A Linear-Order Based Access Method for Efficient Network Computations

Sung-Ho Woo and Sung-Bong Yang

Department of Computer Science
Yonsei University,
Seoul, Korea, 120-749
{shwoo, yang}@mythos.yonsei.ac.kr

Abstract. It is well known that the disk-I/O costs of network computations, for example, finding a shortest path between two vertices, can be reduced by clustering the connected vertices in the network to the same disk page. This paper studies a network access method which is based from a linear order for the vertices in a given network. In order to compute network queries efficiently, a linear order should preserve a "good" clustering property of a network. For this requirement, we present a hierarchical vertex ordering method which obtains a linear order of the vertices in a simple but effective way. In experiments, various access methods are evaluated by comparing the I/O costs of the network computations. The access method constructed from the proposed ordering method outperforms other methods in most cases.

1 Introduction

Efficient network computations are very crucial in geographical information systems (GIS) and intelligent transportation systems which include various applications in transportation, utility and communication networks. Network data are usually stored in a topological format with vertices, edges, connectivity information along with application-dependent attributes. They are stored on a secondary-storage device such as magnetic disks, because the size of a network is usually quite large. During some network computations, the desired vertices are accessed by transferring the appropriate disk pages into the main memory. If the transferred pages include the vertices required for the next step of the computations, additional access to the disk device can be saved. In general the adjacent vertices to a vertex processed currently are accessed in the later steps of the computation. Therefore, the overall I/O cost for the network computations would be decreased if we cluster the vertices so that the adjacent vertices are stored into the same disk page.

In the literature of recursive query processing, access methods based on topological ordering have been developed for I/O-efficient network traversals. The vertices in a network are sorted in a topological order and stored in an one-dimensional access structure [3, 4, 10, 13, 14, 19]. However, they are limited to a
directed acyclic graph which is a special type of a network. Furthermore, topological orders cannot take advantage of clustering the adjacent vertices to the same disk page.

To store the networks with spatial information (such as coordinates of vertices), spatial access methods have been mainly used in the applications of GIS. There have been several research efforts in the design of spatial access methods for the efficient processing of spatial queries [1, 7, 16]. Since the spatial access methods are optimized for spatial queries based on spatial proximity, they may incur high I/O costs for network computations based on connectivity.

In order to reduce the I/O costs of network computations, sophisticated clustering methods have been proposed to allocate the vertices in a network to disk pages of a given size. The approach of [18] uses $B^+$-tree for a secondary index to facilitate search operations and employs an existing graph partitioning algorithm of [6] to split an overflowed page into two pages. Since the graph partitioning algorithm improves the probability that adjacent vertices are clustered into the same disk page, it is a very efficient access structure for network computations.

This paper focuses on a network access structure which is based on a linear order of the vertices in a given network. In the access structure, the I/O costs of the network computations depend on the property of a linear order. To reduce the I/O costs of the computations, a linear order should preserve a clustering property of the network, so the chances that adjacent vertices are assigned to the same disk page should be increased. Topological ordering and space-filling curves have also been used to achieve a linear order of the vertices in a given network. Topological ordering, such as breadth-first order, depth-first order or their variants, is ill-suited for our purpose since they may visit a different region in the network rather than cluster a dense region. For the networks with spatial information, space-filling curves, such as Z-ordering and Hilbert curve, can be used for ordering the vertices in a network. There have been various researches in the design of linear-ordering based spatial access methods for efficient spatial query processing [1, 7]. However, these approaches limit the opportunities for clustering according to connectivity, since they are based mainly on the spatial proximity.

If a linear order preserves a “good” clustering property of a network, an access method based on a linear order has following advantages:

- The access method can be easily incorporated into the DBMS by using an existing one-dimensional access method. This advantage can be found in the Z-ordering which is one of the few spatial access methods used in commercial database products [7]. Oracle has adopted the technique and offered it as a product.

- By transferring several consecutive pages, the access method can overcome the limitation that the clustering effect gets diminished as the amount of the information required for a vertex increases. Transferring multiple consecutive pages at a time is called the chained-I/O technique which can be implemented without any extra I/O costs compared with a single page transfer. It is available in almost all modern computer systems as well as in DBMS [11].
In this paper, we present a simple yet effective hierarchical vertex ordering (HVO) method. The goal of HVO is to find a linear order that captures a connectivity structure of a given network. HVO generates a series of networks through graph matching to construct a matching tree which represent a hierarchical structure of the clusters in a network. A total order is then obtained by visiting each node in the tree in a top-down manner and by ordering its elements properly. Since each cluster has at most two elements, the ordering of its elements is a simple task. Our experiments evaluate various access methods based on linear-ordering with respect to the I/O costs of the network computations. Linear orders derived by HVO are compared with those by breadth-first search, depth-first search and Z-ordering. A network access structure using a graph partitioning approach [18] is also compared. The results show that the access method based on HVO outperforms other methods in terms of the I/O costs for the network computations.

The rest of this paper is organized as follows. In Section 2, network model and network computations are described. Section 3 presents HVO in detail. In Section 4, an extended version of HVO is described for large networks. Section 5 presents our experimental results and performance comparisons. Section 6 provides the conclusions.

2 Network Model and Network Computations

Consider a network $N = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ represents a set of vertices, and $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ represents a set of edges. Each vertex $v_i$ in $V$ has $(x, y)$ representing the spatial location of the vertex. Each edge $(v_i, v_j)$ in $E$ is an edge that connects $v_i$ and $v_j$, $i \neq j$. In general, networks can be represented in various ways. We model a network as the adjacency-list oriented representation [11]. In this representation, a network is modeled as a list of vertices and each vertex has a record which holds the vertex data, the coordinates and its neighbor-list. The neighbor-list includes a set of edges which are represented by the connected vertex-id and its associated weight.

Linear ordering is to find an one-to-one mapping $\varphi : V \rightarrow \Pi$ such that $\Pi = [\pi_1, \pi_2, \ldots, \pi_n]$, $\varphi(v_i) = \pi_j$ means that $v_i$ is the $j$th vertex in the order and $\pi_j$ represents $v_i$. Let $w_{v_iv_j}$ denote the relative frequency of a query accessing vertices $v_i$ and $v_j$ together. If $(v_i, v_j) \in E$, $w_{v_iv_j}$ is positive. Otherwise, $w_{v_iv_j}$ is zero. This model is proposed on the basis of the available database statistics on access frequencies [18].

The vertices in a network are assumed to be stored in contiguous space on a disk. They are arranged physically on the disk space according to a linear order. To facilitate the searches by vertex-id, $B^+$-tree is used. A network dataset is assumed to be available before the $B^+$-tree is constructed. So, it can be constructed in a bottom-up fashion so that the number of pages needed to store the entire dataset can be minimized. We do not consider the cost for reading the access structure, that is, the pages are assumed to be directly accessible. The data transfer between the main memory and the secondary storage is assumed
to be carried out by a disk page. However, if the chained-I/O technique is used, the transfer-unit is a multiple of the size of a disk page.

One of the typical network computations is the path computation which includes path evaluation and path finding. To compute the path evaluation, the relevant vertices in a given path are retrieved sequentially and the associated attributes are evaluated. Path finding algorithms such as Dijkstra’s shortest path algorithm and $\mathcal{A}^*$ algorithm require high disk access costs, since they consists of iterations in which the adjacent vertices of the currently accessed vertex are required for the next iteration. The I/O cost models of these algorithms are well described in [11, 17].

In a real time application like a car-navigation system, path precomputation methods have been developed for quick responses [2, 12]. They provide a balance between the efficiency of path finding and the feasibility of storage. However, they require that a record of each vertex should include precomputed information. The storage overhead of [2] is $O(|V| + k^2)$ and that of [12] is $O(|V|^2/k)$, where $|V|$ is the number of vertices in the network and $k$ is the number of clusters. As the record size of each vertex increases, the number of vertices that a disk page can hold decreases. This reduces the effect of clustering the adjacent vertices to disk pages. In this case, the chained-I/O may improve the access structure based on the linear orders by transferring consecutive disk pages without any extra I/O costs.

3 Hierarchical Vertex Ordering

In this section, we illustrate HVO in detail. HVO has two steps: 1) Constructing a matching tree $T$ and 2) hierarchical ordering. In the first step, a graph matching is used to construct a matching tree. In the second step, a vertex order is achieved by visiting each nonleaf node from the highest level and by ordering the children of the currently visited node.

3.1 Constructing a Matching Tree

The algorithm for constructing $T$ is described in Figure 1. ConstrHierarchy() accepts $N = (V, E)$ as input and returns $T$ along with a series of networks $N^0, N^1, \ldots, N^h$, where $N^0 = N$. For $1 \leq t \leq h$, $N^t = (V^t, E^t)$ is constructed from $N^{t-1} = (V^{t-1}, E^{t-1})$. We call $N^t$ the supernetwork of $N^{t-1}$. The vertex set of $N^t$ is represented by $V^t = \{v^t_1, v^t_2, \ldots, v^t_{n^t}\}$, where $n^t = |V^t|$. Vertex $v^t_i$ is created by merging $v^{t-1}_a$ and $v^{t-1}_b$ (or possibly only one vertex $v^{t-1}_a$) in $N^{t-1}$ as shown in Figure 2 (a), so we call $v^t_i$ the supervertex of $v^{t-1}_a$ and $v^{t-1}_b$.

To generate $N^t$ from $N^{t-1}$, all the vertices in $N^{t-1}$ are initialized to unmatched (line 4). In the inner while loop (lines 6-22), an unmatched vertex $v^{t-1}_a$ is selected randomly (line 7) and one of its unmatched adjacent vertex $v^{t-1}_b$ is selected to be paired with $v^{t-1}_a$ (line 11). $v^{t-1}_a$ and $v^{t-1}_b$ become the children of supervertex $v^t_i$ in $T$ (line 13) as shown in Figure 2 (b). Note that HVO does not care who becomes the left or the right child of $v^t_i$. If there is no unmatched vertex
**ConstrHierarchy(N)**

Input: \( N = (V, E) \)
Output: \( N^0, N^1, \ldots, N^h \) and \( T \)

1. \( l = 1; \)
2. While \( |V^{l-1}| > 2 \) do
   3. Generate a supernetwork \( N^l = (V^l, E^l) \) with \( V^l = \emptyset \) and \( E^l = \emptyset; \)
   4. Each vertex in \( V^{l-1} \) is marked with \textit{unmatched};
   5. \( i = 1; \)
   6. While there exists an \textit{unmatched} vertex in \( V^{l-1} \) do
      7. Randomly select an \textit{unmatched} vertex \( v^{l-1}_a; \)
      8. Generate a new supervertex \( v^l_a; \)
      9. \( i = i + 1; \)
     10. Add \( v^l_a \) to \( V^l; \)
    11. If there exist \textit{unmatched} vertices among the neighbors of \( v^{l-1}_a \) then
        12. \( v^{l-1}_a \) is paired with \( v^{l-1}_b \), where \( w_{v^{l-1}_a-v^{l-1}_b} \) is the largest among \( v^{l-1}_a \)'s \textit{unmatched} adjacent vertices;
        13. \( v^{l-1}_a \) and \( v^{l-1}_b \) are marked with \textit{matched};
        14. \( v^{l-1}_a \) left = \( v^{l-1}_b \), right = \( v^{l-1}_a \);
    15. else
        16. \( v^{l-1}_a \) is marked with \textit{matched};
        17. \( v^{l-1}_a \) left = \( v^{l-1}_b \), right = NIL;
    18. endif
   19. For each \( v^l_j \) which has a child (or children) adjacent to either \( v^l_j \)_left or \( v^l_j \)_right
      20. Add \( (v^l_j, v^l_j) \) to \( E^l; \)
      21. \( w_{v^l_j v^l_j} = w_{v^l_j \text{left} v^l_j \text{left}} + w_{v^l_j \text{right} v^l_j \text{left}} + w_{v^l_j \text{left} v^l_j \text{right}} + w_{v^l_j \text{right} v^l_j \text{right}}; \)
   22. endwhile
3. \( l = l + 1; \)
4. endwhile

**Fig. 1.** A pseudo-code for constructing a matching tree \( T \)
Fig. 2. A series of networks and the matching tree

among the adjacent vertices (line 14), \( v_l^i \) will have a single child in \( T \) (line 16).
If \( v_j^i \in V^i \) has a child (or children) adjacent to either \( v_l^i \).left or \( v_l^i \).right (line 18), then \((v_l^i, v_j^i)\) is added to \( E^i \) (line 19) and \( w_{v_j^i[v_j^i]} \) is the sum of the weights between each child of \( v_l^i \) and each child of \( v_j^i \) (line 20). For the sake of simplicity, we assume that the weight value associated with a NIL pointer is zero.

### 3.2 Hierarchical Ordering

In the second step of HVO, a vertex order \( \Pi^h \) of \( N^h \), from \( l = h \) down to 1, is determined. Let \( \Pi^l \) be \( [\pi^l_1, \pi^l_2, \ldots, \pi^l_{n^h}] \). As shown in Figure 3, \( \Pi^h \) is decided first (line 1). Since \( N^h \) has only two vertices, the order between these two vertices may be determined arbitrarily.

Once \( \Pi^h \) is obtained, \( \Pi^{h-1} \) is determined by visiting \( \pi^h \) in \( T \) first and then by visiting \( \pi^h \). In general, when \( \pi^l_1 \) is visited, its children on level \( l - 1 \) are ordered according to the connectivities between the left or right child of \( \pi^l_1 \) and each of the previous node \( \pi^l_{i-1} \) and the next node \( \pi^l_{i+1} \) of \( \pi^l_1 \) in \( \Pi^l \). We call these four possible connectivities LPW, LNW, RPW, and RNW as shown in Figure 4.

If the value of (LPW-LNW) is positive, the left child \( \pi^l_1\.left \) is more tightly connected to the children of \( \pi^l_{i-1} \) than those of \( \pi^l_{i+1} \). Similarly, if (RPW-RNW) is positive, the right child \( \pi^l_1\.right \) is more tightly connected to the children of \( \pi^l_{i+1} \) than those of \( \pi^l_{i-1} \). In order to determine the order between the two children, \( \pi^l_1\.left \) and \( \pi^l_1\.right \), we compare the value of (LPW-LNW) with that of (RPW-RNW) (line 9). If (LPW-LNW) \( \geq \) (RPW-RNW), then \( \pi^l_1\.left \) becomes \( \pi^l_{j-1} \) and \( \pi^l_1\.right \) is placed at the next position, that is, \( \pi^l_{j+1} \) in \( \Pi^{h-1} \) (line 10).

Otherwise \( \pi^l_1\.right \) becomes \( \pi^l_{j-1} \) and \( \pi^l_1\.left \) becomes \( \pi^l_{j+1} \) in \( \Pi^{h-1} \) (line 12).

Note that when (LPW-LNW) \( = \) (RPW-RNW), \( \pi^l_1\.left \) and \( \pi^l_1\.right \) in \( \Pi^{h-1} \) can be ordered arbitrarily.
After all the nodes on level $l$ are visited according to the sequence of $H^l$, we obtain $H^{l-1}$. In this way, visiting each non-leaf node of $T$ and ordering its children are iterated level by level. Then $H^0$ can be achieved finally as a vertex order of the given network $N$.

$\textbf{Hierarchical
Ordering()}$

\textbf{Input}: $T$ and $N^0, N^1, \ldots, N^n$

\textbf{Output}: $H : [x_1, x_2, \ldots, x_n]$

1. $x_1 = v_1^1; \quad x_2 = v_2^1$
2. For $l = h$ down to 1
   3. $i = 1; \quad j = 1; \quad /\star$ Note that $i$ is used for $x_i^l$ and $j$ is used for $x_j^{l-1}$. $/\star$
   4. For $i = 1$ to $n_1$
      5. If $x_i^l.right = NIL$ then /* $x_i^l$ has only one child. */  
         $x_j^{l-1} = x_i^l.left; \quad j = j + 1;$
      6. else
         7. $LPW = w_{x_i^l.left} + w_{x_i^l.left, left + w_{x_i^{l-1}, right}'}$
         8. $LNW = w_{x_i^l.left} + w_{x_i^{l-1}, left; + w_{x_i^{l-1}, right}'}$
         9. $RPW = w_{x_i^{l-1}, right} + w_{x_i^{l-1}, right; + w_{x_i^{l-1}, right}'}$
         10. $RNW = w_{x_i^{l-1}, right} + w_{x_i^{l-1}, right; + w_{x_i^{l-1}, right}'}$
      11. If $(RPW - RNW) < (LPW - LNW)$ then
         12. $x_j^{l-1} = x_i^l.left; \quad j = j + 1; \quad x_j^{l-1} = x_i^l.right$
      13. else
         14. $x_j^{l-1} = x_i^l.right; \quad j = j + 1; \quad x_j^{l-1} = x_i^l.left$
      15. endif
      16. endif
      17. endfor

\textbf{Fig. 3.} A pseudo-code for hierarchical ordering

4 Multi-level HVO

HVO requires matching computation and a data structure for the tree. If the size of a network is very large and the available main memory is limited, it may not be possible to find a linear order by loading the whole network into main memory. Therefore, we extend HVO to have multi-levels of a network. The main idea is to divide the vertices in a network into several clusters. Then a super-network whose vertices correspond to the obtained clusters is generated. To achieve a total order for a network, two kinds of linear orders are required: i) a linear order of a super-network and ii) a linear order of each cluster. The total order can be achieved by concatenating the linear orders of all the clusters according to the linear order of the super-network.
Fig. 4. Criteria for the ordering policy

<table>
<thead>
<tr>
<th>Networks</th>
<th>Vertices</th>
<th>Edges</th>
<th>Degrees (Max/Avg/Min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airfoil</td>
<td>4,253</td>
<td>12,289</td>
<td>9 / 5.7 / 3</td>
</tr>
<tr>
<td>FFT9</td>
<td>5,120</td>
<td>9,216</td>
<td>4 / 3.6 / 2</td>
</tr>
<tr>
<td>N1</td>
<td>9,000</td>
<td>37,522</td>
<td>18 / 8.3 / 2</td>
</tr>
<tr>
<td>N2</td>
<td>9,000</td>
<td>23,268</td>
<td>15 / 5.2 / 1</td>
</tr>
</tbody>
</table>

Table 1. Sample network datasets

For the multi-level HVO, we should solve the problem of how to partition the vertices into several clusters of the same size in such a way that for any pair of vertices in a cluster there exists a path. In [2], several heuristics for the clustering are presented in the context of path computations.

5 Experimental Results

In this section, various network access methods are evaluated by comparing the I/O costs of the network computations. The access method by HVO (HVO-AM) is compared with those by BFS (BFS-AM), DFS (DFS-AM) and Z-ordering (Z-AM). BFS-AM and DFS-AM arrange the vertices by traversing the network with BFS and DFS schemes from a random start vertex, respectively. The topological ordering based access methods in [14] can be extended for general networks. Z-AM is implemented by the bits interleaving technique for the binary representation of vertex coordinates. We also compare a network access method, called a connectivity-clustered access method (CCAM), which uses a graph partitioning algorithm in [6]. We implemented CCAM using dynamic clustering for the creation of access structures and the secondary reorganization policy for the reflection of the changes by insert operations.

We assume that the consecutive disk pages are reserved sufficiently to store all the entire network. Therefore, the vertices can be stored contiguously on the disk according to a linear order. To measure the I/O costs, the number A of
page accesses and the number $M$ of page misses are measured. The I/O costs are compared by the page miss ratio which is computed by $M/A$.

The experiments are conducted on four sample networks: Airfoil, FFT9, N1, and N2. The characteristics of the sample networks are given in Table 1. Airfoil and FFT are from a network partitioning library, called CHACO [9]. N1 and N2 are randomly generated networks. In N1, two vertices located within a predetermined distance are connected with a certain probability. Any pair of vertices in N2 are connected by an edge only with a fixed probability. Figure 5 shows how the experiments are performed.

The experimental results are presented in the following three subsections. First, the I/O costs are compared when the path computations are implemented by Dijkstra's algorithm, $A^*$ algorithm and precomputing method in [2] for finding the shortest path between two vertices. Second, the I/O costs are compared for path evaluation. Finally, the effect of the buffer capacity and the page size are evaluated on the network computations.

5.1 The I/O Costs for Path Computations

This section presents the comparisons of the I/O costs of the path computations which are implemented by Dijkstra's algorithm, $A^*$ algorithm with the Euclidean distance heuristic and precomputing method in [2]. The page size is set to 8KB and the size of internal buffer is 8 times larger than that of a disk page. The index pages are excluded from calculating the page miss ratio.

Figure 6 (a) presents the I/O costs when the paths are computed by Dijkstra's algorithm. The paths are computed for randomly selected 10 pairs of vertices each of which represent the start vertex and the terminal vertex, respectively. The I/O costs are presented by the average page miss ratio for computing the 10 paths on each network. HVO-AM outperforms other methods on all the networks.
Figure 6 (b) presents the average page miss ratios for finding the paths when the paths are computed by $A^*$ algorithm. HVO-AM outperforms other methods except for CCAM on network FFT9. Even though we do not show all the experiments, HVO-AM outperformed CCAM on FFT9 for different page sizes and internal buffer sizes. In subsection 5.3 we show the performance comparison of the access methods on the average over various sizes of a page and the internal buffer.

The path precomputing method in [2] is implemented and the I/O costs comparisons are presented in Figure 6 (c). A network is divided into $k$ clusters by the heuristic presented in [2], where $k$ is the square root of the number of vertices in a given network. The precomputed information includes the weight of the shortest distance between each vertex and the center vertex of each cluster. For each center vertex, the weights of all shortest paths among the center vertices should also be computed and stored. The precomputed information should be stored within the record of the associated vertex. In this approach, the path computa-


The I/O costs for path evaluation

In this section, the I/O costs of path evaluation are compared. Paths are generated by random walks on the network. Two sets of paths are generated according to their lengths. Each set includes 10 paths. Short paths have 100 – 300 vertices while long paths have 1000 – 1500 vertices. Figure 7 presents the average page miss ratio for each set of paths. The page size is set to 8KB and the capacity of the internal buffer is $8 \times P$. HVO-AM outperforms other methods except for
Z-AM on the short paths in Airfoil. Since the network Airfoil has a characteristic that the connectivity among the vertices is closely related to their spatial proximity, the performance of Z-AM which is based on the space-filling curve is improved on Airfoil in most experiments.

5.3 The Effect of Buffer Capacity and Page Size

In this section, the effect of the capacity of the internal buffer and the size of a disk page are evaluated. The I/O costs are compared when the path computations discussed in Section 5.1 are performed. First, the I/O costs are compared as the size of a disk page varies. In this experiment, the capacity of the internal buffer is fixed to 32KB. Figure 8 shows the average page miss ratios for all the networks. HVO-AM shows better performance than other methods in all the cases.

Figure 9 shows the average page miss ratios for all the networks as the capacity of the internal buffer changes. The size of a disk page is set to 4KB. As a whole, we can observe that the average page miss ratios of all the access methods decrease as the capacity of the internal buffer increases. HVO-AM outperforms other methods consistently.

6 Conclusion

In this paper, we presented a linear ordering method which finds a vertex order through a matching tree and hierarchical ordering. The vertices are arranged in the one-dimensional disk space according to the achieved order. An one-dimensional access structure is constructed to facilitate search operations. In this structure, the linear order should preserve a “good” clustering property of a network in order to reduce the I/O costs for the network computations. Our experiments confirm that the access structure derived from HVO is superior to other methods in terms of the I/O costs. Our future work includes the design of a access method for the dynamic environment in which vertices and edges are frequently updated.

References

Fig. 8. The effect of page size
Fig. 9. The effect of buffer capacity