What Links Alice and Bob?
Matching and Ranking Semantic Patterns in Heterogeneous Information Networks

Jiongqian Liang†, Deepak Ajwani‡, Patrick K. Nicholson§, Alessandra Sala∥ and Srinivasan Parthasarathy¶
†Computer Science and Engg., The Ohio State University, Columbus, OH, USA
‡Bell Labs Ireland, Dublin, Ireland
§{liangji,srini}@cse.ohio-state.edu
∥{deepak.ajwani, pat.nicholson, alessandra.sala}@alcatel-lucent.com

ABSTRACT
An increasing number of applications are modeled and analyzed in network form, where nodes represent entities of interest and edges represent interactions or relationships between entities. Commonly, such relationship analysis tools assume homogeneity in both node type and edge type. Recent research has sought to redress the assumption of homogeneity in both node type and edge type. Guiding on such efforts, in this work we articulate a novel approach for mining relationships across entities in such networks while accounting for user preference over relationship type and interestingness metric. We formalize the problem as a top-k lightest paths problem, contextualized in a real-world communication network, and seek to find the k most interesting path instances matching the preferred relationship type. Our solution, PROphetic HEuristic Algorithm for Path Searching (PRO-HEAPS), leverages well-designed heuristics and the venerable A* search algorithm. We run our algorithm on real-world large-scale graphs and show that our algorithm significantly outperforms a wide variety of baseline approaches with speedups as large as 100X. We also conduct a case study and demonstrate valuable applications of our algorithm.

Keywords
Heterogeneous Information Networks, Semantic Relationship Queries, Graph Algorithms

1. INTRODUCTION

Many learning systems, used in a diverse range of application domains such as semantic search, financial fraud detection, intelligence gathering, root-cause analysis of distributed systems, recommendations, contextualization, personalization of services, biological networks, security, etc., rely on mining Heterogeneous Information Networks (HINs) that have semantic labels on vertices and/or edges. HINs are particularly useful in applications where information from diverse sources must be linked and mined in a holistic way. Mining such networks has also attracted a lot of academic interest in recent years (e.g., [28, 30, 29, 26, 22]).

A fundamental problem in mining heterogeneous information networks is to find interesting (possibly complex and derived) relationships between entities that are modeled as vertices in the heterogeneous graph. Past literature on mining the relationship between entities either builds on homogeneous networks [9, 31] or performs generic mining of heterogeneous networks [10] without taking into consideration the specific type of relationship that an application/user is searching. When used in real applications involving HINs, such techniques often end up in discovering trivial and/or non-interesting relationships. Thus, there is a need for techniques that can discover semantic relationships — queries where the search is focused on a particular type of relationship that is specified using a sub-graph with semantic vertex and edge labels.

In addition to the advantage of returning only the relationship instances that the user actually cares about, we show that specifying semantic query patterns also enables an application developer to prune the search space of possible relationship instances and thereby support queries in near real-time. In cases where even the elimination of irrelevant relationship instances (that do not match the specified semantic pattern) still leaves a plethora of matches, a user can specify a ranking metric to further prioritize the search results. For instance, Figure 1 shows a real-world example of a heterogeneous network modeling the communication between different people. These people use various explicit channels for communication — Emails, SMS, phone calls — which are modeled as vertices in this network. One may further supplement explicit information with implicitly derived information (e.g., conversation topics) using standard NLP tools. An analyst may be interested in indirect communication between two people, such as Person 1 sending an email to an intermediate person who then calls Person 2. Among all such instances, the analyst may be interested in prioritizing the most recent communication exchanges.

In this paper, we present an algorithm for prioritized relationship mining, where the prioritization lies in both the relationship type and the interestingness metric over the relationship. The relationship type is defined in terms of a path pattern (or more generally as a sub-graph pattern) between
entities and the detailed interpretation of this relationship will be inferred from the semantic labels and other attributes on vertices and edges of path instances. The interestingness metric over the relationship is captured by a weight function on the edges of the graph. For instance, the analyst’s query for the indirect communication (as mentioned above) between Person 1 and Person 2 in Figure 1 can be modeled by the path pattern: \((\text{Person 1}) \rightarrow V \rightarrow E \rightarrow V \rightarrow \ldots \rightarrow \text{Person 2}\). The interestingness metric of recency can be captured by a weight function where the weight of the edges (e.g., \(V \rightarrow E\)) is exponential to the difference between the current time and the time of communication. We then formalize the problem as \(k\)-lightest paths problem, targeting the \(k\)-lightest loopless paths between the entities, matching the path pattern (see Section 2.1).

The problem of finding the \((k,\ell)\) lightest loopless path, matching a pre-specified pattern, is NP-hard and furthermore, simple heuristics and straightforward approaches are unable to efficiently solve the problem in real time (see Section 2.3). We propose PROphetic HEuristic Algorithm for Path Searching (PRO-HEAPS) to efficiently solve our problem using effective preprocessing and by employing elaborately selected heuristics. We preprocess the graph based on the query provided to facilitate follow-up searching and generate a prophet graph, which is a new graph designed for efficient search. We devise a consistent heuristic which can be obtained by conducting breadth-first search on the prophet graph. Adapting the A* algorithm with the heuristic, PRO-HEAPS is able to discover prioritized relationships in real-time even when dealing with large-scale graph and reasonably complex relationships.

The main contributions of this work are: 1) The prioritization of relationship mining by specifying the relationship of interest and weighting the edges of the graph. 2) The design of a simple but novel tool called prophet graph for efficient path searching. 3) That we devise a consistent but computationally cheap heuristic to ensure the optimality of A* algorithm. 4) That we demonstrate that PRO-HEAPS can answer relationship queries in real-time while allowing the weights to be dynamic (e.g., depending on recency).

2. PRELIMINARIES

2.1 Problem Formulation

We first provide some definitions that formalize the concept of a Heterogeneous Information Network (HIN), borrowing from previous work [28, 30].

**Definition 1. Weighted Heterogeneous Information Network.** A Weighted Heterogeneous Information Network is a directed graph \(G = (V, E, \Phi, \Psi, W)\), where: \(V\) is a set of vertices; \(E\) is a set of edges; \(\Phi : V \rightarrow \mathcal{L}\) is a vertex labeling function; \(\Psi : E \rightarrow \mathcal{R}\) is an edge labeling function; and \(W : E \rightarrow \mathbb{R}\) is a weight function.

In our problem, vertices represent entities, of which there are \(|\mathcal{L}|\) types in the network, while edges indicate relationships or interactions between entities, of which there are \(|\mathcal{R}|\) types. Weight is defined according to the specific application and interestingness, and is further discussed in Section 2.2. Moreover, for a vertex \(u \in V\), we distinguish outgoing and incoming neighbors by denoting them as \(N_{\text{out}}(u)\) and \(N_{\text{in}}(u)\), respectively.

In this paper, we use paths to explain the relationship between entities, following the idea from Fang et al. [10], where they showed that complex relationships expressed by a subgraph can be decomposed into paths. To convey user preference on relationship, we use the vertex and edge labels along a path to represent the type of relationship, and the weights of edges to interpret importance. Specifically, when mining relationships between entities, we are provided with a path pattern.

**Definition 2. Path Pattern.** Given a weighted HIN \(G = (V, E, \Phi, \Psi, W)\), a path pattern \(\mathcal{P}\), or metapath, is a sequence \(L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} \ldots \xrightarrow{R_{\ell-1}} L_\ell\), where \(L_0, \ldots, L_\ell \in \mathcal{L}\) are vertex labels and \(R_0, \ldots, R_{\ell-1} \in \mathcal{R}\) are edge labels.

Here we define the length of a path pattern \(\mathcal{P}\) to be the number of edges in \(\mathcal{P}\) and denote it as \(|\mathcal{P}| = \ell\). Given a weighted HIN \(G = (V, E, \Phi, \Psi, W)\), if a directed path \(p = v_0 e_0 v_1 e_1 \ldots e_{\ell-1} v_\ell\) in the graph \(G\) and a path pattern \(\mathcal{P} = L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} \ldots \xrightarrow{R_{\ell-1}} L_\ell\) satisfy \(\Phi(v_i) = L_i, \forall i = 0, \ldots, \ell\) and \(\Psi(e_i) = R_i, \forall i = 0, \ldots, \ell-1\), then we say path \(p\) is a legitimate path of pattern \(\mathcal{P}\) and \(p\) is a path instance of \(\mathcal{P}\), denoted as \(p \in \mathcal{P}\). The weight of path \(p\) is defined as \(W(p) = \sum_{i=0}^{\ell-1} W(e_i)\).

In addition to a path pattern, we specify a start and end vertex \(v_s, v_t \in V\) as input to our problem. Thus, a tuple \(Q = (v_s, v_t, \mathcal{P})\) specifies a query. We furthermore assume that path instances are loopless, as loops (or cycles) in a path are rarely useful in understanding relationships between entities. Besides, the query is non-trivial, i.e., that \(\Phi(v_s) = L_0\) and \(\Phi(v_t) = L_\ell\). We now define our prioritized relationship mining problem as the \(k\)-lightest paths problem.

**Definition 3. Top-\(k\) Lightest Paths Problem.** Given a weighted HIN \(G = (V, E, \Phi, \Psi, W)\) and a query \(Q = (v_s, v_t, \mathcal{P})\), find the \(k\) loopless paths having smallest weights among all path instances of \(\mathcal{P}\) that start at vertex \(v_s\) and end at \(v_t\).

Figure 2a is an example of a weighted HIN with different shapes representing different vertex labels. A query is provided at the bottom of Figure 2a specifying the path pattern and start and end vertices. The top-\(k\) lightest paths problem is to find the \(k\) lightest loopless paths among those between vertices 1 and 4 that are path instances of the pattern.
when the network is homogeneous (i.e., problem can be formally proved to be NP-complete by a
2.3 Lightest Paths Complexity

A straightforward way to solve the top-k lightest paths problem is to enumerate all paths matching the given path pattern and pick the top-k lightest paths. However, enumerating all possible paths of length $|P|$ between two vertices in a graph can be exponential in $|P|$ and quickly becomes intractable in large graphs. In fact, the top-k lightest path problem can be formally proved to be NP-complete by a reduction from the MINIMUM-WEIGHT PATH problem [24], when the network is homogeneous (i.e., $|L| = |R| = 1$). To see this, consider the worst case when the network is homogeneous and the query pattern has length $|P| = |V| - 1$: this is the well-known Hamiltonian Path problem.

Despite the worst-case complexity of the top-k lightest paths problem, it seems easy to adapt standard graph traversal algorithms to solve the problem in practice, at least in the case where $|P|$ is small. Breadth-first search (BFS) can be adapted to enumerate all the matched paths, which we call breadth-first match (BFM). The basic idea of BFM is to conduct a BFS starting from the $v_s$ and explore the neighborhoods of the frontier vertices following the path pattern. Instead of storing only the frontier vertices, BFM stores all candidates paths reaching the frontier in order to determine if a path contains a loop. Similarly, we can modify depth-first search and have depth-first match (DFM) algorithm. Another method is to greedily explore vertices without enumerating all the paths. Dijkstra’s algorithm can be adapted to this problem by placing additional constraints on the path pattern, which we call Dijkstra’s Matching (DijkstraM) algorithm. Instead of enumerating all paths, DijkstraM preferentially explores those paths with lower weights. While these algorithms solve our problem, we point out the top-k lightest paths problem is actually significantly more complex than the standard shortest path problem.

This additional difficulty can be attributed to two issues:
1. Searching for loopless paths makes our problem more difficult. If we allow loops in the path, we only need to store frontier vertices of the searcher and simple methods such as BFS and DFS will work. However, since we require the path to be loopless, we need to keep track of each vertex in each path in order to avoid loops, which is computationally more expensive.
2. The same vertex might be explored multiple times. With query path pattern, approaches such as BFM, DFM or DijkstraM may explore the same vertex multiple times since the same vertex can be matched to different nodes in the path pattern. In the example shown in Figure 2a, when using DFM to enumerate all the legitimate paths, vertex 6 will be explored once following the path 1 → 6 and another time following the path 1 → 3 → 8 → 6. Here vertex 6 can be mapped to both the second vertex label and the fourth vertex label in the path pattern.

3. ALGORITHM

Motivated by the two issues just discussed, we propose our algorithm to solve the top-k lightest paths problem efficiently, called PROphetic HEuristic Algorithm for Path Searching (PRO-HEAPS). The algorithm is divided into two phases. The first is a preprocessing phase, where we construct a prophet graph that reduces the search space of our problem. In the second phase, we use the prophet graph to derive a heuristic function that estimates the distance to the target. This heuristic then guides an A* search, which takes place directly on the prophet graph. The key feature of the prophet graph is that we can use it to compute the solution to the query without having to refer to the original graph $G$. Though PRO-HEAPS still has exponential computational complexity in the worst case, in practice it is able to execute queries in real time as shown in our Section 4.

3.1 Prophet Graph

As mentioned, one difficulty of the top-k lightest paths problem lies in repeated exploration of vertices in the graph. In addition, when searching for legitimate paths, there are many candidate paths to explore, most of which cannot finally reach the target entity following the path pattern. This motivates us to define a prophet graph $G'$, which is a new graph derived from both the graph $G$ and the query $(v_s, v_t, P)$, in which vertices are assigned levels, which range between 0 and $|P|$. Intuitively, the $i$-th level of $G'$ contains the set of vertices $V_i$ matching the $i$-th vertex label in the path pattern $P$, which also appear $i$ steps away from $v_s$ in some path instance of $P$.

Formally, we have the following definition:

**Definition 4. Prophet Graph.** Suppose we are given a weighted HIN $G = \langle V, E, \Phi, \Psi, W \rangle$ and non-trivial query
The prophet graph is a level-wise graph \( G' = (V_0 \cup V_1 \ldots \cup V_\ell, E_0 \cup E_1 \ldots \cup E_{\ell-1}) \) where

1. \( V_0 = \{v_0\} \) and \( V_\ell = \{v_\ell\}; 
2. For \( i \in [1, \ell-1] \), a vertex \( v \in V_i \) iff \( \Phi(v) = L_i \), and there exists a vertex \( u \in V_{i-1} \) such that \( u, u' \in V_i, e_{u,v}, e_{v,u'} \in E, \Psi(e_{u,v}) = R_{i-1} \) and \( \Psi(e_{v,u'}) = R_i; 
3. For \( i \in [0, \ell-1] \), an edge \( e_{u,v} \in E_i \) iff \( u \in V_i, v \in V_{i+1} \) and \( \Phi(e_{u,v}) = R_i \).

Crucially, a vertex \( v \in V \) can appear in multiple levels of \( G' \), and thus, \( G' \) is not a subgraph of \( G \). Furthermore, the prophet graph itself does not enforce the paths be loopless: this is handled at a later step. Figure 2b shows the prophet graph for the graph and query in Figure 2a.

We now describe an algorithm for computing the prophet graph \( G' \), given a HIN \( G \) and a query \( Q \), shown in pseudocode in Algorithm 1. The major task of creating prophet graph is to determine the vertices in each level. Obviously, \( G' \) contains \(|P| + 1 \) levels and we store the vertices in each level as a set (line 1-2). To determine the vertices for each level, we then perform bidirectional BFS from the vertices \( v_0 \) and \( v_\ell \). When the two searches meet in the middle, the intersection of their frontiers becomes the middle level of \( G' \) (line 3-11). In Line 6, the \( i \)-th level is obtained by traversing towards neighbors of \((i-1)\)-th level following the outgoing edge with label matched by \((i-1)\)-th edge label in \( P \). Vertices on \( i \)-th level should contain the same label matching \( i \)-th vertex label in \( P \). Line 10 is similar to Line 6 but works in the reverse direction. The algorithm then continues the bidirectional BFS and only retain vertices visited by both searches until they reach \( v_0 \) and \( v_\ell \) respectively (line 12-13). After determining the vertices in each level, we can construct the prophet graph \( G' \) by linking each consecutive levels with edges from \( G \) matching the edge labels in \( P \).

**Algorithm 1 Create prophet graph.**

**Input:** The given HIN \( G \) and a query \( Q = (v_s, v_t, P) \).

**Output:** Prophet graph \( G' \).

1. Create an array of empty sets levels[0...|P|].
2. Set levels[0] = \{v_0\} and levels[|P|] = \{v_\ell\}.
3. Let \( mid = \frac{|P|+1}{2} \).
4. \( \triangleright \) Forward BFS from \( v_0 \).
5. \( \triangleright \) Backward BFS from \( v_\ell \).
6. for \( i = 1 \rightarrow \text{mid} \) do
   6.1. levels[i] ← \( \text{Next}(\text{levels}[i-1]) \) matching \((i-1)\)-th edge label and \( i \)-th vertex label in \( P \).
    6.2. midLevel = levels[mid]; levels[mid] = 0.
6.3. \( \triangleright \) Backward BFS from \( v_0 \).
6.4. for \( i = |P| \rightarrow 1 \) do
   6.4.1. levels[i] ← \( \text{Next}(\text{levels}[i+1]) \) matching \( i \)-th edge label and \( i \)-th vertex label in \( P \).
6.4.2. levels[mid] = levels[mid] \( \cap \) midLevel.
7. Construct prophet graph \( G' \) as mentioned in Section 2.3 can effectively avoid enumerating all the paths. A more appropriate choice will be \( A^* \) (best-first-search) algorithm, considering we are given both the source and target vertex in this problem. However, \( A^* \) algorithm requires a heuristic estimation of the minimum weight to reach the target and it is nontrivial to obtain the heuristic in the graph, especially when we expect the heuristics should ensure the optimality of \( A^* \) algorithm.

While an appropriate heuristic is difficult to obtain in the original graph, we show that it can be easily derived from the prophet graph. The value we intend to estimate, i.e. the minimum weight of acyclic path in the prophet graph from current vertex to the target vertex, is expensive to compute since we need to keep track of vertices in the path to avoid loops. However, if we allow loops, the problem becomes much easier. Here the idea of our heuristic estimator is to relax the constraint of paths by allowing loops, and to use the smallest weight loop path as an estimation of loopless smallest weight. Algorithm 2 shows how the smallest weight loop path can be efficiently obtained using backward BFS on prophet graph. Specifically, the algorithm starts backward BFS from \( v_\ell \) and propagates the heuristics in a bottom-up fashion till reaching \( v_s \). For each vertex on \( i \)-th level of \( G' \), it propagates its heuristic value to its incoming neighbors in \((i-1)\)-th level, during which the heuristic value increases by the weight of the edge between them. Each vertex in \((i-1)\)-the level will accept the minimum value among all the heuristic values propagated into it (Line 3-6).

**Algorithm 2 Calculate heuristic function.**

**Input:** Prophet Graph \( G' \) and weight function \( W \).

**Output:** Heuristic function \( H \) on vertices of prophet graph.

1. Set \( H(v_\ell) = 0 \) and for other vertices \( u \in G' \), set \( H(u) = \infty \).
2. \( \triangleright \) Backward BFS from \( v_\ell \).
3. for \( i = |P| \rightarrow 1 \) do
   3.1. for vertex \( u \in \text{levels}[i] \) do \( \triangleright \) \( i \)-th level in \( G' \).
   3.2. for vertex \( v \in u \)'s incoming neighbors do \( \triangleright \) last level.
   3.3. \( H(v) = \min(H(v), H(u) + W(e_{v,u})) \).
5. Return heuristic function \( H \).

As an example, Figure 3a shows the prophet graph with a specified weight function. Figure 3b demonstrates how the heuristic values are calculated in the prophet graph, where the green dashed lines indicate the traces of propagation of
heuristic values. For instance, knowing the heuristic values of vertex 5 and vertex 6 on 3rd level is 3 and 2 respectively, we can easily derive the heuristic value of vertex 8 on 2nd level is 3 by adding edge weight 1 to 3 and 2 respectively and take the minimum one. Note that the heuristic value of vertex $v$ is an estimation of the distance of the loopless shortest path from $v$ to $v_t$ in $G'$. It might not be a correct estimation since our heuristics allow loops in the paths. For instance, the heuristic value of vertex 6 on 1st level is 4 while the correct one should be 5 with path $6 \rightarrow 8 \rightarrow 5 \rightarrow 4$.

With the heuristic, we propose PROphetic HEuristic Algorithm for Path Searching (PRO-HEAPS), which is adapted from A* algorithm to solve our top-k lightest paths problem. Algorithm 3 describes the procedures of PRO-HEAPS. It uses the heuristic value in addition to the distance from $v_s$ to current vertex as the key in the priority queue (Line 13) and explores vertices in a greedy way. The while loop executes until we extract $k$ loopless paths, or the priority queue is empty (Line 6). Each loop extracts a path with the smallest key in the priority queue. Outgoing neighbors of the tail vertex in the path are explored if the tail vertex is not the target (Line 10 to 13). Otherwise, the path is stored (Line 8 to 9). Note that we still need to check the loop in the path (Line 12), but the priority queue along with our heuristic function ensure our algorithm exploring only a small number of paths.

![Diagram](image)

Figure 3: Calculating the heuristic on our running example.

**Algorithm 3** PROphetic HEuristic Algorithm for Path Searching (PRO-HEAPS).

| Input: HIN $G$ with weight function $W$, a query $Q = (v_s, v_t, P)$ and parameter $k$. |
| Output: Top-$k$ shortest path from $v_s$ to $v_t$ following $P$. |

1. Run Algorithm 1 to create prophet graph $G'$.
2. Run Algorithm 2 to calculate heuristics $H$.
3. $result \leftarrow$ empty array for storing top-$k$ loopless paths.
4. $frontier \leftarrow$ priority queue with entities of format (path, key).
5. Initialize $frontier$ with single-vertex path $v_s$ and key 0.
6. while $result.size() < k$ and $frontier.size() > 0$ do
   7. (path, key) $\leftarrow$ frontier.pop().
   8. if path reaches $v_t$ then
      9. $result \leftarrow$ result; Go to Line 6.
   10. vertex $u \leftarrow$ tail vertex in path.
   11. for $v \in u$'s outgoing neighbors in next level of $G'$ do
      12. if $v$ does not exist in path then
         13. Push (path + $v$, $W(path) + H(v)$) to frontier.
14. Return paths stored in result.

We now prove Algorithm 3 outputs the correct answer, i.e. top-$k$ lightest paths if any. We first propose a lemma.

**Lemma 1.** Using the heuristic from Algorithm 2, the first path added to $result$ (if any) in Line 9 of Algorithm 3 is the lightest loopless path from $v_s$ to $v_t$ following pattern $P$.

**Proof.** For a vertex $p$ in $i$-th level and each $q$ of its outgoing neighbors in $(i+1)$-th level of $G'$, the heuristic satisfies $H(p) \leq H(q) + W(pq)$ according to the property shown in Line 6 of Algorithm 2. Therefore, the heuristic is consistent [23] and our algorithm adapted from A* algorithm can guarantee to attain the lightest path once the path popped out of the priority queue reaches the target vertex.

With Lemma 1, we can induce that the $i$-th path added to $result$ (if any) is the $i$-th lightest loopless path from $v_s$ to $v_t$ following $P$. Therefore, by the end of Algorithm 3, it will return top-$k$ lightest paths if any.

## 4. EXPERIMENTS AND ANALYSIS

In this section, we compare the performance of PRO-HEAPS with a series of baselines using three different real-world datasets.

### 4.1 Experimental Setup

#### 4.1.1 Datasets Description

The three datasets used in our experiments are as follows:

- **Enron**: This is a dataset containing Email messages sent between employees of the Enron corporation. We created a HIN based on the raw dataset with four types of vertex labels: person, Email address, Email message, and topic. For the topics, we created fifty topics using the LDA model [6], and linked each Email message to the closest three topics.
- **DBLP**: A dataset of computer science bibliographic information. We created a HIN by categorizing the entries into vertex labels: author, paper, conference, and terminology.
- **Stack Overflow**: This dataset comes from a popular question answering service found among the datasets of the Stack Exchange XML dump. We parsed each post to create a HIN by dividing entities into the vertex labels: question, answer, tag and user.

Table 1 provides detailed information about each dataset. We defined weight functions over edges for these datasets as follows. For edges of action (eg. publishing, asking and emailing), we defined the weights based on the recency, exponential to the time difference between the action time on edges and the query time. For edges of relation (eg. with topics and with email address), we defined the weights as the probability of derivation (1.0 if it is certain).

| Dataset     | # Vertices | # Edges | $|L|$ | $|R|$ |
|-------------|------------|---------|-----|-----|
| Enron Dataset | 46,463     | 613,838 | 4   | 8   |
| DBLP         | 2,241,258  | 14,747,328 | 4 | 6 |
| Stack Overflow | 21,579,657 | 53,325,635 | 4 | 8 |

Table 1: The datasets used: $|L|$ is number of different vertex labels and $|R|$ is number of different edge labels.

#### 4.1.2 Baselines

We implemented three groups of algorithms, containing ten algorithms in total, to use as baseline comparisons to

[1]https://www.cs.cmu.edu/~./enron/
We used three different metrics to measure the performance of algorithms for solving the top-k lightest paths problem.

**Query Time**: We measured the time to execute each query and reported the mean query time from multiple executions. Note that for algorithms that make use of a prophet graph, the time to construct the prophet graph is included.

**Memory Consumption**: The memory consumption of executing a query, the average number of paths stored in memory, and the average number of paths explored for a query, respectively. Lines/points are missing for algorithms that do not finish within 48 hours or use more than 48 GB of RAM.

**Distance Oracle**: A distance oracle can be used to efficiently return the length of the shortest path between a source and target vertex, and is constructed in a preprocessing phase on the entire graph. In our problem, we used the distance oracle from Akiba et al. [1] with the hope of improving the algorithms in group 1. A distance oracle can be used to efficiently return the length of the shortest path between a source and target vertex, and is constructed in a preprocessing phase on the entire graph. In our problem, we used the distance oracle to prune vertices when searching for top-k lightest paths. Specifically, if the distance oracle indicates that the distance between a vertex $v$ and the target entity is larger than the remaining part of the path pattern, then vertex $v$ can be pruned since continuing current match will not reach the target entity. We applied distance oracles to the four approaches in the first group to obtain the four methods in this group. Our implementation used the exact distance oracle from Akiba et al. [1].

**Group 3**: PRO-Bidir-BFM, PRO-DijkstraM. In this final group, we implemented two additional methods that make use of the prophet graph. They run respectively Bidir-BFM and DijkstraM on the prophet graph.

**4.1.3 Evaluation metrics**

We used three different metrics to measure the performance of algorithms for solving the top-k lightest paths problem.

**Query Time**: We measured the time to execute each query and reported the mean query time from multiple executions. Note that for algorithms that make use of a prophet graph, the time to construct the prophet graph is included.

**Memory Consumption**: The memory consumption of ex-
executing a query is primarily dominated by the storage of the candidate paths. Thus, we consider the maximum number of paths stored in memory at any time during query execution as an indication of the memory consumption, and report the mean of this value over multiple executions.

**Search Space:** To gain better insights into the running time of each algorithm, we measured the mean number of candidate paths explored during a query. Here a candidate path is a path from $v_0$ (or $v_i$) to an intermediate vertex that follows the appropriate pattern.

### 4.1.4 Experimental Design

All experiments were conducted on a machine running Linux with an Intel Xeon x5650 CPU (2.67GHz) and 48GB of RAM. All algorithms were implemented in C++ and compiled using the gcc compiler. For each dataset, we generated queries with path patterns of different length, ranging from 2 to 9, as follows. To generate a query $Q' = (v'_0, v'_1, P')$, we first randomly select a vertex $v'_0$ and start random walk on the HIN of specified length to a vertex $v'_1$ to obtain a path pattern $P'$. We used the time we generated the query as the query timestamp, which is used during the query to calculate the weights on-the-fly. For each dataset, we generated 800 queries: 100 for each pattern length between 2 and 9. Each algorithm was independently executed by one process, and assigned a memory limit of 48GB and time limit of 48 hours for the set of queries. The queries were executed in order of path length, and the process was terminated if it exceeded the memory or time limit.

### 4.2 Performance

Figure 4 shows the performance of PRO-HEAPS compared to other baselines. Figure 5 presents a more detailed view of the execution time by ranking all algorithms for the pared to other baselines. For longer patterns, the overhead of constructing the prophet graph is well-compensated for by the search space reduction that it allows. Table 2 shows the speedup of PRO-HEAPS compared to the best baselines over the three datasets. For path patterns longer than 5 the speedup can be over a factor of 100 for the larger datasets: DBLP and Stack Overflow. For these datasets, the query time of PRO-HEAPS is typically around 2 seconds when $|P| = 7$, while the best baselines take more than half an hour.

Next, we describe the reasons for the significantly faster query times with PRO-HEAPS.

1. The use of the prophet graph drastically reduces the search space. As shown in Figure 4, the search space for PRO-Bidir-BFM is smaller than that of Bidir-BFM +oracle, which in turn is much smaller than that of Bidir-BFM. Similarly, the search space of PRO-DijkstraM is much smaller than that of DijkstraM+oracle, which is much smaller than that of DijkstraM. This is a strong indication that the prophet graph is more aggressive in pruning the search space compared to the distance oracle. The prophet graph not only considers the distance between a vertex and source/target vertex, but also considers the labels in the path pattern when pruning the search space. This reduction in search space clearly offsets the small time required to construct the prophet graph.

2. Leveraging loopy paths in the prophet graph as the heuristic function in A* helps to prune the search space even further. This can be seen by considering the difference in performance between PRO-HEAPS and PRO-DijkstraM in Figure 4. Owing to this heuristic, we found that PRO-HEAPS is up to 1000 times faster than PRO-DijkstraM. Moreover, this heuristic is a computationally inexpensive primitive (cf. Algorithm 2).

### 4.3 Analysis of PRO-HEAPS

We now study how the performance of PRO-HEAPS varies with respect to the properties of the graph. We considered the following properties of the input graph:

**Distribution of Weights**: Since we aim to search for the top-$k$ lightest paths in weighted HIN, we considered the effect of weight distribution on the performance of PRO-HEAPS.

**Density of HIN**: We considered the effect of graph density on PRO-HEAPS compared to baselines. In particular, we considered the rate of computational growth as the graph gets denser.

**Heterogeneity of HIN**: Intuitively, heterogeneity, i.e. the number of different labels in the graph, also affects the performance of the algorithm. Thus, we considered the variations of performance as the number of vertex labels in HIN is reduced.

To conduct experiments for analyzing our algorithm, we used the Enron dataset and manipulated it with the following procedures. 1) In order to understand the effect of weight distribution, we used edge weights generated from various distributions. We considered the following distributions: constant weight, uniform distribution, Gaussian distribution, power law distribution skewed towards high values (most weights are large, denoted as power1), and power
law distribution skewed towards low values (most weights are small, denoted as power2). We forced the weights to be positive and all distributions to have the same mean. 2) To study the effect of graph density, we randomly added edges to the Enron graph, so that the number of edges reached up to 16X the number in the original graph. In this case, we used edge weights drawn from a uniform distribution. 3) We manually made the HIN more homogeneous. This was done by reducing the number of types of vertex labels, by selecting two types and merging them into one type.

For each of these variants of the Enron graph, we ran all the previously generated queries. The results can be seen in Figure 6. Figures 6a-6c shows the averaged query time of PRO-HEAPS compared to two fastest baseline approaches under different edge weight distributions. The result of queries with path pattern length |P| between 4 and 6 are presented, respectively. We observe that the distribution of edge weights does not affect the performance of approaches that are based on enumerating paths, such as PRO-Bidir-BFM and Bidir-BFM. For PRO-HEAPS, we observe that it runs slightly faster with weights from a uniform distribution, while it tends to be a slightly slower for weights with a power law distribution skewed towards high values (power1 in Figure 6a-6c). However, overall we observe that the performance of PRO-HEAPS is relatively consistent across different weight distributions.

Figures 6d-6f show the relationship between the query time of different methods and the density of graph. The densification factor is the ratio of the number of edges in the modified graph divided by the number of edges in the original Enron graph. We can observe that the query time increases for denser graphs for all three methods. However, for queries with longer path patterns, the performance discrepancy between PRO-HEAPS and baselines becomes significantly larger as the graph get denser (Figures 6e and 6f).

Finally, Figures 6g-6i present the variations of query time as the graph gets more homogeneous. We observe that, in general, more query time is required for a more homogeneous graph. We also observe that the slow-down due to homogeneity is similar for all tested algorithms. We note that the slowdown of PRO-HEAPS is relatively small, implying possible applications for the PRO-HEAPS algorithm for solving similar problems on homogeneous graphs.

5. DISCUSSION

In this section, we first present some example use cases to demonstrate the efficacy of PRO-HEAPS in dynamically adapting to the analyst’s requirements and preferences. Then, we describe some generalizations of PRO-HEAPS that make it more flexible and usable in a wider range of applications.

5.1 Example Use Cases

Relations between Stack Overflow users. In this use-case, we show how an analyst can use our solution, PRO-HEAPS, to study (e.g., to verify the expertise of people) publicly available forums such as Stack Overflow. Consider two user accounts, Gordan Linoff and Mureinik, both of whom claim to be experts in the SQL and MySQL area. To verify this, an analyst can easily formulate a query: \( P' = (\text{Gordan Linoff}, \text{Mureinik}, \text{account} \rightarrow \text{answer} \rightarrow \text{question} \rightarrow \text{answer} \rightarrow \text{account}) \) and use some weight function depending on recency. For a query in June 2014, PRO-HEAPS finds a relationship instance where the two users answered a question on Oracle database \(^{5}\) in April 2014. For a query in Jan 2014, PRO-HEAPS finds a question on MySQL answered by both in December 2013 \(^{6}\) as the best match. This example demonstrates the efficacy of path patterns and the weight function in expressing the intent of an analyst and the effectiveness of PRO-HEAPS in understanding that intent and returning the relevant relationship instance.

Next, we consider Martijn Pieters whose interest is mostly in Python and Gordan Linoff (active in SQL and MySQL). A query with the same path pattern as \( P' \) results in no matches between these accounts. This validates that there is no question which was answered by both these users, indicating that their expertise is on different topics. However, \(^{5}\)http://stackoverflow.com/questions/23298310

\(^{6}\)http://stackoverflow.com/questions/20828174

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Speedup w.r.t Best Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>length</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Enron</td>
<td>0.7X(PRO-Bidir-BFM)</td>
</tr>
<tr>
<td>DBLP</td>
<td>14.1X(PRO-Bidir-BFM)</td>
</tr>
<tr>
<td>Stack Overflow</td>
<td>1.0X(PRO-Bidir-BFM)</td>
</tr>
</tbody>
</table>

Figure 5: Timing comparisons for path patterns of length 4 (in seconds). Methods are ranked in decreasing order of query time. Missing bars (e.g., DijkstraM in DBLP) are due to not finishing within time/memory limit.
when the path pattern is modified to `account → answer → question → tag → question → answer → account`, PRO-HEAPS reveals that these two accounts did answer questions with the same tag of `String` in 2014. This indicates that their interests are still on the same general subject.

**Common publication venues between authors.** In this use-case, we use PRO-HEAPS to examine the relationship between authors of publications indexed by DBLP. We consider Jiawei Han and Ion Stoica, two famous researchers in the areas of data-mining and systems respectively, as objects of our queries. When the query is formulated as $P' = (\text{Jiawei Han}, \text{Ion Stoica}, \text{author} \rightarrow \text{paper} \rightarrow \text{venue} \rightarrow \text{paper} \rightarrow \text{author})$ with query time in 2009 to discover the common venue where the two researchers (from different areas) publish and the weight function is set based on recency, PRO-HEAPS identifies that Jiawei Han has a paper in ICDCS’09 and Ion Stoica has a paper in ICDCS’08. However, when the weights are defined based on the influence of papers and venues (to determine the common venue where they published their most influential work), PRO-HEAPS returns SIGMOD. Again, this illustrates the efficiency of PRO-HEAPS in handling complex queries.

---

8. Modeling Probabilistic Measurement Correlations for Problem Determination in Large-Scale Distributed Systems
7. Adaptive Distributed Time-Slot Based Scheduling for Fairness in Multi-Hop Wireless Networks
9. Jiawei Han published a paper called *Mining Frequent Patterns without Candidate Generation* in SIGMOD’00 and Ion Stoica published a paper called *Declarative networking: lan-
ficacy of weight functions and path patterns in capturing complex intuitions regarding relationships and shows that PRO-HEAPS is able to find and rank the relevant instances.

5.2 Generalizations of PRO-HEAPS

Our solution PRO-HEAPS is very flexible and can be further generalized in the following ways:

1. The specification of node and edge labels in the query path pattern can be a logical statement with OR/AND/NOT. This also enables wildcard labels “*” that matches any other label. For instance, an analyst can ask the following query on the DBLP graph: Author 1 \( \rightarrow \) paper OR poster \( \rightarrow \) NOT workshop AND NOT journal \( \rightarrow \). Author 2 \( \rightarrow \) find an instance of a connection between two authors in which the first author published a paper or a poster in a venue that is neither a workshop, nor a journal (e.g., conference/symposium), where the second author also published something. This relaxation can be easily incorporated in the construction of the prophet graph and the follow-up steps of the algorithm can also be easily adapted to support this flexibility.

2. Since in PRO-HEAPS, there is no pre-processing of the graph based on weights, the weights function can, as well, be specified at the time of query. This allows an analyst to rank the relationship instances in many different ways, learning different insights in the process.

3. PRO-HEAPS can also be used to mine relationships between two groups of entities, rather than just two entities, where it will return the top-\(k\) lightest paths between the two groups. To support this extension, the first and the last level of prophet graph have to contain multiple vertices belonging to the two groups. Other algorithmic steps are similar to the ones described in Section 3.

4. An extension we intend to investigate is to support privacy-preserving analysis for applications where user privacy is sacrosanct. Recent efforts on the privacy preserving publishing of social network data\cite{36} coupled with anonymization strategies\cite{21} of user profiles, can potentially be implemented on top of our current efforts.

6. RELATED WORK

\(k\)-shortest Path Problems and Variants: The \(k\)-shortest path problem seeks to find the top-\(k\) shortest path between two vertices in a graph \cite{35,27}. A variant of this problem, closer to our work, is restricted to disallow loops within paths. Exact algorithms for solving this problem is too expensive for large graphs \cite{35,17,14}, and have led to the development of parallel \cite{33} and approximation approaches \cite{15,27}. Another related problem, the minimum-weight path problem, aims to find the minimum-weight simple path of a user-defined length and its solution relies on a randomized algorithm based on color coding \cite{3}, which typically works on smaller-scale networks \cite{24}. In contrast to these techniques that primarily focus on homogeneous graphs, we leverage the idea of a path pattern with node and edge labels to both prioritize paths and drastically prune the search space of possible paths matching the pattern.

Generalized Pattern Matching in Graphs: The classic structural graph pattern match problem, known as subgraph isomorphism, aims to find matches for a given graph pattern among a graph database. It is known to be NP-complete \cite{12} and will be too expensive for our purpose, even with various optimizations \cite{32,11} and approximations \cite{8,25,11}. Beyond focusing just on structure, there are semantic variants, where pattern graph and data graph contain labels on nodes and/or edges. GraphGrep \cite{13} provides an exact algorithm for this problem. Specifically, it represents the graph database as a set of all possible paths and parses a query graph into a series of label paths. Then the matching becomes straightforward after filtering unpromising path matching. This algorithm has exponential complexity and works only for small graphs. Even approximate \cite{2,7,34,18,19} and distributed variants \cite{4} often do not scale to large graphs. While the above semantic graph pattern matching algorithms focus on general pattern matching in graph database, our work can be thought of as a specialization of the above, for which a scalable algorithm is realized through novel graph pre-processing and smart heuristics.

Meta-Paths in Heterogeneous Networks: Meta-paths, essentially a labeled path within a HIN \cite{28}, have found significant use as a mechanism to quantify the similarity between a pair of nodes within a HIN \cite{30,29,20,26,28}. Variants of the above include work by Lao et al. \cite{20} which uses path constraint random walks to quantify the similarity of nodes while Shi et al. \cite{26} defined a symmetric relevance measurement based on the pair-wise random walk. More recently, Meng et al. \cite{22} studied how to discover relevant metapaths given pairs of related nodes, where they defined the problem in supervised learning context and leveraged a greedy algorithm. These works are distinct from ours with respect to application (e.g. similarity search, clustering and link prediction) and the fact that they do not leverage users’ preference and are limited to unweighted graphs.

Relationship Mining in Graphs: Keyword search in relational databases is an important problem in web-based search where vertices represent tuples and edges represent the foreign-key relationships. Solutions seek tree- (e.g. Steiner trees) or subgraph- patterns to explain the relationships among a given set of keywords \cite{5,16}. Related to the above is the notion of center-piece subgraphs where the authors \cite{9,31} seek to find the connected subgraphs between two or more entities to explain the relationship. Their algorithm is guided by a specific goodness function while restricting the size of the subgraph to some budget. Fang et al. extend the above and in a heterogeneous graph setting \cite{10}. However, none of them incorporates the users’ preference and prioritizes relationship mining.

7. CONCLUSIONS

In this paper, we solve the problem of prioritized relationship mining considering user preference and formalize it as the top-\(k\) lightest paths problem. Our algorithm for this problem, PRO-HEAPS, outperforms numerous baseline approaches with speedups as large as 100X and is able to execute queries in real time even on large-scale graphs with complex relationships. We show our algorithm can be extended to solve more generalized problems and has the potential to enable many other applications involving the search of paths in HINs.

Acknowledgements. This work is supported by NSF Award NSF-EAR-1520870 and NSF-DMS-1418265.
8. REFERENCES