What Links Alice and Bob? Matching and Ranking Semantic Patterns in Heterogeneous 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 and focused on mining heterogeneous information networks (HINs) where both nodes and edges can be of different types. Building 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 a combination of novel graph preprocessing techniques, 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

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2883007.

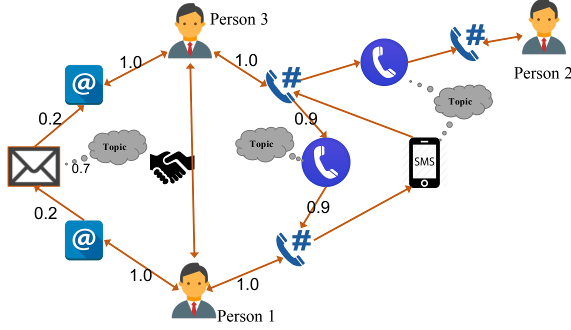


Figure 1: Communication Network: different icons represent person, email address, email message, phone number, etc.[1]

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: (Person 1) → → → → → → → (Person 2). The interestingness metric of recency can be captured by a weight function where the weight of the edges (eg. →) is exponential to the difference between the current time and the time of communication. We then formalize the problem as *top-k lightest paths problem*, targeting the top-k lightest loopless paths between the entities , matching the path pattern (see Section 2.1).

The problem of finding the (top-k) 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; $\Phi : V \to \mathcal{L}$ is a vertex labeling function; $\Psi : E \to \mathcal{R}$ is an edge labeling function; and $W : E \to \mathbb{R}$ 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 $N_{out}(u)$ and $N_{in}(u)$, 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 relationship 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 $L_0, ..., L_\ell \in \mathcal{L}$ are vertex labels and $R_0, ..., R_{\ell-1} \in \mathcal{R}$ are edge labels.*

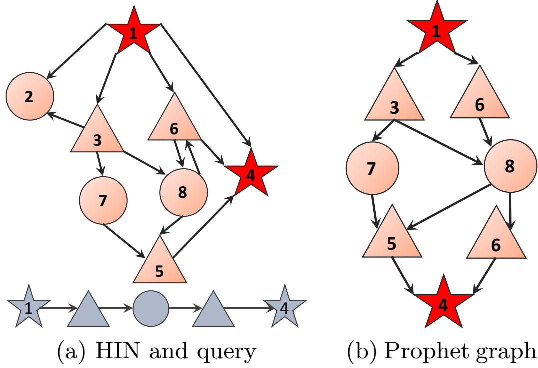
Here we define the length of a path pattern $\mathcal{P}$ to be the number of edges in $\mathcal{P}$ and denote it as $|\mathcal{P}| = \ell$. Given a weighted HIN $G = (V, E, \Phi, \Psi, W)$, if a directed path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{\ell-1}} v_\ell$ in 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 $\Phi(v_i) = L_i, \forall i = 0, ..., \ell$ and $\Psi(e_i) = R_i, \forall i = 0, ..., \ell-1$, 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 $\mathcal{W}(p) = \sum_{i=0}^{\ell-1} W(e_i)$.

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 $\mathcal{Q} = (v_s, v_t, \mathcal{P})$ 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 *top-k lightest paths problem*.

Definition 3. ***Top-k Lightest Paths Problem.*** *Given a weighted HIN $G = (V, E, \Phi, \Psi, W)$ and a query $\mathcal{Q} = (v_s, v_t, \mathcal{P})$, 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.

[1]We note that in such applications user privacy is an important consideration. While it is out-of-scope in the current effort, we discuss it as future directions in Section 5.2.



(a) HIN and query (b) Prophet graph

Figure 2: Our running graph and query example.

2.2 Path Pattern and Edge Weights on HIN

To mine a prioritized relationship, the path pattern is provided to specify the relationship type. Given the path pattern, we avoid other relationship types and can avoid being overloaded by other trivial or overcomplex relationships. In many scenarios, users can determine appropriate path patterns to indicate their preference. However, in cases where it is difficult to reason about path patterns, domain experts become necessary. Automatic systems can also be used to find potentially interesting path patterns [22].

Moreover, the edges in a HIN can be assigned weights based on interestingness metrics given by experts or users. For instance, the relationships can be prioritized based on the recency of the individual relationships captured in the information network. Alternatively, in a noisy environment (e.g., information extracted using NLP techniques), the weight can be based on the reliability of the information. For derived attributes (such as the topic of a message), it can be based on the probability of successful derivation, e.g. probability of the text belonging to a given topic. Note that the weights might not be static and sometimes should be calculated on-the-fly at query time. For example, weights defined on recency cannot be calculated until the query time.

2.3 Top-k 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 $|\mathcal{P}|$ between two vertices in a graph can be exponential in $|\mathcal{P}|$ 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., $|\mathcal{L}| = |\mathcal{R}| = 1$). To see this, consider the worst case when the network is homogeneous and the query pattern has length $|\mathcal{P}| = |V|-1$: 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 $|\mathcal{P}|$ 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 $1 \rightarrow 6$ and another time following the path $1 \rightarrow 3 \rightarrow 8 \rightarrow 6$. 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 $(v_s, v_t, \mathcal{P})$, in which vertices are assigned *levels*, which range between 0 and $|\mathcal{P}|$. Intuitively, the i-th level of G' contains the set of vertices V_i matching the i-th vertex label in the path pattern $\mathcal{P}$, which also appear i steps away from v_s in some path instance of $\mathcal{P}$.

Formally, we have the following definition:

Definition 4. ***Prophet Graph.*** *Suppose we are given a weighted HIN $G = (V, E, \Phi, \Psi, W)$ and non-trivial query*

$\mathcal{Q} = (v_s, v_t, \mathcal{P} = L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} ... \xrightarrow{R_{\ell-1}} L_\ell)$. *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. $V_0 = \{v_s\}$ *and* $V_\ell = \{v_t\}$*;*
2. *For* $i \in [1, \ell-1]$*, a vertex* $v \in V_i$ *iff* $\Phi(v) = L_i$*, and there exists a vertex* $u \in V_{i-1}$*,* $u' \in V_{i+1}$ *such that* $u, u' \in V$*,* $e_{u,v}, e_{v,u'} \in E$*,* $\Psi(e_{u,v}) = R_{i-1}$ *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 $\mathcal{Q}$, shown in pseudocode in Algorithm 1. The major task of creating prophet graph is to determine the vertices in each level. Obviously, G' contains $|\mathcal{P}|+1$ 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_s and v_t. 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 $\mathcal{P}$. Vertices on i-th level should contain the same label matching i-th vertex label in $\mathcal{P}$. 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_t and v_s 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 $\mathcal{P}$.

Algorithm 1 Create prophet graph.

Input: The given HIN G and a query $\mathcal{Q} = (v_s, v_t, \mathcal{P})$.
Output: Prophet graph G'.

1: Create an array of empty sets $levels[0...|\mathcal{P}|]$.
2: Set $levels[0] = \{v_s\}$ and $levels[|\mathcal{P}|] = \{v_t\}$.
3: Let $mid = \lfloor \frac{|\mathcal{P}|+1}{2} \rfloor$.
4: ▷ *Forward BFS from* v_s.
5: **for** $i = 1 \to mid$ **do**
6: $levels[i] \leftarrow N_{out}(levels[i-1])$ matching $(i-1)$-th edge label and i-th vertex label in $\mathcal{P}$.
7: $midLevel = levels[mid]$; $levels[mid] = \emptyset$.
8: ▷ *Backward BFS from* v_t.
9: **for** $i = |\mathcal{P}| - 1 \to mid$ **do**
10: $levels[i] \leftarrow N_{in}(levels[i+1])$ matching i-th edge label and i-th vertex label in $\mathcal{P}$.
11: $levels[mid] = levels[mid] \cap midLevel$.
12: Continue forward BFS and prune $levels$ till reaching v_t.
13: Continue backward BFS and prune $levels$ till reaching v_s.
14: Construct G' based on vertices from $levels$ and edges from G matched by $\mathcal{P}$.

Note that the prophet graph can be created efficiently. It is more efficient than searching for paths in a brute-force manner, since we need not store all paths, but rather only keep track of candidate vertices in each level. This can be done in a dynamic programming fashion as shown in Algorithm 1. In addition, the adoption of bi-directional BFS helps prune many vertices from the prophet graph.

The prophet graph will be a powerful tool for our top-k lightest paths problem. After pruning vertices that are too far away from v_s or v_t according to the path pattern, it only retains vertices that lie in the path from v_s to v_t following the path pattern. With the prophet graph in hand, we can directly search on it, without needing to refer to the original graph G and query $\mathcal{Q}$. Walking down from v_s level by level to v_t will obtain a path following the path pattern though might contain loops. It can be verified that all the legitimate paths for the query $\mathcal{Q}$ can be derived from the prophet graph by traversing from v_s to v_t level by level.

3.2 PRO-HEAPS

We can run BFM or DFM (see Section 2.3) on top of prophet graph to enumerate all the paths and can correctly find the top-k lightest paths. However, it might be too expensive to perform since the number of legitimate paths of the query can potentially be very huge and we are interested in merely a few of them. Therefore, greedy algorithms with priority at exploring vertices will be a better option. For example, methods adapted from Dijkstra's algorithm (DijkstraM) 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 loopy path as an estimation of loopless smallest weight. Algorithm 2 shows how the smallest weight loopy path can be efficiently obtained using backward BFS on prophet graph. Specifically, the algorithm starts backward BFS from v_t and propagate 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.

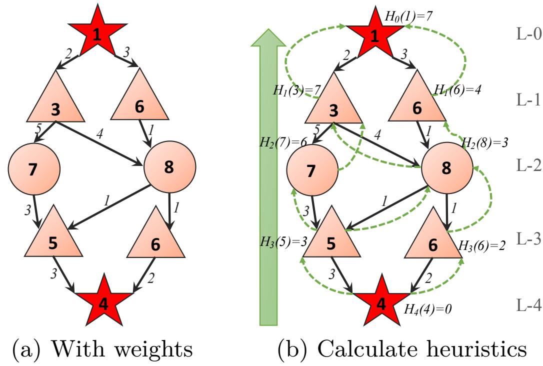
Input: Prophet Graph G' and weight function W.
Output: Heuristic function H on vertices of prophet graph.

1: Set $H(v_t) = 0$ and for other vertices u in G', set $H(u) = \infty$.
2: ▷ *Backward BFS from* v_t.
3: **for** $i = |\mathcal{P}| \to 1$ **do**
4: **for** vertex $u \in levels[i]$ **do** ▷ *i-th level in* G'.
5: **for** vertex $v \in u$'s incoming neighbors **do** ▷ *last level.*
6: $H(v) = \min(H(v), H(u) + W(e_{v,u}))$.
7: 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 $6 \to 8 \to 5 \to 4$.

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 priory 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 property of priority queue along with our heuristic function ensure our algorithm exploring only a small number of paths.



(a) With weights (b) Calculate heuristics

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 $\mathcal{Q} = (v_s, v_t, \mathcal{P})$ and parameter k.
Output: Top-k shortest path from v_s to v_t following $\mathcal{P}$.

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 lightest 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: Add $path$ to $result$; 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 $(path + v, \mathcal{W}(path) + H(v))$ 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 $\mathcal{P}$.*

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 $H(p) \leq H(q) + W(e_{p,q})$ 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 $\mathcal{P}$. 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[2]: 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[3]: A dataset of computer science bibliographic information. We created a HIN by categorizing the entities into vertex labels: author, paper, conference, and terminology.
Stack Overflow[4]: 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).

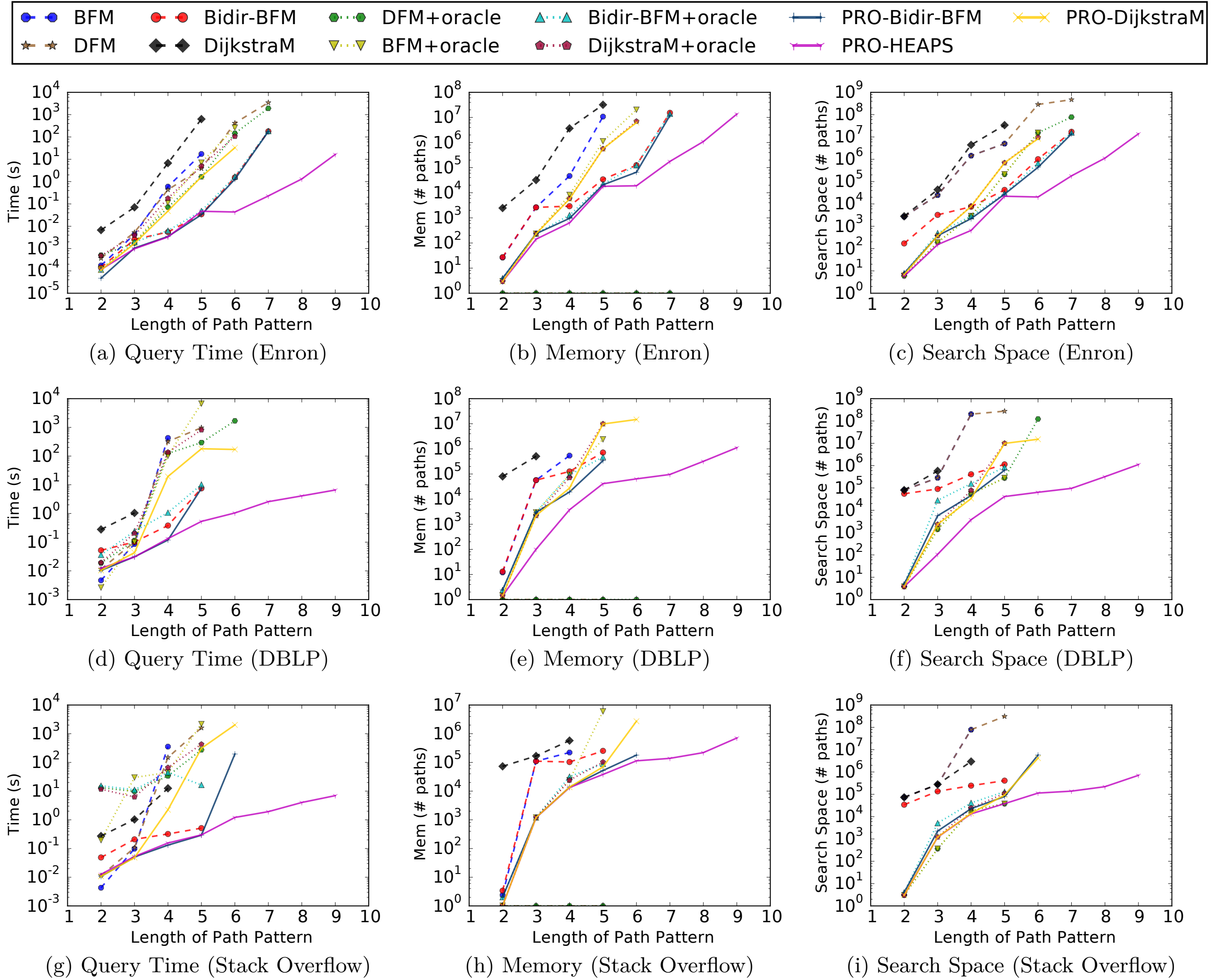
Dataset	# Vertices	# Edges	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$
Enron Dataset	46,463	613,838	4	8
DBLP	2,241,258	14,747,328	4	6
Stack Overflow	21,579,657	53,325,635	4	8

Table 1: The datasets used: $\lvert\mathcal{L}\rvert$ is number of different vertex labels and $\lvert\mathcal{R}\rvert$ 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

[2]https://www.cs.cmu.edu/~./enron/
[3]http://dblp.uni-trier.de/xml/
[4]https://archive.org/details/stackexchange



(a) Query Time (Enron) (b) Memory (Enron) (c) Search Space (Enron)

(d) Query Time (DBLP) (e) Memory (DBLP) (f) Search Space (DBLP)

(g) Query Time (Stack Overflow) (h) Memory (Stack Overflow) (i) Search Space (Stack Overflow)

Figure 4: Performance comparisons of different algorithms. Top to bottom: The rows correspond to the results for the Enron, DBLP, and Stack Overflow datasets, respectively. Left to right: The columns show the average time for 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.

PRO-HEAPS.

Group 1: *DFM, BFM, Bidir-BFM, DijkstraM*. This group of methods contains four basic approaches. Section 2.3 describes depth-first matching (DFM), breadth-first matching (BFM) and Dijkstra's matching (DijkstraM) algorithm. The other one, Bidirectional breadth-first matching (Bidir-BFM), is similar to BFM but searches from v_s and v_t alternately until meeting in the middle.

Group 2: *DFM+oracle, BFM+oracle, Bidir-BFM+oracle, DijkstraM+oracle*. In this group, we employed an off-the-shelf distance oracle tool [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-

ecuting 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_s (or v_t) 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 complied 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 $\mathcal{Q}' = (v_s', v_t', \mathcal{P}')$, we first randomly select a vertex v_s' and start random walk on the HIN of specified length to a vertex v_t' to obtain a path pattern $\mathcal{P}'$. 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 case where the length of path pattern $|\mathcal{P}| = 4$. Table 2 compares PRO-HEAPS with the fastest baseline to demonstrate the speedup of our algorithm. The main observations are highlighted and discussed as follows:

1. In general, both the distance oracle and prophet graph usually speed up the searching algorithm. In Figure 5, it is obvious that DFM/BFM with distance oracle is significantly faster than the plain DFM/BFM over all three datasets. However, the distance oracle does not always speed up Bidir-BFM and DijkstraM, which can be observed from Stack Overflow dataset in Figure 5c. On the other hand, the prophet graph tends to bring more improvements for both DijkstraM and Bidir-BFM. We observe that PRO-DijkstraM is much faster than DijkstraM+oracle and DijkstraM. Similarly, PRO-Bidir-BFM is much faster than Bidir-BFM+oracle and Bidir-BFM.

2. PRO-HEAPS reduces memory consumption and search space drastically. PRO-HEAPS uses much less memory than other baseline methods, with the exception of DFM and DFM+oracle, as they only store one path. Furthermore, PRO-HEAPS prunes the search space far more aggressively compared to other baseline methods. The reduction in memory consumption and search space becomes increasingly evident as the length of the path pattern increases (cf. 4b-4c, 4e-4f and 4h-4i).

3. From Figures 4a, 4d and 4g it can be seen that PRO-HEAPS is significantly faster on query time compared to other baselines, for longer patterns (i.e., $|\mathcal{P}| > 5$). For shorter patterns (of length between 2 and 5), PRO-HEAPS performs comparably to the best baseline methods, with queries taking up to a few hundreds of milliseconds. This is consistent with our intuition that for longer path 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 $|\mathcal{P}| = 7$, 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 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

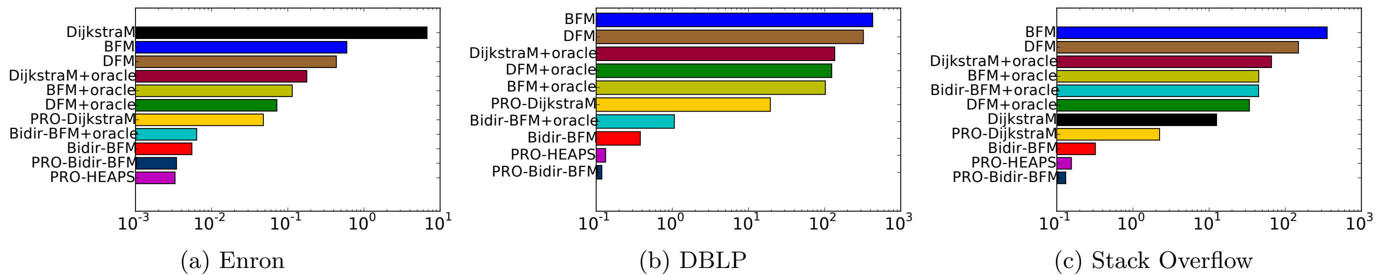


Figure 5: Timing comparisons for path patterns of length 4 (in seconds). Methods are ranked in decreasing order of query time. Missing bars (e.g., DijkstraM in DBLP) are due to not finishing within time/memory limit.

Dataset	Speedup w.r.t Best Baseline		
	length $\|\mathcal{P}\| = 5$	length $\|\mathcal{P}\| = 6$	length $\|\mathcal{P}\| = 7$
Enron	0.7X(PRO-Bidir-BFM)	28.7X(PRO-Bidir-BFM)	735.6X(PRO-Bidir-BFM)
DBLP	14.1X(PRO-Bidir-BFM)	163.1X(PRO-DijkstraM)	>670.4X(-)
Stack Overflow	1.0X(PRO-Bidir-BFM)	165.1X(PRO-Bidir-BFM)	>932.6X(-)

Table 2: PRO-HEAPS compared with the fastest baselines. The best baseline method is listed inside the parenthesis. A hyphen ("-") indicates that none of the baselines finished in time, and only a lower-bound on the speedup is shown.

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 $|\mathcal{P}|$ 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

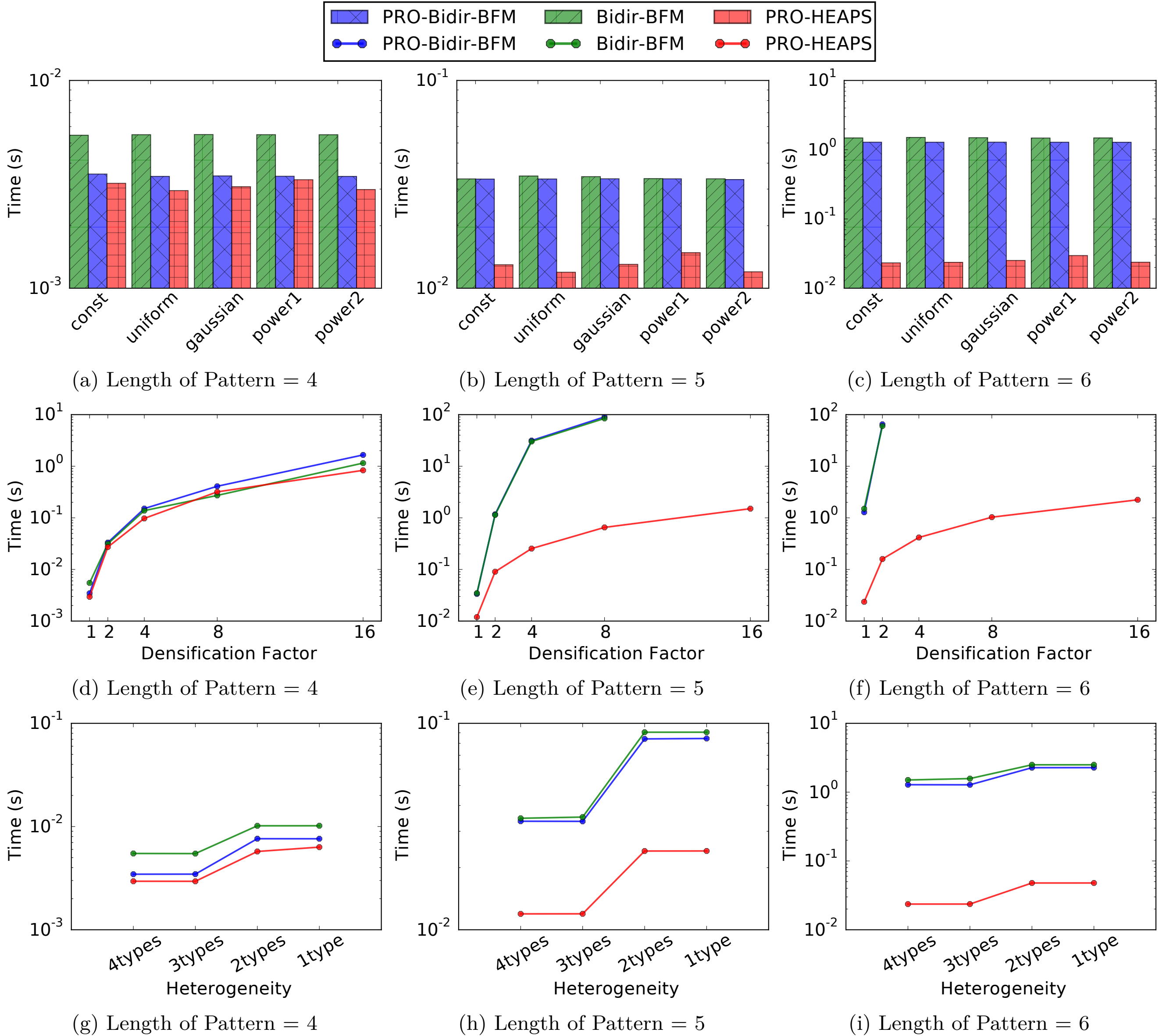
In this section, we first present some example use cases to demonstrate the efficacy of PRO-HEAPS in dynamically adapting to the analyst's requirements and preferences. Then, we describe some generalizations of PRO-HEAPS that make it more flexible and usable in a wider range of applications.

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: $\mathcal{P}' =$ (Gordan Linoff, Mureinik, $account \rightarrow answer \rightarrow question \rightarrow answer \rightarrow account$) 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 $\mathcal{P}'$ 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



(a) Length of Pattern = 4 (b) Length of Pattern = 5 (c) Length of Pattern = 6

(d) Length of Pattern = 4 (e) Length of Pattern = 5 (f) Length of Pattern = 6

(g) Length of Pattern = 4 (h) Length of Pattern = 5 (i) Length of Pattern = 6

Figure 6: Effect of weight distribution, graph density and heterogeneity. (a)-(c) show different weight distributions on the x-axis. On the x-axis of (d)-(f), the densification factor is the ratio of # edges in the graph to the # edges in the original graph. For (g)-(i), the x-axis indicates the total number of different vertex labels in the graph.

when the path pattern is modified to *account* $\rightarrow$ *answer* $\rightarrow$ *question* $\rightarrow$ *tag* $\rightarrow$ *question* $\rightarrow$ *answer* $\rightarrow$ *account*, PRO-HEAPS reveals that these two accounts did answer questions with the same tag of *String* in 2014[7]. 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 $\mathcal{P}' = (\text{Jiawei Han}, \text{Ion Stoica}, \textit{author} \rightarrow \textit{paper} \rightarrow \textit{venue} \rightarrow \textit{paper} \rightarrow \textit{author})$ 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 [8] and Ion Stoica has a paper in ICDCS'08 [9]. 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 [10]. Again, this illustrates the ef-

[7]http://stackoverflow.com/questions/18744391, http://stackoverflow.com/questions/17299581

[8]*Modeling Probabilistic Measurement Correlations for Problem Determination in Large-Scale Distributed Systems*

[9]*Adaptive Distributed Time-Slot Based Scheduling for Fairness in Multi-Hop Wireless Networks*

[10]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 → *paper OR poster* → *NOT workshop AND NOT journal* → . → Author 2 to 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[36] coupled with anonymization strategies[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 [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 [35, 17, 14], and have led to the development of parallel [33] and approximation approaches [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 [3], which typically works on smaller-scale networks [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 [12] and will be too expensive for our purpose, even with various optimizations [32, 11] and approximations [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 [13] provides an exact algorithm for this problem. Specifically, it represents the graph database as a set of all possible paths and parse a query graph into a series of label paths. Then the matching becomes straightforward after filtering unpromising path mapping. This algorithm has exponential complexity and works only for small graphs. Even approximate [2, 7, 34, 18, 19] and distributed variants [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.

guage, execution and optimization in SIGMOD'06

Meta-Paths in Heterogeneous Networks: Meta-paths, essentially a labeled path within a HIN [28], have found significant use as a mechanism to quantify the similarity between a pair of nodes within a HIN [30, 29, 20, 26, 28]. Variants of the above include work by Lao et al. [20] which uses path constraint random walks to quantify the similarity of nodes while Shi et al. [26] defined a symmetric relevance measurement based on the pair-wise random walk. More recently, Meng et al. [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 [5, 16]. Related to the above is the notion of center-piece subgraphs where the authors [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 [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

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360. ACM, 2013.

[2] B. Aleman-Meza, C. Halaschek-Wiener, S. S. Sahoo, A. Sheth, and I. B. Arpinar. Template based semantic similarity for security applications. In *Intelligence and Security Informatics*, pages 621–622. Springer, 2005.

[3] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[4] Y. Bai, C. Wang, X. Ying, M. Wang, and Y. Gong. Path pattern query processing on large graphs. In *IEEE BdCloud*. IEEE, 2014.

[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*. IEEE, 2002.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[7] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[8] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, pages 231–255, 1994.

[9] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *SIGKDD*, pages 118–127. ACM, 2004.

[10] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. Rex: explaining relationships between entity pairs. *Proceedings of the VLDB Endowment*, 5(3), 2011.

[11] B. Gallagher. *The state of the art in graph-based pattern matching*. United States. Department of Energy, 2006.

[12] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. wh freeman, 2002.

[13] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 112–115. IEEE, 2002.

[14] E. Hadjiconstantinou and N. Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, (34.2):88–101, 1999.

[15] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, 3(4):45, 2007.

[16] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[17] N. Katoh, I. Toshihide, and M. Hisashi. An efficient algorithm for k shortest simple paths. *Networks*, (12.4):411–427, 1982.

[18] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912. ACM, 2011.

[19] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. In *VLDB*, volume 6, pages 181–192. VLDB Endowment, 2013.

[20] N. Lao and W. W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine learning*, 81(1):53–67, 2010.

[21] K. Liu, K. Das, T. Grandison, and H. Kargupta. Privacy-preserving data analysis on graphs and social networks, 2008.

[22] C. Meng, R. Cheng, S. Maniu, P. Senellart, and W. Zhang. Discovering meta-paths in large heterogeneous information networks. In *WWW*, pages 754–764. International World Wide Web Conferences Steering Committee, 2015.

[23] J. Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[24] J. Scott, T. Ideker, R. M. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *Journal of Computational Biology*, 13(2):133–144, 2006.

[25] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *ACM SIGMOD symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[26] C. Shi, X. Kong, Y. Huang, P. S. Yu, and B. Wu. Hetesim: A general framework for relevance measure in heterogeneous networks. *TKDE*, 26(10):2479–2492, 2014.

[27] Y.-K. Shih and S. Parthasarathy. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12):i49–i58, 2012.

[28] Y. Sun and J. Han. Mining heterogeneous information networks: a structural analysis approach. *ACM SIGKDD Explorations Newsletter*, 14(2):20–28, 2013.

[29] Y. Sun, J. Han, C. C. Aggarwal, and N. V. Chawla. When will it happen?: relationship prediction in heterogeneous information networks. In *WSDM*, pages 663–672. ACM, 2012.

[30] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB*, 2011.

[31] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *SIGKDD*. ACM, 2006.

[32] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[33] A. Ullrich and C. V. Forst. k-pathA: k-shortest path algorithm. In *IEEE International workshop on High Performance Computational Systems Biology*, 2009.

[34] M. Wolverton, P. Berry, I. W. Harrison, J. D. Lowrance, D. N. Morley, A. C. Rodriguez, E. H. Ruspini, and J. Thomere. Law: A workbench for approximate pattern matching in relational data. In *IAAI*, volume 3, pages 143–150, 2003.

[35] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, (17.11):712–716, 1971.

[36] B. Zhou, J. Pei, and W. Luk. A brief survey on anonymization techniques for privacy preserving publishing of social network data. *ACM SIGKDD Explorations Newsletter*, 10(2):12–22, 2008.