Graph Indexing for Shortest-Path Finding over Dynamic Sub-Graphs

Mohamed S. Hassan
Purdue University
West Lafayette, IN, USA
msaberab@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN, USA
aref@cs.purdue.edu

Ahmed M. Aly
Purdue University
West Lafayette, IN, USA
aaly@cs.purdue.edu

ABSTRACT

A variety of applications spanning various domains, e.g., social networks, transportation, and bioinformatics, have graphs as first-class citizens. These applications share a vital operation, namely, finding the shortest path between two nodes. In many scenarios, users are interested in filtering the graph before finding the shortest path. For example, in social networks, one may need to compute the shortest path between two persons on a sub-graph containing only family relationships. This paper focuses on dynamic graphs with labeled edges, where the target is to find a shortest path after filtering some edges based on user-specified query labels. This problem is termed the *E*dge-*C*onstrained *S*hortest *P*ath query (or ECSP, for short). This paper introduces *E*dge-*D*isjoint *P*artitioning (EDP, for short), a new technique for efficiently answering ECSP queries over dynamic graphs. EDP has two main components: a dynamic index that is based on graph partitioning, and a traversal algorithm that exploits the regular patterns of the answers of ECSP queries. EDP partitions the graph based on the labels of the edges. On demand, EDP computes specific sub-paths within each partition and updates its index. The computed sub-paths are cached and can be leveraged by future queries. To answer an ECSP query, EDP connects sub-paths from different partitions using its efficient traversal algorithm. EDP can dynamically handle various types of graph updates, e.g., label, edge, and node updates. The index entries that are potentially affected by graph updates are invalidated and are re-computed on demand. EDP is evaluated using real graph datasets from various domains. Experimental results demonstrate that EDP can achieve query performance gains of up to four orders of magnitude in comparison to state of the art techniques.

1. INTRODUCTION

The ubiquity of location-based services, social networks, and other graph-dependent systems calls for ongoing research efforts in graph data management systems. One of the most important operations performed over graphs is finding a shortest path. Many applications require filtering the underlying graph first before computing a shortest path. To illustrate, let $\sigma_{predicate}(G)$ be a relational select operator that yields a sub-graph $\acute{G}$ as a result of filtering Graph G using some select predicate. Let $\S_{s,d}(\acute{G})$ be an operator that finds the shortest path from Node s to Node d in Graph $\acute{G}$. The following example queries illustrate useful applications of these operators when interacting together:

Biological networks: One may need to find the shortest path (that acts as a relatedness measure) between two proteins, say p_1 and p_2, under stable or covalent interactions among these proteins [22]. This can be expressed by $\S_{p_1,p_2}(\sigma_{interaction\ in(stable,\ covalent)}(\textit{ProteinNetwork}))$. In this example, a shortest-path computation needs to be performed on a subset graph that is dynamically specified at query time. Hence, any preprocessing on the original graph will be useless as only a subset of the graph is of interest to the shortest-path operation.

Road networks: A traveler from Location loc_1 to Location loc_2 may be interested in the shortest distance route with certain types of roads, e.g., avoid roads with construction work to prevent delays, or avoid toll roads. The latter can be expressed by $\S_{loc_1,loc_2}(\sigma_{roadtype \neq toll}(\textit{RoadNetwork}))$. Again, only the subset of the graph that is specified at query time is needed for the shortest-path operation, hence invalidating any preprocessing that took place on the entire graph.

Social networks: Some graph analysis techniques compute the shortest path between two persons, say p_1 and p_2, where the returned path has to use certain types of relationships (e.g., family relationships). This can be expressed by $\S_{p_1,p_2}(\sigma_{relation\ =\ family}(\textit{SocialNetwork}))$.

This paper introduces a new technique for finding the shortest path on a sub-graph, say G_{sub}, that is dynamically selected at query time. G_{sub} is selected by a predicate that uses a set of labels, say A, so that each edge in G_{sub} is labeled by at least one label in A. We term this query the *E*dge-*C*onstrained *S*hortest-*Path* query (or ECSP, for short). An ECSP query, say Q, is expressed as $Q(s,\ d,\ A)$, where s is the source vertex, d is the destination vertex, and A is the set of allowed labels.

A straightforward approach to answer an ECSP query is to use Dijkstra's algorithm [11]. During the traversal of the graph, we check the label for each edge, say e, on the fly. e is processed only if it satisfies the filtering predicates, or otherwise is discarded. Although this approach leads to a correct answer to the query, it does not leverage any preprocessing, and hence can be inefficient as it may traverse a large portion of the graph. This calls for novel techniques that can efficiently answer ECSP queries.

Several existing techniques address unconstrained shortest-path finding (e.g., [5, 7, 15, 19]). However, these techniques are not directly applicable to ECSP queries because they rely on indexes that do not assume any constraints on the graph edges. To illustrate, consider the ECSP query $Q(1,\ 6,\ \{Blue, Red\})$ using the graph in Figure 1, where Vertex 1 is the source vertex, Vertex 6 is the

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882933

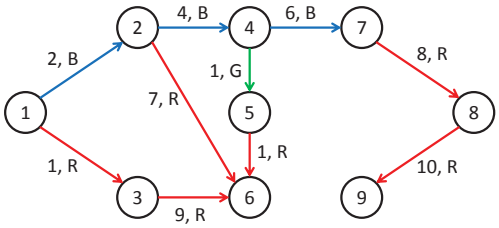


Figure 1: Edge-labeled Graph G with each edge having two values: $Weight$, $Label$. R, G, and B refer to the labels Red, $Green$ and $Blue$, respectively.

destination vertex, and $\{Blue, Red\}$ is the set of allowed edge labels. A typical index that pre-computes the shortest paths without considering the labels will find a shortest path from Vertex 1 to Vertex 6 with cost 8, i.e., the path given by $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$. However, this path is useless for Q (except for acting as a lower-bound). The reason is that the green edge from Vertex 4 to Vertex 5 is not allowed by Q. The correct answer to an ECSP query should be a feasible path with the least cost. The answer to Q is the path of cost 9 given by $1 \rightarrow 2 \rightarrow 6$.

This paper introduces *E*dge-*D*isjoint *P*artitioning (EDP, for short), a new technique that provides an exact answer to ECSP queries. EDP exploits regular expressions over the labels in the ECSP query-answers. Refer to Figure 1. Consider the ECSP Query $Q_j(1, 9, \{Blue, Red\})$. The shortest path P of Q_j is $1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9$. P has the following edge labels in order: $Blue$, $Blue$, $Blue$, Red, Red. The regular expression $(Blue* \; Red*)$ matches the ordered labels of P's edges, i.e., P begins with three $Blue$ edges followed by two Red edges. In general, Let Q_i be an ECSP query whose predicates allow only Labels $\{L_1, L_2, ..., L_k\}$. The labels for Q_i's correct answer should match the regular expression $(L_1 \mid L_2 \mid ... \mid L_k)$*. Specifically, EDP optimizes for paths of the form $(L_1* \mid L_2* \mid ... \mid L_i* \mid ... \mid L_k*)$+, i.e., the paths that have consecutive edges of the same label. We term each sub-path L_i* of the same label a *monochrome* sub-path. EDP supports fast access to the monochrome sub-paths corresponding to each label in the graph. For example, Path P has a monochrome sub-path that consists of three consecutive $Blue$ edges: $1 \rightarrow 2 \rightarrow 4 \rightarrow 7$. EDP provides fast access to this monochrome sub-path to construct Path P efficiently.

EDP realizes an incremental index to evaluate ECSP queries efficiently. During preprocessing, the input graph is partitioned so that any monochrome sub-path is constructed by consulting only one partition, i.e., each partition has edges of the same label. Incrementally, EDP maintains monochrome shortest sub-paths within each partition in a fixed-size cache. These sub-paths are shortcuts to efficiently answer future ECSP queries. However, these sub-paths are not aggressively precomputed for each partition. Instead, EDP applies a *build-as-you-go* mechanism that distributes the cost of the precomputations over all ECSP queries. To illustrate, assume that a shortest path, say P_i, of a query has a certain monochrome sub-path, say P_{ij}, and that P_{ij} has already been computed and cached for a previously answered query. Instead of recomputing P_{ij}, EDP directly reuses the cached index entry for P_{ij}. In other words, each sub-path, if needed, is computed only once unless it is invalidated by an update operation or is removed from the cache. Also, EDP uses an efficient traversal algorithm to build an ECSP from its monochrome sub-paths. EDP's fixed-size cache puts a cap on the storage size designated for the stored sub-paths, and hence avoids the quadratic growth in space requirements (see Section 5.4).

To the best of our knowledge, only two research efforts [8, 18] study ECSP queries. Bonchi et al. [8] compute an approximate answer to the problem. In contrast, EDP can compute an exact answer in a sub-millisecond. Rice et al. [18] present CHLR, an exact answer to ECSP queries. CHLR is tailored to road-network graphs as CHLR extends the contraction hierarchies technique [15]. In contrast, EDP is not tailored to any specific graph domain. As demonstrated in Section 8, EDP outperforms CHLR by up to four orders-of-magnitude w.r.t. query-processing time. Both [8,18] supports only static graphs, i.e., if the graph is updated by inserting or deleting edges or nodes, or by changing edge labels or weights, the indexes in [8, 18] need to be rebuilt. EDP is the first technique to process ECSP queries over dynamic graphs. Furthermore, EDP outperforms the state-of-the-art techniques on static graphs.

The contributions of this paper are as follows:

- We introduce EDP, a new technique for answering ECSP queries that is applicable to any graph domain. We present its complexity analysis as well as a proof of its correctness.
- We demonstrate how EDP handles graph updates efficiently.
- We present a dynamic indexing mechanism that amortizes the indexing cost by incrementally building the index of EDP. We introduce a fixed-size cache in EDP to cache and reuse already computed monochrome shortest sub-paths. Use of this cache puts a cap on the worst-case storage requirements for the stored sub-paths.
- We present an efficient index traversal algorithm that exploits the regular expressions in the answers of ECSP queries.
- We conduct extensive experiments using six real datasets including the Tiger dataset [3] and other graphs from different domains. Results demonstrate that EDP can achieve more than four orders-of-magnitude enhancement in query performance compared to CHLR, the state-of-the-art technique. Moreover, EDP's performance is stable and robust, as its variance when applying 95% confidence intervals is small for all reported speedup measurements.

2. EDGE-CONSTRAINED SHORTEST-PATH QUERIES

2.1 Problem Definition

Let $G = (V, E, L, l, w)$ be a directed weighted graph, where V is a set of vertexes, E is a set of positively weighted edges, L is a set of labels (that can be viewed as colors), l is a function that assigns a label to each edge, and w is a function that assigns a weight to each edge (i.e., $\forall e \in E, \exists l(e) \in L$ and $\exists w(e) \in \mathbb{R}^+$). Let $Q(s, d, A)$ be an edge-constrained shortest-path (ECSP) query, where $s \in V$ is the source vertex, $d \in V$ is the destination vertex, and $A \subseteq L$ is the set of allowed labels by Q. Q searches for a path $P = (e_1, e_2, \ldots, e_c)$ from s to d that uses only edges with labels from A (i.e., $\forall\, e \in P \mid l(e) \in A$) such that the summation $\sum_{e \in P} w(e)$ is minimized. Refer to Table 1 for a listing of the notations used in this paper.

2.2 Straightforward Approaches

One way to answer an ECSP query is to use an index, e.g., as in [5, 7, 15], that is designed for the unconstrained shortest-path problem (no restricted edges). This can be achieved by building $2^{|L|}$ indexes, where L is the set of labels of the underlying graph G (i.e., the powerset of L). To answer a query, one of the $2^{|L|}$ indexes that corresponds to the labels permitted by the query is selected. Clearly, the exponential space of the required indexes

Notation	Description
G	A directed, weighted, and labeled graph
V	A set of vertexes
E	A set of edges
L	A set of labels
$w(e)$	Weight of Edge e
$l(e)$	Label of Edge e, e can be represented by the vertex-identifiers of its endpoints
$Q_i(s, d, A)$	ECSP Query Q_i from Vertex s to Vertex d with the allowed Labels-set A
$I(G)$	Edge-disjoint index for Graph G
$sp(s, d)$	Unlabeled shortest path from Vertex s to d
$CP(P)$	Contracted sub-paths of Path P
$Pr_i(v)$	Vertex v in Partition Pr_i

Table 1: Frequently used notations

makes this approach impractical. Also, this approach cannot handle graph updates efficiently.

Another straightforward approach is to modify any traditional shortest path algorithm (e.g., Dijkstra) to consider only the allowed edges of an ECSP query. The main drawback of this approach is that it may explore most of the graph edges if the query has low selectivity (i.e., most of the labels are allowed by the query). Moreover, this approach will perform an exhaustive traversal if the destination is not reachable from the source using the allowed labels. Hence, this approach is unlikely to scale for large graphs.

3. RELATED WORK

Since the 1950s, the problem of finding the shortest path has gained extensive attention (e.g., see [20,21]). Tremendous research efforts have been conducted to support shortest-path querying (e.g., see [5,7,15,19]). In this section, we discuss two main categories of related work: 1) existing approaches for answering unconstrained shortest-path queries that can be modified to support ECSP queries, and 2) existing approaches for answering ECSP queries.

In the first category, several techniques have been proposed to preprocess a graph, say G, to enable fast computation of unconstrained shortest-path queries. Goldberg et al. [16] present a two-hop approach for answering unconstrained shortest-path queries. The main idea is to select a set of landmark vertexes $LM \subseteq G.V$ such that for any shortest path, say $sp(u,\ v)$, where $u, v \in G.V$, $\exists$ a vertex, say $w \in\ LM$, that lies on $sp(u,\ v)$. The two-hop approach is not directly applicable to ECSP queries as the shortest paths from/to the vertexes of LM do not consider any labels. One modification to this approach is to build a separate graph G_{sl} for each possible set of labels sl. G_{sl} will contain only edges of G with labels in sl. Unfortunately, this modified approach requires $O(2^{|L|})$ space and time complexities as the number of different subsets of labels is $O(2^{|L|})$, and hence it will not scale.

In [9], the idea of landmarks is extended to dynamic graph settings, where the weights of the edges may change. However, the techniques in [9] consider only unconstrained shortest path queries. One approach to answer ECSP queries using the techniques in [9] is as follows. Given a query, we set the weights of all the disallowed edges to positive infinity, and re-adjust these weights to their original values after the query is answered. Clearly, this approach is not scalable to large graphs especially if the queries are highly selective (many graph updates will be performed).

In the second category, Bonchi et al. [8] present approximate answers for ECSP queries based on landmark vertexes. They propose two types of indexes, namely, PowCov and ChromLand that exhibit interesting trade-offs between index size and accuracy. However, both indexes are not suitable for applications that require exact shortest-path computations.

CHLR [18] is the state-of-the-art technique that can answer ECSP queries exactly. CHLR has been extended to support more flexible edge restrictions [14]. CHLR adopts Contraction Hierarchies (CH, for short) [15] to provide an exact answer to the following query: Given a source, say s, a destination, say d, and a set of restricted labels, say R, retrieve the shortest path from s to d that avoids all the edges with labels in R. An ECSP Query $Q(s,\ d,\ A)$ can be answered by CHLR after computing the complement of Set A. The main idea of CHLR is similar to that in [15]. The main difference is that CHLR considers edge-labels while contracting the underlying graph nodes. In particular, when contracting a node, say v, and there is a sub-path passing through the vertexes u, v, and w (i.e., $u \rightsquigarrow v \rightsquigarrow w$) such that a shortcut ($u \rightsquigarrow w$) is added, the shortcut ($u \rightsquigarrow w$) is labeled by the labels of the sub-paths ($u \rightsquigarrow v$) and ($v \rightsquigarrow w$). Labeling the shortcuts with labels enables CHLR to avoid these shortcuts if they contain any restricted label.

CH is originally designed for road-network graphs, and so is CHLR as it uses CH at its core. Consequently, CH has assumptions that are not valid for other networks. For example, CH assumes that the average fan-out of the vertexes is small (e.g., 2). The small fan-out assumption is valid for road networks, but is invalid for graphs from other domains. For example, an average fan-out of 15 is common in biological networks. In CHLR, when contracting a vertex with high fan-out, many shortcuts will be added. Adding many shortcuts by CHLR negatively affects its performance as the sparsity of the traversed graph will decrease. EDP, as proposed in this paper, performs up to four orders-of-magnitude faster than CHLR. Moreover, EDP supports dynamic graphs and is not tailored to road networks. To the best of our knowledge, EDP is the first technique that processes ECSP queries over dynamic graphs, and yet outperforms the state-of-the-art on static graphs by up to four orders-of-magnitude.

Variants for finding Regular-Language-Constraint-Paths (RLCP) have been studied [6, 10, 17]. An RLCP query works on edge-labeled graphs where the concatenation of the labels of the found path satisfies a query-specified regular expression. Unlike ECSP, RLCP queries assume that the user knows the exact order of the labels. Hence, RLCP queries are orthogonal to ECSP queries because ECSP queries do not impose any order on the allowed labels.

4. OVERVIEW OF EDP

A monochrome sub-path consists of edges with the same label. We define a contracted path of an ECSP query answer P as follows:

DEFINITION 1. *The* **contracted path** of Path P, denoted by $CP(P) = (P_1,\ P_2, \ldots,\ P_i,\ \ldots,\ P_m)$, is an ordered list of monochrome sub-paths that, if concatenated, will produce P. A monochrome sub-path $P_i \in CP(P)$ is the maximum contraction of a set of consecutive edges with the same label.

To illustrate how a contracted path is formulated, let $Q_i(v_a,\ v_f,\ \{L_1,\ L_2\})$ be an ECSP query. Assume that the answer P' to Q_i is represented by the edges $((v_a,\ v_b),\ (v_b,\ v_c),\ (v_c,\ v_d),\ (v_d,\ v_e),\ (v_e,\ v_f))$, where the labels of these edges are (L_1, L_1, L_1, L_2, L_2), respectively. Then, $CP(P') = ((P'_1(v_a \rightsquigarrow v_d),\ P'_2(v_d \rightsquigarrow v_f))$ represents the contracted sub-paths of P', namely P'_1 and P'_2, such that each sub-path is monochrome and contains the maximum possible number of consecutive edges with the same label.

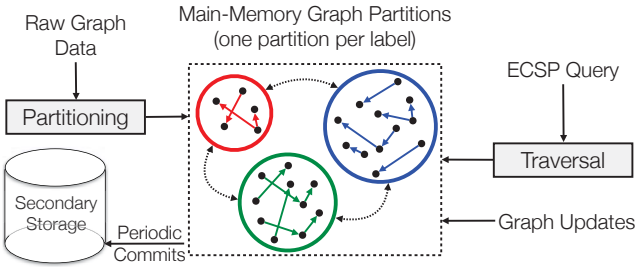


Figure 2: An Overview of EDP.

EDP consists of two main components:

1. **Indexing:** treats the contracted sub-paths of any ECSP query as independent tuples, and indexes these tuples.
2. **Traversal:** connects the appropriate contracted sub-paths efficiently to form the answer of any ECSP query.

Figure 2 gives a high-level overview of EDP. First, a raw graph is partitioned so that constructing any monochrome sub-path is doable by consulting only one partition. Thus, EDP constructs one partition per graph label. A partition corresponding to Label L_i hosts all the graph edges with Label L_i. Given an ECSP query, say Q, the traversal algorithm visits the partitions necessary to produce Q's answer. Each partition computes local shortest sub-paths. These local shortest sub-paths are cached in main memory to be leveraged by future queries (as explained in Section 6). To process graph updates, EDP assigns timestamps to each index entry. These timestamps help decide if some index entries need to be recomputed. EDP periodically commits the cached shortest sub-paths to secondary storage to avoid recomputing them upon system restart.

5. INDEX CONSTRUCTION

For a directed weighted graph G, the index $I(G)$ can be viewed as two integrated components: 1) a set of graph partitions, and 2) a set of lookup tables, where some shortest monochrome sub-paths are stored. The lookup tables are continuously updated as queries arrive. Section 5.1 discusses the graph partitioning of EDP while Section 5.2 describes the lookup tables of the shortest sub-paths.

5.1 Graph Partitioning

EDP partitions an input graph so that any two edges of different labels cannot co-exist in the same partition. Thus, each graph label, say c, has a corresponding partition, say Pr_c, that holds all the graph edges with Label c. The intuition behind this partitioning is twofold. First, only the partitions corresponding to a given set of query labels are considered and all the other partitions are safely discarded. This reduces the traversal search space. Second, any contracted sub-path, say p_i, that is part of the query answer will be computed using only the edges of the partition corresponding to the label of p_i (recall that a contracted sub-path is monochrome). Thus, each partition can cache all the contracted sub-paths it computes so that they can be leveraged by future queries. Hence, a contracted sub-path is computed once from scratch unless it is potentially affected by a graph update (see Section 7). Afterwards, it is recalled from the cache as needed.

Algorithm 1 constructs an edge-disjoint index I for a given graph G. Let L be the set of graph labels in G. Algorithm 1 creates a partition for each label and assigns an integral identifier to it (Lines 2-4). Each Label $i \in \{0, 1, ..., |L| - 1\}$ has a corresponding Partition Pr_i. The edges in Pr_i are defined by $Pr_i.E = \{(u, v) \in G.E \mid l(u, v) = i\}$, i.e., Pr_i contains all the edges of G that are labeled with Label i. To assign edges to each partition, the graph edges E are scanned only once (Lines 5-18). The $|L|$ partitions form an edge-disjoint partitioning of G.

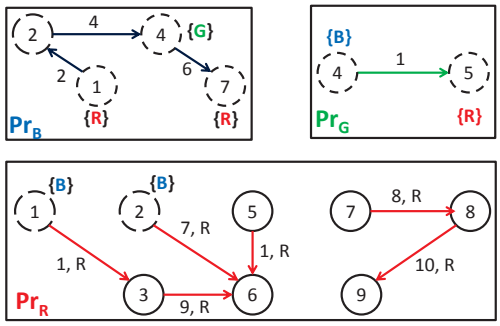


Figure 3: Edge-disjoint partitioning of the graph in Figure 1.

To illustrate Algorithm 1, we partition Graph G of Figure 1 into the partitions of $I(G)$ in Figure 3. G has three labels. Thus, Algorithm 1 creates 3 partitions and assigns an integer identifier to each label, say $\{R = 0, B = 1, G = 2\}$. Thus, we have the 3 partitions Pr_R, Pr_B, and Pr_G that contain the red, blue, and green edges, respectively. As we explain in Section 6, the traversal algorithm traverses only $I(G)$ and not G. Thus, $I(G)$ should preserve all the connections of G. To preserve the connectivity of G, Algorithm 1 annotates some vertexes with properties as defined below.

DEFINITION 2. **Bridge Vertex:** Given Partitions Pr_i and Pr_j, a vertex $v \in Pr_i$ is termed a bridge vertex in Pr_i if and only if $\exists$ an edge, say $(v,\ u) \in\ Pr_j$ where $i \neq j$.

In Figure 3, bridge vertexes are marked with dashed circles. For instance, Vertex 5 in Partition Pr_G is a bridge vertex because Vertex 5 has an outgoing edge hosted by another partition (i.e., Pr_R). However, Vertex 5 in Partition Pr_R is not a bridge vertex because there are no outgoing edges from Vertex 5 that are hosted by a partition other than Pr_R. We denote a vertex instance v hosted by Partition Pr_i by $Pr_i(v)$.

DEFINITION 3. **OtherHosts List:** Given a bridge vertex, say $v \in Pr_i$ (i.e., $Pr_i(v)$), the OtherHosts list of $Pr_i(v)$ is the set of label identifiers of all the outgoing edges of $v \in G$ not equal to the identifier of Label i, where i is the label identifier of Partition Pr_i. Formally, $Pr_i(v).OtherHosts = \{j | \exists (v, u) \in G.E(l(v, u) = j \wedge j \neq i)\}$.

In Figure 3, the OtherHosts list of each bridge vertex is listed in curly braces. For instance, the bridge vertex $Pr_G(5)$ has its OtherHosts list set to $\{R\}$, where R is the label identifier of the red label. The reason is that Vertex 5 in G (as in Figure 1) has an outgoing Red edge to Vertex 6. Notice that there is a one-to-one mapping between a label identifier, say R, and its corresponding partition, say Pr_R. Hence, the OtherHosts list of $Pr_G(5)$ indicates that Partition Pr_R hosts an outgoing Red edge of Vertex $Pr_G(5)$. In Algorithm 1, Lines 9-13 flag the bridge vertexes and set their OtherHosts lists. The OtherHosts lists are used by the traversal algorithm to connect contracted sub-paths of different labels to form the requested shortest path (see Section 6). Notice that if a vertex, say v, is replicated in more than one partition, v keeps the same local identifier in each hosting partition. However, each vertex instance in a partition has a global identifier consisting of its local identifier as well as the partition identifier hosting that instance.

Algorithm 1 EDP-Partitioning ($G < V, E, L, l, w >$)

```
1: I.PlainGraph ← G
2: for each label identifier i ∈ G.Labels do
3:     I.CreatePartition(i)
4: end for
5: for each edge (u, v) ∈ E do
6:     Partition ← I.getPartition(l(u, v))
7:     for each vertex a ∈ {u, v} do
8:         if !Partition.Contains(a) then
9:             isBridge ← false
10:            OtherHosts ← {l(a, b) ∈ E|l(a, b) ≠ l(u, v)}
11:            if OtherHosts ≠ ∅ then
12:                isBridge ← true
13:            end if
14:            Partition.addVertex(a, isBridge, OtherHosts)
15:        end if
16:    end for
17:    Partition.addEdge(u, v, w(u, v))
18: end for
19: return I
```

Observe that it is possible that the partitions become unbalanced, e.g., some partitions have most of the edges while the other partitions are almost empty. This unbalance does not affect the efficiency of EDP. In fact, in some cases, it is beneficial to have large partitions because we will be able to precompute more effective shortcuts in these partitions. These shortcuts are leveraged by the traversal algorithm. The only disadvantage for having large partitions is the inflation in the number of reachable bridge vertexes from the partition's vertexes. We address this issue in Section 6.2.

5.2 Repository for Shortest Sub-Paths

EDP stores a set of shortest paths in each partition. These shortest paths are used by the traversal algorithm as monochrome sub-paths to construct the final query answer. Index traversal is discussed in detail in Section 6.1. For a given ECSP Query $Q(s,\ d,\ A)$, the traversal algorithm keeps a current vertex instance, say $Pr_i(v)$. When $Pr_i(v)$ is the current vertex instance, EDP needs to be aware of only the following two types of paths in Pr_i, if any exists: 1) the shortest path from $Pr_i(v)$ to Destination d if d is hosted by Pr_i, 2) the shortest paths from $Pr_i(v)$ to the bridge vertexes in Pr_i. The latter allows EDP to consider other possible sub-paths hosted by other partitions.

DEFINITION 4. **Bridge Shortcut Edge:** Given any two vertexes hosted by the same partition, say $Pr_i(u)$ and $Pr_i(v)$, where $Pr_i(v)$ is a bridge vertex, if there is a monochrome shortest path of Label i from $Pr_i(u)$ to $Pr_i(v)$, that path is termed a bridge shortcut edge, or **bridge edge**, for short.

For example, consider the two vertexes $Pr_B(1)$, and $Pr_B(7)$ in Partition Pr_B of Figure 3. Vertex $Pr_B(7)$ is a bridge vertex that has an incoming monochrome shortest path from Vertex $Pr_B(1)$ (call it $P = 1 \rightarrow 2 \rightarrow 4 \rightarrow 7$). P is a bridge edge and is represented as a shortcut edge from $Pr_B(1)$ and $Pr_B(7)$ in Figure 4.

So, each partition in EDP holds a set of shortest paths stored in hash tables. Each vertex in a partition can possibly be a source, destination, or an intermediate node in a shortest path. This suggests that we might need a comprehensive list of all the shortest paths between all the possible pairs in any partition in order to support any query. Although this process can be performed as part of Algorithm 1 when constructing the index, this does not scale for large partitions. The space complexity of a partition's shortest-path lookup will be quadratic w.r.t. the number of the partition's vertexes ($O(|V|)$); also the pre-processing time for computing the all-pairs shortest paths will be $O(|V| * (|E| + |V| log(|V|))$. Fortunately, EDP does not have to aggressively precompute these comprehensive shortest paths. Instead, EDP builds incrementally the shortest-paths repository in response to the queries received. Also, EDP limits the index size so that it does not exceed a user-specified size. EDP may replace some existing index-entries by new entries in order not to exceed the index maximum size (See Section 5.4).

A comprehensive construction of all-pairs shortest paths of a partition's vertexes assumes that all the vertexes in that partition will be used as sources and destinations for queries. However, usually not all the vertexes in a partition are queried as sources and destinations. For example, on a road network, we may have hotspot vertexes (as destinations) to which the shortest paths need to be computed, while many other vertexes are unlikely to be queried as destinations (e.g., a vertex on a highway that does not represent any point of interest). Hence, we construct a partition's shortest-paths list as queries arrive. The computations performed to serve a query, say Q_i, can serve a future query, say Q_j. Thus, the cost to update the index by computing a shortest path, say sp, is amortized over all the queries that use sp as part of their path discovery process.

5.3 Index Operations

EDP supports the following index operations:

isBridge(pr, v): Returns true if Vertex $pr(v)$ is marked as a bridge vertex. This operation takes constant time.

getCost(pr, v, u): Returns the cost of the shortest path from Vertex v to Vertex u inside Partition pr. If one or both of u and v are not hosted by pr, or they are both hosted by pr but not connected in pr, positive infinity is returned, and an index entry is added to flag that $pr(v)$ and $pr(u)$ are not connected. $getCost$ uses a hash table to store and retrieve a path cost. The key of the hash table is a function of both v and u.

getBridgeEdges(pr, v): Returns all the bridge edges from Vertex $pr(v)$ to all the reachable bridge vertexes in pr. The returned list is sorted in ascending order by the cost of the edges in a hash table. Section 6.2 explains how $getBridgeEdges$ is implemented as a non-blocking operator, i.e., a caller to $getBridgeEdges$ will not be suspended until the whole list is computed.

5.4 Index Size

Although EDP replicates some vertexes in more than one partition (e.g., Vertex 1 is replicated in Pr_R and Pr_B of Figure 3), the replicated vertexes are *light-weight*, i.e., only the vertex identifier and some connectivity information are replicated, but not the whole vertex's objects that are stored in G.

For Graph G, let n be the number of vertexes, m be the number of edges, c be the number of labels, and a be the number of vertexes that have outgoing edges of different labels. Assume that the maximum fan-out of vertexes in G is f. Then, the number of bridge vertexes in Index $I(G)$ is $O(af) = O(nf)$ in the worst-case, when all the vertexes are bridge vertexes. For the partitions component of the index, in the worst case, the space complexity is $O(m + nfc)$. Observe that the term (nfc), has the factor c because, in the worst-case, a bridge vertex will be replicated in c partitions. As the edges are distributed among the partitions without replication, the space complexity of the index has the term m.

The repository for shortest sub-paths determines the size of EDP's index, however, this size is query-driven. Notice that the repository for shortest sub-paths does not store full query answers. Instead, monochrome sub-paths are stored to help construct answers of different queries. Although the number of queries that vary only in the source/destination vertexes is $O(n^2)$, the number of index entries in practice does not reach $O(n^2)$ for two rea-

sons: 1) EDP does not store full paths, and 2) Not all the vertexes of ECSP queries are designated as sources/destinations. To avoid any chance of quadratic space-growth, EDP limits the index size by adopting a least-recently-used (LRU) replacement policy. Section 8.4 shows that EDP's index does not exceed a few gigabytes after processing millions of queries over real large graphs. The replacement policy of EDP tracks the index entries usage by a doubly linked list to quickly determine the LRU entries. Although the index of EDP has a light memory-footprint (see Section 8.4), EDP regularly monitors the hit-rate of the index and writes it into an event log. Writing the hit-rate into the event log helps an administrator decide if the limit of the index needs to increase according the current query-workload.

6. QUERY PROCESSING

EDP's query processing algorithm (termed EDP-QP, for short) follows a greedy traversal approach. The greedy traversal connects the source vertex to the vertexes with least cost. The traversal continues until reaching the destination vertex. Once the destination vertex is reached, the shortest cost is obtained, and the shortest path can be determined.

6.1 EDP Index Traversal

Given an ECSP Query $Q(s,\ d,\ A)$ over Graph G, EDP-QP uses Index $I(G)$ to answer Q. As EDP-QP traverses I(G), all the vertexes referenced by EDP-QP are identified by their global identifiers (e.g., $Pr_i(v)$ to refer to the instance of Vertex v hosted in Partition Pr_i). When EDP-QP traverses Partition Pr_i, it only traverses the shortcut edge to the destination node if hosted by Pr_i as well as some bridge edges of Pr_i. When processing Query Q, only the partitions corresponding to the allowed labels of Q are considered; the other partitions are ignored.

EDP-QP uses a min-priority Queue PQ of vertexes whose final shortest-path weights from the source vertex have not been determined yet. The structure of PQ is keyed by the 2D global vertex-identifiers of EDP (i.e., partition Id and vertex Id). An element in PQ has the following attributes: 1) Pr: a partition's identifier, 2) v: the vertex's identifier that is reachable from the source and is hosted by Pr, and 3) $cost$: the least cost observed so far that connects the source vertex to Vertex $Pr(v)$, where the elements in the min-priority queue are ordered by $cost$. Two additional attributes in the queue will be introduced in Section 6.2.

EDP-QP uses a two-dimensional $cost$ table to maintain the least-costs observed so far to connect the source vertex to the visited vertexes of $I(G)$ (regardless of whether the visited instances of vertexes are dequeued or not). A key of the $cost$ lookup table is identified by: 1) a partition identifier, and 2) a vertex identifier. For a given key of the $cost$ table, say k, the value that corresponds to k is a structure holding k as well as the least-cost observed so far to reach the vertex identified by k from the source vertex.

Algorithm 2 outlines the traversal Algorithm EDP-QP that returns only the cost of the shortest path. Constructing the actual path is straightforward (see Appendix B). Given an ECSP Query $Q(s,\ d,\ A)$ and $I(G)$, the partitions of $I(G)$ are traversed in a greedy way to find a feasible path with minimum cost if one exists. At any point in time, Algorithm 2 will have a current vertex, say $Pr_i(v)$. Algorithm 2 checks if there is a sub-path in Partition Pr_i that reaches Destination d from Vertex $Pr_i(v)$. If this sub-path exists, Algorithm 2 will have a feasible path, say P_i, that can be a shortest path. To consider other feasible paths that can be shorter than P_i, Algorithm 2 considers other allowed edges hosted by other partitions. Edges of other allowed partitions can be reached through the bridge vertexes of Partition Pr_i. Algorithm 2 uses the bridge

Algorithm 2 EDP-QP(I, s, d, A)

```
1:  PQ ← ∅, cost(p, v) ← ∅
2:  if s and d have edges labeled by at least one label of A then
3:      PQ.Insert(getPr(s, A), s, 0))
4:  end if
5:  while PQ.NotEmpty() do
6:      t ← PQ.ExtractMin()
7:      if t.v = d then
8:          return t.Cost
9:      end if
10:     PQ.InsertIfRelaxed(t.Pr, d, t.Cost + I.w(t, d)))
11:     if I.isBridge(t) then
12:         for each pr ∈ t.v.OtherHosts ∩ A do
13:             PQ.InsertIfRelaxed(pr, t.v, t.Cost)
14:         end for
15:     end if
16:     for each e ∈ I.getBridgeEdges(t.Pr, t.v) do
17:         for each pr ∈ e.To.OtherHosts ∩ A do
18:             PQ.InsertIfRelaxed(pr, e.To, t.Cost + e.getWeight())
19:         end for
20:     end for
21: end while
22: return ∞
```

vertexes of Partition Pr_i that are reachable from Vertex $Pr_i(v)$ to explore other feasible shorter paths. To explore other feasible paths from Vertex $Pr_i(v)$ through a bridge vertex, say $Pr_i(b)$, the $OtherHosts$ list of Bridge Vertex $Pr_i(b)$ is intersected with the allowed query labels (i.e., $Q.A$). The intersection of the label sets $Pr_i(b).OtherHosts$ and $Q.A$ determines partitions that can form other feasible paths.

Algorithm 2 does not explore all possible feasible paths. As soon as the current vertex instance of Algorithm 2 is a destination vertex instance, Algorithm 2 yields the shortest-path cost and terminates. To illustrate all the logical branches of Algorithm 2, we use two query examples using the partitions in Figure 3.

Consider Query $Q_1(1,\ 6\ \{R\})$. EDP-QP tests if a feasible path can exist. No feasible path exists if the source vertex has no outgoing edge labeled by Label R. Q_1 passes this necessary-but-not-sufficient test (Lines 2-4), and the vertex $Pr_R(1)$ will be added to Priority Queue PQ with cost zero. EDP-QP traverses $I(G)$ as long as the shortest path is not determined and PQ is not empty (Lines 5-21). For Q_1, the initial status of PQ is $((Pr_R,\ 1,\ 0))$. After extracting $(Pr_R,\ 1,\ 0)$ from PQ, the current vertex will be $Pr_R(1)$. As the current partition Pr_R hosts an instance of the destination Vertex 6, EDP-QP will compute the shortest path $Pr_R(1) \rightsquigarrow Pr_R(6)$, and will relax (decrease) the cost to the destination Vertex $Pr_R(6)$. Initially, the cost of reaching all the vertexes except the source is positive infinity. Relaxing the cost of reaching $Pr_R(6)$ will update $cost(Pr_R,\ 6)$ to 10, and will add $Pr_R(6)$ with the updated cost to PQ (Line 10). The status of PQ becomes $((Pr_R,\ 6,\ 10))$. As only red edges are allowed, EDP-QP will not explore other partitions. In the next iteration, the current vertex becomes $Pr_R(6)$. As $Pr_R(6)$ corresponds to the Destination Vertex 6, EDP-QP returns the shortest cost of 10 (Line 8).

To illustrate how EDP-QP traverses different partitions, consider Query $Q_2(1,\ 6\ \{R,\ B\})$. Q_2 uses the Index $I(G)$ of Figure 3. The edges traversed to answer Q_2 are given in Figure 4. The dashed edges in Figure 4 do not physically exist. They are there to illustrate the sequence of traversing the different partitions to answer Q_2. As the source Vertex 1 has two allowed partitions hosting two instances of Vertex 1, one instance should be selected to start the traversal. Procedure $getPr$ (Line 3) selects a partition hosting an allowed outgoing edge of Vertex 1 with the minimum edge cost. For Q_2, $getPr$ selects Partition Pr_R as it hosts an outgo-

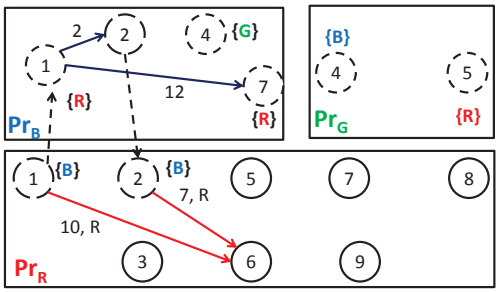


Figure 4: Index traversal for the shortest path from Vertex 1 to Vertex 6 using only Red and Blue edges.

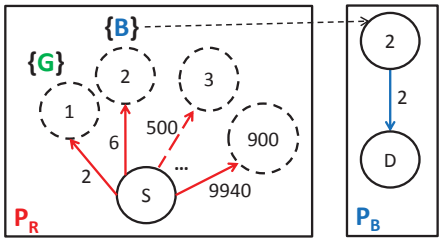


Figure 5: Example of a vertex with many reachable bridge vertexes (Vertex S). Dashed vertexes are bridge vertexes.

ing edge of Vertex 1 with the least cost (see Figure 1). So, PQ is initially set to $((Pr_R,\ 1,\ 0))$. At the beginning of the traversal, the current Vertex becomes $Pr_R(1)$. As the destination Vertex 6 is hosted by Pr_R and is reachable from the current vertex, the shortest Red path from $Pr_R(1)$ to $Pr_R(6)$ is retrieved from the index, and PQ is updated to $((Pr_R,\ 6,\ 10))$. As $Pr_R(1)$ is a bridge vertex, its $OtherHosts$ list is intersected with the allowed labels $\{R,\ B\}$); the set intersection determines other partitions hosting outgoing edges of the current vertex that are allowed by Q_2. The bridge vertex logic handling (Lines 11-15) causes the following update to PQ $((Pr_B,\ 1,\ 0),\ (Pr_R,\ 6,\ 10))$. Notice that the entry corresponding to $Pr_R(6)$ in PQ imposes an upper bound of 10 to the query answer. In the next iteration, $Pr_B(1)$ becomes the current vertex. Although $Pr_B(1)$ is a bridge vertex, its $OtherHosts$ list will not cause any cost relaxation, i.e., the cost of $Pr_R(1)$ is already zero. Thus, Lines 11-15 will not update PQ. EDP-QP will not find Vertex 6 in Pr_B. However, EDP-QP will try to leave Partition Pr_B searching for the destination vertex. As in Figure 4, leaving Pr_B is possible through the bridge vertexes $Pr_B(2)$, $Pr_B(4)$ and $Pr_B(7)$ that are reachable from $Pr_B(1)$. Edges $(Pr_B(1),\ Pr_B(2))$, $(Pr_B(1),\ Pr_B(4))$ and $(Pr_B(1),\ Pr_B(7))$ are called the bridge edges of Vertex $Pr_B(1)$. As the $OtherHosts$ list of the Bridge-vertex $Pr_B(4)$ does not contain allowed labels, the Bridge-vertex $Pr_B(4)$ is discarded. However, $Pr_B(2)$ and $Pr_B(7)$ have allowed labels in their $OtherHosts$ list. Lines 16-20 handle the bridge edges of the current node and cause the following update to PQ $((Pr_R,\ 2,\ 2),\ (Pr_R,\ 6,\ 10),\ (Pr_R,\ 7,\ 12))$. In the next iteration, the current vertex becomes $Pr_R(2)$ that can reach the destination Vertex $Pr_R(6)$. Relaxing the cost of reaching $Pr_R(6)$ from $Pr_R(2)$ (Line 10) causes the following status of PQ $((Pr_R,\ 6,\ 9),\ (Pr_R,\ 6,\ 10),\ (Pr_R,\ 7,\ 12))$. The following iteration will have $Pr_R(6)$ as the current vertex. Hence, EDP-QP will return the exact shortest cost 9 for Q_2.

Notice that any computed bridge edges of a vertex (e.g., bridge edges of Vertex $Pr_B(1)$ computed by Q_2) will be computed from scratch only once unless it is invalidated by a recent graph-update. Other queries that need any saved bridge edges will leverage them from the index to avoid any recomputations as long as they are not out-of-date (see Section 7). As we explain in Section 6.2, EDP-QP computes bridge edges by a non-blocking operator that runs on a separate thread. This allows the traversal algorithm to continue traversing $I(G)$ while the bridge edges are still being computed.

6.2 Handling Large Bridge Vertexes

For large graph datasets, the partitioning scheme formed by Algorithm 1 may lead to many bridge vertexes in one partition. The disadvantage of this case is the increased fan-out of the vertexes (recall that the traversal algorithm can explore all the outgoing bridge edges of a given vertex). EDP handles this issue by not exploring all the possible transitions at once, i.e., the bridge edges of the current node are not relaxed all together. To illustrate the intuition behind this handling, consider a road-network graph. If the current node is a commuter's location on a highway, and EDP wants to compute a toll-free exit to a certain destination, it is not wise to consider all the possible reachable exits from the current position (some of them will be very far and not of interest). The intuition is to consider a small near subset of all the possible exits, and the commuter will most likely reach his/her destination by a cost less than the cost of reaching other far exits (i.e., bridge vertexes that are reachable by high costs).

For example, in Figure 5, Vertex S has 900 bridge edges. If we insert all 900 reachable bridge vertexes in PQ, we will issue 900 insertion operations in PQ. Moreover, ExtractMin on PQ will be a function of an enlarged number of elements in PQ. Thus, we modify the way Algorithm 2 handles bridge vertexes. Instead of performing a breadth exploration when processing a current node (e.g., S in Figure 5), EDP follows a hybrid traversal approach without losing the correctness of Algorithm 2. Recall that the outgoing bridge edges of any vertex are sorted by the edge weights. EDP defines a system parameter, termed $MaxBreadth$, that is set to a positive integer. EDP sets $MaxBreadth$ to the average fan-out of the vertexes in G having at least two edges with different labels (in order not to increase the average fan-out observed by the traversal algorithm). $MaxBreadth$ ensures that at any iteration of the main while loop of Algorithm 2 (Lines 5-21), the maximum number of explored outgoing edges of the current vertex is not going to exceed the value of $MaxBreadth$.

To preserve the correctness of EDP-QP, we add two new attributes to the structure of elements of PQ (namely, $edgeId$ and $costRank$). An element PQ becomes a tuple having the following five attributes: 1) Pr: a partition's identifier, 2) v: the identifier of a vertex that is reachable from the source and hosted by Pr, 3) $cost$: the least cost observed so far that connects the source vertex to Vertex v, 4) $edgeId$: the identifier of the outgoing edge of Vertex v that should be investigated when the element is dequeued (initialized to zero), and 5) $costRank$: the key of the PQ element (initialized to the same value of cost); $costRank$ has the same domain as the $cost$ attribute, but its value may differ.

We describe the modifications required for Algorithm 2 to handle the possibility of having many reachable bridge vertexes from the current vertex. In particular, we describe the modifications to Lines 16-20. Before the for-loop at Line 16, EDP-QP defines a counter that starts at 0 and that is incremented at each iteration of the for-loop. The body of the for-loop will have an extra check using the defined counter and the value of $MaxBreadth$. If EDP-QP already investigated $MaxBreadth$ outgoing edges of the current vertex, it performs these steps:

Step 1: Update the $edgeId$ and $costRank$ attributes of the current

element t so that t can be safely re-enqueued into PQ. Notice that EDP-QP should remember the id of the edge, say e, at which the investigation stops as well as the potential cost if e is relaxed. For example, if the last investigated outgoing edge of the current vertex has index k (w.r.t. the sorted list of outgoing bridge edges from the current vertex), we set $edgeId$ to $k+1$ and set $costRank$ to $t.Cost$ + the cost of the $(k+1)^{th}$ edge.

Step 2: Insert Element t to PQ again with an updated $costRank$ (to ensure that we do not lose a potential path).

Step 3: At every investigation of the outgoing bridge edges of a current vertex, EDP-QP starts from the edge specified by $t.edgeId$ (edges with id less than t.edgeId have been already investigated).

Refer to Figure 5. Let $MaxBreadth$ be equal to 2. PQ is initialized to $((P_R, S, 0, 0, 0))$. Then, we relax up to 2 outgoing bridge edges. So, PQ becomes $((P_R, 1, 2, 0, 2), (P_R, 2, 6, 0, 6), (P_R, S, 0, 2, 500))$. Notice that only two elements are added to PQ plus the element $(P_R, S, 0, 2, 500)$ (name it $delayedEntry$) instead of adding 900 elements. The $delayedEntry$ indicates that EDP-QP should consider refetching Vertex $P_R(S)$ from PQ to explore its third outgoing edge with $costRank$ 500 ($edgeId$ is zero-based). The cost of the $delayedEntry$ does not change as we have not reached Vertex 3 yet. However, costRank is set to 500 as EDP-QP should keep the correct order of resuming the processing of Vertex $P_R(S)$ (500 is the cost of reaching Vertex $P_R(3)$ from Vertex $P_R(S)$).

In the figure, Element $(P_R, 1, 2, 0, 2)$ will not be further explored as it does not have allowed access paths (marked as having only outgoing edge(s) with Label G), and hence will be ignored. Entry $(P_R, 2, 6, 0, 6)$ causes a transition to Partition P_B. Now, PQ contains $((P_B, 2, 6, 0, 6), (P_R, S, 0, 2, 500))$. Element $(P_B, 2, 6, 0, 6)$ gets processed in the next iteration and PQ will then include: $((P_B, D, 8, 0, 8), (P_R, S, 0, 2, 500))$. EDP-QP reaches the destination node when extracting the first element of PQ. Hence, a minimum cost of 8 is returned without exploring the delayed outgoing edges of S.

Non-blocking Operator for Computing Bridge-edges. Recall that the bridge edges of any vertex, say $Pr(v)$, are computed from scratch only once. Assume that EDP-QP needs $Pr(v)$'s bridge-edges for the first time. If $Pr(v)$ has many bridge-edges, EDP-QP would block until all bridge-edges are computed, and the query response time would increase. EDP-QP handles this issue by computing the bridge-edges of $Pr(v)$ by a non-blocking operator, namely $BridgeEdgesOp$. Recall that EDP-QP asks for a maximum of $MaxBreadth$ bridge-edges at any iteration. So, when EDP-QP calls $BridgeEdgesOp$ to compute $Pr(v)$'s bridge-edges, a separate thread is initiated. This thread runs Dijkstra's algorithm to search for the shortest paths from $Pr(v)$ to all the bridge-vertexes in Partition Pr. Dijkstra's algorithm finds the bridge-edges in increasing order of their cost. So, once a bridge-edge is computed, $BridgeEdgesOp$ pipelines it to EDP-QP that can then resume its traversal. The thread running $BridgeEdgesOp$ computes all bridge-edges even if all the requests of EDP-QP are satisfied. The computed bridge-edges are stored in EDP's index. This avoids any recomputations of the bridge-edges in the future. EDP-QP runs a separate thread for the first request to compute a bridge-edges set from scratch. To address synchronization issues among concurrently executing queries that attempt to simultaneously build the bridge-edges of the same vertex, EDP-QP follows a many-consumer-one-producer synchronization model. In other words, each bridge-edges set, say Set_i, can be requested by more than one query (consumers). However, only one thread (producer) is responsible for computing Set_i.

6.3 Time Analysis of EDP Traversal

In this section, we analyze the time complexity of the traversal algorithm EDP-QP, and we present some strategies that make EDP-QP fast (e.g., parallel execution). For an underlying graph, say G, recall that EDP-QP traverses Index $I(G)$ and not G. Let n' and m' be the number of vertexes and edges of G, and let n and m be the number of vertexes and edges of $I(G)$, respectively. We analyze Algorithm 2, and then discuss the implications of the modification presented in Section 6.2 on the time analysis.

EDP uses a Fibonacci heap implementation for its priority queue, and hence insertion into the priority queue, and decreasing the key of an entry are $O(1)$ operations, however, the ExtractMin operation is $O(\log n)$, where n is the number of vertexes in the priority queue [13]. In Algorithm 2, the number of iterations of the While loop in Lines 5-21 is bounded by $O(n)$, and the most costly operation (i.e., ExtractMin) is $O(\log n)$. As each bridge edge is examined at most once (Lines 11-20), Algorithm 2 runs in $O(m + n \log n)$. The number of vertexes of $I(G)$, n, can be expressed as $n = n' + \sum_{v \in G.V}(l(v) - 1)$, where n' is the number of vertexes in G, and $l(v)$ is the number of distinct labels of the outgoing edges of Vertex v. Similarly, the number of edges of $I(G)$, m, can be expressed as $m = \sum_{p \in I.P} \sum_{c \in p.C} |c.V| \times |c.BV|$, where $I.P$ represents the partitions of Index I, $p.C$ represents the disconnected components of Partition p, $|c.V|$ represents the number of vertexes in Component c, and $|c.BV|$ represents the number of bridge vertexes in c. This formula is a theoretical upper bound. In practice, not all the vertexes of a component are connected to all the bridge vertexes of that component. Also, notice that the bridge edges are computed on demand based on the query workload.

Observe that the bridge edges in $I(G)$ are path contractions of the edges in G. This means that $I(G)$ has a smaller diameter than G, and hence traversing $I(G)$ is faster than traversing G. Also, in order not to increase the average fan-out of $I(G)$ observed by the traversal algorithm, Algorithm 2 is modified as discussed in Section 6.2. The modification described in Section 6.2 may add the current vertex to the priority queue again in order not to explore all its bridge edges all at once. This may lead to $k = \sum_{v \in I.V} \frac{|v.BE|}{MaxBreadth}$ priority-queue insertions, where $|v.BE|$ is the number of bridge edges emerging from Vertex v. Thus, the worst-case time-complexity of the modified algorithm is $O(m + k \log n)$.

In practice, not all the bridge edges of a node are examined. The main reason is that bridge edges have high-variability in their costs (e.g., in a road network, the bridge edges from a restaurant to all the highways will have costs with high variance). Hence, in practice, it is unlikely to scan all the bridge edges of a graph before finding the shortest path. In addition, the number of bridge edges m is query-workload dependent. This is because the bridge edges are computed only on demand according to the received queries. Also, recall that the modified algorithm allows parallel computation of the bridge edges if they are not cached (see Section 6.2).

In addition to the parallel computation of bridge edges, Algorithm 2 early computes (in parallel) some shortest paths that may be required by the main traversal thread. Recall that Line 10 checks the cost between the current node and the destination node if the destination is hosted by the same component hosting the current node. EDP uses parallel execution and caching so that Line 10 can run in constant amortized time. When a new vertex, say v, is added to the priority queue, a thread from an active thread-pool computes the distance between v and the destination node. This computation is cached to be ready if Vertex v becomes the current node. In many cases, this distance is marked as infinity in $O(1)$ when the destination node is not hosted by the component hosting Vertex v.

7. GRAPH UPDATES

The ubiquity of dynamic graphs calls for indexing techniques that can handle graph updates efficiently. For example, the roads of a road network have dynamic travel-costs that vary with time (e.g., long travel-times during rush hours). Also, roads are sometimes closed for maintenance/accidents. Similarly, other graph types like social networks are dynamic in nature (e.g., new relationships are added with time). To the best of our knowledge, EDP is the first work to answer ECSP queries on dynamic graphs.

Graph updates can be categorized into two categories: 1) topological updates (e.g., adding or removing an edge), and 2) non-topological updates (e.g., updating an edge's weight). EDP supports all kind of updates on the underlying graph, i.e., topological and non-topological updates.

7.1 Timestamp Approach

A straightforward approach for EDP to handle updates is to invalidate the pre-computations that are only affected by the updates. However, determining exactly the pre-computations to invalidate for each update is time consuming and does not scale for online applications. Hence, EDP adopts a timestamp-based approach to decide whether an indexed monochrome sub-path needs to be recomputed or not.

After the graph partitioning phase (see Section 5.1), EDP finds the disconnected components of each partition. The main idea is to host any index entry by a single component. As a result, EDP can invalidate a component instead of invalidating its index entries individually. The process of finding the disconnected components in EDP places any two nodes that are connected by any path in the same component. EDP uses a variation of Tarjan's algorithm [24] to find the disconnected components in linear time. For example, the partitioned graph in Figure 3 is processed to find the disconnected components of each partition as shown in Figure 6. Each disconnected component is given an identifier that is unique with respect to the hosting partition. For instance, in Figure 6, Partition Pr_R has two disconnected components: C_1 and C_2.

Each disconnected component, say C_i, holds Timestamp $TS(C_i)$. Also, each indexed monochrome sub-path in Component C_i, say P_{ij}, has Timestamp $TS(P_{ij})$. The timestamp values in EDP are assigned values from a global clock. The global clock is advanced by each update operation. Each update results in advancing the timestamp of the affected components. Also, each indexed monochrome sub-path, say P_{ij}, is associated with a timestamp that stores the value of the global clock at the time of computing P_{ij}. Whenever the traversal algorithm asks for Sub-path P_{ij}, the timestamps of the Sub-path and its hosting component are compared to decide if P_{ij} needs to be recomputed, i.e., if $TS(P_{ij})$ is less than $TS(C_i)$, Sub-path P_{ij} is recomputed and is assigned a new higher timestamp.

Notice that adding a new edge may lead to the merge of two components. Also, deleting an existing edge may cause a component to split into two disconnected components. EDP does not perform the merging/splitting operations instantly to support online updates and querying. Delaying the merging/splitting operations in these cases clearly do not affect the correctness of EDP. However, there is a thread pool that run periodically in the background to perform the merge/split operations if necessary.

For each update type, the global clock is advanced, and EDP maintains the timestamp of the affected components as follows:

Update edge weight: On updating the weight of an edge, its hosting component, say C_i, is determined and its timestamp (i.e., $TS(C_i)$) is set to the value of the global clock.

Update edge label: On updating an edge's label, its original hosting component, say C_o, and its new hosting component in the new partition, say C_n, are determined. The edge is removed from Component C_o and is added to Component C_n (new vertexes may be added to the new hosting component). The timestamps $TS(C_o)$ and $TS(C_n)$ are updated.

Add edge: On adding a new edge, say e, the label of Edge e determines the hosting partition. The endpoint vertexes of Edge e can be either hosted by one or two components. After adding Edge e, the timestamp of the affected component(s) is updated. If the endpoint vertexes of Edge e is hosted by different components, EDP adds an entry to a system hash-table that these components are connected.

Delete edge: After deleting the edge, the timestamp of its hosting component is updated.

Add vertex: Nothing takes place until an edge connected to this vertex is added.

Delete vertex: On deleting a vertex, say v, all its instances in all the partitions are determined. For each instance of v, all the incoming/outgoing edges of that instance are deleted and the timestamp of the affected components are updated.

EDP performs any graph-update operation in $O(1)$ time except when deleting a vertex, which takes linear time in the number of partitions hosting that vertex's instances, and the number of their in/out edges. Recall that an indexed monochrome sub-path, say P_{ij}, hosted by a component, say C_i, is associated with a timestamp. On demand, P_{ij} is recomputed only if $TS(P_{ij})$ is less than $TS(C_i)$.

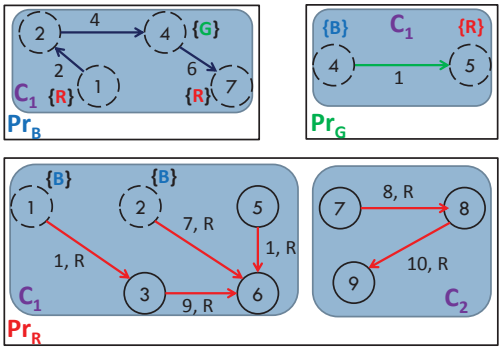


Figure 6: Disconnected Components of the graph in Figure 1.

8. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of EDP against that of CHLR [18] that is the state-of-the-art technique. Mainly, we measure the average speedup in query-time that EDP achieves in comparison to CHLR. Our experiments are conducted on a machine running Linux kernel 3.17.7 on 32 cores of Intel Xeon 2.90 GHz and 192 GB of main-memory. Our implementation is based on Java 1.8. For fair comparison with CHLR, we implement all the optimizations mentioned in [18] and [14], e.g., the node ordering for the contraction algorithm.

8.1 Datasets

We use six real graph datasets: Tiger [3], BioGrid [4], BioMine [12], String [2], DBLP [1], and Youtube [23]. Table 2 summarizes the properties of the datasets. Tiger is a road network dataset covering the entire U.S., where the labels describe the types of roads (e.g., primary road, ramp, alley). The BioGrid and String datasets are protein-interaction networks, where the vertexes represent proteins, the edges represent interactions among the proteins,

and the labels represent the interaction types. The BioMine dataset is a network that captures a set of relationships among biological entities. DBLP and Youtube datasets are subsets of the popular co-authorship network, and the popular video sharing service, respectively. In DBLP, the vertexes represent authors, the edges represent co-authorship, and the labels represent publication topics as described in [8]. For the Youtube dataset, the vertexes represent users, the edges represent user relationships, and the edges are labeled by relation types as described in [23].

8.2 Preprocessing Time

The preprocessing overhead performed by EDP is much less than that of CHLR (see Table 2). EDP builds $I(G)$ by scanning the edges of G in one pass. In contrast, during preprocessing, CHLR performs many shortest path computations, i.e., it is workload-independent. On the other hand, EDP computes shortcut edges on-demand according to the query workload received. The preprocessing time of EDP is affected by the size and the density of the graph as well as the number of bridge vertexes to be created (see Table 2). For example, the time for preporocessing BioGrid is greater than the time for preprocessing Tiger because the average fan-out of BioGrid is greater than that of Tiger by an order-of-magnitude. Also, the maximum fan-out for Tiger is 7, while BioGrid has a maximum fan-out of 36987.

For static graphs, EDP computes any shortcut edge, say e, only once. afterwards, other queries can leverage e. However, the first query that asks for e pays the cost of its computation. In order to have a tangible measure of the cost to answer shortest path queries by EDP using a fresh index (i.e., one that has no precomputations), in our experiments, we run EDP on the query workload twice. We refer to the first run as cold-run, where we measure the performance of EDP without having any precomputations in its cached index. The second run, called warm-run, assumes that all the required shortcuts are already computed in the first run.

8.3 Effect of the Number of Query Labels

We study the effect of the number of labels specified in the query on the speedup of EDP compared to CHLR. For each dataset, we generate random queries with $1 \cdots |L|$ labels. For each label size and using $1000 * |L|$ random source nodes, we generate more than one million random queries. From these queries, we select 1000 queries randomly for each label Size s.

Figure 7 gives the speedup of EDP when changing the query label set size for each dataset. Figure 7 also gives the 95%-confidence interval bars. Notice that the speedup of EDP downgrades as the query label set size increases. In this case, EDP explores more partitions and traverses a larger number of bridge edges.

EDP achieves the highest speedup when the query has only one label. For example, the Tiger dataset shown in Figure 7(a) reaches a warm-run speedup of 2068 ± 211 with a 95% confidence-interval. The speedup downgrades as more labels are considered. The lowest warm-run speedup is 61 ± 14 when all the labels are considered. For the cold-run, when no precomputations exist in EDP's index, the best speedup is 152 ± 19 and the lowest speedup is 10.4 ± 0.78. For the Tiger dataset, EDP provides answers in a sub-millisecond on average during the warm-runs.

Graph Updates: Figure 7 also gives query-performance for dynamic graphs. After performing the cold and warm runs, we run a set of random updates on the graph, and then measure the speedup using the same queries. For each edge in the graph, an update is selected based on the following probability distribution: 40% no-update, 40% weight-update, 10% label-update, 5% edge deletion, and 5% random edge addition. These probabilities are based on the occurrence-likelihood of the update types in real applications. From Figure 7, EDP achieves significant speedups on dynamic graphs without any downtime for index reconstruction in contrast to CHLR which requires rebuilding its index from scratch (and may lead to downtimes of several hours). In this experiment, EDP performs an update operation in 3.43 ± 1.2 microseconds on average with a 95% confidence interval for the Tiger dataset. The other datasets experience similar behavior as EDP handles any graph-update operation in $O(1)$ time. Figure 7 illustrates the effectiveness of the EDP's timestamp invalidation approach, and demonstrates that EDP can handle dynamic graphs efficiently. Note that the after-update speedup curve is constantly better than the cold-run curve that outperforms CHLR.

In contrast to CHLR, EDP is not tailored to road networks. For instance, in the case of the BioGrid dataset in Figure7(b), EDP achieves up to four orders-of-magnitude speedup over CHLR for static graphs, and up to three orders-of-magnitude speedup for dynamic graphs. EDP shows significant speedups of 14305 ± 2718 in the best case and 11.7 ± 6.4 in the worst case compared to CHLR with 95% confidence intervals.

Observe that CHLR and the contraction hierarchies are originally designed for road-network graphs, and hence they have assumptions that are not valid for other networks. For instance, it is assumed that the graph vertexes can be placed in hierarchies based on their importance. Also, it is assumed that the average fan-out of the vertexes is small (e.g., 2). However, these assumptions are invalid for graphs of other domains (e.g., a protein-interaction network). For example, the Tiger dataset has an average fan-out of 1.2, but the BioGrid dataset has an average fan-out of 14. Moreover, the maximum fan-out for the Tiger dataset is 7 in contrast to a maximum fan-out of 36987 for the BioGrid dataset.

One of the reasons of the superiority of EDP over CHLR is that CHLR's shortcuts are not sufficient to answer ECSP queries and they have to be combined with edges from the original graph. In contrast, the shortcuts of EDP are sufficient to discover any ECSP query shortest path. Moreover, EDP prunes the irrelevant portions of the search space at low-cost. The reason is that the graph is partitioned, and all the partitions corresponding to disallowed labels are discarded from the search space.

8.4 Index Size

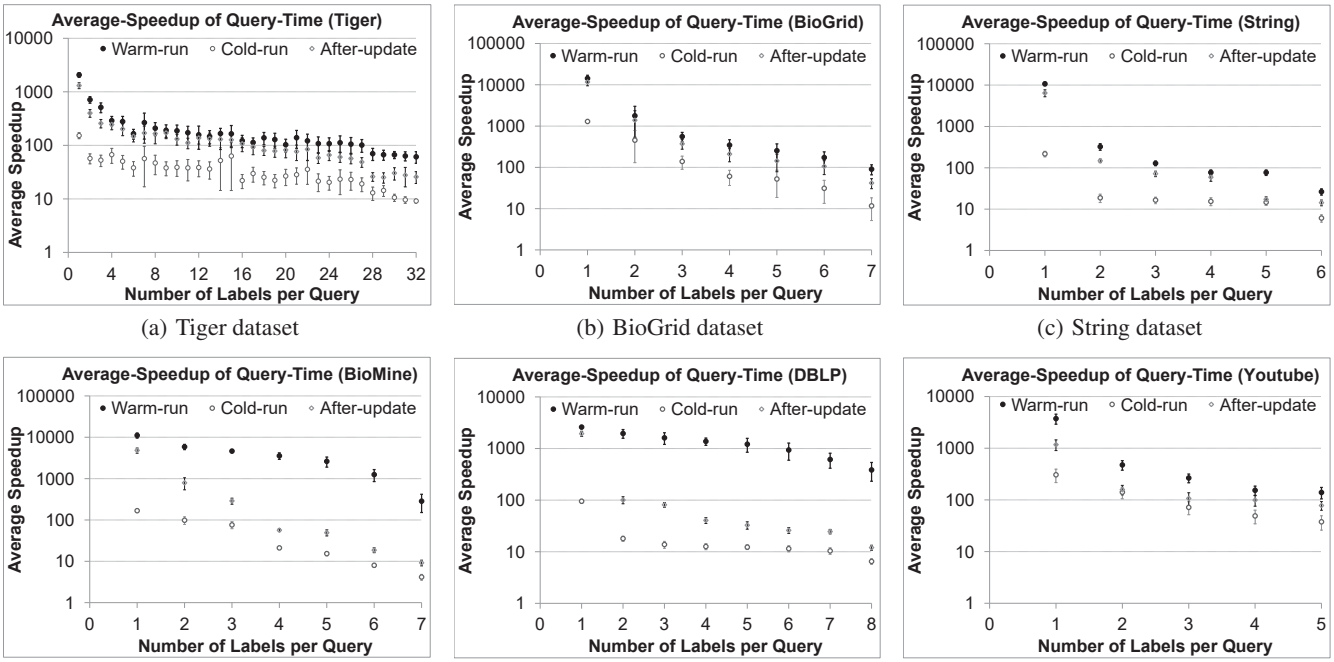
In this set of experiments, we measure the size of EDP's index as queries are processed. We also measure the effect of varying the maximum index size on the query performance.

To measure the index size of EDP, we process millions of randomly generated queries and measure the index size. Because the total number of different queries is exponential (i.e., $2^{|L|} \times n^2$), we use a workload of 10 millions random queries. The number of labels per query is fixed to half of the number of the labels in each dataset. Figure 8 shows the size of EDP's index grows linearly with the number of processed queries. The reason is that the precomputations added to EDP's index after processing ECSP queries are reused by newly arriving queries. The figure also demonstrates that EDP can process millions of random queries using few gigabytes of main memory, which can be easily provided by commodity servers. We observe that the main factor of the index size corresponding to the Tiger dataset is the indexed monochrome sub-paths with long lengths, while for the other graphs (e.g., BioMine), the main factor of the index size was the computed bridge edges in response to the processed queries.

Figure 9 (cold-run curves) shows the effect of limiting the index size (i.e., cache) on the query-time speedup using the same

Dataset	# Vertexes	# Edges	# Labels	Avg. Fan-out	% Bridge Vertexes	EDP Preprocessing	CHLR Preprocessing
Tiger	24412259	58698439	32	1.2	4%	46 seconds	6.9 hours
BioGrid	56395	1578358	7	14	34%	3.6 minutes	5.1 hours
String	35423	444331	6	13	63%	14 seconds	70 minutes
BioMine	47175	286372	7	3	56%	3 seconds	16 minutes
DBLP	47598	252881	8	6	66%	6 seconds	15 minutes
Youtube	15088	13628895	5	909	73%	9 minutes	10.7 hours

Table 2: The datasets used and the preprocessing time of EDP and CHLR.



(a) Tiger dataset (b) BioGrid dataset (c) String dataset

(d) BioMine dataset (e) DBLP dataset (f) Youtube dataset

Figure 7: Query-Time Speedup of EDP vs. CHRL with 95% confidence intervals w.r.t. the number of labels per query.

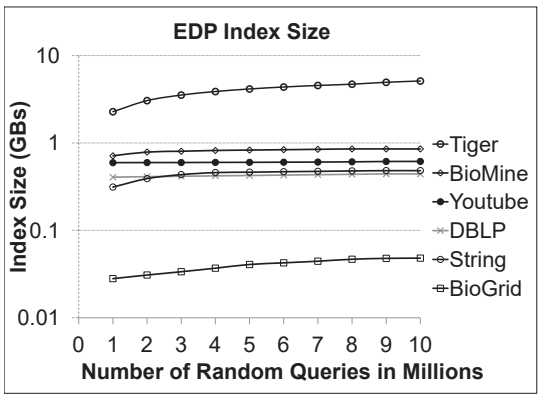


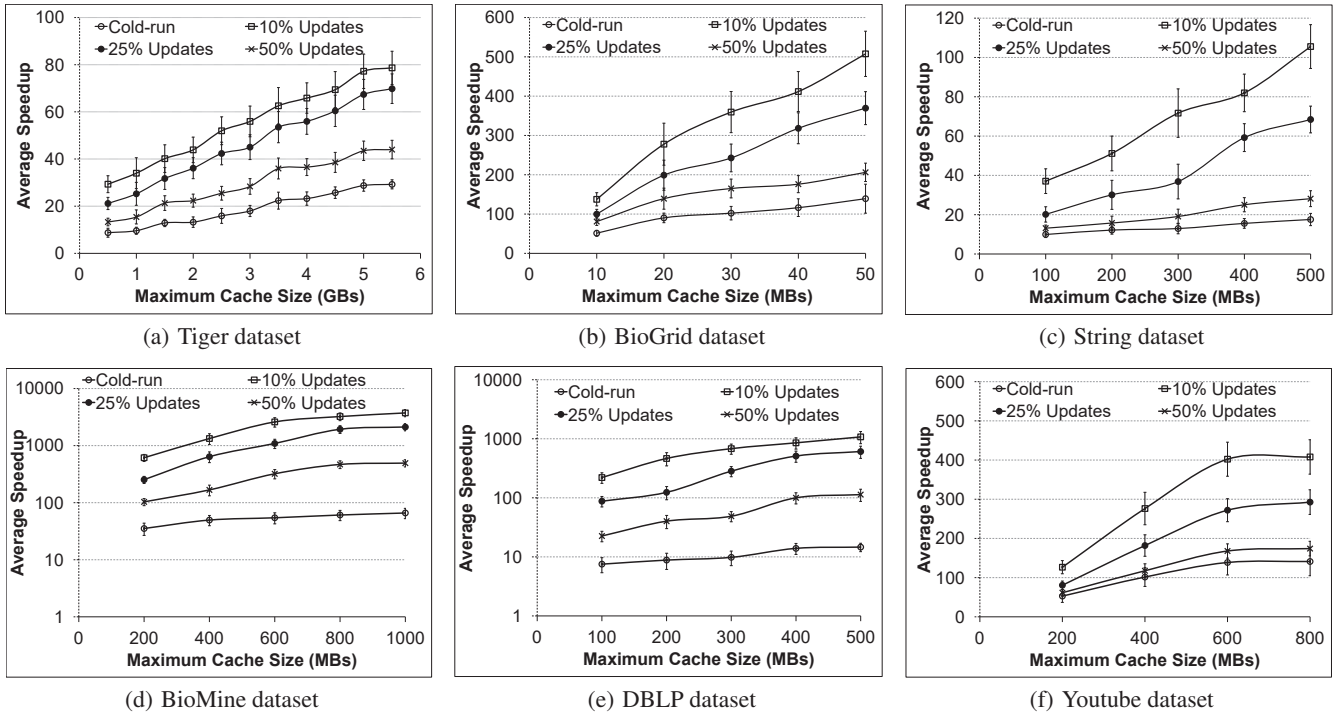
Figure 8: Index Size of EDP w.r.t number of processed queries.

query workload described above. In this experiment, we measure the average query-time speedup of processing 10 million randomly generated ECSP queries using both EDP and CHLR. For EDP, we run the queries for different maximum index-size values (according to Figure 8). EDP has significant speedup for limited cache sizes. For instance, in Figure 9(a) for the Tiger dataset, EDP achieves a speedup of 8.79 ± 1.7 with 95% confidence interval when using only 500 MB of memory (i.e., 9% of the total memory required after processing the entire workload), and an order-of-magnitude speedup is achieved when limiting the index-size to just one GB.

8.5 Interleaving Updates

In this experiment, we monitor the performance of EDP when graph updates are interleaved with ECSP queries. Recall that the index entries of EDP are invalidated based on the level of the disconnected components of a partition (see Section 7.1). So, after processing 10 million random queries, and after ensuring that the cache is full, we select a workload of 1000 ECSP queries, say QS. We run a graph-update workload associated with a percentage value, say p, to ensure that p% of the components serving QS are updated, and hence, these components will be invalidated. Determining the components serving an ECSP query is straightforward as the monochrome sub-paths forming the query answer are associated with their hosting components identifiers. After running the graph-update workload, we measure the average query-time speedup when executing QS. We interleave this process three time for three different values of p, specifically 10%, 25%, and 50%. For instance, in Figure 9, the curve corresponding to 10% updates means that 10% of the components serving QS are updated.

As Figure 9 shows, EDP has significant speedup for different



(a) Tiger dataset (b) BioGrid dataset (c) String dataset

(d) BioMine dataset (e) DBLP dataset (f) Youtube dataset

Figure 9: Query-Time Speedup with interleaved updates w.r.t maximum cache size.

graph-updates frequencies under limited cache sizes. Observe that, the 10% update curves show the highest speedups because the queries of these curves have high probabilities of being partially served from the cache (i.e., similar to a warm-run). The speedup decreases as the graph-updates frequency increases, however, at the worst case, the queries will experience a performance similar to that of a cold-run. Recall that a cold-run of EDP still outperforms CHLR, and that a cache entry is replaced in $O(1)$ time.

Discussion

EDP outperforms CHLR due to EDP's natural partitioning based on edge labels. A query processed by EDP visits only the partitions of interest based on the query labels. Another reason for EDP's good performance is that it only uses shortcuts to discover a shortest path. EDP uses the original graph edges to create a shortcut for the first time only (cold-run). Hence, the added shortcuts of EDP would not much decrease the sparsity of the underlying graph. Also, by using the $MaxBreadth$ parameter as described in Section 6.2, EDP does not necessarily investigate the whole shortcuts of a vertex visited during the traversal. While limiting the index-size of EDP, EDP still achieves orders-of-magnitude speedup as it efficiently frees space for new index-entries, besides the aforementioned reasons of why EDP performs well on cold-runs. EDP executes online graph updates efficiently in $O(1)$ time by flagging the affected components as invalidated. An index entry in an invalidated component will be computed on demand based on the query-workload. Hence, with no downtime, EDP presents query-time speedups that lie between warm-run and cold-run speedups for dynamic graphs.

9. CONCLUSION

EDP is a technique for answering edge-constrained shortest path queries (ECSP). It assumes a dynamic graph where each edge has one label. EDP also works for multi-graphs with edges having multiple labels. EDP has two components: an index and a traversal algorithm. We exploit the notion of contracted monochrome sub-paths of any ECSP shortest path to design the index for EDP. We illustrate how to decrease recomputations of monochrome sub-paths by caching them once computed. Thus, the costs of on-demand computation of these sub-paths are amortized over all future queries that use the already constructed shortcuts. Moreover, we demonstrate that the cache size needed is small. The monochrome sub-paths are only re-computed on demand if they are potentially affected by some graph updates. We present EDP's traversal algorithm and provide its proof of correctness. The experimental study over real six graphs spanning different domains show up to four orders-of-magnitude speedup compared to the state-of-the-art with 95% confidence intervals.

10. ACKNOWLEDGMENTS

We would like to thank Engy M. Abdallah for her help in editing this paper. This research was supported in part by National Science Foundation under Grant IIS 1117766.

11. REFERENCES

[1] http://dblp.uni-trier.de/xml/.
[2] http://string-db.org/.
[3] https://www.census.gov/geo/maps-data/data/tiger.html.
[4] http://thebiogrid.org.
[5] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 24–35, 2012.
[6] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, and M. V. Marathe. Classical and contemporary shortest path problems

in road networks: Implementation and experimental analysis of the transims router. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, 2002.

[7] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*, 2007.

[8] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.*, pages 547–558, 2014.

[9] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 52–65, 2007.

[10] J. Dibbelt, T. Pajor, and D. Wagner. User-constrained multimodal route planning. *J. Exp. Algorithmics*, 19, Apr. 2015.

[11] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[12] L. Eronen and H. Toivonen. Biomine: Predicting links between biological entities using network models of heterogeneous database. In *BMC Bioinformatics*, pages 13–119, 2012.

[13] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[14] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras. Route planning with flexible edge restrictions. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

[15] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 319–333, 2008.

[16] A. V. Goldberg, H. Kaplan, and R. F. F. Werneck. Better landmarks within reach. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 38–51, 2007.

[17] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6), Dec. 1995.

[18] M. N. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *PVLDB*, 4(2):69–80, 2010.

[19] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 43–54, 2008.

[20] A. Shimbel. Applications of matrix algebra to communication nets. *Bulletin of Mathematical Biophysics*, 13:165–78, 1951.

[21] A. Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biophysics*, 15:501–507, 1953.

[22] S. N. Simões, D. C. Martins-Jr, H. Brentani, and R. Fumio. Shortest paths ranking methodology to identify alterations in ppi networks of complex diseases. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12, pages 561–563, 2012.

[23] L. Tang, X. Wang, and H. Liu. Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.*, 25(1):1–33, July 2012.

[24] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

APPENDIX

A. CORRECTNESS OF EDP

THEOREM 1. The traversal Algorithm EDP-QP finds a correct shortest path if one exists.

$Proof$. The proof for this theorem follows very closely the proof of correctness of Dijkstra's algorithm. Recall that $I(G)$ preserves the connectivity of G by construction (refer to Section 5.1). Hence, a shortest path that exists in G also exists in $I(G)$. To prove the correctness of EDP-QP, we need only to prove that when a vertex, say $pr_m(u)$, is dequeued from PQ, the cost of $pr_m(u)$ in the $cost$ table is the cost of the shortest path from the source vertex to Vertex $pr_m(u)$. This can be proven by contradiction.

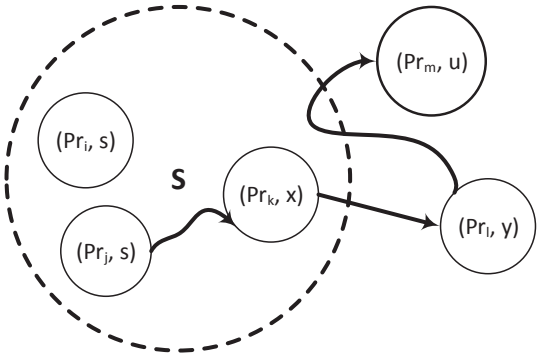
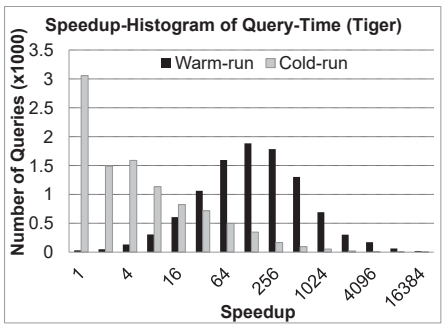


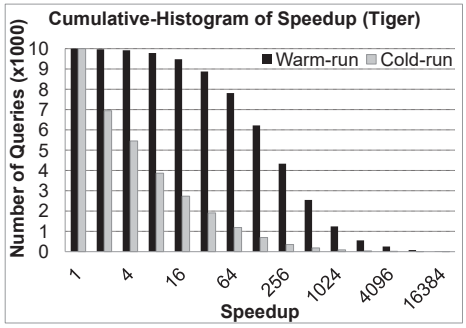
Figure 10: Proof of correctness for EDP-QP. The big dashed circle encloses the set S of dequeued elements. A vertex is represented by a pair (pr, v) indicating Vertex v in Partition pr.

Let S be the set of all $I(G)$ vertexes that are extracted from PQ. We claim that the following invariant holds till the termination of Algorithm 2: $\forall$ element $e \in S$, we have $cost(e) = \delta(s,\ e.v)$, where s is the source vertex and $\delta(s, e.v)$ is the cost of the shortest path from s to $e.v$. Notice that when a bridge vertex is visited, all of its copies in other allowed partitions are added to PQ with the same cost of the visited bridge-vertex. For the sake of contradiction, assume that $e = (pr_m, u)$ in Figure 10 is the first element added to S, where cost(e) $\neq \delta(pr_j(s),\ pr_m(u))$. $pr_m(u)$ cannot represent $pr_j(s)$ because $cost(pr_j, s) = 0$. Hence, there must be a path from (pr_j, s) to (pr_m, u). Otherwise, $cost(pr_m, u)$ would be infinity and it would not appear in S. If a path exists from (pr_j, s) to (pr_m, u), then a shortest one is there.

At the time, say t, when (pr_k, x) gets added to S, $cost(pr_k, x)$ equals $\delta(pr_j(s),\ pr_k(x))$ (recall our hypothesis that $pr_m(u)$ is the first element that violates the claimed invariant). Also, $cost(pr_l(y)) = \delta(pr_j(s),\ pr_l(y))$ (because edge $(pr_k(x),\ pr_l(y))$ has been relaxed at Time t). Then, $cost(pr_l(y)) = \delta(pr_j(s),\ pr_l(y)) \leq \delta(pr_j(s),\ pr_m(u)) \leq cost(pr_m(u))$. However, as both vertexes $pr_m(u)$ and $pr_l(y)$ were not in S when $pr_m(u)$ was chosen, it follows that $cost(pr_m(u)) \leq cost(pr_l(y))$. Then, the previous inequality relation becomes equality as follows: $cost(pr_l(y)) =$



(a) Speedup Histogram of EDP vs. CHLR



(b) Cumulative Speedup of EDP vs. CHLR

Figure 11: Speedup of EDP vs. CHLR for the Tiger dataset

$\delta(pr_j(s),\ pr_l(y)) = \delta(pr_j(s),\ pr_m(u)) = cost(pr_m(u))$. Consequently, $cost(pr_m(u)) = \delta(pr_j(s),\ pr_m(u))$, which contradicts our choice of $pr_m(u)$. Therefor, this proves that $cost(pr_m(u))$ = $\delta(pr_j(s),\ pr_m(u))$ when $pr_m(u)$ is added to S, and this equality is maintained afterwards. Hence, when Line 10 of Algorithm 2 extracts a PQ entry that corresponds to a destination vertex instance, then its associated cost is the cost of the shortest path cost.

B. SHORTEST PATH CONSTRUCTION

In this section, we illustrate how EDP-QP builds the actual shortest path. Each entry in the $cost$ lookup table used by EDP-QP keeps an attribute called $parent$. Whenever EDP-QP relaxes the cost of an edge, say $(pr_i(u), pr_j(v))$, it sets the parent attribute of $cost(pr_j(v))$ to $pr_i(u)$'s $cost$ entry. When the cost of the shortest path is found, the chain of entries ending at $cost(pr_k(d)).parent$ is constructed by a backward traversal (d is the final destination vertex). The constructed chain indicates the vertex/partition pairs that construct the shortest path in order. This chain gives the contracted monochrome sub-paths $CP(P)$ and not P. Using $I(G)$, the details of each monochrome sub-path in $CP(P)$ are to form the final shortest path P. The complexity of this construction algorithm is $O(|CP(P)|)$, where $|CP(P)|$ is the number of monochrome sub-paths in P.

C. MULTI-GRAPHS AND MULTI-LABELS SUPPORT

A multi-graph is a graph where any two vertexes can have more than one edge connecting them. EDP can safely handle multi-graphs. EDP has an edge disjoint partitioning. Given any two vertexes u, v with multi-edges, each of these edges, say e, is hosted by a separate partition based on e's label. EDP treats each edge independently and the semantics of an ECSP query does not change. However, if a certain edge has more than one label, an ECSP query can have two different but valid semantics.

Let $Q(s,\ d,\ A)$ be an ECSP query. Given a labeled graph G' where an edge $e \in G'.E$ can have multi-labels (i.e., $l(e) \subseteq G'.L$), Q can have two different possible semantics:

- **Semantic 1:** An edge in $|G'|$ is allowed if its labels contain at least one query label (i.e., $\forall$ e $\in sp(s,\ d,\ A) \mid l(e) \cap Q.A \neq \Phi$).
- **Semantic 2:** An edge in $|G'|$ is allowed if all its labels are in the query labels (i.e., $\forall$ e $\in sp(s,\ d,\ A) \mid l(e) \subseteq Q.A$).

ECSP supports $Semantic$ 1 by viewing a multi-labeled edge as a set of independent edges. If interested in $Semantic$ 2, the user may transform the graph to have single label edges, if possible.

D. EXCLUSION LOGIC AND QUERY REWRITING

Given a labeled graph where each edge has only one label, an ECSP Query $Q(s,\ d,\ A)$ can take one of the following two forms:

- **Inclusion logic form:** Labels in A are allowed in the shortest path.
- **Exclusion logic form:** Labels in A are not allowed in the shortest path, and hence the labels in $overline A$ are allowed.

The inclusion logic form is suitable for social network graphs, e.g., a query will most likely include only a small subset of the edge labels, e.g., two or three social relation types. In contrast, the exclusion logic form is suitable for road network graphs, e.g., avoiding toll roads. The two forms are equivalent for graphs with single-label edges, and can be transformed to each other easily. EDP supports both forms and can rewrite a query accordingly, e.g., rewrite $Q_i(s,\ d,\ A)$ into $Q_e(s,\ d,\ \overline{A})$ to support the exclusion semantics based on a simple yet effective cost model.

D.1 Query Rewriting

EDP-QP can process an ECSP query in either of the inclusion and exclusion forms. It can choose the form that makes the processing faster. For example, since the $OtherHosts$ list of the index $I(G)$ is fixed w.r.t. any query, being a frequent operation, the intersection operation between $OtherHosts(Pr(v)) \cap Q.A$ for every visited bridge-edge $Pr(v)$ will depend on the size of $Q.A$. Clearly, some queries are better written in a particular form over the other depending of the size of A vs. $L - A$. Hence, we can rewrite the query to be in either the inclusion or the exclusion-forms, and Algorithm 2 adapts its edge selection logic, accordingly (e.g., the comparison logic of Line 18 in Algorithm 2).

When executing an ECSP Query, say Q, a query-rewrite step takes place, and an attribute is set for Q, say $Q.Type$, to indicate its logic-form. If $Q.Type$ is inclusion and $|Q.A| > |L$ - $Q.A|$, $Q.Type$ is changed to exclusion and $Q.A$ is reset to L - $Q.A$. Similarly, if $Q.Type$ is exclusion and $|Q.A| > |L$ - $Q.A|$, $Q.Type$ is changed to inclusion and $Q.A$ is rest to L - $Q.A$. When EDP-QP processes Q, it checks $Q.Type$ to perform the right label filtering.

E. EDP FOR NON-DIRECTED GRAPHS

As every non-directed graph can be represented by an equivalent directed graph, EDP easily supports non-directed graphs. EDP takes a parameter to indicate if the input graph is directed or non-directed. If the input graph is non-directed, then an edge, say e, is stored only once. However, at each of the endpoint vertexes of

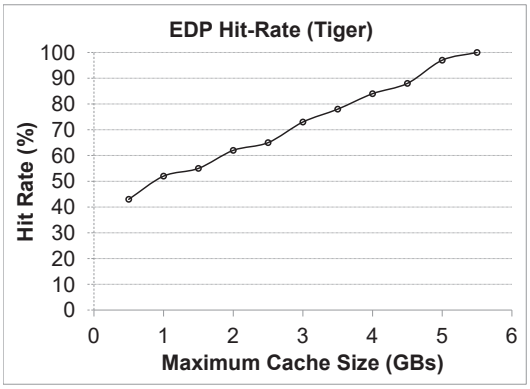


Figure 12: Hit Rate of EDP w.r.t. Maximum Cache Size.

Edge e, EDP stores the identifier of Edge e in both the "InEdges" and "OutEdges" lists of its endpoints. Hence, in a non-directed graph, each vertex is aware of all of its edges.

F. ADDITIONAL EXPERIMENTS

F.1 Fixing Query-Label Size

In this experiment, we fix the query label size, and study the performance speedup of individual queries. We generate more than 50K random queries such that each query has $|L|/2$ allowed labels. The reason for setting the number of labels per query to $|L|/2$ is to cancel the effect of the query labels size on both EDP and CHLR. Recall that for a given ECSP query with allowed labels A, we compute a restricted set $R = L - A$ as the input to the CHLR technique. So we choose A so that $|A| = |L| / 2$ to give CHLR a query with restricted labels R where $|R|$ is of size $|L| / 2$. This cancels the effect of query labels-set size when comparing EDP to CHLR.

From the 50K queries generated, we execute $10,000$ random queries, measure the running time of EDP and CHRL per query, and compute the speedup. Figure 11(a) gives a histogram of the results. The x-axis indicates buckets of speedup ranges and the y-axis indicates the number of queries out of the 10,000 queries that fall in each bucket of the x-axis.

As shown in Figure 11(a), EDP and CHLR perform similarly in the warm-run for only 31 queries out of the 10k queries (i.e., there is no speedup for only 0.3% of the random queries). However, for the remaining 99.7% random queries, EDP has a speedup of more than two over CHLR. For the cold-run, where the precomputation time is added to each query, there is no speedup for 30% of the random queries while 70% of the queries experience a speedup of at least two. Figure 11(b) gives the cumulative distribution of the speedups. About 95% of the queries have at least a speedup of one order-of-magnitude for the warm-run while, for the cold-run, near 38% of the queries experience at least an order-of-magnitude speedup.

F.2 Cached-Index Hit-Rate

In this experiment, we measure the hit rate of EDP's cached-index w.r.t. different index-size limits. This experiment uses the 10 millions randomly-generated ECSP queries used before to measure the index size. Although the workload is random and EDP uses an LRU replacement-policy, the hit rate is still a good measure to represent the likelihood of finding an index entry in the cache when needed by new ECSP queries.

Starting from a maximum index-size of 500 MBs to 5.5 GBs, we measure the hit rate of EDP's cached index after processing the entire workload. As Figure 12 shows, the hit rate increases linearly when increasing the maximum index-size. EDP achieves a hit rate of more than 50% when using only 1 GB cache, and a hit rate of 84% when using 4 GBs cache. Figure 12 shows that EDP has a high potential of not missing requested index-entries even when limiting the index size by small values (e.g., 500 MBs).