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

Lucas Jacques
May 2014

Abstract
This is a report for the Seminar Algorithms for Database Systems, organized by the University of Zürich and by ETH Zürich. This paper summarizes the contents of the paper Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach [1] by James Cheng, Shumo Chu, Carter Cheng — all affiliated with the Nanzang Technological University, Singapore — and by Yiping Ke, from the Institute of High Performance Computing in Singapore.

The paper presents an I/O-efficient algorithm for processing single-source shortest path or distance queries over a large graph, with the aid of a disk-resident tree of vertex covers, called VC-index.

Contents
1. Introduction ................................................................. 2
   1.1 Problem Definition and Notations .................................... 2
2. Vertex Covers .................................................................. 3
   2.1 Motivation and Definition .............................................. 3
   2.2 Generating a Vertex Cover ............................................ 3
   2.3 Typical Size .................................................................. 4
3. A Simple Approach: Using One Vertex Cover ........................ 5
   3.1 Issues ....................................................................... 6
4. VC-Index ........................................................................ 6
   4.1 Overview .................................................................... 6
   4.2 Distance Graphs .......................................................... 7
   4.3 Overall Index Construction .......................................... 9
   4.4 Query Processing .......................................................... 10
5. Performance .................................................................. 11
   5.1 Complexity ................................................................. 11
   5.2 Experimental Results ................................................... 11
6. Conclusion .................................................................... 12
7. References ..................................................................... 13
1. Introduction

Determining the shortest path or the distance between two vertices in a graph is a task found in a broad number of problems, including route planning, social network analysis and the routing of a message in a network [2].

Due to the fact that graphs are becoming increasingly large, new problems arise as they no longer fit into main memory. New approaches are necessary to process large graphs because conventional methods cannot be applied anymore or are not sufficiently fast.

The algorithm of this paper proposes an I/O-efficient solution to answer single-source shortest path or distance queries over a graph with the aid of a tree – the so-called VC-index, where each node contains a subset of vertices of the original graph. The set of vertices in a node of VC-index is increasingly smaller as one travels away from the root of the tree.

After introducing some notations and the precise problem VC-index has been designed for, we will dedicate a section to vertex covers as they are central to this report. Subsequently, a simple approach on how to answer shortest path or distance queries with one mere vertex cover will be demonstrated.

Finally, VC-index will be presented in contrast to the simple approach, followed by a discussion of its performance.

1.1 Problem Definition and Notations

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. We assume all weights to be 1 if the graph is not weighted.

The set of adjacent vertices (or neighbors) of a vertex $v$ in a graph $G$ is denoted by $\text{adj}_G(v)$.

The length of a path $p$ in a graph is referred to as $\text{len}(p)$ and defined as the sum of all edges’ weight in the path, i.e. $\text{len}(p) = \sum_{e \in p} \omega_G(e)$. The shortest path between two vertices $s$ and $v$ in a graph, $SP_G(s, v)$, is defined as the path whose length is the shortest of all possible paths.

The distance between two vertices $s$ and $v$ in a graph $G$, written as $\text{dist}_G(s, v)$, is defined as the length of the shortest path, i.e. $\text{dist}_G(s, v) = \text{len}(SP_G(s, v))$. Furthermore, for any $s \in V_G$, $\text{dist}_G(s, s)$ is defined as 0.

The goal of the VC-index paradigm is to efficiently compute the single-source shortest path $SP_G(s, v)$ or distance $\text{dist}_G(s, v)$ from any vertex $s \in V_G$ to all vertices $v \in V_G$ for graphs so large that they do not fit into main memory.

Graphs are assumed to be stored as adjacency list, in memory as well as on disk. Furthermore, we expect that a unique ID is assigned to each vertex and that the vertices in the adjacency list are ordered by their ID in an ascending fashion.

The focus of VC-index lies on sparse graphs as most large real-world graphs fall into that category [4]. A sparse graph is defined as a graph whose number of edges is not close to the possible maximum of approximately $N^2 / 2$ edges, where $N$ is the total number of vertices.

To analyze I/O complexity, the standard I/O model will be used, where writing or reading a piece of data is $O(1)$. We introduce two parameters: $M$ denotes the size of the main memory and $B$ stands for the disk block size. Reading or writing a piece of data of size $N$ thus requires $(N/B)$ I/Os.
2. Vertex Covers

2.1 Motivation and Definition

Since the size of graphs can be prohibitively large to allow operations on all of the data directly, the first step to prepare for efficient computations is to reduce the size of the material in a sensible manner. Building a vertex cover of a graph is a simple way to achieve this.

A vertex cover \( C \) of a graph \( G = (V_G, E_G, \omega_G) \) is defined by a subset of vertices \( V_G' \) of \( V_G \) such that all edges \( e \in E_G \) have at least one end point in \( V_G' \). In other words, every edge of the graph \( G \) has at least one vertex that is in the vertex cover \( C \) (in the set \( V_G' \)).

One important property of vertex covers considerably facilitates certain operations: given a vertex \( v \) in \( V_G \) and a vertex cover \( C \) of \( G \), if \( v \notin C \), then \( u \in C \) for all \( u \in \text{adj}_G(v) \). If we suppose the opposite, namely that \( v \) is not in \( C \) and that an adjacent vertex \( u \in \text{adj}_G(v) \) exists which is not in \( C \) either, it means that the edge \((u, v)\) is not covered by \( C \), which by definition implies that \( C \) is not a vertex cover of the graph \( G \).

It is clear that no big portions of the graph are ever left out in vertex covers, as all adjacent vertices of any vertex \( v \in C \) are elements of \( C \).

2.2 Generating a Vertex Cover

Trivially, the graph \( G = (V_G, E_G, \omega_G) \) is a vertex cover of itself. However, it is crucial to compute a vertex cover whose size is as small as possible because all further computation is based on it. The minimum vertex cover refers to the vertex cover of minimum size; it is therefore the best suited vertex cover in terms of size. However, generating the minimum vertex cover is NP-hard [5].

The paper presents an efficient algorithm to build a vertex cover with an acceptable size using a so-called 2-approximation. A 2-approximation guarantees that the resulting vertex cover will not exceed twice the size of the minimum vertex cover. As such, the gain in computing time for some excess in size seems like a satisfying trade-off.

Moreover, seeing as the input graph is disk-resident and ordered by vertex ID, the aim of the algorithm is to keep I/O operations at a minimum and to ensure that the result is ordered by vertex ID as well.

The graph is traversed once in a streaming fashion and makes use of a hashtable to temporarily store vertices. For every vertex \( v \), the algorithm checks whether \( v \) is in the hashtable. If so, \( v \) is added to the vertex cover and removed from the hashtable.

If the hashtable does not contain the vertex \( v \), the algorithm considers its adjacent vertices. If an adjacent vertex \( u \) is found whose ID is higher than that of \( v \) and not present in the hashtable, \( v \) is added to the vertex cover and \( u \) is saved into the hashtable.

The pseudocode of the 2-approximation is shown in Figure 1. Figure 2 shows a graph, where the vertices \( \{a, c, d, f, g, h\} \) make up the vertex cover \( C \) generated by the algorithm.
Any vertex cover $\mathcal{C}$ generated by the algorithm in Figure 1 never exceeds twice the size of the minimum vertex cover $\mathcal{M}$, i.e. $|\mathcal{C}| \leq 2|\mathcal{M}|$. The algorithm always selects pairs of vertices to add to the vertex cover, adding a vertex $v$ to the cover immediately and adding an adjacent vertex $u$ into the hashtable, such that it is added to $\mathcal{C}$ at a later time.

In other words, both $u$ and $v$ are added to $\mathcal{C}$ for the edge $(u, v)$. In $\mathcal{M}$, the vertex $u$, $v$ or both $u$ and $v$ are part of $\mathcal{M}$, or the edge would not be covered. This is the case for every pair added to the vertex cover, which means that $|\mathcal{C}|/2 \leq |\mathcal{M}|$.

Apart from the aforementioned guarantee in size, the algorithm has two additional useful properties:

(1) The disk-resident graph can be read sequentially from the disk and does not have to remain in memory. As mentioned in 1.1, it is assumed that graphs are represented in their adjacency list form. This representation allows processing the whole graph with only one read per block, as shown in the pseudocode of Figure 1.

(2) When the algorithm decides to add a pair of vertices to the vertex cover $\mathcal{C}$, it temporarily stores one of the vertices, $u$, into the hashtable. Once the iteration arrives to $u$, it is removed from the hashtable and added to the vertex cover $\mathcal{C}$. Consequently, vertices are added to $\mathcal{C}$ in an ordered manner. The fact that $\mathcal{C}$ is ordered helps reduce the number of I/O operations in computations which use the vertex cover as input.

2.3 Typical Size

The algorithm described in the previous section has been applied to randomly generated graphs in order to gain some figures on how big a vertex cover typically is, compared to the original graph.

For instance, the graph $H$ in Figure 3 was generated by initializing 10 vertices with a desired degree of 25, as well as 30 vertices each with a degree of 2, 3 and 4. Edges were randomly added to the graph until each vertex had reached the desired degree. Thereafter, a vertex cover was built using the 2-approximation in section 2.2.

It appears that the size of the resulting vertex cover is around 50% of the original graph in the best case, but up to 95% for dense graphs. Since more edges have to be covered in dense graphs, it is intuitive that the size of the vertex covers increase.

The resulting vertex cover of the graph $H$ in Figure 3 is made up by all vertices with a filled background — namely of 49 vertices. In contrast, a graph of 100 vertices with 30 edges each typically has a vertex cover of 94 to 96 vertices.

Figure 3: A random graph $H$, consisting of 100 vertices.

From the experiments conducted, it is clear that operating over a vertex cover will not provide much gain in computation speed for dense graphs. As specified in 1.1, we assume that we operate on sparse graphs, as most real-world large graphs are not dense.
3. A Simple Approach: Using One Vertex Cover

We now illustrate a simple way to handle distance queries over a graph \( G \) with the aid of a vertex cover \( \mathcal{C} \). The same approach can be applied to shortest path queries by additionally saving path information.

The idea is to compute the distance between all vertices of the vertex cover \( \mathcal{C} \) with a given algorithm, e.g. Dijkstra’s algorithm [9], and to store the results in a so-called distance matrix \( \text{DIST}_G \) in order to be able to retrieve the values in a fast manner at a later time.

In other words, dist\(_{\mathcal{C}}(u,v) \) is computed and stored into \( \text{DIST}_G(u,v) \) for all vertices \( u,v \in \mathcal{C} \). Finding the distance between any \( u \) and \( v \) in \( \mathcal{C} \) afterwards is thus simply a matter of retrieving the corresponding entry in the distance matrix. In this manner, an expensive algorithm does not have to be executed multiple times to compute the same value.

The pseudocode to generate a distance matrix is given in Figure 4.

To compute dist\(_{\mathcal{C}}(s,v) \) from any vertex \( s \in \mathcal{V}_G \) to \( \forall v \in \mathcal{V}_G \) with help of a distance matrix, it is useful to differentiate between two cases: one where the source vertex \( s \) is part of \( \mathcal{C} \) and another where it is not.

If the source vertex \( s \) is an element of the vertex cover \( \mathcal{C} \) (and thereby of \( \text{DIST}_G \)), the distance between \( s \) and every vertex \( v \in \mathcal{C} \) is available in \( \text{DIST}_G \). In the case that \( s \) is not part of \( \mathcal{C} \), a new vertex cover \( \mathcal{C}' \) of \( G \) is constructed, adding \( s \) to \( \mathcal{C} \). The distance matrix \( \text{DIST}_G \) is subsequently updated to include \( \text{DIST}_G(s,\mathcal{C}') \) — ergo, for \( \forall v \in \mathcal{C}' \), dist\(_{\mathcal{C}}(s,v) \) is computed and stored.

Similarly, the destination vertex \( v \) in dist\(_{\mathcal{C}}(s,v) \) can be part of \( \mathcal{C} \), or not. The value can be retrieved from \( \text{DIST}_G(s,v) \) in the first case. If \( v \notin \mathcal{C} \), there exists no entry \( \text{DIST}_G(s,v) \), but all adjacent vertices of \( v \) must be part of \( \mathcal{C} \), as discussed in 2.1.

Therefore, given any vertex \( v \in \mathcal{C} \), \( \text{DIST}_G(s,u) \) exists for all vertices \( u \in \text{adj}_G(v) \). Determining the length of a path from \( s \) to \( v \) via an adjacent vertex \( u \) only requires the addition of the edge weight \( \omega_G(u,v) \) to the known value dist\(_{\mathcal{C}}(s,u) \). The distance from \( s \) to \( v \), dist\(_{\mathcal{C}}(s,v) \), is simply the shortest length of all possible paths, i.e. dist\(_{\mathcal{C}}(s,v) = \min\{ \text{dist}_G(s,u) + \omega_G(u,v) : \forall u \in \text{adj}_G(v) \} \).

For example, given the graph \( G \) in Figure 2, the vertex cover \( \mathcal{C} \) is made up of \( \{a, c, d, f, g, h\} \). Let \( \text{DIST}_G \) contain the distance between all vertices in \( \mathcal{C} \) — computed once with an appropriate algorithm and then stored into the distance matrix.

To compute the distance from \( a \) to \( j \), where \( a \in \mathcal{C} \) and \( j \notin \mathcal{C} \), all adjacent vertices of \( j \) are considered, namely \( d \) and \( g \). Since the adjacent vertices must be part of \( \mathcal{C} \), dist\(_{\mathcal{C}}(a,d) \) and dist\(_{\mathcal{C}}(a,g) \) can be looked up in \( \text{DIST}_G \).

Getting the length of a path from \( a \) to \( j \) only requires adding one additional weight to a known value. The distance from \( a \) to \( j \) is thus calculated as: dist\(_{\mathcal{C}}(a,j) = \min\{\text{dist}_G(a,d) + \omega_G(d,j), \text{dist}_G(a,g) + \omega_G(g,j)\}\).

Likewise, if the source vertex \( s \) is not part of \( \mathcal{C} \), a new vertex cover \( \mathcal{C}' = \mathcal{C} \cup s \) is created by computing dist\(_{\mathcal{C}}(s,v) \) for \( \forall v \in \mathcal{C} \) with the vertices adjacent of \( s \). For instance, if \( s = e \in \mathcal{C} \), dist\(_{\mathcal{C}}(e,v) \) is computed for all \( v \in \mathcal{C} \) as follows: \( \min\{\omega_G(e,a) + \text{dist}_G(a,v), \omega_G(e,b) + \text{dist}_G(b,v)\} \).

---

Algorithm 1 A Simple VC-Based Index

**Input:** A graph \( G = (\mathcal{V}_G, \mathcal{E}_G, \omega_G) \)

**Output:** A VC-based index, \( \text{DIST}_G \)

1. compute a vertex cover, \( \mathcal{C} \), of \( G \);
2. for each \( u \in \mathcal{C} \) do
   3. compute \( \text{DIST}_G(u,\mathcal{C}) = \{(v, \text{dist}_G(u,v)) : v \in \mathcal{C}\} \);
4. return \( \text{DIST}_G(u,\mathcal{C}) \) for all \( u \in \mathcal{C} \);

**Figure 4: Generation of a Distance Matrix**

---

[1]
3.1 Issues
To return \( \text{dist}_G(s, v) \) for a vertex \( s \in \mathcal{C} \) and all vertices \( v \in V_G \), the distance matrix \( \text{DIST}_G \) only needs to be read once and the graph \( G \) has to be scanned once, so \( O(|\mathcal{C}| + |G|) \) I/Os are required for this case, where \( |G| \) is the size of the graph and \( B \) is the file block size.

However, if \( s \not\in \mathcal{C} \), \( \text{DIST}_G(u, \mathcal{C}) \) needs to be read for each \( u \in \text{adj}_G(s) \) to build the new distance matrix for \( \mathcal{C}' = \mathcal{C} \cup s \). The required number of I/Os is therefore \( O((\deg_G(s) \cdot |\mathcal{C}| + |G|)/B) \), where \( \deg_G(s) \) denotes the degree of \( s \) in the graph \( G \).

Saving the graph as an adjacency list ordered by vertex ID allows less data reads; this approach is efficient when it comes to I/O. Nonetheless, this simple approach still leaves room for improvement. Computing \( \text{dist}_G(u, v) \) for all \( u, v \in \mathcal{C} \) results in a prohibitively large matrix for vertex covers of considerable size.

The large size is of particular concern because the distance matrix has to be regenerated if the graph is altered even in a minor way (e.g., if an edge is removed), as a large portion of the entries in \( \text{DIST}_G \) may be wrong after the modification.

Computing the distance matrix \( \text{DIST}_G(v, \mathcal{C}) \) for all \( v \in \mathcal{C} \) requires a lot of I/O operations, and the space of the resulting matrix \( \text{DIST}_G(v, \mathcal{C}) \) for \( \forall v \in \mathcal{C} \) is \( O(|\mathcal{C}|^2) \). It is clear that this approach does not scale well, especially when working with large graphs.

For instance, a graph showing all editing activities on Wikipedia from its creation until January 2008 contains 5.8 million vertices and 250 million edges, amounting to a handful of terabytes in its uncompressed form. [3]

4. VC-Index
4.1 Overview

The VC-index paradigm aims to alleviate the shortcomings discussed in the previous section. It is a tree-structured index of vertex covers which are increasingly smaller as one travels away from the root of the tree.

Every node in the tree contains a vertex cover \( \mathcal{V} \), based on which a so-called distance graph \( D \) is created. Rather than computing and storing \( \text{dist}_G(u, v) \) for all \( u, v \in \mathcal{V} \) as in the simple approach, the goal is to connect the vertices in a local manner, while guaranteeing that \( \text{dist}_G(u, v) \) can still be computed for any \( u, v \in \mathcal{V} \). Distance graphs are discussed in detail in 4.2.

The root \( \lambda \) of VC-index holds a vertex cover \( \mathcal{C} \) of \( G \) and a distance graph \( D \), where \( \mathcal{C} \) is the vertex set of \( D \). A child node \( N \) of any given node \( M \) in the tree holds a vertex cover \( \mathcal{N} \) of the distance graph \( D_M \) in \( M \). The child \( N \) also contains a distance graph \( D_N \) whose set of vertices is \( \mathcal{N} \).

Furthermore, the vertex covers of the children are constructed such that their union is equal to the vertex cover of the parent.\(^2\) For instance, if the root \( \lambda \) has two children \( X \) and \( Y \), then \( X \) and \( Y \) are vertex covers of \( D \) (the root’s distance graph based on \( \mathcal{C} \)). Their union returns \( \mathcal{C} \), i.e., \( X \cup Y = \mathcal{C} \).

\[ \text{Figure 5: The Layout of VC-index [1]} \]

\(^1\) Note that normal letters refer to the node in the tree, while double-struck letters (such as \( X \) or \( Y \)) refer to the vertex cover of the node.

\(^2\) This is achieved by giving preference to vertices that are not yet in any vertex cover during the construction of another vertex cover.
4.2 Distance Graphs

Given a graph $H = (V_H, E_H, w_H)$ and a vertex cover $\mathbb{X}$, 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: $\forall u, v \in \mathbb{X}$, $\text{dist}_H(u, v) = \text{dist}_X(u, v)$, i.e. $\text{dist}_H(u, v)$ can be computed in $D_x$ for any $u$ and $v$ in $\mathbb{X}$.

The aim of the distance graph $D_x$ is to connect the vertices of $\mathbb{X}$ in a sensible manner and to guarantee that $\text{dist}_H(u, v)$ can be computed for any $u, v \in \mathbb{X}$. It is desirable to keep the number of edges at a minimum as to reduce the size of the graph.\(^3\)

For any $u, v \in \mathbb{X}$, we differentiate all possible paths in $H$ from $u$ to $v$ into two types. Paths which do not contain any intermediate vertex $w \in \mathbb{X}$ fall into the first type, while we consider all paths with one or more intermediate vertices $w \in \mathbb{X}$ to belong to the second type. A path belongs to the first type if there is no intermediate vertex whatsoever.

For example, given the graph $G$ in Figure 2 and the vertex cover $\mathbb{C} = \{a, c, d, f, g, h\}$, the path $(d, j, g)$ is of the first type, seeing as $j \notin \mathbb{C}$. On the other hand, the path $(d, f, g)$ is a second type path, since there is at least one intermediate vertex belonging to $\mathbb{C}$, namely $f$.

Every possible path in $G$ belongs either to the first type or to the second type. Hence, the shortest path $SP_{\mathbb{C}}(u, v)$ for any $u, v \in \mathbb{C}$ must belong to the first or second type.

Paths of the second type are simply concatenations of first type paths. For instance, in the graph $G$ of Figure 2, the path $p = (a, e, d, f, i, c)$ can be easily constructed from first type paths as follows: $(a, e, d) + (d, f) + (f, i, c)$.

Therefore, it is sufficient to only consider paths of the first type to construct a distance graph: each first type path is represented in the distance graph by an edge between the starting and ending vertex. The weight of the edge is the length of the path.

We also note that no path $(u, w, t, v)$ with $u, v \in \mathbb{X}$ and $w, t \notin \mathbb{X}$ exists. Such a path would imply that the edge $(w, t)$ is not covered by any vertex in $\mathbb{X}$, which contradicts to the fact that $\mathbb{X}$ is a vertex cover. Thus, first type paths can have one intermediate vertex at most.

The graph on the left-hand side is a distance graph $D$ of the graph $G$, using the vertex cover $\mathbb{C} = \{a, c, d, f, g, h\}$ of $G$ as the set of vertices.

Note that an edge $(u, v)$ for any $u, v \in \mathbb{C}$ in the distance graph $D$ does not imply that such an edge exists in the original graph $G$. For instance, while there is an edge $(d, g)$ in $D$, this is not the case in $G$. The edge in $D$ corresponds to the first type path $(d, j, g)$ in $G$.

Moreover, as we will see at the end of this section, the edges $(d, f)$ and $(f, h)$ are not part of the final version of the distance graph.

Figure 6: A Distance Graph $D$ of the Graph $G$

---

$^3$ The distance matrix used in the simple approach (section 3) is essentially a graph where an edge $(u, v)$ exists for all vertices $u, v \in \mathbb{X}$, which results in an impractically high number of edges.
For any \( u, v \in \mathbb{X} \), any first type path from \( u \) to \( v \) is either a direct path \( (u, v) \) or a path with one intermediate vertex \( (u, w, v) \), where \( w \not\in \mathbb{X} \). Therefore, the first type paths from a certain vertex can be easily determined by looking at its 2-hop neighborhood.

The algorithm to create a distance graph \( D_\mathbb{X} \) based on a given vertex cover \( \mathbb{X} \) is shown in Figure 7. It iterates over all vertices of the original graph \( H \) and only considers the vertices contained in the vertex cover \( \mathbb{X} \).

For each vertex \( u \) in \( \mathbb{X} \), all adjacent vertices of \( u \) are read from the adjacency list. If an adjacent vertex \( v \) is part of \( \mathbb{X} \) as well, the path \( (u, v) \) is a first type path, so an edge is created from \( u \) to \( v \) in the distance graph, with the path length as weight, i.e. \( \omega_G(u, v) \).

On the other hand, if an adjacent vertex \( v \) is not part of \( \mathbb{X} \), we recall that, in turn, all of its adjacent vertices must be. Thus, the adjacency list is accessed again to retrieve all \( w \in \text{adj}(v) \).

For each vertex \( w \in \text{adj}(v) \), an edge \( (u, w) \) is created in \( D_\mathbb{X} \) to represent the first type path \( (u, v, w) \). The weight of the edge in \( D_\mathbb{X} \) is the path length, namely \( \omega_G(u, v) + \omega_G(v, w) \).

It is possible to have multiple first type paths between two vertices in the original graph. The weight of the edge in \( D_\mathbb{X} \) is only updated if the length of a new path is smaller, such that the shortest first type path is represented by the edge.

For instance, the path \( (a, d) \) in the graph \( G \) of Figure 2 has a length of 3, so an edge \( (a, d) \) with weight 3 is added to \( D \). The path \( (a, e, d) \) in \( G \) is also represented in \( D \) by the edge \( (a, d) \), and since this path’s length is \( \omega_G(a, e) + \omega_G(e, d) = 2 \), the weight of \( (a, d) \) in \( D \) is updated to 2.

The algorithm in Figure 7 has I/O efficiency in mind: accessing the first hop of a vertex is easy, given that the graph is stored in form of an adjacency list and will follow sequentially by iterating over all vertices. However, reading the second hop for adjacent vertices which are not part of \( \mathbb{X} \) is less efficient, since it requires various disk reads at different places to get their adjacency lists.

Once the distance graph \( D_\mathbb{X} \) has been generated, one final manipulation is made to gain a smaller distance graph by ensuring that there are no triangles in \( D_\mathbb{X} \) which do not satisfy triangle inequality. Triangle inequality in graphs is defined as follows.

Given three edges which form a triangle \( \{e_1 = (u, v), e_2 = (u, w), e_3 = (w, v)\} \) in \( D_\mathbb{X} \), for \( \omega(e_1) \geq \omega(e_2) \) and \( \omega(e_3) \geq \omega(e_2) \), the following must be valid: \( \omega(e_1) \leq \omega(e_2) + \omega(e_3) \). If \( \omega(e_1) \) is greater than or equal to \( \omega(e_2) + \omega(e_3) \), the edge \( e_1 \) is removed. After all, \( u \) can reach \( v \) directly via edge \( e_1 = (u, v) \) or through \( e_2 = (u, w) \) and \( e_3 = (w, v) \). If \( \omega(e_1) + \omega(e_2) \leq \omega(e_3) \), \( e_1 \) contains useless (as it is not the shortest path) or redundant information and can therefore be removed.

For example, in the distance graph \( D \) of Figure 6, we consider the following edges which form a triangle: \( e_1 = (d, f), e_2 = (d, g) \) and \( e_3 = (f, g) \). Since \( \omega_D(e_1) > \omega_D(e_2) \) and \( \omega_D(e_2) > \omega_D(e_3) \), \( e_1 \) is removed because \( \omega_D(e_1) > \omega_D(e_2) + \omega_D(e_3) \). Similarly, the edge \( (f, h) \) can be removed, too.

Note that the existence of an edge \( (u, v) \) in a distance graph does not necessarily mean that the edge’s weight corresponds to \( \text{dist}_{\mathbb{X}}(u, v) \). Enforcing triangle inequality is simply a fast manner to remove some unnecessary edges. As we will see in 4.5, omitting triangle inequality would not be an issue in regards to the correctness of query results.
4.3 Overall Index Construction

As mentioned in 4.1, given a node \( X \) with vertex cover \( \mathcal{X} \) and a distance graph \( D_X \) in VC-index, each child \( Y \) will hold a vertex cover \( \mathcal{Y} \) of \( D_X \) and the union of the vertex covers of \( X \)'s children must result in \( \mathcal{X} \). During the creation of VC-index, prioritizing missing vertices to be part of a new child's vertex cover can guarantee that the number of children per node will not go beyond reason.

The pseudo-code to construct VC-index is given in Figure 8. It requires a graph \( G \) as input and a threshold \( \sigma \) to define when the tree should not be grown further at a node, once the node's vertex cover is smaller than the value in \( \sigma \).

Figure 5 shows the layout of the VC-index tree. A vertex cover \( \mathcal{C} \) of \( G \) is generated as discussed in 2.2, based on which a distance graph \( D \) is created, which is then stored into the root \( \lambda \).

Once \( D \) has been stored into \( \lambda \), a child node \( X \) of \( \lambda \) is created, a vertex cover \( \mathcal{X} \) of \( D \) is generated and a distance graph \( D_X \) is computed, using \( \mathcal{X} \) as the set of vertices. \( D_X \) is saved into \( X \).

Children are added to the root as long as there are vertices in \( \mathcal{C} \) which are not in any child’s vertex cover yet, i.e. the goal is that every vertex in the parent’s vertex cover is present in at least one child’s vertex cover.

The procedure of adding children to the node until each of its vertices is part of a child’s vertex cover is repeated for every child of \( \lambda \). This method is applied recursively for every node, until the vertex cover inside a node is smaller than the user-defined threshold \( \sigma \).

At the end of the construction, each vertex in \( \mathcal{C} \), the root’s vertex cover, will be present in at least one leaf node, and every leaf node will contain a distance graph whose vertex set is smaller than the supplied threshold \( \sigma \).

Figure 6 shows the distance graph \( D \),\(^4\) constructed from the graph \( G \) in Figure 2 and the vertex cover \( \mathcal{C} = \{a, c, d, f, g, h\} \), as generated by the algorithm shown in 2.2. Two possible vertex covers of \( D \) are \( \mathcal{X} = \{a, c, d, f\} \) and \( \mathcal{Y} = \{a, c, g, h\} \). The distance graphs \( D_X \) and \( D_Y \) — based on \( \mathcal{X} \) and \( \mathcal{Y} \), respectively — are shown in Figure 9.

---

\(^4\) Note that the edges \((d, f)\) and \((f, h)\) are not part of the final version of \( D \) as they are the longest edge in triangles which do not satisfy triangle inequality (cf. section 4.2).
4.4 Query Processing

To process distance queries of the type dist\(_G(s,v)\) from any vertex \(s \in V_G\) to \(v \in V_G\) with VC-index, let \(\mathcal{C}\) be the vertex cover at the root of VC-index. As such, \(\mathcal{C}\) is a vertex cover of the graph \(G\).

If \(s \in \mathcal{C}\), the procedure starts by locating the distance graph \(D_S\) of \(X\), the smallest leaf containing the source vertex \(s\). If multiple nodes come into question, the node closest to the root is selected.\(^5\)

Finding \(X\) can be reduced to O(1) I/O operations with an external look-up table, e.g. by creating entries during the construction of VC-index: if a node \(X\) is created and its distance graph \(D_X\) is smaller than \(\sigma\), all vertices in \(X\), the vertex set of \(D_X\), which are not yet present in the look-up table could be added to it with a reference to \(X\).

Once \(X\) has been established, dist\(_G(s,v)\) is computed for all \(v \in \mathcal{X}\), where \(\mathcal{X}\) is the vertex set of \(D_X\). As mentioned in 4.2, the existence of an edge \((u,v)\) in \(D_X\) does not necessarily mean that the edge’s weight corresponds to dist\(_G(u,v)\). \(D_X\) must be treated like a normal graph, and a shortest path algorithm such as Dijkstra’s algorithm must be applied to compute the distances.

After computing dist\(_G(s,v)\) for all \(v \in \mathcal{X}\), we can move up to \(X\)'s parent, \(Y\). The distance graph of the parent, \(D_Y\), is loaded. We recall that \(\mathcal{X}\) is a vertex cover of \(D_Y\), and as such, it must hold that for any vertex \(v \in \mathcal{Y} \cap \mathcal{X}\) and all \(u \in \text{adj}_D(v), u \in \mathcal{X}\), so dist\(_G(s,u)\) is known. In other words, for any new vertex \(v\) that is not present in \(\mathcal{X}\), all of the adjacent vertices of \(v\) in \(D_Y\) are part of \(\mathcal{X}\).

Therefore, computing dist\(_G(s,v)\) for any \(v \in \mathcal{Y} \cap \mathcal{X}\) simply requires adding a weight to the known distance of all vertices adjacent to \(v\) in the distance graph \(D_Y\) and then selecting the minimum. Thus, dist\(_G(s,v)\) = min{ dist\(_G(s,u)\) + \(\omega(v,u)\) : \(v \in \text{adj}_D(v)\) }.

The procedure of loading the parent node and computing all unknown distances through adjacent vertices is repeated until the root is reached, at which point dist\(_G(s,v)\) is known for all vertices \(v\) in \(\mathcal{C}\), i.e. for all vertices of the vertex cover at the root. Since \(\mathcal{C}\) is a vertex cover of \(G\), computing the remaining distances is similar to moving up to a parent node in VC-index: for any vertex \(v \in \mathcal{C}\), dist\(_G(s,v)\) = min{ dist\(_G(s,u)\) + \(\omega(u,v)\) : \(u \in \text{adj}_C(v)\) }.

If \(s\) is not part of \(\mathcal{C}\), the distance from \(s\) to all \(v \in V_G\) is computed via the adjacent vertices of \(s\), akin to the simple approach presented in section 3. The distance dist\(_G(s,v)\) to any \(v \in V_G\) is simply the minimum \((\omega(s,u) + \text{dist}(u,v))\) of all \(u \in \text{adj}_C(s)\).

As an example, we will look at how dist\(_G(g,v)\) is processed for all \(v \in V_G\) of the graph \(G\) in Figure 2. Given \(D_g\) and \(D_y\) in Figure 9, we select \(D_g\) to start the computations, as \(D_g\) does not contain \(g\). With an algorithm such as Dijkstra’s [9], we compute dist\(_G(g,v)\) for all \(v\) present in \(\mathcal{Y}\). We establish the following distances: dist\(_G(g,a)\) = 4, dist\(_G(g,c)\) = 2, dist\(_G(g,h)\) = 3.

Then, we move up to the parent of \(D_y\), namely to \(D\), shown in Figure 6. Given \(D \in \mathcal{C} \cap \mathcal{Y}\), we compute the length of all possible paths from \(g\) to \(d\) by considering the adjacent vertices of \(d\) in \(D\). The path of minimum length is the distance from \(g\) to \(d\), so dist\(_G(g,d)\) = min{ dist\(_G(g,a)\) + \(\omega(a,d)\), dist\(_G(g,b)\) + \(\omega(b,d)\), dist\(_G(g,h)\) + \(\omega(h,d)\) } = min{ 4+2, 0+2, 3+1 } = 2, since \(a\), \(g\), and \(h\) are adjacent to \(d\) in \(D\).

We determine the same way that dist\(_G(g,f)\) = 3 and can then move on to the graph \(G\). The distance to vertices that are not in \(\mathcal{C}\) (the vertex set of \(D\), which is a vertex cover of \(G\)) can be calculated by considering their adjacent vertices, which are all elements of \(\mathcal{C}\).

For instance, dist\(_G(g,e)\) = min{ dist\(_G(g,a)\) + \(\omega(a,e)\), dist\(_G(g,d)\) + \(\omega(d,e)\) } = min{ 4+1, 2+1 } = 3. In this manner, an expensive algorithm such as Dijkstra’s algorithm has to be executed only once on a relatively small graph, namely on a distance graph of a leaf node in VC-index.

\(^5\) The vertex cover of two siblings in VC-index may be of different size. Therefore, leaf nodes are not necessarily all on the same level, since no children are created for a node whose vertex cover is smaller than the user-defined threshold \(\sigma\). This situation may occur sooner or later.
5. Performance

After discussing the complexity of VC-index, we will view some experimental results of VC-index, before reaching some concluding statements.

5.1 Complexity

The algorithm in section 2.2 generates a 2-approximate minimum vertex cover of a given graph by scanning the graph once sequentially and maintaining a hash table to temporarily store vertices in order to determine whether or not they will be part of the vertex cover. Scanning the graph requires $O(|H|/B)$ I/Os and maintaining the hash table to generate the vertex cover $\mathbb{C}$ can require $|\mathbb{C}|/2$ memory in the worst case, but in practice it is usually much less.

Generating the distance graph based on the vertex cover uses a block nested-loop join (Algorithm 5 in Figure 7). For a graph $H$, let $b_1 = b_2 = O(|H|/B)$ be the number of disk blocks necessary to read for the outer and inner loop, and $b_3 = O(|D|/B)$ the number of I/Os necessary to write the distance graph $D$ to the disk. If the graph fits in memory, $O(b_1 + b_3)$ I/O operations are required. Otherwise, the algorithm requires $O(b_1 + (b_1/(M/B-2))b_2 + b_3)$ I/O operations.

A precise overall complexity analysis is difficult as the resulting size of VC-index depends on the size and topology of the input graph. However, it is possible to give a worst-case analysis.

Assuming that VC-index has $N$ nodes, let $b = b_1 = b_2 = b_3 = O(|D|/B) = O(|H|/B)$, where $D$ is a distance graph, which is usually far smaller than the original graph $H$. In the worst case, constructing VC-index requires $O(C_{VC} + C_{DG})$, where $C_{VC} = O(b)$ denotes the I/O cost to approximate the minimum vertex cover, and $C_{DG} = O(b_1 + (b_1/(M/B-2))b_2 + b_3)$ is the worst-case I/O cost to compute a distance graph, as mentioned above.

Generating all distance graphs for all $N$ nodes is $O(N(C_{VC} + C_{DG}))$, which is $O((N/(M/B-2))(|H|/B)^2)$ in the very worst case — the complexity analysis assumes that all distance graphs, aside from those in the leaf nodes, do not fit in memory.

5.2 Experimental Results

In order to gain some concrete figures as to the performance of VC-index, the authors conducted experiments on real-world datasets. The system was implemented in C++ and run on a machine with CentOS 5.4, using 4 GB of RAM and an Intel Xeon 2.67GHz CPU.

The following datasets were used:

- **USRN**: a weighted graph representing the full USA road network [6] with 24 million vertices and 58 million edges. The total storage size is 1 GB.
- **Web**: a subgraph of the UK web graph; vertices are pages and the edges represent hyperlinks. The graph was modified to render it undirected. The dataset is 13 GB in size and consists of 106 million vertices with a total of 1,092 million edges.
- **BTC**: an unweighted graph with 165 million vertices and 361 million edges, which puts persons, documents and other objects into relationships such as “has-author” or “has-title” [7]. The graph amounts to a size of 6 GB.

Attempting the simple approach discussed in section 3, namely generating a distance matrix based on a vertex cover of the graph, was not possible for any of the datasets due to their large size.

There are two aspects to the construction of VC-index: generating the vertex covers on the one hand, and computing distance graphs at each node of the tree on the other.

Generating vertex covers with the streaming 2-approximation algorithm shown in section 2 takes very little time compared to the required time to construct the distance graphs. Creating the entire VC-index structure for USRN, Web and BTC took 289; 23,030 and 4,241 seconds, respectively.

The time required to build the tree for USRN is substantially shorter than for the other two graphs due to the smaller size of the graph and also due to the fact that the graph’s maximum vertex degree is considerably smaller.

The storage size of the created indices for USRN, Web and BTC are 1.8, 13.5 and 3.1 GB, respectively. The index for BTC is half the original graph’s size because it has a very small vertex cover.

Figure 10 shows the average time it took to process 20 random distance queries. Since it makes a difference whether or not the source vertex of the query is in VC-index, both cases are listed separately in the table.

In order to provide a comparison for the performance of VC-index, the same queries have been processed using an external-memory breadth-first search algorithm (EM-BFS) or an in-memory shortest distance algorithm (IM-SSdist). It is clear from the vast amount of time these methods require why the authors limited the number of queries they tested to 20.

VC-index performs considerably better. The difference is remarkable on large datasets — while it takes thousands of seconds to process Web and BTC with EM-BFS, VC-index requires mere tens of seconds.

The substantial increase in time in Web for non-VC queries, i.e. queries whose source vertex is not part of the vertex cover, stems from the fact that the dataset is much denser than BTC or USRN.

6. Conclusion

The fact that a distance graph in VC-index only holds local information (namely about the 2-hop neighborhood of a vertex) facilitates the process of maintenance. When a new edge is inserted or an existing one is deleted, the index must be updated to reflect the change. This only requires that the entries be recomputed for the vertices whose set of edges has been altered.

A minor update in the graph therefore doesn’t require the entire tree to be recomputed. Moreover, VC-index solves problems which arise when attempting to apply conventional methods for small graphs to large, disk-resident graphs.

It is a novel disk-based tree which processes shortest path and distance queries efficiently for large graphs. The employed algorithms are designed with I/O efficiency in mind, thus being suitable for graphs so large that they have to be read from the disk.

---

7 Insertion or deletion of a vertex can be treated as a special case of an edge update: after adding or deleting the vertex, all edges from the vertex are also added or deleted.
7. References


