2023 Large Assignment - Reading Material

1 PRELIMINARIES

1.1 Problem Formulation

Given an edge-weighted graph $G(V, E, w)$, where V is the set of vertices, E is the set of edges, and w is a function which maps each edge $(u, v) \in E$ between a pair of vertices u and v to a positive value $w(u, v)$ that we refer to as edge weight. Let $n = |V|$ be the number of vertices, $m = |E|$ be the number of edges, and $N(v)$ be the set of adjacent vertices of v. We consider the hop of a path as the number of edges in this path. Given a natural number $k \in \mathbb{N}$, a k-hop-constrained path is a path with a hop no larger than k. We use $d_k(v_i, v_j)$ to denote the k-hop-constrained shortest distance between a pair of vertices v_i and v_j, which is the minimum value of the total weight of edges in any k-hop-constrained path between v_i and v_j. Furthermore, we use $P_k(v_i, v_j)$ to denote a k-hop-constrained shortest path between v_i and v_j, which is a path that corresponds to $d_k(v_i, v_j)$. For example, in the figure below, the 1-hop-constrained shortest distance between vertex 0 and 1 is 1.0 and the corresponding 1-hop-constrained shortest path is $\{(0, 1)\}$. If we raise the hop-constraint to 2, and the shortest distance become 0.8 and the corresponding shortest path is $\{(0, 2), (2, 1)\}$. We focus on the following hop-constrained shortest distance (or path) problem.

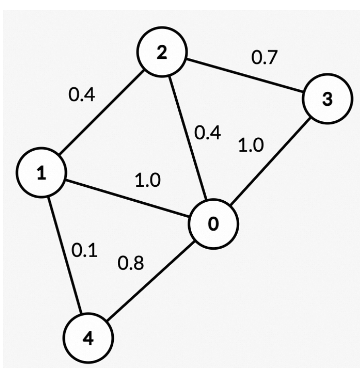

Figure 1: Example

Problem 1. *Given an edge-weighted graph $G(V, E, w)$ and a hop upper bound $k \in \mathbb{N}$, the hop-constrained shortest distance (or path) problem is to query the k-hop-constrained shortest distance (or path) between any pair of vertices v_i and v_j in the above graph.*

Querying hop-constrained shortest distances or paths is useful for retrieving information in relational databases , detecting frauds in E-commerce transaction networks, and routing data in communication networks. In this paper, we exploit the indexing approach to efficiently perform this task.

1.2 Hop-constrained Labels

The classical 2-hop labels enable us to query hop-unconstrained shortest distances and paths between vertices efficiently, and have been intensively studied in the last decade. Most recently, Zhang *et al.* incorporate hop constraints into the classical 2-hop labels for querying hop-constrained shortest distances and paths. We introduce the resulting hop-constrained labels as follows. For each vertex $v \in V$, there is a set $L(v)$ of labels. Each label in $L(v)$ is a three-element tuple $(u, h, d_h(v, u))$, where u is a vertex that is called a hub of v, h is a hop upper bound, and $d_h(v, u)$ is the h-hop-constrained shortest distance between v and u. We use L to denote the total set of hop-constrained labels, *i.e.,* $L = \{L(v) | \forall v \in V\}$. We let L satisfy the hop cover constraint.

Definition 1 (the hop cover constraint). *L satisfies the hop cover constraint for the hop upper bound $k \in \mathbb{N}$ if and only if, for an arbitrary pair of vertices v_i and v_j, if there is a k-hop-constrained path between v_i and v_j, then there are two labels $(u, h_1, d_{h_1}(v_i, u)) \in L(v_i)$ and $(u, h_2, d_{h_2}(v_j, u)) \in L(v_j)$ such that (i) $h_1 + h_2 \le k$; and (ii) $d_{h_1}(v_i, u) + d_{h_2}(v_j, u) = d_k(v_i, v_j)$, which indicates that the common hub u is in a k-hop-constrained shortest path between v_i and v_j.*

With a set L of labels that satisfies the hop cover constraint, we can query $d_k(v_i, v_j)$ using the following equation.

$$d_k(v_i, v_j) = \min_{\substack{(u,h_1,d_{h_1}(v_i,u)) \in L(v_i) \\ (u,h_2,d_{h_2}(v_j,u)) \in L(v_j) \\ h_1+h_2 \le k}} d_{h_1}(v_i, u) + d_{h_2}(v_j, u). \tag{1}$$

Furthermore, by incorporating the predecessor information into labels, we can also query every edge in a k-hop-constrained shortest path between v_i and v_j in a recursive way. Specifically, we extend each label in $L(v)$ to be a four-element tuple $(u, h, d_h(v, u), p_{vu})$, where p_{vu} is the predecessor of v in a h-hop-constrained shortest path between v and u. An example of using such labels to recursively query a k-hop-constrained shortest path is as follows. In figure 1, we assume that $(0, 2, 0.8, 2) \in L(1)$ and $(0, 0, 0, 0) \in L(0)$. We can use these two labels to get the shortest distance between 0 and 1. The predecessor of 1 shows that the next vertex of vertex 1 on the corresponding shortest path is 2, so edge $(0, 2)$ must be on the shortest path. Then we only need to find the path between 1 and 2, which is $(1, 2)$ of course. As a result, the shortest path is $\{(0, 2), (2, 1)\}$.

2 THE HSDL ALGORITHM

To our knowledge, the HBLL algorithm is the only existing method for generating hop-constrained labels. We note that HBLL may not have a sufficiently high efficiency in practice. The major reason is that it may generate a large number of redundant labels in a multi-thread environment. To address this issue, here, we propose the Hop-constrained Shortest Distance Labeling (HSDL) algorithm, which can efficiently generate a minimal set of labels parallel.

The details of HSDL. The algorithm inputs the graph $G(V, E, w)$ and a parameter K, which is the maximum value of the possibly queried hop upper bound k. First, it initializes an empty set L^{temp} of labels (Line 1). We assume that vertices are ranked by their degrees from large to small. Specifically, let $V = \{v_1, \cdots, v_{|V|}\}$. We use $r(v_i)$ to denote the rank of v_i. We have $deg(v_1) \ge \cdots \ge deg(v_{|V|})$,

Algorithm 1 The HSDL Algorithm

Input: a graph $G(V, E, w)$ and the maximum hop upper bound K
Output: a set L^{HSDL} of hop-constrained labels

1: $L^{temp} = \emptyset$
2: **for** each sorted vertex $v_i \in V$ in parallel **do**
3:     Initialize Q that contains $(v_i, 0)$ with the priority of 0
4:     **while** $Q \neq \emptyset$ **do**
5:         Pop (u, h_u) out of Q with the priority of d_u
6:         **if** $r(v_i) \geq r(u)$ **then**
7:             **if** $Query(u, v_i, h_u) > d_u$ **then**
8:                 Insert (v_i, h_u, d_u) into $L^{temp}(u)$
9:                 **if** $h' = h_u + 1 \leq K$ **then**
10:                     **for** each vertex $v \in N(u)$ **do**
11:                         $d_v = d_u + w(u, v)$
12:                         **if** $d_v < Q((v, h')).priority$ **then**
13:                             Push (or update) $\{(v, h')|d_v\}$ into Q
14:                         **end if**
15:                     **end for**
16:                 **end if**
17:             **end if**
18:         **end if**
19:     **end while**
20: **end for**
21: Return $L^{HSDL} \leftarrow$ **sort** L^{temp}

and $r(v_1) > \cdots > r(v_{|V|})$, where $deg(v_i)$ is the degree of v_i. HSDL pushes each $v_i \in V$ into a thread pool in a decreasing order of vertex ranks, for parallel and roughly sequentially processing each $v_i \in V$ as follows (Line 2).

HSDL generates labels with the hub vertex of v_i via a Dijkstra-style search. First, it initializes a min priority queue Q that contains a two-element tuple $(v_i, 0)$ with the priority of 0 (Line 3). While $Q \neq \emptyset$ (Line 4), it pops the top tuple (u, h_u) out of Q with the priority of d_u (Line 5). It tries to insert a label (v_i, h_u, d_u) into $L^{temp}(u)$ as follows. It only performs this insertion when $r(v_i) \geq r(u)$ (Line 6). Furthermore, it queries the h_u-hop-constrained shortest distance between u and v_i using L^{temp}. If the queries distance is larger than d_u (Line 7), then it inserts the label (v_i, h_u, d_u) into $L^{temp}(u)$ (Line 8). After that, if $h' = h_u + 1 \leq K$ (Line 9), it tries to push new elements into Q as follows. For each vertex $v \in N(u)$ (Line 10), it computes $d_v = d_u + w(u, v)$ (Line 11). If d_v is smaller than the priority of the two-element tuple (v, h') in Q (Line 12; if (v, h') is not in Q, then the priority is ∞), then it pushes (or updates) (v, h') into Q with the priority of d_v (Line 13).

An example of HSDL**.** Using Figure 1 as an example, after the HSDL-algorithm, the generated labels are as follows, with the format $(vertex, distance, hop - constraint, predecessor)$.

```
print_L:
L[0]={0,0,0,0}
L[1]={0,1,0,1}{0,0.8,2,2}{1,0,1,0}
L[2]={0,0.4,0,1}{1,0.4,1,1}{2,0,2,0}
L[3]={0,1,0,1}{1,1.1,2,2}{2,0.7,2,1}{3,0,3,0}
L[4]={0,0.8,0,1}{1,0.1,1,1}{4,0,4,0}
```

Figure 2: generated labels