$ do
9.                             CreateEdge($u, w, \omega_H(u,v)+\omega_H(v,w)$);
10.                    else  /* $v \in \mathbb{X}$ */
11.                        CreateEdge($u, v, \omega_H(u,v)$);
12. return $D_X = (V_X, E_X, \omega_X)$;
```

Procedure 6 *CreateEdge*(u, v, *new-weight*)

```
1. if(edge $(u,v)$ is not in $E_X$)
2.     add $(u,v)$ to $E_X$ with $\omega_X(u,v)$ = new-weight;
3. else  /* edge $(u,v)$ is already created */
4.     $\omega_X(u,v) \leftarrow min\{\omega_X(u,v)$, new-weight$\}$;
```

In the block nested-loop join in Algorithm 5, both the outer relation and the inner relation are the input graph H. The outer relation reads a vertex $u \in \mathbb{X}$, together with the first hop of u, i.e., $adj_H(u)$, which is then joined with a vertex v in the inner relation by the condition that $v \in adj_H(u)$. Since the graph is represented in adjacency lists, the second hop of u (i.e., $adj_H(v)$ for each $v \in adj_H(u)$) can then be accessed in the inner relation. The outer relation is read into memory in blocks by a simple sequential scan of the input graph H, but only those vertices in $\mathbb{X}$ are processed since only vertices in $\mathbb{X}$ are in D_X (Step 4). For each block B_1 of the outer relation, we sequentially scan each block B_2 of the inner relation once to find matches (Step 5). If there is a match, i.e., a vertex v in B_2 is in $adj_H(u)$ of some u in B_1 (Step 6), we construct an edge (u,v) for the path $p = \langle u, v\rangle$ in $P^1_H(u,v)$ if $v \in \mathbb{X}$ (Steps 10-11) or construct an edge (u,w) for the path $p = \langle u, v, w\rangle$ in $P^1_H(u,w)$ for each $w \in adj_H(v)$ if $v \notin \mathbb{X}$ (Steps 7-9). Procedure 6 is used to ensure that the weight on each edge (u,v) in D_X is the minimum length of the paths in $P^1_H(u,v)$.

We now prove the correctness of Algorithm 5.

THEOREM 4. *Given a graph $H = (V_H, E_H, \omega_H)$ and a VC $\mathbb{X}$ of H, the graph $D_X = (V_X, E_X, \omega_X)$ computed by Algorithm 5 is a distance graph of H.*

PROOF. We prove that $D_X = (V_X, E_X, \omega_X)$ is the same as the graph $D = (V_D, E_D, \omega_D)$ defined in Definition 3.

First, $V_X = V_D = \mathbb{X}$. Second, we prove that $E_X = E_D$. Each edge in E_X is created by either Step 9 or Step 11 of Algorithm 5. If an edge (u,w) is created in E_X by Step 9, we have $u \in \mathbb{X}$ (by Step 4) and $w \in \mathbb{X}$ (otherwise, (v,w) is not covered by $\mathbb{X}$ since $v \notin \mathbb{X}$ by Step 7). Thus, we have $\langle u, v, w\rangle \in P^1_H(u,w)$ and $P^1_H(u,w) \neq \emptyset$. Therefore, by Definition 3, edge (u,w) is in E_D too. If an edge (u,v) is created in E_X by Step 11, we have $u \in \mathbb{X}$ (by Step 4) and $v \in \mathbb{X}$ (by Step 10). Thus, $P^1_H(u,v) \neq \emptyset$ since we have $\langle u, v\rangle \in P^1_H(u,v)$. Again by Definition 3, edge (u,v) is in E_D. Since every edge in E_X is in E_D, we have $E_X \subseteq E_D$.

On the other hand, for each edge $(u, v) \in E_D$, we have $u, v \in \mathbb{X}$ and $\exists p = \langle u, \ldots, v\rangle \in P^1_H(u, v)$. By Lemma 3, there is at most one vertex between u and v on p. Whether there is no vertex or one vertex between u and v on p, the edge (u, v) is created in E_X by Algorithm 5. Therefore, we have $E_D \subseteq E_X$ and thus $E_X = E_D$. Finally, we have $\omega_X(u, v) = \omega_D(u, v)$ for every edge (u, v) since the block nested-loop join in Algorithm 5 checks every path in $P^1_H(u, v)$ and $\omega_X(u, v)$ is updated as the minimum length of the paths in $P^1_H(u, v)$ as ensured by Step 4 of Procedure 6. $\square$

A Smaller Distance Graph. We can obtain a smaller distance graph by enforcing the triangle inequality on the D_X constructed by Algorithm 5. D_X satisfies the *triangle inequality* if and only if, for every triangle $\{e_1 = (u, v), e_2 = (v, w), e_3 = (w, u)\}$ in D_X, we have $\omega_X(e_1) < (\omega_X(e_2) + \omega_X(e_3))$, assuming that $\omega_X(e_1) \geq \omega_X(e_2)$ and $\omega_X(e_1) \geq \omega_X(e_3)$. To ensure the triangle inequality in D_X, we postprocess D_X as follows. We perform a block nested-loop join on D_X, which is the same as Steps 2-6 of Algorithm 5 except that the input is D_X instead of H. We then check each triangle $\{(u, v), (v, w), (u, w)\}$ found in D_X. If it does not satisfy the triangle inequality, we remove the edge with the highest weight in the triangle. In this way, we reduce the size of D_X while it is still a distance graph, since the distance between any two vertices in the graph is not changed.

We give an example of distance graph as follows.

Example 3. Given the graph G in Figure 1 and a VC of G $\mathbb{C} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$, the corresponding distance graph D is shown in Figure 2. We can easily verify that for any two vertices u and v in D, $dist_D(u, v) = dist_G(u, v)$. We can also verify that D satisfies the triangle inequality. $\square$

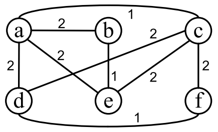

Figure 2: A distance graph D of G in Figure 1

Complexity of Distance Graph Construction. The I/O complexity analysis of Algorithm 5 is similar to that of the standard block nested-loop join. Let $b_1 = b_2 = O(|H|/B)$ be the number of disk blocks occupied by the outer/inner relation, and $b_3 = O(|D_X|/B)$ be the number of I/Os required to write D_X to disk. If H can fit in memory, Algorithm 5 requires $O(b_1 + b_3)$ I/Os. Otherwise, the algorithm requires $O(b_1 + (b_1/(M/B-2)) \cdot b_2 + b_3)$ I/Os. The CPU time complexity is $\sum_{u\in\mathbb{X}} \sum_{v\in adj_H(u)} deg_H(v) = O(|\mathbb{X}| \cdot d^2)$, where d is the average degree in H in the expected case and the maximum degree in H in the worst case. The complexity analysis of the triangle inequality checking is similar except that the input graph is D_X.

5.3 Overall Index Construction

We now present the overall index construction algorithm. As shown in Algorithm 7 and Procedure 8, for each node in VC-index, we invoke Algorithms 4 and 5 to compute the VC and the distance graph. For each node X, we ensure that the union of the VCs computed for X's children is equal to V_X (i.e., $\mathbb{X}$), which can be done by preferring the inclusion into a new VC those vertices that are not yet in any VC (by giving priority to those vertices). During the depth-first construction, a node Y is created as a leaf in VC-index if the size of D_Y is smaller than a pre-defined threshold σ. The setting of σ is discussed in Section 5.4.

We show an example of VC-index as follows.

Algorithm 7 *VC-Index Construction*

Input: A graph G
Output: VC-index of G

```
1. create the root, $\lambda$, of VC-index;
2. compute a VC $\mathbb{C}$ of $G$ for $\lambda$ (by Alg. 4);
3. construct the distance graph $D$ for $\lambda$ from $G$ and $\mathbb{C}$ (by Alg. 5);
4. GrowTree($\mathbb{C}$, $D$, $\lambda$);
```

Procedure 8 *GrowTree*($\mathbb{X}$, D_X, X)

```
1. $S \leftarrow \mathbb{X}$;
2. while($S \neq \emptyset$)
3.     create a child $Y$ of $X$;
4.     compute a VC $\mathbb{Y}$ of $D_X$ (by Alg. 4),
           by giving preference to the vertices in $S$;
5.     construct the distance graph $D_Y$ for $Y$ from $D_X$ and $\mathbb{Y}$ (by Alg. 5);
6.     if($|D_Y| > \sigma$)   /* $\sigma$ is a pre-defined size threshold */
7.         GrowTree($\mathbb{Y}$, $D_Y$, $Y$);
8.     $S \leftarrow S \backslash \mathbb{Y}$;
```

Example 4. Figure 3 shows VC-index of the graph G in Figure 1. The distance graph D at the root, λ, is shown in Figure 2, while the distance graphs at the two children X and Y of the root are shown in Figures 4 (a) and (b). We can easily verify that the vertex set of D_X (or of D_Y) is a VC of D in Figure 2, and each vertex in D (i.e., in the VC of G) appears in at least one distance graph at the leaf node of VC-index. $\square$

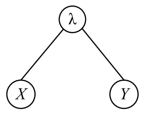

Figure 3: VC-index of G in Figure 1

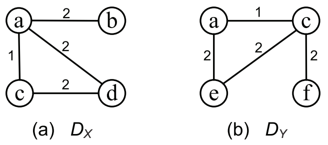

Figure 4: The distance graphs, D_X and D_Y, at nodes, X and Y, of the VC-index in Figure 3

5.4 Complexity of VC-Index Construction

There are two main operations in VC-index construction: minimum VC approximation and distance graph construction. We have analyzed the complexity of each of these two operations at the end of Sections 5.1 and 5.2. However, the input graph at each node of VC-index is different. A precise complexity analysis of the overall index construction is difficult because the size of the distance graph at each node depends on the characteristics of the input graph, which are difficult to model for real-world graphs. Instead, we give a worst-case analysis as follows.

Assume that VC-index has N nodes. Let $b = b_1 = b_2 = b_3 = O(|H|/B) = O(|G|/B)$, where H is a distance graph (note that a distance graph H is in general much smaller than G; if not, we can simply stop growing the VC-index tree from there). In the worst case, constructing VC-index requires $O(N(C_{VC} + C_{DG}))$ I/Os, where $C_{VC} = b$ is the I/O cost of approximating the minimum VC and $C_{DG} = O(b_1 + (b_1/(M/B - 2)) \cdot b_2 + b_3)$ is the I/O cost of computing a distance graph. Thus, $O(N(C_{VC} + C_{DG})) = O((N/(M/B - 2))(|G|/B)^2)$, where N is typically significantly

Algorithm 9 *Query Processing by VC-Index*

Input: A graph $G = (V_G, E_G, \omega_G)$, VC-index of G, and a source vertex s
Output: $dist_G(s, v)$ for all $v \in V_G$

```
1. let $\mathbb{C}$ be the VC at the root of VC-index;
2. $dist_G(s, v) \leftarrow \infty$, for all $v \in V_G$;
3. if($s \in \mathbb{C}$)
4.     return ComputeDistance($s$);
5. else  /* $s \notin \mathbb{C}$ */
6.     for each $u \in adj_G(s)$ do
7.         $DIST_G(u, V_G) \leftarrow$ ComputeDistance($u$);
8.         for each $v \in V_G$ do
9.             $dist_G(s,v) \leftarrow min\{dist_G(s,v), \omega_G(s,u) + dist_G(u,v)\}$;
10.    return $dist_G(s, v)$, for all $v \in V_G$;
```

Procedure 10 *ComputeDistance*(a)

```
1. let $X$ be the smallest leaf closest to the root and containing $a$;
2. load $D_X$ into memory;
3. compute the distance from $a$ to all other vertices in $D_X$;
4. $dist_G(a, v) \leftarrow dist_{D_X}(a, v)$, for all $v \in V_X$;
5. MoveToRoot($a, X$);
6. for each $v \in V_G \backslash \mathbb{C}$ do
7.     $dist_G(a, v) \leftarrow min\{dist_G(a, u) + \omega_G(u, v) : u \in adj_G(v)\}$;
8. return $dist_G(a, v)$, for all $v \in V_G$;
```

Procedure 11 *MoveToRoot*(a, X)

```
1. if($X$ is not the root of VC-index)
2.     scan $D_Y$, where $Y$ is the parent of $X$;
3.     for each $v \in V_Y \backslash V_X$ (in each block of $D_Y$ scanned) do
4.         $dist_G(a,v) \leftarrow min\{dist_G(a,u) + \omega_Y(u,v)$: $u \in adj_{D_Y}(v)\}$;
5.     MoveToRoot($a, Y$);
```

smaller than M/B. Note that this worst-case I/O complexity has assumed that all distance graphs, except those at leaf nodes, cannot fit in memory.

Let d be the maximum degree of a vertex in any distance graph or G. The CPU time complexity is $O(N(|G| + |V_G| \cdot d^2))$, where $O(|G|)$ is the time for approximating the minimum VC and $O(|V_G| \cdot d^2)$ is the time for computing a distance graph. The maximum degree d can be large; however, this is for the worst case and most real-world large graphs have a power-law degree distribution, for which only a very small number of vertices have large degree [13].

Let $\mathcal{X}$ be the set of nodes in VC-index. The storage space required for VC-index is $O(\sum_{X \in \mathcal{X}} |D_X|) = O(N|G|)$.

This size threshold σ for a leaf node X is set such that $|D_X| < M$, since D_X is to be loaded into memory for query processing. We also want running Dijkstra's algorithm [10] in D_X to have a complexity of $O(|V_G|)$, i.e., $O(|D_X| \log |V_X|) = O(|V_G|)$, which is the optimal time complexity for processing an SSdist or SSSP query since the output size is $O(|V_G|)$.

6. VC-INDEX: QUERY PROCESSING

We now discuss how to process an SSdist query by VC-index. Let $\mathbb{C}$ be the VC at the root of VC-index, i.e., $\mathbb{C}$ is a VC of G. As shown in Algorithm 9, query processing consists of two cases: (1) the source vertex s is in $\mathbb{C}$, and (2) s is not in $\mathbb{C}$.

If s is in $\mathbb{C}$, the algorithm invokes Procedure 10 to compute $dist_G(a, v)$ for all $v \in V_G$, where $a = s$ in this case. Procedure 10 loads the distance graph D_X at the leaf node X in VC-index, where D_X is the smallest distance graph containing a and X is closest to the *root* among other leaf nodes. We can locate X using an external lookup table with $O(1)$ I/O. We first compute the single-source distance from a to other vertices in D_X, by running Dijkstra's algorithm [10] in D_X. Then, Procedure 11 is invoked to compute the distance from a to other vertices that are not in D_X.

Procedure 11 reads the distance graph D_Y of X's parent, Y. At this point, $\forall u \in adj_{D_Y}(v)$ for each $v \in V_Y \backslash V_X$, $dist_G(a, u)$ has already been computed. Thus, we use $dist_G(a, u)$ to compute $dist_G(a, v)$ for each $v \in V_Y \backslash V_X$, where $u \in adj_{D_Y}(v)$. Then, we move up one step towards the root by invoking Procedure 11 to process on Y's parent, and recursively until we reach the root of VC-index.

When we return from Procedure 11 to Procedure 10, $dist_G(a, v)$ for all $v \in \mathbb{C}$ has been computed. Then, Steps 6-7 of Procedure 10 compute the distance from a to the rest of the vertices in G.

Finally, if s is not in $\mathbb{C}$, Algorithm 9 first computes the distance from each adjacent vertex u of s to all vertices in V_G. Then, $dist_G(s, v)$ is simply the minimum $(\omega_G(s, u) + dist_G(u, v))$, for $u \in adj_G(s)$.

We show an example of query processing as follows.

Example 5. Given the graph G in Figure 1 and VC-index of G in Figure 3, where the distance graphs are shown in Figures 2 and 4, let $\mathtt{a}$ be the source vertex. Since $\mathtt{a}$ appears in both D_X and D_Y, we assume that D_X is chosen for processing $\mathtt{a}$. We compute SSdist from $\mathtt{a}$ in D_X in memory and find that $dist_G(\mathtt{a}, \mathtt{b}) = 2$, $dist_G(\mathtt{a}, \mathtt{c}) = 1$, and $dist_G(\mathtt{a}, \mathtt{d}) = 2$. Then, we move up to the parent of X, i.e., the root, and scan D (block by block for a disk-resident graph). For those vertices that are in D but not in D_X, i.e., $\mathtt{e}$ and $\mathtt{f}$, the distance from $\mathtt{a}$ to their adjacent vertices must have been computed. Thus, we compute $dist_G(\mathtt{a}, \mathtt{e})$ and $dist_G(\mathtt{a}, \mathtt{f})$ from their adjacent vertices in D. That is, $dist_G(\mathtt{a}, \mathtt{e}) = min\{ (dist_G(\mathtt{a}, \mathtt{a}) + \omega_D(\mathtt{a}, \mathtt{e})), (dist_G(\mathtt{a}, \mathtt{b}) + \omega_D(\mathtt{b}, \mathtt{e})), (dist_G(\mathtt{a}, \mathtt{c}) + \omega_D(\mathtt{c}, \mathtt{e}))\} = min\{2, 3, 3\} = 2$. Similarly, $dist_G(\mathtt{a}, \mathtt{f}) = min\{ (dist_G(\mathtt{a}, \mathtt{c}) + \omega_D(\mathtt{c}, \mathtt{f})), (dist_G(\mathtt{a}, \mathtt{d}) + \omega_D(\mathtt{d}, \mathtt{f}))\} = min\{3, 3\} = 3$. Finally, for the remaining vertices in G, their distance from $\mathtt{a}$ is computed in a similar way as in Example 1.

If the source vertex is not in the VC of G, for example, $\mathtt{h}$, we first compute the distance to all vertices from $\mathtt{h}$'s adjacent vertices in G, $\mathtt{a}$ and $\mathtt{b}$. Then, the distance from $\mathtt{h}$ to any vertex, for example, $\mathtt{m}$, can be computed as $dist_G(\mathtt{h}, \mathtt{m}) = min\{(\omega_G(\mathtt{h}, \mathtt{a}) + dist_G(\mathtt{a}, \mathtt{m})), (\omega_G(\mathtt{h}, \mathtt{b}) + dist_G(\mathtt{b}, \mathtt{m}))\} = min\{2, 3\} = 2$. □

6.1 Correctness of Query Processing

We prove the correctness of query processing as follows.

THEOREM 5. *Given a graph $G = (V_G, E_G, \omega_G)$, VC-index of G computed by Algorithm 7, and a source vertex s, Algorithm 9 correctly computes $dist_G(s, v)$ for all $v \in V_G$.*

PROOF. We first prove the correctness for the case $s \in \mathbb{C}$, where $\mathbb{C}$ is a VC of G at the root of VC-index. This is essentially to prove the correctness of Procedure 10.

First, we show that $\forall v \in V_X$, $dist_G(a, v)$ is correctly computed by Step 4 of Procedure 10. By Theorem 4 and Definition 1, we know that $\forall u, v \in V_X$, $dist_{D_X}(u, v) = dist_H(u, v)$, where H is the distance graph at the parent of X, or $H = G$ if X is the root of VC-index. Thus, by Definition 1, $dist_{D_X}(u, v) = dist_{D_Y}(u, v) = \cdots = dist_{D_\lambda}(u, v) = dist_G(u, v)$, where D_λ is the distance graph at the root, λ, of VC-index. Thus, $\forall v \in V_X$, $dist_G(a, v)$ is correctly computed as $dist_{D_X}(a, v)$ in Steps 3-4 of Procedure 10.

Next, we prove that $\forall v \in \mathbb{C} \backslash V_X$, $dist_G(a, v)$ is correctly computed by invoking Procedure 11 in Step 5 of Procedure 10. We first show that, $\forall v \in V_Y \backslash V_X$, where Y is the parent of X in VC-index, Step 4 of Procedure 11 gives the correct $dist_G(a, v)$. Since $SP_{D_Y}(a, v)$ for any $v \in V_Y \backslash V_X$ must pass through a neighbor u

of v (or $a = u$), we have $dist_{D_Y}(a, v) = min\{dist_{D_Y}(a, u) + \omega_Y(u, v) : u \in adj_{D_Y}(v)\}$. Since V_X is a VC of D_Y, by Lemma 1, $\forall v \in V_Y \backslash V_X$, if $u \in adj_{D_Y}(v)$, then $u \in V_X$. Thus, $dist_G(a, u)$ has already been computed, using which Step 4 of Procedure 11 correctly computes $dist_G(a, v) = dist_{D_Y}(a, v)$, $\forall v \in V_Y \backslash V_X$. Procedure 11 then moves one step towards the root to process Y's parent recursively. When Procedure 11 reaches the root of VC-index, $dist_G(a, v)$, $\forall v \in \mathbb{C}$, is correctly computed.

Finally, we prove that $\forall v \in V_G \backslash \mathbb{C}$, $dist_G(a, v)$ is correctly computed by Step 7 of Procedure 10. Since $SP_G(a, v)$ for any $v \in V_G \backslash \mathbb{C}$ must pass through a neighbor u of v (or $a = u$), we have $dist_G(a, v) = min\{dist_G(a, u) + \omega_G(u, v) : u \in adj_G(v)\}$. For each $v \in V_G \backslash \mathbb{C}$, by Lemma 1, $\forall u \in adj_G(v)$, we have $u \in \mathbb{C}$. Note that up to now, $dist_G(a, u)$, $\forall u \in \mathbb{C}$, has been correctly computed. Thus, $dist_G(a, v)$, $\forall v \in V_G \backslash \mathbb{C}$, is correctly computed.

Since Procedure 10 is invoked with $a = s$, $dist_G(s, v)$, $\forall v \in V_G$, is correctly computed.

Now we prove the correctness for the case $s \notin \mathbb{C}$. For each $u \in adj_G(s)$, by Lemma 1, we have $u \in \mathbb{C}$. Thus, $dist_G(u, v)$ for all $v \in V_G$ can be correctly computed by Procedure 10 with $a = u$. Since $SP_G(s, v)$ for each $v \in V_G$ must consist of a neighbor u of s, we have $dist_G(s, v) = min\{\omega_G(s, u) + dist_G(u, v) : u \in adj_G(s)\}$. Thus, $dist_G(s, v)$, $\forall v \in V_G$, computed in Steps 6-9 of Algorithm 9 is correct. $\square$

6.2 Complexity of Query Processing

We now analyze the complexity of processing an SSdist query.

Let $\mathcal{Y}$ be the set of nodes in VC-index accessed during query processing. We first analyze the case $s \in \mathbb{C}$. For each $Y \in \mathcal{Y}$, Algorithm 9 only scans D_Y once. It also scans G once. Thus, in total we need $O((\sum_{Y \in \mathcal{Y}} |D_Y|/B) + |G|/B) = O(|\mathcal{Y}| \cdot |G|/B) = O(h|G|/B)$ I/Os in the worst case, where h is the height of VC-index.

In the above analysis, we assume that $M > |V_G|$. When $M < |V_G|$, Steps 3-4 of Procedure 11 and Steps 6-7 of Procedure 10 need to be implemented as a block nested-loop join as follows. The inner relation is the set $\{(u, dist_G(a, u)) : dist_G(a, u) \neq \infty\}$ (i.e., $dist_G(a, u)$ has already been computed), which is of size $|V_X| = O(|\mathbb{C}|)$ for each X along the path from the leaf to the root of VC-index. The outer relation is the distance graph, which is read in $O(|G|/B)$ I/Os. If we use $(M/B - 2)$ pages for reading the outer relation, we need $O(h(((|G|/B)/(M/B - 2)) \cdot (|\mathbb{C}|/B))) = O(h(|G|/B) \cdot (|\mathbb{C}|/M))$ I/Os in the worst case. This is only within a factor of $O(h|\mathbb{C}|/M)$ of $(|G|/B)$, i.e., $O(h|\mathbb{C}|/M)$ scans of G. For sparse real-world graphs, typically h is very small and $|\mathbb{C}|$ is significantly smaller than M.

We now analyze the CPU time complexity. In the worst case, Steps 3-4 of Procedure 11 and Steps 6-7 of Procedure 10 require $O(|D_Y|)$ and $O(|G|)$ time, respectively, since we at most process each edge once (or all adjacent vertices of v for all v). Let A be the time for computing SSdist in D_X using Dijkstra's algorithm [10]. According to Section 5.4, we have $A = O(|V_G|)$. When $M \geq |V_G|$, the worst-case CPU time complexity is $O(A + |\mathcal{Y}| \cdot |G|) = O(h|G|)$. When $M < |V_G|$, we need $O(A + |\mathcal{Y}| \cdot ((|G|/B)/(M/B - 2) \cdot |\mathbb{C}|)) = O((h|\mathbb{C}|/M) \cdot |G|)$ time.

Finally, if $s \notin \mathbb{C}$, the worst case I/O and CPU time complexity of Algorithm 9 is $deg_G(s)$ times those of the case $s \in \mathbb{C}$. We can easily make $deg_G(s)$ to be smaller than the average degree by preferring the inclusion of high-degree vertices into the VC $\mathbb{C}$.

7. VC-INDEX FOR SSSP AND ITS VARIANTS

In this section, we discuss how VC-index can be extended to process SSSP queries and a few types of closely related queries.

7.1 Single-Source Shortest Path Queries

To support SSSP queries, we only need to make the following change to VC-index.

As shown in Algorithm 5, there are two types of edges created in a distance graph D_X: (u, w) for a path $\langle u, v, w \rangle$ in H (Step 9), or (u, v) for a path $\langle u, v \rangle$ in H (Step 11). To process SSSP queries, we need to attach v to the edge (u, w) to indicate that (u, w) in D_X represents the path $\langle u, v, w \rangle$ in H.

For query processing, at the leaf node X, we compute the SSSPs from the source vertex a to all other vertices in D_X by Dijkstra's algorithm. The algorithm maintains a *predecessor* vector as a concise representation from which the exact shortest path can be obtained [9]. Then, when we move up from X towards the root of VC-index, in addition to finding the distance from a to a vertex v, we also put u as the predecessor of v if $u \in adj_{D_Y}(v)$ is on the shortest path from a to v in the current distance graph D_Y (Step 4 of Procedure 11). When we move up the tree, we may also need to update the predecessor of some vertices whose distance from a has already been computed. This is because when moving towards the root of VC-index, the distance graphs are "unfolded" step by step to capture more path information in the original graph G. The predecessor u of v is updated as another vertex w if w is attached with the edge (u, v) in the current distance graph. Finally, when we process Step 7 of Procedure 10 or Step 9 of Algorithm 9, we put u as the predecessor of v if $u \in adj_G(v)$ is on the shortest path from a or s to v in G.

Attaching an extra vertex with an edge in D_X increases the space requirement by at most $|E_X|$ for each distance graph D_X, which adds at most $O(|E_X|/B)$ I/Os for both index construction and query processing involving each D_X. Updating the predecessor can be easily done during the process of scanning a distance graph. Thus, asymptotically the complexity remains the same for both index construction and query processing.

7.2 Other Related Query Types

VC-index can be easily extended to answer some other related types of queries. We briefly discuss a few as follows.

BFS Queries. For processing BFS queries, no change is needed to our algorithm, since BFS is a special case of SSdist by treating each edge in the input graph as of unit weight.

Synopsis of Shortest Paths. In some applications we may not need the entire shortest path but only a synopsis of the path, i.e., not all vertices on the shortest path are returned. In the simple case, no change needs to be made to VC-index, but we can simply return the predecessor of a vertex in the distance graph where the distance of the vertex is computed. If we want to retain some important vertices on a shortest path, VC-index can be changed by always keeping those important vertices in the VCs or giving preference to them when computing the VCs, with more important vertices retained in the VCs at the nodes closer to the leaves of VC-index.

K-Level BFS Queries. Some applications, such as finding k-neighbors in social networks or finding the k-hops from a source vertex, require processing BFS for only k levels. We can use VC-index to process such a k-level BFS query by simply ignoring those vertices v in any distance graph if the BFS level of v from s is greater than k. This can save considerable amount of unnecessary processing. If k is known in advance, or if we can set a limit to the maximum value of k, we may further ignore all edges in any distance graph that have weight greater than k. In doing so we can further reduce the size of VC-index and speed up query processing.

8. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of VC-index (implemented in C++). We ran all experiments on a machine with Intel Xeon 2.67GHz CPU and 4GB RAM, running CentOS 5.4.

Datasets. We use the following three datasets in our experiments: *USRN*, *Web*, and *BTC*. The *USRN* graph is a weighted graph, which is the full *USA road network* (`http://www.dis.uniroma1.it/~challenge9/download.shtml`), where vertices represent intersections and endpoints, and edges represent the roads connecting these intersections or endpoints. *Web* (`http://barcelona.research.yahoo.net/webspam`) is a subgraph of the *UK Web graph*, where vertices are pages and edges are hyperlinks. The original graph $\vec{G}$ is directed and is converted into an undirected weighted graph G as follows: two vertices have an undirected edge in G with weight w if and only if they are reachable from each other within w steps in $\vec{G}$, where we set $w \in \{1, 2\}$. The *BTC* graph is an unweighted graph, which is a semantic graph converted from the *Billion Triple Challenge 2009 RDF dataset* (`http://vmlion25.deri.ie/`), where each vertex represents an object such as a person, a document, and an event, and each edge represents the relationship between two nodes such as "has-author", "links-to", and "has-title". We give the number of vertices and edges, the maximum vertex degree *max-deg*, and the storage size on disk, of the datasets in Table 2.

Table 2: Datasets (M $= 10^6$)

	$\|V\|$	$\|E\|$	*max-deg*	disk size	weighted?
USRN	24M	58M	9	1GB	yes
Web	106M	1,092M	36,561	13GB	yes
BTC	165M	361M	105,618	6GB	no

8.1 Results of Index Construction

We first report the results of index construction. We could not construct the simple VC-based index described in Section 3 because pre-computing the distance matrix is too expensive for such large datasets. Note that constructing the distance matrix is equivalent to processing many SSdist queries (see Table 4 for the cost of answering such a query without an index).

For the construction of VC-index, we report two settings of available memory size, $M = 2$GB and $M = 4$GB. Table 3 reports the total index construction time, together with the total time taken for the two main operations of VC-index construction, i.e., minimum VC approximation and distance graph (DG) construction.

Table 3: Min. VC approximation time, distance graph construction time, total indexing time (wall-clock time in seconds)

	VC time	DG time	Total time
USRN ($M = 2$GB)	6	283	289
USRN ($M = 4$GB)	6	283	289
Web ($M = 2$GB)	41	62,262	62,303
Web ($M = 4$GB)	41	22,989	23,030
BTC ($M = 2$GB)	25	13,074	13,099
BTC ($M = 4$GB)	25	4,216	4,241

The results show that our streaming 2-approximation algorithm for computing minimum VC is efficient and it takes only a very small portion of the total indexing time. The most time consuming part of the index construction is to construct the distance graphs at each node of the VC-index tree.

The total indexing time for *USRN* is much smaller because the graph is relatively small compared to the other two graphs, and also because road network graph is much simpler and has a much lower maximum vertex degree. We also note that constructing VC-index for *USRN* uses less than 1GB of memory. Thus, the indexing time is the same for $M = 2$GB and $M = 4$GB.

The indexing time for the other two graphs, *Web* and *BTC*, is much longer. However, given the large size of these two graphs and the large maximum vertex degree, constructing VC-index is efficient especially given the fact that our algorithm is not an in-memory algorithm. As we will show in Table 4, even just processing a few queries using an external-memory algorithm would take longer time than constructing the whole index.

Table 3 also shows that when more memory is available, the indexing time reduces significantly. As the available memory M is doubled from 2GB to 4GB, the total indexing time decreases by approximately 3 times for both *Web* and *BTC*. The tremendous speedup is because in processing the block nested-loop join in the construction of a distance graph, more memory reduces the number of times the inner relation being loaded and processed, and thus reducing both the I/O and CPU cost. On the contrary, the VC approximation algorithm is a streaming algorithm, and thus its running time remains stable with the increase in available memory.

Finally, the storage sizes of the indexes (excluding the original graph) for *USRN*, *Web*, and *BTC* are 1.8, 13.5, and 3.1 GB, respectively. Note that the index size of *BTC* is only half of its graph size because *BTC* has a very small VC.

8.2 Results of Query Processing

We next report the results of query processing using VC-index. According to Algorithm 9, query processing with VC-index has two cases: **VC queries** and **non-VC queries**, i.e., the query vertices that are in the VC of the input graph and those that are not. Section 6.2 also shows that VC-index has different complexity for processing VC queries and non-VC queries. Therefore, we tested queries of both types. We randomly selected 20 queries. We have actually tested much more queries for VC-index but processing these queries using the external-memory (EM) algorithm **EM-BFS** is too time consuming (see Table 4). Since the querying time for single-source distance queries is very stable for both VC-index and EM-BFS, we only report the results for 20 queries with which EM-BFS was tested.

The first two rows of Table 4 report the query processing time of VC-index, averaged over all queries of each type. The remaining four rows in Table 4 will be discussed shortly.

Table 4: Average query processing time (wall-clock time in seconds) of VC-index, IM-SSdist, and EM-BFS

	USRN	*Web*	*BTC*
VC-index (VC queries)	5.54	18.83	5.03
VC-index (non-VC queries)	6.64	52.78	8.14
IM-SSdist (VC queries)	29.97	–	–
IM-SSdist (non-VC queries)	30.07	–	–
EM-BFS (VC queries)	–	7,486.77	3,160.16
EM-BFS (non-VC queries)	–	7,506.42	3,166.59

The result shows that even for large graphs with over 100 million vertices and 1 billion edges, processing SSdist queries takes at most tens of seconds. We remark that processing an SSdist query has a lower-bound complexity of $\Theta(|V_G|)$ even with an index, because the output size is $\Theta(|V_G|)$. Thus, the query performance of VC-index is very competitive, which will be made even clearer when we compare with IM-SSdist and EM-BFS next.

Table 4 also shows that processing non-VC queries takes only slightly longer time than processing VC queries for *USRN* and *BTC*. However, for the *Web* graph, processing a non-VC query is about three times slower than processing a VC query. This is mainly because the *Web* graph is much denser than *USRN* and *BTC*.

Before we report the results for other related algorithms, we note that query processing by VC-index uses about 0.5GB to at most 1.5GB memory for different datasets. VC-index does not need to use up the available 4GB memory, because it only needs to load the distance graph at a leaf node into memory and then scans the distance graph at an internal node in the VC-index tree.

Baseline Reference. Since there is no existing index for processing single-source distance queries, we use some closely related works as comparison baselines to give readers some reference regarding the performance of VC-index.

One baseline reference is the indexes for processing *source-to-target* distance queries [31, 30], since we can issue $(|V_G| - 1)$ source-to-target distance queries to answer a single-source distance query. However, all these indexes required more than 4GB memory (plus an addition of 2GB virtual memory) to construct even on the smallest *USRN* dataset. Although we are not able to obtain any query result for these indexes, it does reflect that there is a need to design I/O-efficient algorithms for index construction when memory is insufficient, even though the constructed index may be resident on disk.

Another baseline reference is to directly run a single-source distance algorithm on the input graph. For the *USRN* dataset that fits in memory, we use Dijkstra's algorithm with a binary heap [10, 9], denoted by **IM-SSdist**. For the *Web* and *BTC* graphs that cannot fit in memory, we use EM algorithms. However, we were not able to obtain any implementation of existing EM SSdist algorithms [17, 21, 22, 19, 20]. Thus, we use the latest released implementation of the state-of-the-art EM BFS algorithm [18] instead, denoted by **EM-BFS**. We remark that in practice SSdist is significantly more costly than BFS since the factor $O(\log_2 |V_G|)$ cannot be ignored for a graph with even just 1 million vertices, i.e., SSdist can be $\log_2 10^6 \approx 20$ times slower than BFS, while our graphs are much larger.

The last four rows of Table 4 report the query processing time of IM-SSdist (for *USRN*) and EM-BFS (for *Web* and *BTC*), averaged over all queries. Note that there is no difference for IM-SSdist and EM-BFS in processing VC or non-VC queries, but we report the two types separately for clearer reference with the performance of VC-index.

The result shows that query processing using VC-index is about 5.4 times faster for VC queries and 4.5 times faster for non-VC queries than IM-SSdist on *USRN*. The significantly improved running time is because our algorithms runs IM-SSdist on a much smaller distance graph (at a leaf of VC-index) in $O(|V_G|)$ time, while computing the distance from the source vertex to other vertices not in the distance graph takes only $O(|V_G|)$ time as well. However, running IM-SSdist on *USRN* takes $O(|G| \log |V_G|)$ time.

For *Web* and *BTC*, the advantage of using VC-index over EM-BFS is obvious. Query processing by VC-index is over two orders of magnitude faster than EM-BFS on both *Web* and *BTC* for all queries tested. This result demonstrates the efficiency of VC-index for processing SSdist or BFS queries.

Applications. Based on the above results, we now demonstrate how VC-index may benefit some popular applications. Consider the approximation of the *closeness* measure [12], which requires to answer SSdist queries from $k = (\alpha \log |V_G| / \epsilon^2)$ source vertices, where $\alpha \geq 1$ is a constant and ϵ is a parameter for error control.

To obtain a reasonably small error, e.g., $\epsilon = 0.1$, the corresponding values of k for *USRN*, *Web*, and *BTC* (assuming $\alpha = 1$), are given in Table 5. The estimated running time required for the approximation of closeness without an index and with an index is also given in Table 5. The running time of closeness approximation is estimated by multiplying the querying time of BFS/SSdist (without/with an index) by the corresponding k.

Table 5: Sample size and estimated running time (in seconds) for the approximation of *closeness* without an index and with VC-index

	USRN	*Web*	*BTC*
Sample size k	2,451	2,334	2,730
No index	73,589	17,494,872	8,634,604
VC-index	15,217	106,558	22,213

In Table 5, to estimate the time needed for approximating closeness without an index, we use IM-SSdist for *USRN*, and EM-BFS for *Web* and *BTC*. For approximating closeness with an index, we use VC-index for all the three datasets, while we also include the construction time of VC-index into the total time. We use the average querying time reported in Table 4 for the estimation. For VC-index, the querying time is averaged over VC and non-VC queries.

The result clearly demonstrates the advantage, in fact the necessity, of using an index for approximating the closeness measure in a large graph. Without an index, it is obviously impractical to approximate or compute the closeness measure in large graphs.

The approximation or exact computation of many other important centrality measures for network analysis, such stress [26], betweenness [15], global efficiency [27], characteristic path length [8], integration and radiality [29], etc., are all processed in a similar way as closeness (i.e., require to invoke SSdist queries lots of times). To compute these measures in a large network, using an index is clearly a more feasible and efficient way. In addition, traditional applications such as urban planning and network routing, as well as others listed in Section 1, also reveal the need for such an index especially in today's continually growing networks.

9. A DISCUSSION ON UPDATE MAINTENANCE OF VC-INDEX

We consider two types of updates in the input graph G: edge insertion and edge deletion. Vertex insertion and deletion can be handled as inserting/deleting an isolated vertex followed/preceded by edge insertion/deletion.

Rebuilding the entire VC-index from scratch is expensive when the update is frequent; however, we can limit the scope of the update in VC-index to small local areas and perform incremental index maintenance. According to Lemma 3, the construction of the edge set of a distance graph is restricted to the 2 hops of each vertex only. For the insertion or deletion of an edge (u, v), we only need to examine the 2 hops of u and v in the input graph from which a distance graph is constructed, and add or remove edges in the distance graph accordingly to restore the conditions defined in Definition 1. For edge insertion, vertex insertion into the distance graph is also needed if both u and v are not in the distance graph. For edge deletion, vertex deletion from the distance graph is needed if either u or v now becomes isolated. The update is performed in the same way at each node of VC-index, and may propagate to the leaf nodes but it stops early when there is no update needed at an internal node. Finally, we note that this update is lazy update in the sense that the VC at each node may no longer be a 2-approximate minimum VC. This should not affect the performance of the index much and we can rebuild the index periodically at system idle time.

10. RELATED WORK

Many indexes have been proposed for processing source-to-target shortest path and distance queries. Most of them are for planar graphs or road networks (see [1] and the references therein). For general graphs, there are a few proposed recently [31, 30], all of them are in-memory indexes (thus not suitable for processing large graphs) and most are for un-weighted graphs.

For processing SSSP or BFS on large graphs that cannot fit in memory, the existing solutions are mainly EM algorithms [17, 21, 22, 19, 20] (for SSSP) and [24, 6, 18, 4, 5] (for BFS). Note that these algorithms actually find single-source distance, but can be extended to report the actual path, as does in VC-index. However, these algorithms are expensive for applications where SSSP or BFS needs to be processed repeatedly and frequently (see a list of applications in Section 1). Our index offers a practical solution for these applications in massive graphs.

11. CONCLUSIONS

We presented a disk-based index for processing single-source shortest path or distance queries. We verified by experiments that VC-index is efficient to construct in large graphs with more than 100 million vertices and 1.1 billion edges. Query processing using VC-index can be several hundred times faster than a non-index approach for the large graphs that cannot fit in memory. In addition, our experimental results also reflect the need of such an index in real applications.

12. ACKNOWLEDGMENTS

This research is supported in part by the A*STAR Thematic Strategic Research Programme Grant (102 158 0034), and the AcRF Tier-1 Grant (M52020092) from Ministry of Education of Singapore.

13. REFERENCES

[1] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] 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.

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

[5] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory bfs implementation. In *ALENEX*, 2007.

[6] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[7] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD Conference*, pages 447–458, 2010.

[8] T. R. Coffman and S. E. Marcus. Dynamic classification of groups through social network analysis and HMMs. In *IEEE Aerospace Conference*, pages 3197–3205, 2004.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[10] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269Ű271, 1959.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[12] D. Eppstein and J. Wang. Fast approximation of centrality. In *SODA*, pages 228–229, 2001.

[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[14] L. Ford and D. R. Fulkerson. Maximum flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[15] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35Ű41, 1977.

[16] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[17] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *In Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[18] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *ESA*, pages 723–735, 2002.

[19] U. Meyer. Via detours to I/O-efficient shortest paths. In *Efficient Algorithms*, pages 219–232, 2009.

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

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

[22] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *ESA*, pages 540–551, 2006.

[23] S. Micali and V. V. Vazirani. An $O(|E|\sqrt{|V|})$ algoithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27, 1980.

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

[25] M. Najork and J. L. Wiener. Breadth-first crawling yields high-quality pages. In *WWW*, pages 114–118, 2001.

[26] A. Shimbel. Structural parameters of communication networks. *Mathematical Biophysics*, 15:501Ű507, 1953.

[27] B. E. Thomason, T. R. Coffman, and S. E. Marcus. Sensitivity of social network analysis metrics to observation noise. In *IEEE Aerospace Conference*, pages 3206–3216, 2004.

[28] J. S. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, 83(1):1 – 20, 1989.

[29] T. W. Valente and R. K. Foreman. Integration and radiality: Measuring the extent of an individual's connectedness and reachability in a network. *Social Networks*, 20(1):89–105, 1998.

[30] F. Wei. TEDI: efficient shortest path query answering on graphs. In *SIGMOD Conference*, pages 99–110, 2010.

[31] Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.