External Memory Algorithms for Shortest Distance Queries

Sandeep Gupta Chinya V. Ravishankar
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{sandeep,ravi}@cs.ucr.edu

Abstract

Efficient computation of shortest distances over large static graphs is vital in several applications including web modeling, large communication systems, network analysis, Intelligent Transportation system and advanced spatio-temporal databases.

A natural strategy is to preprocess the graph so that distance can be answered efficiently. Such a scheme should have following properties: (a) the data structure from preprocessing should be small (b) CPU and Disk I/O cost for per shortest distance computation be low and (c) it should provide worst case performance guarantees for both storage and distance computation cost. Current techniques lack one or more of these properties.

We present two algorithms to bridge this gap for graphs that can be partitioned efficiently. Our experiments with real-world and synthetic datasets show that our algorithms are efficient and represent a tunable mechanism for making trade-offs between CPU costs and storage requirements.

1. Introduction

Shortest distance queries arise frequently in numerous scenarios, including web modeling, large communication systems, transportation systems, search and rescue, water flow analysis, games and robotics. Its research has long and rich history starting with Dijkstra's celebrated algorithm. The graphs in such applications are often static, allowing us the liberty of offline preprocessing to improve the performance of shortest-distance queries. Work such as [3, 6, 7, 11] falls into this category.

However, such graphs are often very large, so they must be stored in external memory. In such cases, our algorithms must be both CPU- and disk-efficient. Several researchers have addressed this problem [1, 9, 12, 19, 21, 22, 23].

Motivation

We focus on two application scenarios where it is important to compute shortest distances efficiently over large graphs. In spatial modeling and network analysis, a given topography is represented as a triangulated planar graph (as a *triangulated irregular network* or TIN). A typical objective is to connect several predetermined locations by new roads or rails. Edge weights in such graphs represent some cost function such as travel time or construction cost. Shortest distance computation helps determine the most efficient connection layout.

Similarly, efficient shortest-distance computation is essential in navigation systems (or Intelligent Transportation Systems) and advanced spatio-temporal databases. Consider a typical fleet management system consisting of a road network, a set of warehouses, and a fleet of trucks. Typical queries may be "Find all warehouses that are within 5 km from a current position of the truck A" or "Which truck will be closest to a given warehouse at time t_1". As observed in [19, 9] the bottleneck many such spatio-temporal queries is finding the shortest distances among pairs of nodes of the road network.

CPU-Storage Tradeoff Issues

Current algorithms for shortest distance computation turn out not to have the properties that we require. Dijkstra's celebrated algorithm [4] takes $O(n)$ space and $O(n \log n)$ time, when implemented with heaps. An $O(n\sqrt{\log n})$ time algorithm [5] that works for planar graphs is the most practical and efficient of all the algorithms available.

However, since we would like to be able to handle very large networks, we would prefer response times even smaller than provided by current methods. One simple approach might be to compute shortest paths in advance and do a simple table lookup to obtain $O(1)$ response times. However, this method becomes impractical, since a network with just 100,000 nodes would require a table with five billion entries.

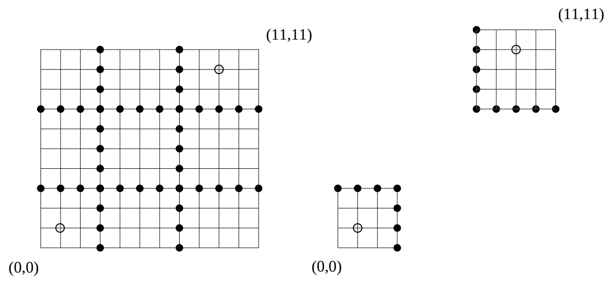


Figure 1. Grid Graph with labeled nodes

Our Contribution

Gavoille et. al. [6] presented a labeling scheme that assigns labels to nodes of the given graph such that shortest distance between any two nodes can be computed using only their labels. For planar graphs, this scheme yields labels of size $O(\sqrt{n}\log n)$, from which shortest distances may be computed in time in $O(\log n)$. While this appears efficient, the storage overhead of $O(n\sqrt{n}\log n)$ still is too high for it to be applied to very large graphs.

To put this in perspective in our experiments a graph of 25000 nodes which can be stored with less than 3 MB of space, its labeling storage cost however was 72 MB. For larger graphs our system ran out of main memory resources to compute the labels for all nodes.

Our algorithms aim to bridge this gap, and yield a data structure that reduces storage overhead without substantially sacrificing shortest distance computation efficiency. Moreover, the distance decoding is disk-efficient and also bounds the worst case performance.

In this work we present two techniques, namely DL and NDL, to this end over graphs that can be partitioned efficiently. As in [6], we assign labels to nodes and define a decoding function on the labels. Our technique uses a preprocessing phase that builds an external memory data structure and a distance decoder which computes shortest distance between any two given nodes of the graph.

Our data-structure consists of several partitions of the input graph (referred to as *fragments* in this work) and the node labels, and our distance decoder combines Dijkstra and a decoding function on the labels. Our data structure has low storage overhead and the distance-decoder is both CPU and disk-efficient.

For the same graph mentioned before the storage cost from DL and NDL is 4.2MB and 3.2MB respectively. The distance computation on average fell marginally, when compared with that of Gavoille's, from 0.1 second to 0.3 second for both the algorithms. On the other extreme Dijkstra's algorithm which does not have any storage overhead took more than 14 seconds with same buffer size.

1.1. Intuition

We present the intuition behind our algorithms using the simple example of a grid graph (see Figure 1). In the algorithm of [6], every node is assigned a label during preprocessing, such that shortest distance between any two nodes can be computed only from there labels.

This method is elegant, but not directly useful in practice since the label sizes are large, although the distance computation is very efficient. The nodes are themselves relatively small, so fetching them incurs a small disk I/O overhead, in comparison.

DL Algorithm

To reduce storage cost, our approach is to label only a subset of nodes, say, the ones shown as dark circles. Shortest distance computation between two labeled nodes proceeds as in [6], and is cheap. Shortest distance computation between non-labeled nodes (empty circles) proceeds as follows.

We define a *fragment* as a connected subgraph that is shielded from the remaining graph by labeled nodes. In Figure 1 the right side shows two fragments. We proceed with a Dijkstra style expansions from the unlabeled source and destination nodes u and v, but limit the expansion to the fragments that contain them.

Let $\langle u_1, \ldots, u_t\rangle$ and $\langle v_1, \ldots, v_k\rangle$ be the labeled nodes bounding u and v, respectively, and let $d(u, u_i)$ and $d(v, v_i)$ be the shortest distances derived from this expansion. Now, the shortest distance between u and v is given by

$$d(u,v) = \min_{\substack{1\le i\le t\\ 1\le j\le k}} \{d(u,u_i) + d(v,v_j) + |L_{u_i}, L_{v_j}|\}, \quad (1)$$

if $|L_{u_i}, L_{v_j}|$ is the distance decoder function of over labels.

Our approach must address several conflicting goals. We need the the number of labeled nodes to be small as well as the labels themselves to be short to reduce the space overhead. To reduce the CPU costs of Dijkstra, we need the fragment sizes and the number of boundary nodes for any fragment to be small.

NDL Algorithm

Conceptually NDL-algorithm is a nested DL-algorithm. It has an outer and inner phase along with a bridge phase to connect both the phases.

In the outer phase, similar to DL algorithm we select are subset of nodes (we refer to these as o-nodes) and label them. Let L^o_u denote the label of o-node u. We define l-fragment as a connected subgraph that is shielded from remaining graph by o-nodes. Figure 1.1 shows grid graph, the corresponding l-fragments and their boundary o-nodes.

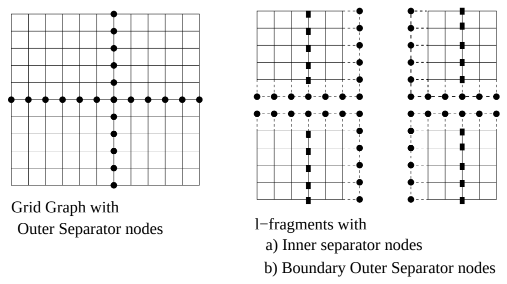
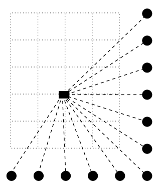


Figure 2. NDL decomposition over grid graph.

In the inner phase we apply DL labeling independently on each of the l-fragment: i.e. a subset of carefully selected nodes of the l-fragment is assigned labels such that given two labeled nodes from the same l-fragment we can decode shortest distance between them from their labels.

Nodes labeled during inner phase is called i-nodes. Let L^i_u be the label for i-node u. Figure 1.1 shows the inner-labeled nodes for each of the l-fragment generated in outer phase. As before a connected subgraph shielded by i-nodes form the rest of l-fragment is called a fragment.

We would like to note that distance decoding between two inner-labeled nodes is applicable only if they belong to the same l-fragment.



In the bridge phase each i-node u is assigned a bridge label L^b_u. L^b_u maintains distance to all o-nodes that shields the l-fragment of u from the rest of the graph as shown in adjacent figure. If x is a o-node shielding the l-fragment of i-node u then let $L^b_u(x)$ be the stored shortest distance from u to x.

The distance decoding of the NDL-algorithm is a nested variation of the DL decoding algorithm with some addition that occurs due to bridge labels. The overall algorithm is bit involved because there are several different scenarios to handle. To present the underlying concept we will discuss only one particular scenario. The exact decoding procedure is discussed later in the paper:

Suppose u and v are nodes that belong to different l-fragments and are neither i-node or o-nodes, referred to as n-nodes, then distance decoding proceeds as follows: Let $\langle u_1, \ldots, u_t\rangle$ and $\langle v_1, \ldots, v_k\rangle$ be the o-nodes bounding u and v, respectively, and let $d(u, u_i)$ and $d(v, v_i)$ be the shortest distance. Now, similar to DL-distance decoding in equation 2, the shortest distance between u and v is given by

$$d(u, v) = \min_{\substack{1\le i\le t\\ 1\le j\le k}} \{d(u, u_i) + d(v, v_j) + |L^o_{u_i}, L^o_{v_j}|\}, \quad (2)$$

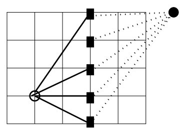
if $|L^o_u, L^o_v|$ is the distance decoder function over labels. This is the outer distance decoding.

But we haven't precomputed $d(u, u_i)$ or $d(v, v_i)$. In the DL-algorithm we would run Dijkstra on the corresponding l-fragment and compute it online. This is costly as l-fragments are significantly large. Instead NDL adopts following technique:

To find $d(w, x)$ where w is a n-node and x is a o-node we proceed as follows: Let F be the fragment that contains w and let $\langle w_1, \ldots, w_t\rangle$ be the i-nodes bounding w in its l-fragment.

Let $d_F(w, w_i)$ be the shortest distance between w and w_i. Then the shortest distance between w and x is given by

$$d(w, x) = \min_{\substack{1\le i\le t\\ 1\le j\le k}} \{d_F(w, w_i) + L^b_{w_i}(x))\}, \quad (3)$$



where $d_F(w, w_i)$ is computed using Dijkstra's algorithm on the fragment F. $L^b_{w_i}(x)$ is the shortest distance from i-node w_i and bounding o-node x. Adjacent figure depicts this execution on one of the non-labeled nodes (shown as empty circle) for the example grid graph. The solid lines depict shortest distance computed by running Dijkstra's algorithm and the dashed lines are shortest distance stored in the bridge label of respective i-nodes.

To obtain lower storage overhead and efficient decoding for the NDL-algorithm we have to simultaneously minimize the number of boundary outer-labeled nodes of the l-fragments, number of boundary inner-labeled node of fragments, the size of l-fragments.

Our algorithm DL and NDL strive to balance these parameters so that distance computation in a external memory system is efficient with small storage overhead. Moreover we provide asymptotic bounds between the inter-play of these parameters if the graph is planar.

2. Previous Work

Shortest paths and distances on graph has been a topic of much research. Here we focus on specifically on work that preprocess graphs to quickly answer shortest distances. We would look at algorithms designed to work in main memory and external memory. Since our scheme relies heavily on separators we would briefly review their literature, then move on to describe application where preprocessing is necessary and then discuss work closely related to this work.

2.1. Separators For Planar Graphs

We will briefly review some key results regarding separators. Lipton and Tarjan's seminal work in [16] showed that that all planar graphs admit separators of size at most

$O(\sqrt{n})$. Several problems, such as shortest path and TSP required separators to be of $O(\sqrt{n})$ size and to also satisfy additional constraint. Fredrickson's [8] algorithm, given a parameter r, yields separators that breaks the graph into $O(n/r)$ regions with $O(r)$ vertices each and $O(\sqrt{n}/r)$ boundary vertices in total. He used this separator to obtain $O(\sqrt{n})$ time algorithm to find shortest distance on planar graphs.

Finally, separators were applied towards labeling graph [6] which we have discussed at greater length since it's concept is central to this paper.

2.2. Application of Shortest Distance

Shahabi et al. [20] solve the k-nearest neighbor problem with distance measured as the shortest path on the road network, using a modified Lipschitz embedding [15] which approximately preserves shortest distances on the road network. Restricted motion was also addressed by Papadias et al. [19], who use R-trees over the road network to prune the search space. A data model and algorithm to answer k-NN queries to objects moving along the road network appears in [13], and [23] addresses the problem of finding the inroute nearest neighbor. The project PLACE [18] is a framework for location-based services and supports many spatial and spatio-temporal queries. To make the system scalable, it uses incremental evaluation mechanism for concurrently answering several continuous spatio-temporal queries.

2.3. Main Memory Algorithms

We begin with the well-known algorithm by Dijkstra [4] that require $O(n)$ storage and $O(n \log n)$ time when implemented with heaps. An $O(n\sqrt{\log n})$ time algorithm [5] that works for planar graphs is the most practical and efficient of all the algorithms available.

Gavoille et al. [6] addressed the problem of labeling the nodes of a graph to allow computation of distances directly from their labels. They show that given any distance labeling scheme on planar graphs, there exist graphs for which the labels need $\Omega(n^{1/3})$ bits per node. They give an encoding scheme based on separators that assigns labels of length $O(\sqrt{n}\log n)$ to each vertex, with the distance computable from the labels in time $O(\log n)$. Their labeling scheme takes O($n\sqrt{n}\log n$) time to compute the labels.

Similar to [6] the authors in [3] assign labels so that reachability and shortest distance queries be answered efficiently. They focus mainly on directed graphs and is based on the concept of 2-hop covers. A 2-hop cover is a subset of path such that shortest path between any two nodes can be represented as concatenation of two paths from the subset. Finding 2-hop cover is NP-hard. However the authors give practical algorithms to find such labels for general graphs.

Goldberg and Harrelson [7] propose to optimize A*-search on graphs as follows: They select a random subset of nodes, lets call them beacons, and each node maintains distances to all the beacons. The search for shortest path is guided and pruned based on the triangle inequality between the source, destination, visited node and the beacons.

2.4. External Memory Algorithms

When the graph is stored on external memory, Dijkstra's algorithm performs very poorly as there is little or no locality of access. Hutchinson et. al. [12] gave an external memory data structure to speed up shortest distance queries for planar graphs. They first propose a blocking for rooted trees with small space overhead to support disk efficient traversal. Using this they provide efficient techniques to find separators in external memory model. Overall it requires $O(n\sqrt{n})$ space and $O(\sqrt{n})$ disk access per distance computation (ignoring the constants the appear due to disk page size).

In the area of Intelligent Transportation System, a several heuristics have been proposed to optimize shortest distance queries over road network (see [21, 1] and references therein). All of them partition the graph into several fragments, create compressed graphs over the resulting boundary nodes. Shortest distances are answered using Dijkstra over the fragment and subgraph. They differ in their approach of partition, the depth of recursion (i.e. if the compressed subgraphs are further partitioned and sub-subgraph is created). However, as in Dijkstra there is little or no locality of access which makes computation over large graphs very costly.

3. Structure of the Paper

In Section 4 we present notations and definition used in this work. Before we present our algorithms we will describe a hierarchical recursive decomposition (or HRD) of the input graph into almost equal size fragments. The concept of HRD is central to our work and also to the labeling scheme of [6]. A byproduct of HRD is a data structure called separator-tree. The data-structure is an abstract notion which we will use to describe our algorithms. For the rest of the paper when we refer to separator-tree we imply the corresponding HRD.

Then in section 5 we describe the labeling of [6] that takes as input a separator-tree and yields label for all the nodes of the graph. The size of a label and the cost of distance decoding is a function of separator tree.

Next we present our DL-algorithm in section 6. It selects a subset of nodes, based on the input separator tree, to label in preprocessing phase. The DL-distance decoder is a combination of Dijkstra and distance decoding of [6].

The NDL-algorithm described in Section 7 has three labeling phases, outer, inner and bridge. In section 7.2 we describe the NDL distance decoding algorithm.

4. Definitions and Notations

Let $G = (V(G), E(G), W(G), X(G))$ be an undirected graph, where $V(G) = \{v_1, \ldots, v_n\}$ and $E(G) = \{e_1, \ldots, e_m\}$ are the set of vertices and the set of edges, respectively. Edges and vertices are assigned weights by the functions $W(G) : E(G) \rightarrow \mathbb{R}^+$ and $X(G) : V \rightarrow \mathbb{R}^+$ respectively.

The input graph is G^0. Subgraph H of G has vertex weight function
$V(H)(u) = \frac{1}{|V(H)|}$, $\forall u \in V'$ and edge weight function $W(H)(e) = W(G^0)(e)$, $\forall e \in E'$.

We say $S^v \subset V$ is a *vertex separator* for graph G if its removal partitions the remaining vertices into two or more disconnected components $\{G/S^v\}_1, \ldots, \{G/S^v\}_k$. To each vertex in S^v, we assign a unique integer index in the range $[0, |S^v| - 1]$. Let $S^v[i]$ denote the vertex with index i

If G' is a connected subgraph of graph G^0 and $S \subset V(G) \setminus V(G')$, then $\partial_{G'}(S) \subset S$ is the set of nodes in S at the boundary of G'. Formally, $\partial_{G'}(S) = \{u | u \in S, (v, u) \in E(G^0), v \in V(G')\}$.

We denote the shortest path from vertex v_1 to vertex v_k in graph G by $p_G(v_1, v_k) = \langle v_1, \ldots v_k \rangle$. The shortest distance between v_1 and v_k is denoted by $d_G(v_1, v_k) = \sum_{1 \leq i \leq k-1} W(v_i, v_{i+1})$.

The shortest distance from u to v via $S \subset V(G)$ in graph G is denoted by

$$\widehat{d}_G(u, v, S) = \min_{x \in S} \{d_G(u, x) + d_G(x, v)\} \tag{4}$$

We denote the label of u on graph G by L_u or $L(u, G^0)$.

A separator is called a *δ-separator* if the sum of the weights of the vertices in any connected component is no more than δw where w is total weight of all vertices in the graph, and $0 < \delta < 1$.

Next we present two lemmas used for correctness for our algorithm. These are well known results and generally implied in all previous works on shortest distance graphs. Lemma 4.1 states that given a subgraph, the shortest distance from node u inside of the subgraph to node v outside of the subgraph is equal to the distance via the boundary nodes of the subgraph. Lemma 4.2 is generalization of this lemma for separators and its components.

Lemma 4.1 *Let G' be a connected subgraph of G, $V' = V(G')$, $V = V(G)$, and let B be the set of all boundary nodes of V'. The shortest distance between $u \in V'$ and $v \in V \setminus V'$ then satisfies $d_G(u, v) = \widehat{d}_G(u, v, B)$.*

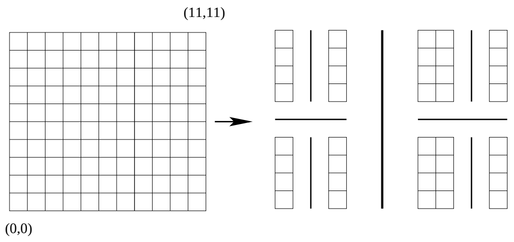


Figure 3. Hierarchical Recursive Decomposition on Grid Graph

Lemma 4.2 *If separator S^v break graph $G = (V, E)$ into connected components $\{G'/S^v\}_1, \ldots, \{G'/S^v\}_k$, then the shortest distance from $u \in \{G/S^v\}_i$ to $v \in \{G/S^v\}_j$ is*

$$d_G(u, v) = \begin{cases} \widehat{d}_G(u, v, S^v) & \textit{if } i \neq j \\ \min\{\, d_{\{G/S^v\}_i}(u, v),\ \widehat{d}_G(u, v, S^v)\} & \textit{if } i = j \end{cases}$$

4.1. Hierarchical Recursive Decompositions

Our algorithms DL and NDL build upon the work in [6], which applies a hierarchical recursive decomposition (or HRD for short) on the input graph $G^0 = (V^0, E^0)$.

HRD starts with the input graph, and recursively applies vertex separators to decompose the graph into connected components, until depth of recursion exceeds some fixed threshold d.

To describe the labeling procedure we will need to refer to the subgraphs, separators, and other entities generated during the recursion. Therefore in order to facilitate the description of our labeling algorithm we introduce a data structure, called Separator Tree $T = (V_T, E_T)$ to capture the complete HRD process. Nodes are associated with several variables, described next, which hold separators, subgraphs and entities generated at intermediate steps of HRD-decomposition.

Each tree-node $\mathbf{n} \in V_T$ is associated with a component by the function $\mathcal{C} : V_T \rightarrow \mathcal{G}$, where $\mathcal{G}$ is the family of all subgraphs of G^0. The root $\mathbf{n}_{root}$ of the separator-tree T is associated with the complete graph i.e.$\mathcal{C}(\mathbf{n}_{root}) = G^0$.

Function $\mathcal{S} : V_T \rightarrow V^+$ maintains the separator applied to the subgraph $\mathcal{C}(\mathbf{n})$. If $\mathbf{n}$ is a leaf of T then $\mathcal{S}(\mathbf{n}) = \emptyset$. Each tree-node $\mathbf{n}_i$ is assigned an identifier by the function $\mathcal{I} : V_T \rightarrow \mathbb{Z}$.

Edges in E_T maintain the hierarchy of the decomposition. That is, $(\mathbf{n}_i, \mathbf{n}_j) \in E_T$ if component $\mathcal{C}(\mathbf{n}_j)$ emerged from applying separator on component $\mathcal{C}(\mathbf{n}_i)$. We say $\mathbf{n}_i$ is a parent of $\mathbf{n}_j$. This parent relationship is maintained by function $\mathcal{P} : V_T \rightarrow V_T$ which maps each node of separator tree T to its parent. Function $\mathcal{L} : V_T \rightarrow \mathbb{Z}$ denotes the height of the node in the separator tree.

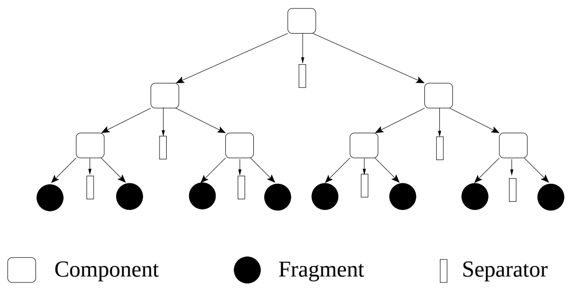


Figure 4. Corresponding Separator Tree

Algorithm 1 presents the pseudo-code for the construction for HRD and the separator tree T. It initializes $\mathbf{n}_{root}$ as the root of T with $\mathcal{C}(\mathbf{n}_{root}) = G^0$ and $\mathcal{L}(\mathbf{n}_{root}) = 0$ and invoke algorithm 1 as $HRD(G^0, \mathbf{n}_{root})$.

Line 4 applies separator S^v on G to decompose G^0 into components $\{G/S^v\}_1, \ldots, \{G/S^v\}_{c-1}$. Line 9 creates $c-1$ children $\mathbf{n}_1, \ldots, \mathbf{n}_c$ of $\mathbf{n}$ where $\mathcal{C}(\mathbf{n}_i) = \{G/S^v\}_i$.

Function $new\_edge(\mathbf{n}_i, \mathbf{n}_j)$ (line 8) adds a directed edge from $\mathbf{n}_i$ to $\mathbf{n}_j$. Figure 4.1 shows the separator tree corresponding to HRD with depth 3 on a sample grid graph shown in figure 4.1.

Algorithm 1 HRD(G, $\mathbf{n}$)

```
1: if ($\mathcal{L}(\mathbf{n}) == d$) return;
2: Let $G = \mathcal{C}(\mathbf{n})$. Find vertex separator $S^v$ for $G$. Let $S^v$ decompose $G$ into $c-1$ connected components.
3: $\mathcal{S}(\mathbf{n}) = S^v$
4: for all $i \in [1, \ldots, c-1]$ do
5:     $\mathbf{n}_i = T.new\_node(\{G/vsep\}_i)$. $I(\mathbf{n}_i) = i$. $\mathcal{C}(\mathbf{n}_i) = \{G/S^v\}_i$. $\mathcal{L}(\mathbf{n}_i) = \mathcal{L}(\mathbf{n}) + 1$.
6:     $T.new\_edge(\mathbf{n}_i, \mathbf{n})$
7:     $HRD(\mathbf{n}_i)$
8: end for
9: $\mathbf{n}_c = T.new\_node(S^v)$. $\mathcal{C}(\mathbf{n}_c) = S^v$. $\mathcal{I}(\mathbf{n}_c) = c$. $\mathcal{L}(\mathbf{n}_c) = \mathcal{L}(\mathbf{n}) + 1$.
```

A *label arc* is a tree-node pair $(\mathbf{n}_1, \mathbf{n}_2)$ and is used to pictorially depict the set of precomputed distances. If a labeling contains a label arc $e = (\mathbf{n}_1, \mathbf{n}_2)$, then its data structure maintains all values of $d_{G^0}(n_i, n_j)$ where $n_i \in \mathcal{S}(\mathbf{n}_1)$ and $n_i \in \mathcal{S}(\mathbf{n}_2)$.

5. Gavoille's Labeling (or GL algorithm)

For consistency, we use the same definitions as in [6]. A vertex-labeling for a graph $G = (V, E)$ is a non-negative function that assigns a label $L(u, G)$ to each vertex u of G. A distance decoder is a function ζ responsible for distance computation; given two labels λ_1, λ_2 it returns a "distance" value $\zeta(\lambda_1, \lambda_2)$. We say $\langle L, \zeta \rangle$ is a distance labeling for G if $\zeta(L(u, G), L(v, G)) = d_G(u, v)$ for any pair of vertices $u, v \in V(G)$, where $d_G(u, v)$ is the shortest distance between u and v.

Gavoille [6] showed that a general graph with n vertices can be labeled with labels of size $9n$ bits, so that the distance between two vertices can be computed from their labels in time $O(\log\log n)$. If an n-vertex graph has a (recursive) $2/3$-separator of size αn, then it can be given labels of size $O(\alpha \log n + \log^2 n)$, such that the distance can be computed in time $O(\log n)$. Since planar graphs have $\frac{2}{3}$-separators of size $O(\sqrt{n})$ [6], planar graphs admit distance labellings with labels of size $O(\sqrt{n} \log n)$.

5.1. Applying Labeling

We now describe the labeling for planar graphs $G^0 = (V^0, E^0)$ as described in [6]. The labels are of form $L(u) = \langle (id_u^1, A_u^1), (id_u^2, A_u^2), \ldots \rangle$.

First, perform HRD and build the separator tree T as described in Section 4.1. Next, starting with $\mathbf{n} = \mathbf{n}_{root}$, proceed as follows. Let $S^v = \mathcal{S}(\mathbf{n})$ and $\mathbf{n}_1, \ldots, \mathbf{n}_{c-1}$ be the $c-1$ children of $\mathbf{n}$ in T. Also let $l = \mathcal{L}(\mathbf{n})$ be the height of $\mathbf{n}$. Each vertex u from component $\mathcal{C}(\mathbf{n}_t)$, $1 \le t \le c$ is given label $L(u, G)$ consisting of:

1. The identifier $id_u^l = \mathcal{I}(\mathbf{n}_t)$ of its component.
2. An array A_u^l whose j^{th} entry contains u's distance to separator vertex $S^v[j]$, that is, $A_u^l[t] = d_G(u, S^v[t])$.
3. The label $L(u, \mathcal{C}(\mathbf{n}_t))$, if $t \neq c$ (created recursively).

id_u^i is the identifier of the component u belongs to at the i^{th} level of recursion, and A_u^i is the array of u's distances to the separator at that level. Let $|A_u^i|$ be the size of the array A_u^i, and $|L(u)|$ be the number of pairs in L_u.

5.2. Distance Decoder

Algorithm 2 outlines the corresponding distance decoder function. To obtain $d_{G^0}(u, v)$, we invoke $DistanceDecoder(0)$ on labels L_u and L_v which is a recursive application of lemma 4.2 i.e. Consider separator $S^v = \mathcal{S}(\mathbf{n})$ for the root tree-node $\mathbf{n}$. Let $\mathbf{n}_1, \ldots, \mathbf{n}_{c-1}$ be the $c-1$ children of $\mathbf{n}$ in T. Let $u \in \mathcal{C}(\mathbf{n}_i)$ and $v \in \mathcal{C}(\mathbf{n}_j)$, then

$$d_G(u, v) = \begin{cases} \widehat{d}_G(u, v, S^v) & \text{if } i \neq j \\ \min\{\, d_{\mathcal{C}(\mathbf{n}_i)}(u, v),\ \widehat{d}_G(u, v, S^v)\} & \text{if } i = j \end{cases}$$

The variable A represents $\widehat{d}_G(u, v, S^v)$, and variable B corresponds to $d_{\mathcal{C}(\mathbf{n}_i)}(u, v)$ at the i^{th} level of recursion. We will use notation $L_u, L_v|_G$ to represent this decoding function on labels L_u and L_v.

Algorithm 2 DistanceDecoder(i)

```
Require: Labels $L(u)$, $L(v)$ and integer $i$
1: $A = \min \{ (A_u^i[j] + A_v^i[j]) \ , 0 \leq j < |A_u^i| \}$
2: if $id_u^i \neq id_v^i$ then
3:     return A
4: end if
5: if $i \leq |L(u)|$ then
6:     $B = DistanceDecoder(i+1)$
7:     $A = \min\{A, B\}$
8: end if
9: return $A$
```

6. Disk Based Labeling (or DL)

In [6], labels are assigned to each node such that shortest distance between two nodes can be computed just from their labels. However, storing these labels for all nodes incurs a prohibitive storage overhead for large graphs. A simple strategy would be to assign labels to subset of nodes and online compute labels for unlabeled nodes. This would reduce storage cost but would make the distance-decoding procedure prohibitively costly.

The DL algorithm performs assigns labels to a subset of nodes and breaks the graph into several components (referred to as fragments). The distance decoder combines Dijkstra on the resulting fragments and the labeling approach of [6] over the labeled nodes. We demonstrate in the experiments that our scheme reduces storage cost significantly yet can answer shortest distance with marginal penalty in CPU-cost. The number of disk accesses per query is low in both scheme.

6.1. Labeling Procedure

The DL-algorithm derives labeling based upon an input separator tree $T = (V_T, E_T)$. A node $n \in V^0$ is a *separator node* if it is a member of any of the vertex separators generated during the decomposition. That is, if n is a separator node then $n \in \mathcal{S}(\mathbf{n})$ for some $\mathbf{n} \in V_T$.

The labels are of same type as Gavoille's labeling, i.e. Let $L(u) = \langle (id_u^0, A_u^0), \ldots, (id_u^t, A_u^t) \rangle$ represent the label for node u, where id_u^i is the identifier of the component u belongs to at the i^{th} level of the recursion. Array A_u^i maintains distances to the separator nodes at i^{th} level of recursion.

If $\mathbf{n}$ be such that $\mathcal{L}(\mathbf{n}) = i$ and $u \in \mathcal{C}(\mathbf{n})$ then $id_u^i = \mathcal{I}(\mathbf{n})$ and $A_u^i[k] = d_{G^0}(u, S^v[k])$ where $S^v = \mathcal{S}(\mathbf{n})$.

Algorithm 3 describes the procedure for assigning labels to the separator-nodes of $\mathbf{n}$ of tree T. We invoke $Label(\mathbf{n})$ for all $\mathbf{n} \in V_T$.

Algorithm 3 Label($\mathbf{n}$)

```
1: $S^v = \mathcal{S}(\mathbf{n})$
2: $\mathbf{n}' = \mathbf{n}$
3: repeat
4:     for all $i \in [1, \ldots, |S^v|]$ do
5:         $u = S^v[i]$
6:         $id_u^{\mathcal{D}(\mathbf{n}')} = \mathcal{I}(\mathbf{n}')$
7:         for all $j \in [1, \ldots, |S^{v\prime}|]$ do
8:             $A_u^{\mathcal{L}(\mathbf{n}')}[j] = d_{G^0}(u, S^{v}‘[j])$
9:         end for
10:    end for
11:    $\mathbf{n}' = \mathcal{P}(\mathbf{n}')$
12: until $(\mathbf{n}' == NULL)$
```

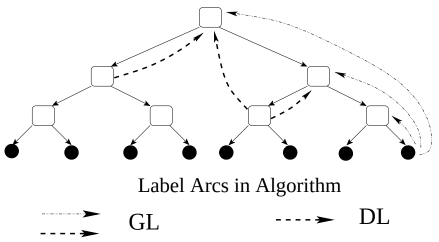


Figure 5. Label arcs for few tree-nodes for the sample grid graphs under Gavoille's and DL labeling scheme

6.2. The Distance Decoder

The labeling phase yields separator-nodes, their labels and several fragments. The distance decoder has two parts; the first obtains inter-separator shortest distance using labels only, and the second computes distances between non-separator nodes using Dijkstra and inter-separator shortest distances.

6.2.1 Shortest Distances Between Separator Nodes

To obtain shortest distance between separator-nodes u and v from their labels, we use the distance decoder function of [6] which is described in Section 5.2. Given labels L_u and L_v, for separator nodes u and v, we can compute $d_{G^0}(u, v) = |L(u), L(v)|_{DL} = |L_u, L_v|_G$.

6.3. Distance Between Non-Separator Nodes

Finding the shortest distance between u and v where $u, v \in V^0$ proceeds as follows. Let $F_u = (V_u, E_u)$ and $F_v = (V_v, E_v)$ be the fragments containing u and v. Let G_u be the subgraph induced by F_u and its boundary nodes i.e. $V(G) = V_u \cup \partial_{F_v}(V^0 \setminus V_u)$. Similarly G_v be the subgraph that representing v's fragment and its boundary nodes.

1. Find shortest distance from u in G_u to all the boundary nodes $\partial_{F_u}(V^0 \setminus V_u)$ using Dijkstra, as in Algorithm 4.

   Let the result be stored in Distance-Vector $DV_u = \langle (u_1, d_i^u, \ldots, (u_t, d_t^u) \rangle$, where $d_i^u = d_{G_u}(u, u_i)$.

2. Similarly, find shortest distance from v in G_v to all nodes boundary nodes $\partial_{F_v}(V^0 \setminus V_v)$, and store them in the Distance-Vector $DV_v = \langle (v_1, d_1^v), \ldots, (v_r, d_r^v) \rangle$ where $d_i^v = d_{G_v}(v_1, v)$.

3. In our third and final step, we obtain the shortest distance using the distance vectors and the labels for the boundary separator nodes:

$$d_G(u,v) = \min_{1 \le i \le t, 1 \le j \le r} \{d_i^u + d_j^v + |L(u_i), L(v_j)|_{DL}\} \tag{5}$$

Our experiments used the pseudo-code in Algorithm 4 to implement Dijkstra. The priority queue (referred to as node_pq in line 1) is a partial function from the nodes to double data type. We use the implementation of [17].

Algorithm 4 $FDB(G = (V, E))$

```
1:  node_pq PQ(G);
2:  node n; edge e;
3:  for all $v \in V$ do
4:      $dist[n] = 0$ if $n == v$ else $dist[n] = \infty$
5:      PQ.insert(n, dist[n])
6:  end for
7:  while $!PQ.empty()$ do
8:      node u = PQ.del_min();
9:      if($u$ is a separator-node) FDV[u] = dist[u]
10:     for all out_edges(e, u) do
11:         v = target(e); c = W(e) + dist[u]
12:         if( $c < dist[v]$){ PQ.decrease_p(v, c); dist[v] = c;}
13:     end for
14: end while
```

6.4. Space and Time Complexity

We now analyze the space and time complexity of the proposed algorithm when the input graph G^0 is a planar graph. Lipton and Trajan in [16] gave an algorithm that constructed $2/3$-separators of size less than $O(\sqrt{n})$ for planar graphs.

It now follows that for any tree node $\mathbf{n}$ the size of the $\mathcal{C}(\mathbf{n})$ is no more than $O(\frac{2}{3}^i n)$ where $i = \mathcal{L}(\mathbf{n})$. If the depth of HRD is $d = O(\log_{\frac{3}{2}} n)$ then the upper bound on the total number of separator-nodes works out to

$$\sum_{0 \le i \le d} O(2^i \sqrt{\frac{2}{3}^i n}) = O(n) \tag{6}$$

Because each label maintains distance to all ancestor separator nodes, the storage cost per label is $\sum_{0 \le i \le l} O(\sqrt{\frac{2}{3}^i n}) = O(\sqrt{n} \log n)$. However, no fragment is larger than $O(\frac{2}{3}^{\log_{\frac{3}{2}} n} n) = O(1)$. Therefore, the cost of Dijkstra (step 3) and the total number of boundary nodes on the fragment is constant. Hence the distance decoder has CPU-time complexity $O(\log_{3/2} n)$, and requires $O(1)$ disk access per shortest distance query.

Asymptotically, this scheme is no better than that of Gavoille's. However, in practice we observe a large reduction in storage cost for a marginal increase in disk access and CPU-time.

6.5. Heuristics in DL-algorithm

Next we describe heuristics that substantially improve the performance of DL labeling.

Expand Fragments

In practice we do not store fragments as is. Instead we expand the graph by one hop to include all of the bounding separator node and store this graph instead. Note that a boundary nodes of fragment will always be a separator-node.

After this procedure a separator node may be stored in several fragments, which increase the storage cost. But distance computation is optimized as this avoid multiple disk access to fetch the boundary nodes.

Disk Layout of fragments

We measured the disk I/O performance of the decoding algorithm with the assumption that each fragment occupies separate and single disk page. However we found that for our datasets and choice of separator algorithm there are several fragments of very small sizes. We pack these fragments into disk pages using a greedy algorithm based on the number of common separator nodes between fragments. Ties are broken arbitrarily.

Auxiliary Data Structure

All of the axillary data structure such as label index (lookup of node to the page that contains the labels), fragment index etc. are map data structures implement with hash [17] and are always main-memory resident.

6.6. Directed Graphs

Although we have focused on undirected graphs, both the lemmas presented in Section 4 that form basis for correctness of our algorithm also hold for directed graphs.

Therefore we can apply this scheme for directed graphs as well. But the labeling storage cost will be double to that required for undirected graphs, as it would have to maintain distances to in both directions per separator pair.

7. The NDL Algorithm

We now describe the Nested Disk Labeling algorithm (NDL) and its distance decoder function. Our scheme trades off distance-decoding efficiency in favor of reduced storage cost.

7.1. Labeling Procedure

NDL Labeling is conceptually a nested variation of DL-Labeling algorithm and has an outer phase and inner phase.

The outer phase build a separator-tree and performs DL labeling to generate a set of labeled nodes and fragments. These fragments are called l-fragments. In the inner phase, for each of the l-fragment separately NDL applies DL-labeling. The l-fragment are thus decomposed into labeled nodes and fragments.

Additionally there is a bridge labeling phase that connects the two phases. Following section describe these three phases in more detail.

Outer Labeling

Given a input graph NDL builds a separator tree T (described later) and applies DL-labeling algorithm with T as input parameter. As a result a subset of nodes (the separator nodes to be precise) will get labeled and the graph will be decomposed into several fragments. If u is separator node then the label of u is denoted by L_u^o. The fragments generated are refereed to as l-fragments.

Figure 7.1 shows the separator tree and the label-arcs for the example grid graph. The labeled nodes and the resulting l-fragments are shown in figure 7.1.

Inner Labeling

In the inner phase, we independently perform DL-labeling for each l-fragment. The separator-tree for each l-fragment is constructed as described previously in section 4.1. The height of the separator tree is $\log n$ if the l-fragment is of size n.

As before the DL-algorithm labels the separator-nodes which break the l-fragment into several fragments. If node u is a separator for some l-fragment in this phase then it is refereed to as inner-separator node and its label is denoted by L_u^i.

We would like to note that since each l-fragment is labeled independently, Gavoille's distance decoder over two inner labels, L_u^i and L_v^i, is defined only if only if u and v belong to the same l-fragment.

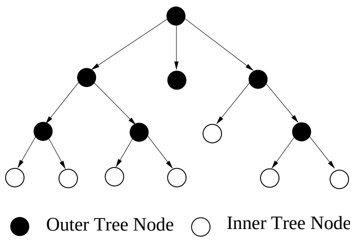


Figure 6. HRD under NDL-algorithm for a hypothetical graph

Bridge Labeling

Bridge phase connects the inner-separator nodes and outer separator nodes i.e. labels assigned during the bridge phase will maintain distance from inner separator to outer-separator.

Let u be an inner separator node that belongs to l-fragment F. Let $v_1, \ldots, v_k$ be the boundary nodes of F. The bridge label of u is given by

$$L_u^b = \langle (v_1, d_{G^0}(u, v_1)), \ldots, (v_k, d_{G^0}(u, v_k)) \rangle \qquad (7)$$

The Resulting Labels

In NDL-labeling input graph is decomposed in several l-fragments by separators. Each separator u is assigned an outer label L_u^o where u is a separator in outer labeling.

Each l-fragment is further decomposed into several fragments using HRD. If node u acts as a separator node during inner labeling it is assigned two labels inner label L_u^i and bridge label L_u^b.

Nodes with no labels are called non-labeled nodes. For convenience, we denote label for node u with tuple $L_u = \langle L_u^o, L_u^i, L_u^b \rangle$. Let $\emptyset$ denote empty string. If u is outer label then $L_u^i = L_u^b = \emptyset$, else if u is inner label then $L_u^o = \emptyset$ and if u is non labeled then $L_u^o = L_u^i = L_u^b = \emptyset$.

7.2. Distance Decoding

Next we describe the distance decoder function, represented as $|L_u, L_v|$ for a pair of NDL-labels L_u, and L_v. We observe first that the NDL-labeling classifies nodes into three disjoint classes: the o-nodes (or class O), the i-nodes (or class I), and n-nodes or (class N).

Given two nodes u and v and their labels L_u and L_v the distance decoding procedure depends on the class to which the nodes u and v belong to and on whether they lie in the same or different l-fragments. Hence the NDL distance decoding is composed of several sub decoding procedures. Next we will enumerate all the scenarios and give their corresponding distance decoding.

The tuple $\langle u\text{–}class, v\text{–}class, pos\rangle$ represents a decoding scenario, where $u\text{–}class$ and $v\text{–}class$ are of type O, I or N. The parameter pos may take values S, D or $-$, indicating that u and v belong to the same or different l-fragments, respectively, while $-$ is used when pos is irrelevant (when $u\text{–}class$ or $v\text{–}class$ is of type O). For each scenario we give a distance decoding scheme and corresponding cost bounds.

We note that the following discussion also acts as proof sketch for correctness and CPU and disk I/O cost for distance decoder. Therefore to aid discussion we do not present the actual formal proofs.

Scenario $\langle O, O, -\rangle$

If u and v are two outer labeled-nodes then shortest distance between them can be computed via the Gavoille's distance decoder i.e. $d_{G^o}(u, v) = |L_u, L_v|_N = |L^o_u, L^o_v|_G$. Clearly, the disk I/O cost for decoding is $O(1)$. The CPU-time is upper bounded by Gavoille's distance-decoder of $O(\log n)$.

Scenario $\langle O, I, -\rangle$

To compute the distance between an o-node u and i-node v we proceed as follows: Let $L^b_v = \langle(v_1, d_{G^o}(u, v_1)), \ldots, (v_k, d_{G^o}(u, v_k))\rangle$. By Lemma 4.1,

$$|L_u, L_v|_N = \min_{0\le i\le k}\{L^o_u L^o_{v_i}|_N + L^b_u(v_i)\} \tag{8}$$

That is, the distance between u and v is the minimum over distances via boundary o-nodes of the v's l-fragment. Figure 7(a) illustrates the scenario. In depicts u,v, and v's boundary o-nodes. The solid and dashed lines depict shortest distance derived by o-labels and b-labels respectively.

As there are at most $O(\sqrt[4]{n})$ boundary o-nodes the disk I/O cost is $O(\sqrt[4]{n})$ and CPU-time has complexity $O(\sqrt[4]{n}\log n)$.

Scenario $\langle O, I, S\rangle$

Next we compute distance between two i-nodes, u and v, that belong to the same l-fragment. Let $L^b_u = \langle(v_1, d_{G^o}(u, v_1)), \ldots, (v_k, d_{G^o}(u, v_k))\rangle$ and $L^b_v = \langle(v_1, d_{G^o}(v, v_1)), \ldots, (v_k, d_{G^o}(v, v_k))\rangle$. Then, using lemma 4.2 the shortest distance $d_{G^o}(u, v) = |L_u, L_v|_N =$

$$\min\{|L^i_u, L^i_v|_G, \min_{1\le i\le k}\{d_{G^o}(u, v_i) + d_{G^o}(v, v_i)\}\}$$

That is, the shortest distance is either via the boundary o-nodes or it is the shortest distance within that l-fragment. In figure 7(a) u and v are i-nodes within same l-fragment and solid lines depict distance via boundary o-nodes and dashed lines is the distance within the l-fragment.

Therefore the disk access per query is $O(\sqrt[4]{n})$ and the decoding takes $O(\sqrt[4]{n}\log n)$ time.

Scenario $\langle I, I, D\rangle$

We now handle the scenario when u and v are i-nodes in different l-fragment. An instance of this scenario is depicted in figure 7(c). Node u lies inside l-fragment whose boundary o-nodes are $u_1, \ldots, u_l$, where $l = 4$, while v lies inside different l-fragment whose boundary nodes are $v_1, \ldots v_m$ where $m = 3$.

The solid lines depict shortest distance from i-node u (or v) to outer labeled nodes u_i's (or v_i's), which is maintained in L^b_u (or L^b_v). The dotted lines lines represent shortest distance between two outer-labeled nodes. This can be computed directly from their respective labels i.e. $d(u_i, v_j) = |L^o_{u_i}, L^o_{v_j}|_G$.

Using lemma 4.2, $d(u, v)$

$$\begin{aligned} &= \min_{\substack{i\in[1:l]\\ j\in[1,m]}}\{d(u, u_i) + d(u_i, v_i) + d(v, v_j)\} && (9)\\ &= \min_{\substack{i\in[1:l]\\ j\in[1:m]}}\{L^b_u(u_i) + |L^o_{u_i}L^o_{v_j}|_G + L^b_v(v_j)\} && (10)\end{aligned}$$

If $l = m = O(\sqrt[4]{n})$, this distance decoder will have time complexity of $O(\sqrt{n}\log n)$ given that Gavoille's decoding takes $O(\log n)$ time.

Scenario $\langle N, N, D\rangle$

We finally consider the case when u and v are non-separator nodes. Let G_u and G_v be the leaf fragments that contain u and v respectively. The distance-decoder proceeds in three steps.

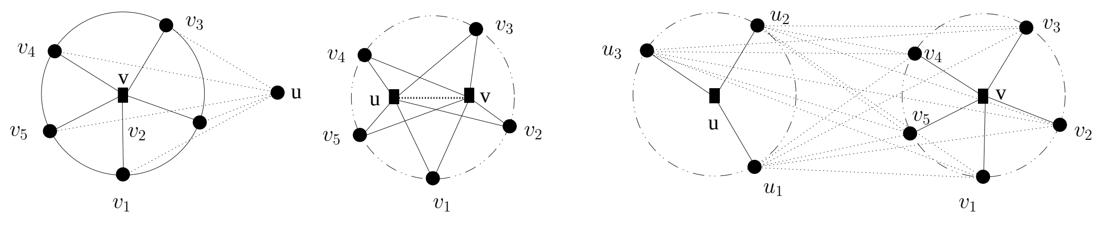
First, it finds shortest distances from u to all the boundary nodes in G_u, and stores these values in distance-vector $DV_u = \langle(u_1, d^u_i, \ldots, (u_t, d^u_t)\rangle$, where $d^u_i = d_{G_u}(u, u_i)$.

Next, it finds the shortest distances from all the boundary nodes of G_v to v, storing them in distance vector $DV_v = \langle(v_1, d^v_1), \ldots, (v_r, d^v_r)\rangle$ where $d^v_i = d_{G_v}(v_i, v)$.

Finally, it obtains the shortest distance using the distance vectors and labels:

$$d_G(u, v) = \min_{\substack{1\le i\le t\\ 1\le j\le r}}\{\, d^u_i + d^v_j + |L(u_i), L(v_j)|_N\} \tag{11}$$

Each leaf fragment is bounded by constant number of inner separator nodes. Therefore, disk access and CPU-time cost is constant times cost for computing i-node dis-



(a) $\langle O, I, - \rangle$ (b) $\langle I, I, S \rangle$ (c) $\langle I, I, D \rangle$

Figure 7. Decoding Scenarios

	GL		DL		NDL	
Dataset	LS	IPD	LS	IPD	LS	IPD
TPG25	72	9.0	4.20	0.53	3.24	0.23
TPG200	??	150.0	94.10	11.74	42.9	5.37
LAC	??	333.7	187.7	23.45	109.3	13.66

Table 1. LS denotes Total label size (in MB) and IPD denotes number of shortest distances maintained (in units of 10^6)

tance. Hence, disk access and CPU-cost are $O(\sqrt[4]{n})$ and $O(\sqrt[4]{n} \log^2_{\frac{2}{3}} n)$ respectively.

The distance decoder for scenarios $\langle N, N, S \rangle$, $\langle N, I, D \rangle$, $\langle N, I, S \rangle$, and $\langle N, O, - \rangle$ can be obtained by trivial modification to the distance decoder of scenario $\langle N, N, D \rangle$, so we omit their details.

8. Experimental Results

We conducted a series of experiments on real-world and synthetic dataset with synthetic workloads to study the space-time trade-offs of our algorithms with that of [6] and Dijkstra's. All experiments are conducted on Linux box with 1.7 GHz CPU and 1 GB of main memory.

8.1. Datasets & Notations

We used three datasets, namely TPG25, TPG200 and LAC. Both TPG25 and TPG200 are synthetically generated random triangulation maps. That is, we choose n random points (where n is the number of nodes in the graph) in a unit square and create a planar triangulation (for detail see [17]). Such graphs are typical in network modeling and analysis. Dataset LAC is the complete road network for Los Angeles County.

Dataset TPG25 has 25,000 nodes and 74,974 edges and occupies 2.8MB of disk space, while dataset TPG200 has 200,000 nodes and 599,963 edges with 28MB of disk space. The LAC dataset has 192,256 nodes and 496,931 edges and requires 21MB to store.

Disk Layout of the Graph and Labels

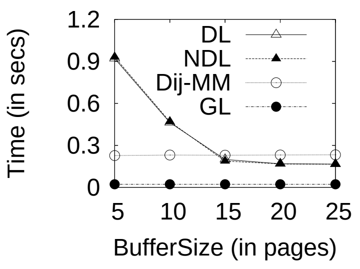
The graph is stored on the disk as follows. Each node of the graph maintains its own 4-byte identifier and 6-byte pointers to each of its adjacent nodes and an 8-byte weight for each out-edge.

The nodes are then Hilbert ordered [14] based on their locations and placed sequentially on the disk in that order. The labels are stored separately and also organized in Hilbert Order. Each label can occupy more than one disk page. Our measurements of disk I/O cost for distance computation in all our experiments explicitly account for sequential and random accesses with respect to this layout. In our model, each sequential access and random access has cost of 0.00055 and 0.011 seconds respectively (based on performance of NTFS file system on 5400 RPM disk [2]).

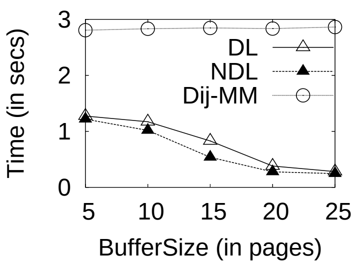
Input Parameters and their Default Values

Let c be the average cost of storage per node. Experiments were carried out with following default values: page size $p = 16384$ bytes, buffer size $b = 10$ pages, Depth of separator-tree $d = \log_{3/2} cn/p$. Algorithm NDL has two additional parameters α and β. For this study we fix a constant $k = 2$ and set $\alpha = \log_{3/2} kcn/p$ and $\beta = d$. All preprocessing was done in main memory, which required 5-10 hours per run.

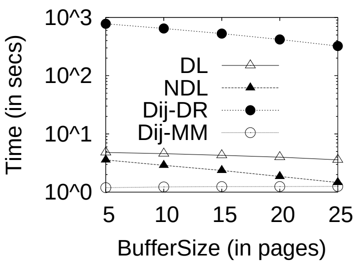
We will now compare the storage costs and distance decoder performance (both CPU and disk I/O cost) between Dijkstra (Dij), Gavoille (GL), Disk-Based Labeling (DL) and Nested Disk Based Labeling (NDL) over various values of input parameters such as pages size (p), buffer size (b) and NDL constant k. In all of the experiments the decoder performance is an average over 5000 randomly selected pairs of nodes.



(a) TPG25



(b) TPG200



(c) LAC

Figure 8. Decoder Performance over varying buffer sizes

In particular, we compare DL and NDL with two variations of Dijkstra, namely main-memory resident Dijkstra (or Dij-MM) and disk-resident Dijkstra (or Dij-DR). In Dij-MM the entire graph is available in main memory, so the buffer parameter is irrelevant. In Dij-DR the graph resides on secondary storage according to the layout described above. Therefore, if a node is to be visited during the execution of Dij-DR, the page holding the node's information (adjacent edges and their weight) must be fetched into the buffer, if not already present.

Our current implementation of NDL algorithm does not balance the boundary nodes. That is, it does not perform the steps mentioned under Balancing Boundary Nodes in Section 7.1. Balancing the boundary will improve performace further.

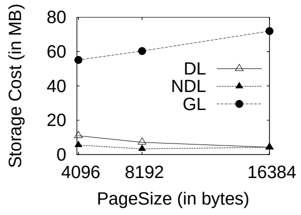
8.2. Algorithm for Graph Partitioning and Separators

All our labeling algorithm perform HRD using separators over graphs. Our discussion on performance of the labeling algorithm was with the separator algorithm given by Lipton and Tarjan [16] which guarantees a $\frac{2}{3}$-separator.

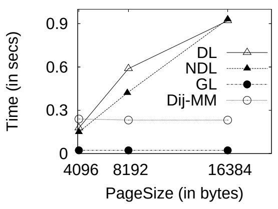
However, we use a spectral method, implemented in the Chaco graph partitioning library [10], to partition the graph and derive separators. Spectral methods [24] are an alternative approach for finding small size separators for general graphs. They use the eigenvalues and eigenvectors of a matrix representation of the graph. Typically, the Laplacian matrix B is used, in which the vertex degrees appear on the diagonal, and entry $b_{ij} = -1$ if the edge (v_i, v_j) exists and 0 otherwise. The eigenvector u_2 corresponding to the second eigenvalue λ_2 is computed and the vertices of the graph partitioned according to entries in u_2.

8.3. Separators & Label Sizes

Here we focus on the storage cost of GL, DL and NDL algorithms for the default values of the input parameter. A labelling essentially maintains, in some fashion, distances between among several pairs of nodes and the storage cost is directly proportional to the number of inter-node distances maintained. Therefore we focus on the resulting total label sizes (denoted as LS) and number of inter-pair distances (denoted as IPD) maintained by each of the labeling algorithm (see table 1). Total number of separator-nodes in the separator-tree of TPG25, TPG200 and LAC was $2,377$, $19,173$, and $15,835$ respectively.



(a) Space



(b) Time

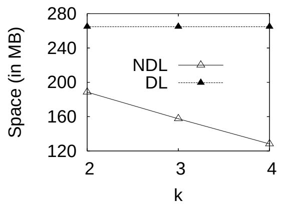
Figure 9. [TPG25] Storage and Decoder Cost vs. varying page size

8.3.1 Infeasibility of GL for Large Graphs

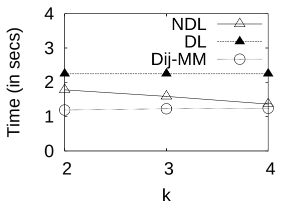
Our results do not include the label sizes for the labeling scheme in [6] for large graphs. Despite using a main memory size of 1GB, the algorithm quickly ran out of memory while computing labels for datasets $TPG200$ and LAC. This result is reflected in Table 1.

8.4. Experiment with Varying Buffer Sizes

Figure 8 studies the effect of buffer size on distance decoding performance on all three datasets. For dataset TGP25, we see that GL has the best performance, but has prohibitively high storage cost (see Table 1). However, our

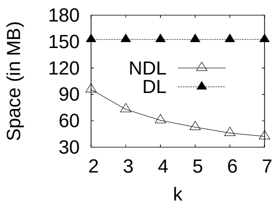


(a) Space

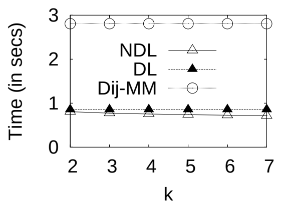


(b) Time

Figure 10. [LAC] Storage and Decoder Cost with varying k



(a) Space



(b) Time

Figure 11. [TPG200] Storage and Decoder Cost with varying k

algorithms DL and NDL perform comparably to Dij-MM and GL, yet use only a fraction of the disk space required for GL and a fraction of the buffer space required for Dij-MM. We did not include results from Dij-DR as its performance is considerably worse (99 secs on average with 5 buffer pages, to 16 secs for 25 buffer pages).

Since TPG200 is a large and dense network, Dij-MM performs considerably worse than DL and NDL in Figure 8(b). We omitted the results for Dij-DR, which takes more 5 minutes on average per computation in the best case. Unexpectedly, NDL delivers performance similar to that of DL, although it uses only half the storage of DL. This interesting result is worthy of further study. Since labels in NDL are smaller then DL, they are packed more densely on the disk and hence may require fewer disk accesses per distance computation.

We emphasize the use of a logarithmic scale in Figure 8(c). Since LAC is a sparser network, Dij-MM has the best performance but disk based Dijkstra still performs quite poorly. Again, NDL outperforms the DL algorithm even though it uses a fraction of storage cost.

8.5. Experiment with Varying Page Size

In the current setup, described in Section 8.1, increasing page-size increases the size of the fragments and reduces the depth of the separator tree. Hence the number of separator nodes is reduced, reducing the total label size for DL and NDL, as shown in Figure 9. However, since GL also maintains distances for all pairs of node within each fragment, increasing fragment size increases the storage cost.

There is no noticeable difference in decoder performance for GL as page size varies. For a page size of 4096, both DL and NDL perform better than Dij-MM, and there is a proportional loss in their decoder's performance with increasing page-size.

8.6. Effect of Varying k

Increasing the parameter k decreases the number and size of outer labels but increases the number and size of inner labels. Although the total number of l-fragments (see Section 7) decreases, the number of boundary separator nodes per fragment is increased. We have work underway to determine the optimal value of k in practice.

For this set of experiments, shown in Figure 10 and 11, we fixed the page size to 8192 bytes to permit larger variations in k. We see that for datasets TPG200 and LAC, NDL requires significantly less storage space and has decoding performance comparable to that of NDL.

In all of the experiments the decoder performance is an average over 5000 randomly selected pairs of nodes.

9. Conclusion

We have demonstrated both theoretically and experimentally that our algorithms provide efficient shortest distance computation at small space overhead. They require significantly less space than precomputing all pair of distances or other labeling schemes and provide shortest distance computation order of magnitude faster than the traditional Dijkstra's algorithm.

Algorithm NDL has distance decoding performance comparable to that of DL although it requires barely over half the space cost of DL. This makes DL particularly attractive method for external memory data structure for shortest distance computation.

Our future work will investigate how to find the optimal value of k in practice. We are also looking into ways to update the data structure when edge weights change.

References

[1] E. P. F. Chan and N. Zhang. Finding shortest paths in large network systems. In *GIS '01: Advances in geographic infor-*

mation systems, pages 160–166, New York, NY, USA, 2001. ACM Press.

[2] L. Chung, J. Gray, B. Worthington, and R. Horst. Windows 2000 disk io performance. Technical Report MS-TR-2000-55, Microsoft Research, Advanced Technology Division, Redmond, Washington, 2000.

[3] Cohen, Halperin, Kaplan, and Zwick. Reachability and distance queries via 2-hop labels. *SICOMP: SIAM Journal on Computing*, 32, 2003.

[4] Cormen, Lieserson, and Rivest. Introduction to algorithms. 1990.

[5] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *In SIAM J. Computing 16(6)*, pages 1004–1022, 1987.

[6] C. Gavoille, D. Peleg, S. Perennes, and R. Raz. Distance labeling in graphs. In *Symposium on Discrete Algorithms*, pages 210–219, 2001.

[7] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA '05: ACM-SIAM symposium on Discrete algorithms*, pages 156–165, 2005.

[8] G. N. Gredrickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal n Computing*, 16(6):1004–1022, December 1989.

[9] S. Gupta, S. Kopparty, and C. Ravishanakar. Roads, codes and spatiotemporal queries. In *ACM Symposium on PODS*, pages 115–124. ACM, 2004.

[10] B. Hendrickson and R. Leland. The chaco user's guide — version, 1994.

[11] Henzinger, Klein, Rao, and Subramanian. Faster shortest-path algorithms for planar graphs. *JCSS: Journal of Computer and System Sciences*, 55, 1997.

[12] D. A. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *COCOON*, pages 51–60, 1999.

[13] C. S. Jensen, J. Kolár, P. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *ACM GIS-03*, pages 1–8, New York, Nov. 7–8 2003. ACM Press.

[14] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. *Proceedings of VLDB*, pages 500–510, Sept. 1994.

[15] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995.

[16] R. Lipton and R. E. Tarjon. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[17] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[18] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad. Place: A query processor for handling real-time spatio-temporal data streams. In *30th International Conference on VLDB*, 2004.

[19] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proc. of the VLDB*, pages 802–813, 2003.

[20] C. Shahabi and M. R. K. M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *In the 10th ACM-GIS*, 2002.

[21] S. Shekhar, A. Fetterer, and B. Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *Advances in Spatial Databases, SSD'97,*, volume 1262, pages 94–111, 1997.

[22] S. Shekhar and D.-R. Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE Knowledge and Data Engineering*, 9(1):102–119, 1997.

[23] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *ACM GIS*, pages 9–16, 2003.

[24] D. A. Spielman. Spectral partitioning works: planar graphs and fi nite element meshes. In *FOCS '96*, page 96, Washington, DC, USA, 1996. IEEE Computer Society.