Prioritized Relationship Analysis in Heterogeneous Information Networks

JIONGQIAN LIANG, The Ohio State University
DEEPAK AJWANI, PATRICK K. NICHOLSON, and ALESSANDRA SALA, Bell Labs
SRINIVASAN PARTHASARATHY, The Ohio State University

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.

To widen the range of applications, we also extend PRO-HEAPS to (i) support relationship analysis between two groups of entities and (ii) allow pattern path in the query to contain logical statements with operators AND, OR, NOT, and wild-card ".". We run experiments using this generalized version of PRO-HEAPS and demonstrate that the advantage of PRO-HEAPS becomes even more pronounced for these general cases. Furthermore, we conduct a comprehensive analysis to study how the performance of PRO-HEAPS varies with respect to various attributes of the input HIN. We finally conduct a case study to demonstrate valuable applications of our algorithm.

CCS Concepts: • **Information systems → Data mining**; • **Theory of computation → Social networks**;

Additional Key Words and Phrases: Heterogeneous information networks, semantic relationship queries, graph algorithms

ACM Reference format:
Jiongqian Liang, Deepak Ajwani, Patrick K. Nicholson, Alessandra Sala, and Srinivasan Parthasarathy. 2018. Prioritized Relationship Analysis in Heterogeneous Information Networks. *ACM Trans. Knowl. Discov. Data.* 12, 3, Article 29 (January 2018), 27 pages.
https://doi.org/10.1145/3154401

This work is supported by NSF Awards NSF-EAR-1520870 and NSF-DMS-1418265.
Authors' addresses: J. Liang and S. Parthasarathy, Department of Computer Science and Engineering, The Ohio State University; emails: {liangji, srini}@cse.ohio-state.edu; D. Ajwani, P. K. Nicholson, and A. Sala, Bell Labs, Ireland; emails: {deepak.ajwani, pat.nicholson, alessandra.sala}@nokia-bell-labs.com.

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, and so on, 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., (Sun and Han 2013; Sun et al. 2011, 2012; Shi et al. 2014; Liang et al. 2018; Meng et al. 2015)).

A fundamental problem in mining HINs 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 (Faloutsos et al. 2004; Tong and Faloutsos 2006) or performs generic mining of heterogeneous networks (Fang et al. 2011) 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. As another example, given a bibliographical HIN where papers, authors, conferences, and terminologies are connected to one another, the analyst might be interested in finding out what are the recent topics two researchers work on or what is the most influential co-authored paper of two researchers.

In this article, 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 *subgraph 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: 👤(Person 1) → @ → ✉ → @ → 👤 → ☏ → 📞 → ☏ → 👤 (Person 2). The interestingness metric of recency can be captured by a weight function where the weight of the edges (e.g., @ → ✉) 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 by employing carefully selected heuristics. Leveraging the input query pattern, we generate an intermediate structure, called a *prophet graph*, that enables efficient pruning of the query search space. We devise a consistent heuristic consisting of a breadth-first search (BFS) on the prophet graph. Adapting the A* algorithm with this heuristic, PRO-HEAPS is able to discover prioritized relationships in real time even when dealing with large-scale graphs and reasonably complex relationships. A preliminary version of this work is published in WWW 2016 (Liang et al. 2016). In this work, we made several major improvements. First, we extend the application of PRO-HEAPS to dynamic weighted HIN and allow changes in both graph structure and weights (Sections 4.1 and 6.1). Second, we generalize PRO-HEAPS to support relationships mining between two groups of entities and demonstrate that the advantage of the PRO-HEAPS is more evident with this extension (Sections 4.2 and 5.3). Third, we generalize the pattern path in the query to support logical statements with operators AND, OR, NOT, and wild-card label ".". We show that this generalization provides a more flexible way for specifying users' interest and has broader applications (Sections 4.3 and 5.3). Fourth, we conduct a comprehensive analysis to study the property of our algorithm, where we review the effects of several key factors on the performance of PRO-HEAPS (Section 5.4).

As a whole, the main contributions of this work are as follows:

(1) Prioritizing relationship by specifying the relationship type of interest and weighting the edges of the graph.
(2) Designing a simple but novel tool called prophet graph for efficient path pattern searching.
(3) Discovering a consistent but computationally cheap heuristic to ensure the optimality of A* algorithm.
(4) Extending the algorithm to support more flexible queries with groups of entities and logical statements in the path pattern.
(5) Conducting empirical analysis to show that PRO-HEAPS can answer relationship queries in real time while allowing the graph structure and weights to be dynamic.

The remainder of this article is organized in the following way. Section 2 formulates the problem and presents the challenges. Section 3 introduces our methodology in full detail, while Section 4 discusses the generalization of PRO-HEAPS to support groups of entities and logical statements. Experiments and analysis are conducted in Section 5, while related work is reviewed in Section 6. We demonstrate several interesting example use cases in Section 7 and make a conclusion of this study in Section 8.

2 PRELIMINARIES

2.1 Problem Formulation

We first provide some definitions that formalize the concept of a HIN, borrowing from previous works (Sun and Han 2013; Sun et al. 2011; Shi et al. 2015, 2017).

Definition 2.1 (Weighted Heterogeneous Information Network). A Weighted HIN 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 ($|\mathcal{L}| > 1$), while edges indicate relationships or interactions between entities, of which there are $|\mathcal{R}|$ types

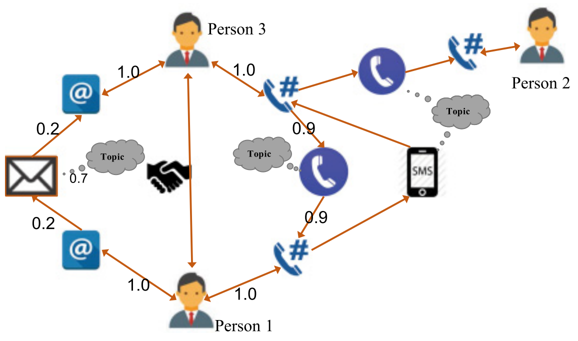


Fig. 1. Communication network: Different icons represent person, email address, email message, phone number, and so on[1].

($|\mathcal{R}| > 1$). 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 article, we use paths to explain the relationship between entities, following the idea from Fang et al. (2011), 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 2.2 (Path Pattern). Given a weighted HIN $G = (V, E, \Phi, \Psi, W)$, a path pattern $\mathcal{P}$, or metapath, is a sequence $L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} \cdot \xrightarrow{R_{\ell-1}} L_\ell$, 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 $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{\ell-1}} v_\ell$ in the graph G and a path pattern $\mathcal{P} = L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} \cdots \xrightarrow{R_{\ell-1}} L_\ell$ 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 $\mathcal{W}(p) = \sum_{i=0}^{\ell-1} W(e_i)$.

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 $Q = (v_s, v_t, \mathcal{P})$ 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 2.3 (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 2(a) is an example of a weighted HIN with different shapes representing different vertex labels. A query is provided at the bottom of Figure 2(a) 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. As it is out-of-scope in the current effort, we leave it as future work.

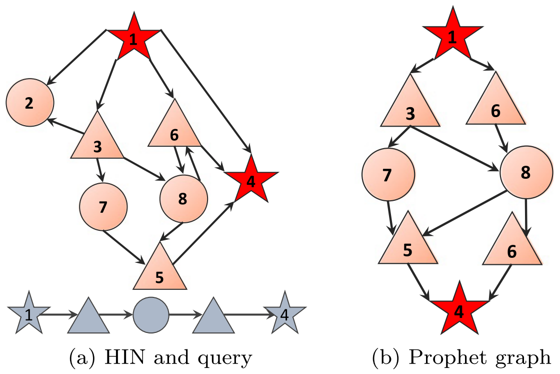


Fig. 2. Our running graph and query example.

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 (Meng et al. 2015).

Moreover, the edges in a HIN can be assigned weights based on interestingness metrics given by experts or users (Shi et al. 2015). 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., the probability of the text belonging to a given topic.

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 (Scott et al. 2006), 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. 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 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 as follows:

(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 2(a), 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 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 5.

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 ith level of G' contains the set of vertices V_i matching the ith 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 3.1 (Prophet Graph). Suppose we are given a weighted HIN $G = (V, E, \Phi, \Psi, W)$ and non-trivial query $Q = (v_s, v_t, \mathcal{P} = L_0 \xrightarrow{R_0} L_1 \xrightarrow{R_1} \cdots \xrightarrow{R_{\ell-1}} L_\ell)$. 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 2(b) shows the prophet graph for the graph and query in Figure 2(a).

We now describe an algorithm for computing the prophet graph G', given a HIN G and a query $\mathcal{Q}$, shown 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 (lines 1 and 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 ith 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 ith level should contain the same label matching ith 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 (lines 12 and 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: $\triangleright$ Forward BFS from $v_s$.
5: for $i = 1 \rightarrow mid$ 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: $\triangleright$ Backward BFS from $v_t$.
9: for $i = |\mathcal{P}| - 1 \rightarrow mid$ 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 non-trivial 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 ith 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 $(i-1)$-th level will accept the minimum value among all the heuristic values propagated into it (lines 3–6).

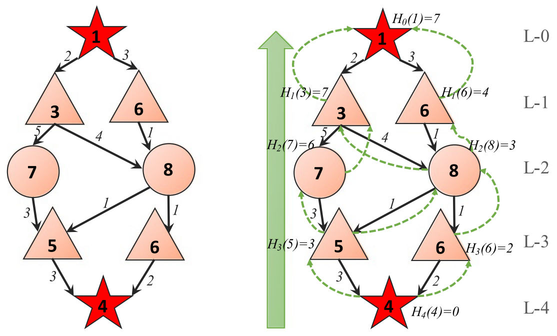
ALGORITHM 2: Calculate Heuristic Function.

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 3(a) shows the prophet graph with a specified weight function. Figure 3(b) 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 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,



(a) A prophet graph with weights

(b) Heuristics calculation

Fig. 3. Calculating the heuristics on our running example.

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 (lines 10–13). Otherwise, the path is stored (lines 8–9). 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.

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

Input: HIN G with weight function W, a query $Q = (v_s, v_t, \mathcal{P})$ 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: $result$ ← empty array for storing top-$k$ lightest paths.
4: $frontier$ ← 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:     $(path, key)$ ← $frontier$.pop().
8:     if $path$ reaches $v_t$ then
9:         Add $path$ to $result$; Go to line 6.
10:    vertex $u$ ← 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 3.2. *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 ith 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* (Pearl 1984) 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 3.2, 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.

We point out the time complexity of our algorithm depends on the weights distribution and the topological structure of the given HIN. In the worst case, our algorithm has a complexity of $O(b^{|\mathcal{P}|})$, where b is the branching factor (Russell and Norvig 1995) and is roughly equal to the averaged node degree. This is aligned with our analysis in Section 2.3 that our problem is NP-complete. Despite the worst-case exponential complexity, the usage of prophet graph and the consistent heuristic can drastically reduce the search space when working on real-world HINs and `PRO-HEAPS` is shown to significantly outperform other baselines in practice (see Section 5).

4 GENERALIZATION AND DISCUSSION

In this section, we describe several generalizations of `PRO-HEAPS` that make it more flexible and usable in a wider range of applications.

4.1 `PRO-HEAPS` in Dynamic HINs

While many efficient path-searching methods rely on building landmarks (or graph sketches) offline and can only work on static networks (Das Sarma et al. 2010; Akiba et al. 2013; Sommer 2014), we point out that `PRO-HEAPS` can be easily adapted to handle the situation where the HIN contains continuous structural changes and weight updates. This adaptation is possible since `PRO-HEAPS` is run on query arrival in a just-in-time fashion. As shown in the previous section, Algorithm 1, 2, and 3 are executed at query time and does not need to access the data before the arrival of queries. The change in graph structure can be easily accommodated by Algorithm 1, given its just-in-time nature. The variation of weights in the graph can be captured by Algorithm 2 on the fly.

Although simple, this extension enables many more applications, such as running `PRO-HEAPS` on online social networks and defining the weights based on recency.

4.2 `PRO-HEAPS` between Two Groups of Entities

While the original `PRO-HEAPS` only supports relationship analysis between two entities, here we generalize it to support two groups of entities. For consistency, we assume that the entities in each group are of the same type and the relationships between the two groups still need to follow the path pattern. We point out that this generalization has broader applications and is particularly useful in describing how different groups interact with each other. For example, we might be interested in finding out the common research topics between two groups of researchers from different research areas (e.g., database and machine learning). This can be achieved by the extended `PRO-HEAPS` with a properly formulated query $Q' = (G_s, G_t, \mathcal{P} = Author \rightarrow Paper \rightarrow Topic \rightarrow Paper \rightarrow Author)$, where G_s and G_s denote the two research groups.

We now introduce the procedure of running `PRO-HEAPS` in this new application. To execute a query with two groups of entities (G_s and G_t), we need to first slightly manipulate the HIN and the query. Specifically, we insert two pseudo-nodes, s' and t', into the original HIN given the query so that s' is connected and only connected to nodes in G_s and t' is connected and only connected to nodes in G_t. The edges associated with these new connections are called pseudo-edges and have weights of 0. We then insert s' and t' into the query as the new source node and target node, respectively. With this manipulation, we have the adapted HIN and query, which is in the same format of our original problem defined in Section 2. Given the adapted HIN and query, the rest of

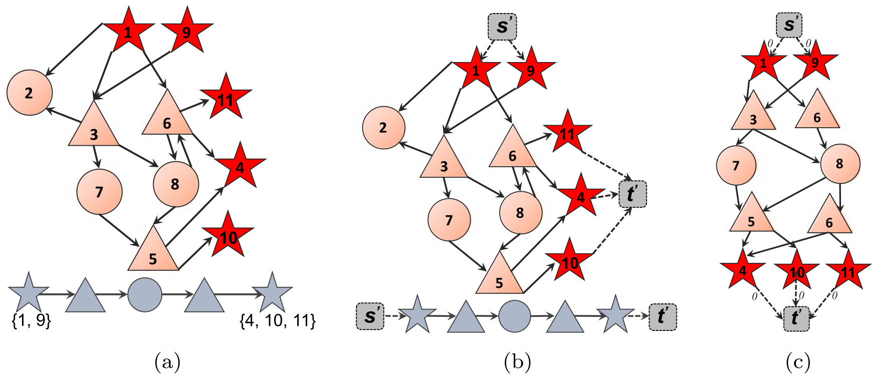


Fig. 4. PRO-HEAPS on the query with two groups of entities. (a) An example HIN with a generalized query, where {1, 9} and {4, 10,11} are two groups of entities of interest. (b) The adapted HIN and query with pseudo-nodes (s' and t') and pseudo-edges (denoted by dash lines). (c) The constructed prophet graph based on the adapted HIN and query.

procedure is the same as described in Algorithm 1, 2, and 3 and it will output the top-k shortest paths. At the last step, we need to post-process the paths by removing the pseudo-nodes in them. It is easy to show that by following this procedure we can obtain the correct answer, i.e., top-k lightest paths in the HIN following the pattern if any.

Figure 4 illustrates the procedure utilizing an example, where the query contains two groups of entities. Figure 4(a) shows the example HIN with a generalized query, where $G_s = \{1, 9\}$ and $G_t = \{4, 10, 11\}$ are two groups of entities of interest. Figure 4(b) presents the insertion of pseudo-nodes (s' and t') and pseudo-edges (denoted by dash lines) in the HIN and the query. Figure 4(c) is the prophet graph constructed following the Algorithm 1 under the adapted HIN and query. The remaining procedure will be identical to the approach described in Section 3 and is therefore not repeated here.

4.3 PRO-HEAPS with Logical Statements in the Query

Though the path pattern is powerful in conveying a users' preference, it might be restricted in some cases as it only allows one type of relationship strictly following the path pattern. To relax the constraint of the path pattern, we allow the specification of node and edge labels in the query path pattern to incorporate logical operators. We begin by describing support for the OR operator. In this case, each node/edge in the path pattern can correspond to multiple labels, which is specified by a logical statement. To make the logical statement more succinct, we also introduce other logical operators, AND, NOT and wild-card label "." that matches any label. This relaxation enables more interesting applications. 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.

To support the relaxed path pattern in the query, we adapt the construction of the prophet graph in Algorithm 1 and the modified algorithm is shown in Algorithm 4. While the original prophet graph contains one type of nodes in each level, we now allow multiple types of nodes in one level, i.e., each level corresponds to one logical statement. For this purpose, we parse each of the logical statements and evaluate whether the label of a node satisfies a logical statement or not

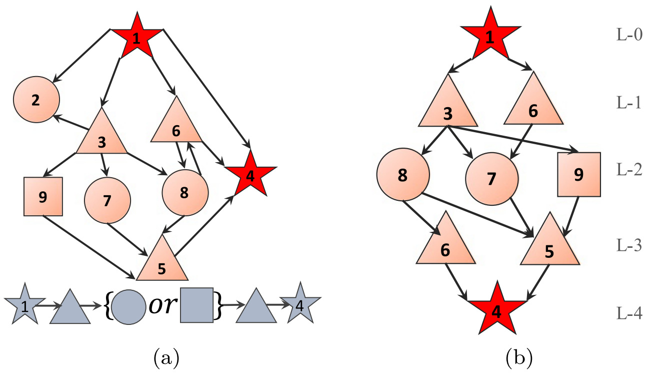


Fig. 5. PRO-HEAPS on the query with logical statements. (a) An example HIN and a query with logical statements. (b) The generalized prophet graph allowing nodes of different labels in the same level.

when constructing the prophet graph. This is done in line 6 and line 10 of Algorithm 4, where we examine whether a node satisfies the logical statements when running bi-directional BFS. Once we create the prophet graph satisfying the path pattern with logical statements, the calculation of heuristic functions and search for the paths are exactly the same as Algorithm 2 and 3.

Figure 5(a) shows an example of queries with logical statements and Figure 5(b) is the prophet graph given the extended query and the HIN. As shown in the Figure 5(b), L-2 contains nodes of two different labels (circle and square) corresponding the logical statement with OR in Figure 5(a). Note that the heuristics computation and path searching algorithm will be similar to the original algorithm.

ALGORITHM 4: Create Prophet Graph (Supporting Logical Statements).

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

```
1: Create an array of empty sets $levels[0 \ldots |\mathcal{P}|]$.
2: Set $levels[0] = \{v_s\}$ and $levels[|\mathcal{P}|] = \{v_t\}$.
3: Let $mid = \lfloor \frac{|\mathcal{P}|+1}{2} \rfloor$.
4: $\triangleright$ Forward BFS from $v_s$.
5: for $i = 1 \rightarrow mid$ do
6:     $levels[i] \leftarrow N_{out}(levels[i-1])$ satisfying the logical statements on $(i-1)$-th edge and $i$th vertex in $\mathcal{P}$.
7: $midLevel = levels[mid]$;   $levels[mid] = \emptyset$.
8: $\triangleright$ Backward BFS from $v_t$.
9: for $i = |\mathcal{P}| - 1 \rightarrow mid$ do
10:     $levels[i] \leftarrow N_{in}(levels[i+1])$ satisfying the logical statements on $i$th edge and $i$th vertex 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 $\mathcal{P}$.
```

Table 1. The Datasets Used

Dataset	# Vertices	# Edges	$\|\mathcal{L}\|$	$\|\mathcal{R}\|$
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

$|\mathcal{L}|$ is the number of different vertex labels and $|\mathcal{R}|$ is the number of different edge labels.

5 EXPERIMENTS AND ANALYSIS

In this section, we compare the performance of PRO-HEAPS with a series of baselines using three different real-world datasets. We also investigate its performance under the generalized applications with support of entities groups and logical statements. A series of analysis is conducted to study the properties of PRO-HEAPS.

5.1 Experimental Setup

5.1.1 Datasets Description. The three datasets used in our experiments are as follows[2]:

Enron[3]*:* 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 (Blei et al. 2003), and linked each Email message to the closest three topics.

DBLP[4]*:* A dataset of computer science bibliographic information. We created a HIN by categorizing the entities into vertex labels: author, paper, venue, and terminology.

Stack Overflow[5]*:* 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. Related entities are connected using labeled links, e.g., an answer is connected to its corresponding question.

Table 1 provides detailed information about each dataset. We defined weight functions over edges for these datasets as follows. For edges of action (e.g., *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 (e.g., *with topics* and *with email address*), we defined the weights as the probability of derivation (1.0 if it is certain).

5.1.2 Baselines. We implemented three groups of algorithms, containing 10 algorithms in total, to use as baseline comparisons to PRO-HEAPS.

Group 1: *DFM, BFM, Bidir-BFM, DijkstraM*. This group of methods contains four basic approaches. Section 2.3 describes DFM, BFM and DijkstraM algorithm. The other one, Bidirectional BFM (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 (Akiba et al. 2013) 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

[2]Datasets are available at http://jiongqianliang.com/PRO-HEAPS/.
[3] https://www.cs.cmu.edu/~./enron/.
[4]http://dblp.uni-trier.de/xml/.
[5] https://archive.org/details/stackexchange.

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. (2013).

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.

5.1.3 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 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_s (or v_t) to an intermediate vertex that follows the appropriate pattern.

5.1.4 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 $Q' = (v'_s, v'_t, \mathcal{P}')$, 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.

5.2 Performance of `PRO-HEAPS`

Figure 6 shows the performance of `PRO-HEAPS` compared to other baselines. Figure 7 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 7, 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 7(c). 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.

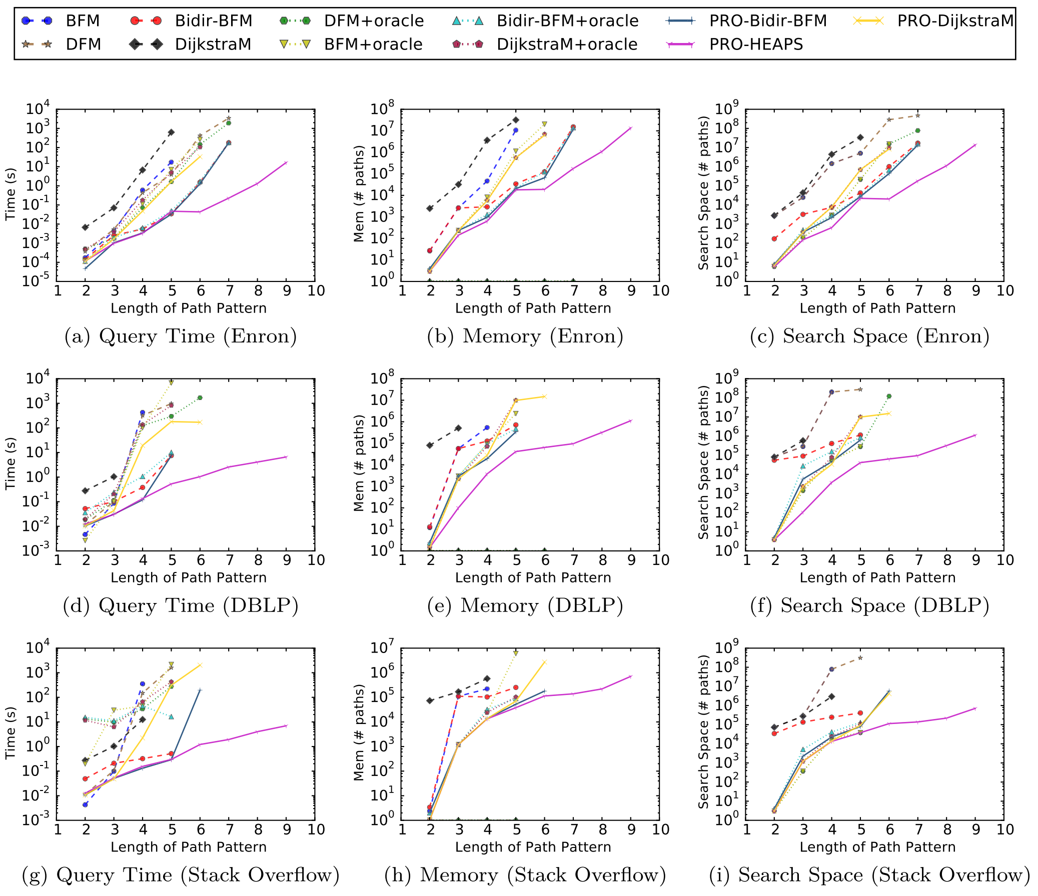


Fig. 6. Performance comparisons of different algorithms. Top to bottom: The rows correspond to the results for the Enron, DBLP, and Stack Overflow datasets, respectively. Left to right: The columns show the average time for executing a query, the average number of paths stored in memory, and the average 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 48GB of RAM.

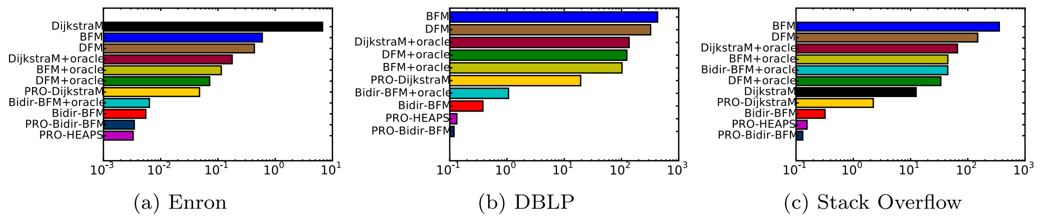


Fig. 7. Timing comparisons for path patterns of length 4 (in seconds). Methods are ranked in decreasing order of query time. Missing bars (e.g., DijkstraM in DBLP) are due to not finishing within time/memory limit.

Table 2. PRO-HEAPS Compared with the Fastest Baselines

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(-)

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.

(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. Figures 6(b)–6(c), 6(e)–6(f), and 6(h)–6(i)).

(3) From Figure 6(a), (d), and (g) 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 6, 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 6. Owing to this heuristic, we found that PRO-HEAPS is up to 1,000 times faster than PRO-DijkstraM. Moreover, this heuristic is a computationally inexpensive primitive (cf. Algorithm 2).

5.3 Performance of Generalized PRO-HEAPS

As we discussed in Section 4, PRO-HEAPS can be generalized to support relationship analysis among groups of entities, while allowing logical statements in the path pattern. In this part, we demonstrate the performance of the generalized PRO-HEAPS through extensive experiments. As a point of comparison, we adapt the previous baselines in Section 5.1.2. To allow path pattern with logical

Table 3. PRO-HEAPS Compared with the Fastest Baselines on Quires Supporting *two Groups of Entities*

Dataset	Speedup w.r.t. best baseline (Two groups)		
	Length $|\mathcal{P}| = 5$	Length $|\mathcal{P}| = 6$	Length $|\mathcal{P}| = 7$
Enron	4.3X(PRO-Bidir-BFM)	103.4X(Bidir-BFM)	756.8X(Bidir-BFM)
DBLP	28.8X(PRO-Dijkstra)	513.2X(PRO-Dijkstra)	>274.5X(-)
Stack Overflow	5.3X(PRO-Bidir-BFM)	362.5X(PRO-Dijkstra)	>322.2X(-)

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.

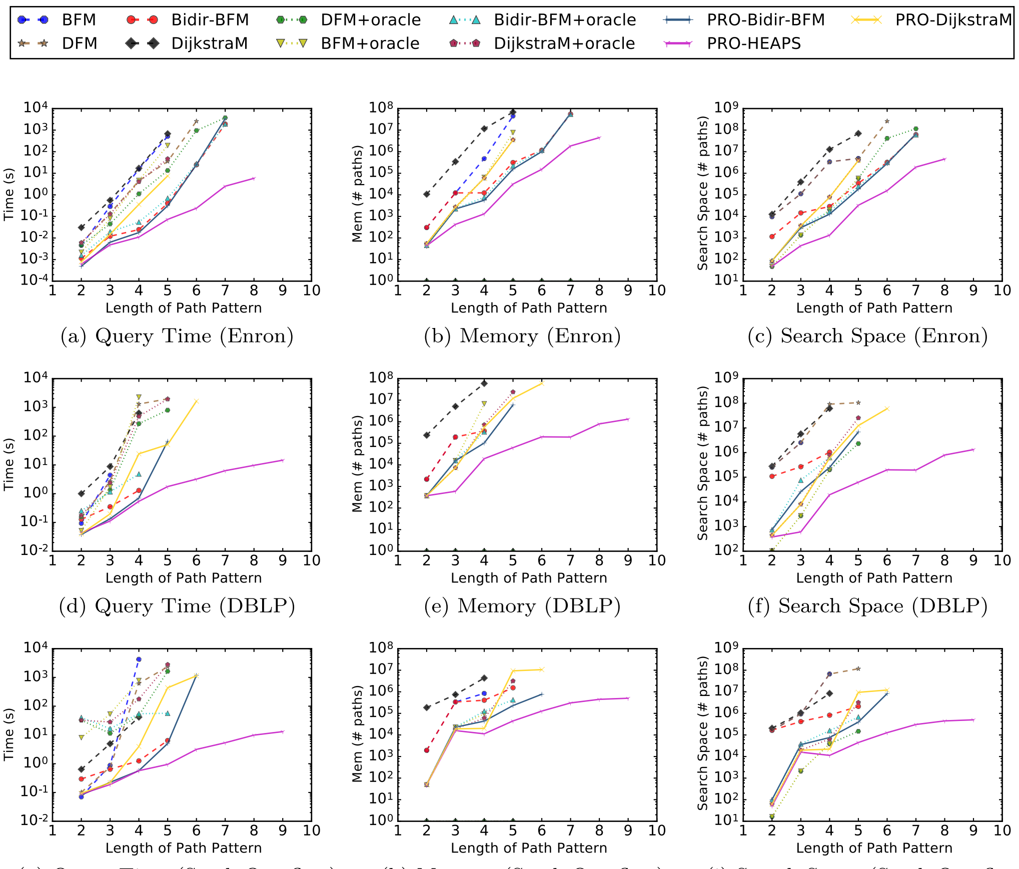
statements, we replaced the steps of label matching in the baselines with the evaluation of logical statements. To support query with two groups of entities, we added pseudo-nodes and pseudo-edges to the HIN for all the baselines. Note that the distance oracle in the baselines of Group 2 cannot directly work on pseudo-nodes since distance oracle is constructed before the arrival of queries, and does not incorporate pseudo-nodes. To address this, we perform distance queries between all pairs of nodes in $G_s \times G_t$.

To evaluate the generalized PRO-HEAPS, we extend the randomly generated queries in the previous experiment (see Section 5.1.4) to incorporate two groups of entities. For a given query $Q' = (v'_s, v'_t, \mathcal{P}')$, we expand v'_s and v'_t to be group G'_s and G'_t, respectively. The expansion procedures of the vertex v'_s to a group G'_s is as follows. We first generate a random number $|G|$ from the uniform distribution $U(1,4)$ as the size of the group. Note we limit the size of the group to be at most four in the experiment so that the baseline methods can finish a reasonable number of queries in the allocated time. In practice, it can be much larger. We then start BFS from v'_s and store all the visited vertices with the same label as v_s into a set S. We terminate the BFS when the size of S is not less than $|G|$, at which point we randomly sample $|G|$ different vertices from S and make them as G'_s. The expansion of v'_t to G'_t follows similar steps.

Likewise, we extend each of the previous queries $Q' = (v'_s, v'_t, \mathcal{P}')$ to contain logical statements. We keep v'_s and v'_t as the source and target vertex and keep the length of path pattern as $|\mathcal{P}|$. In the extension, we retain the first and last vertex label of $\mathcal{P}$ while replacing each of the $|\mathcal{P}| - 2$ intermediate labels with a logical statement. Note that only a subset of labels is valid for a node in $\mathcal{P}$ since our HINs carry semantic meaning. To find out the legitimate subset of labels for each node in $\mathcal{P}$, we perform bi-directional BFS between v'_s and v'_t at the label level (i.e., only the labels of nodes are stored). The legitimate labels for a node are then concatenated by the OR operator to generate a logical statement in place of the original vertex label. We do not include AND, NOT, and "." in the logical statements here as they are only used for the purpose of succinctness (see Section 4.3).

Besides the generated queries, the rest of experimental setting, including datasets, evaluation metrics, and computation resource, is the same as the ones described in Section 5.1. Figures 8 and 9 show the performance comparisons of different algorithms for these two types of generalized queries. Tables 3 and 4 highlight the speedups of PRO-HEAPS compared to the two best baselines.

We can observe that the performance gain of PRO-HEAPS still holds when supporting two groups of entities and logical statements. In fact, the speedup increases for most of the cases compared to the previous results in Figure 6 and Table 2. In particular, the performance gap between PRO-HEAPS and other baselines is much larger when executing queries with two groups of entities. Even when the length of pattern path is as small as 5, the speedup is 28.8 in Table 3 (much larger than 14.1 in Table 2). From the standard PRO-HEAPS to the one supporting two groups of entities, the speedup of the algorithm increases by at least 100% (compare Table 3 with Table 2). For the performance of



(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)

Fig. 8. Performance comparisons for queries containing two groups of entities (G_s and G_t). The number of entities in each group is a random number from 1 to 4.

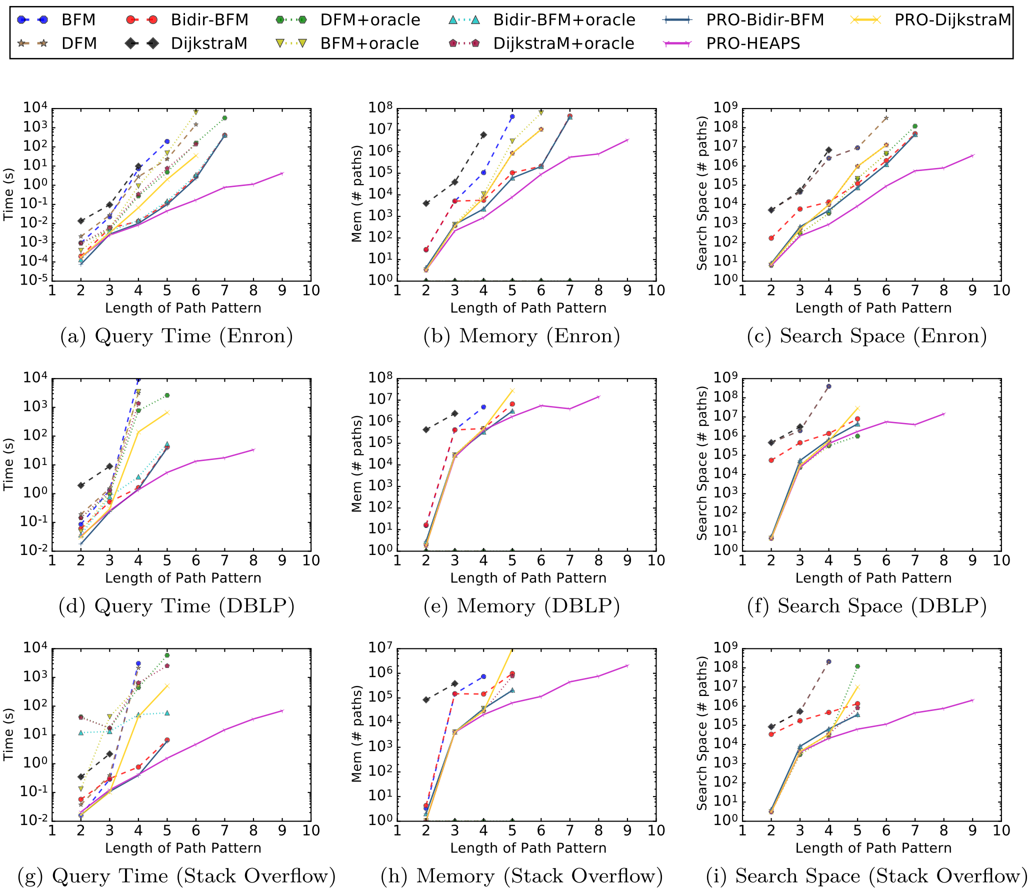
Table 4. PRO-HEAPS Compared with the Fastest Baselines on Queries Supporting *Logical Statements*

Dataset	Speedup w.r.t. best baseline (Logical statements)		
	Length $\lvert\mathcal{P}\rvert = 5$	Length $\lvert\mathcal{P}\rvert = 6$	Length $\lvert\mathcal{P}\rvert = 7$
Enron	2.2X(PRO-Bidir-BFM)	13.6X(PRO-Bidir-BFM)	508.6X(PRO-Bidir-BFM)
DBLP	7.1X(PRO-Bidir-BFM)	>128.2X(-)	>97.5X(-)
Stack Overflow	3.9X(PRO-Bidir-BFM)	>365.3X(-)	>114.4X(-)

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.

PRO-HEAPS supporting logical statements, there are also improvements on speedup although it is less obvious.

The improvements can be explained by the fact that the generalized queries lead to an increase in the size of search space and the advantage of prophet graph at reducing search space becomes more evident (For example, see the difference among Figures 6(f), 8(f), and 9(f)). These observations again validate the power of PRO-HEAPS and indicate the proposed algorithm is particularly useful for dealing with complex relationship patterns.



(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)

Fig. 9. Performance comparisons of different algorithms on queries with logical statements.

5.4 Analysis of PRO-HEAPS

We now study how the performance of PRO-HEAPS varies with respect to the various attributes of the input HINs. We consider the following attributes for this study:

(1) 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.
(2) 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.
(3) 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.
(4) Degree distribution of HIN: We studied the effect of degree distribution of the input HIN on the relative performance of PRO-HEAPS and the baselines. Specifically, we sought to evaluate the performance of PRO-HEAPS on the harder case of HIN with uniformly random degree distribution (where the search space is considerably bigger).

(5) Clustering coefficient of HIN: We also investigated the role of clustering coefficient of the input HIN on the performance of PRO-HEAPS to check whether or not the running time of PRO-HEAPS is robust to more structural changes in the HIN.

To conduct experiments for analyzing our algorithm, we used the Enron dataset and manipulated it with the following procedures.[6]

(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.
(4) We rewired the edges in the HIN to ensure it follows a Pareto degree distribution with different shape parameters α. Specifically, the probability of a vertex with degree x is $Pr(x) = \frac{\alpha m^\alpha}{x^{\alpha+1}}$, where m is a scale parameter to ensure the total degree of all vertices is fixed. We varied α from 1.0 to 3.0 while maintaining the same number of edges. A larger α means the degree distribution is more skewed.
(5) We adopted algorithm of Holme and Kim (2002) for manipulating graphs with Pareto degree distribution and approximate average clustering coefficient. We retained the same number of edges and varied the clustering coefficient from 0.02 to 0.14.

For each of these variants of the Enron graph, we ran all the previously generated queries. The results can be seen in Figures 10 and 11. Figure 10(a)–(c) 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 10(a)–(c)). However, overall we observe that the performance of PRO-HEAPS is relatively consistent across different weight distributions.

Figure 10(d)–(f) shows 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 (Figure 10(e) and (f)).

In addition, Figure 10(g)–(i) presents 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.

[6]We also ran the same experiments on DBLP and SOF datasets, where we observed similar results.

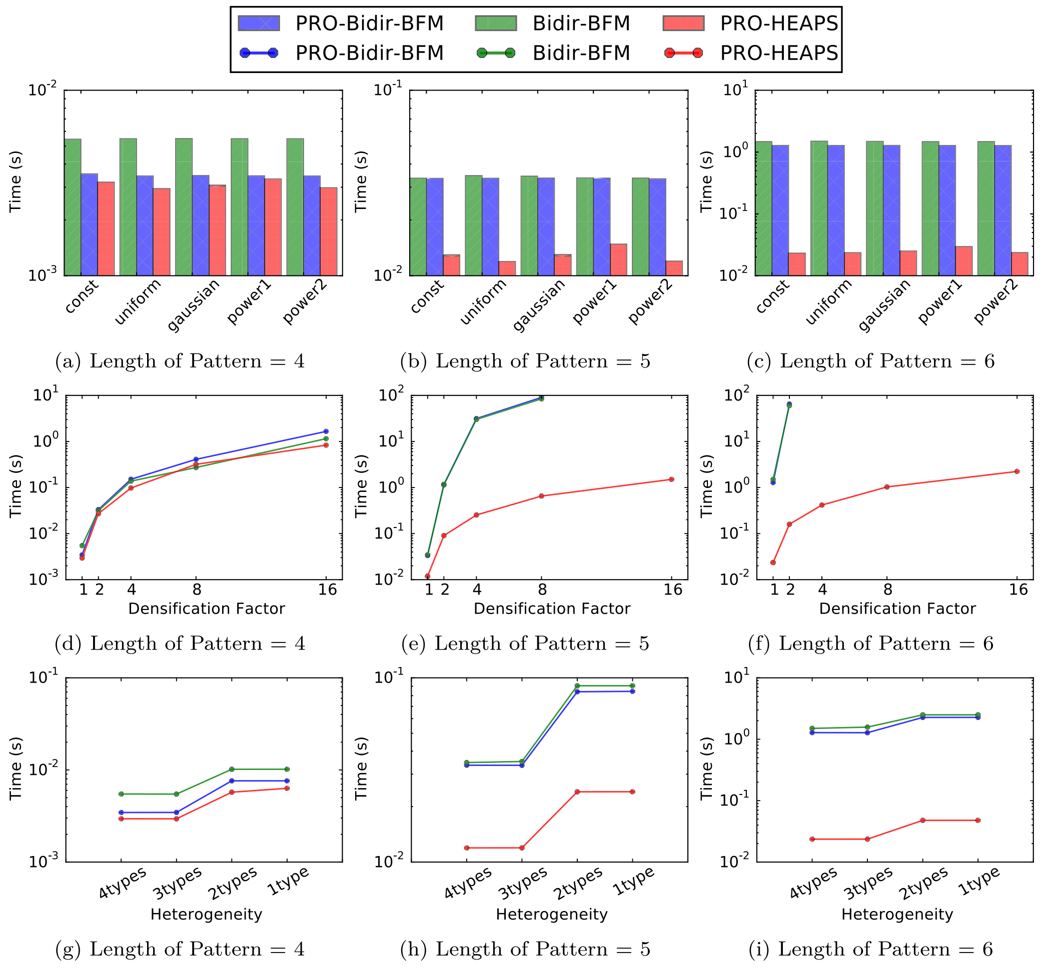


Fig. 10. Effect of weight distribution, graph density and heterogeneity. (a)–(c) show different weight distributions on the x-axis. On the x-axis of (d)–(f), the densification factor is the ratio of # edges in the graph to the # edges in the original graph. For (g)–(i), the x-axis indicates the total number of different vertex labels in the graph.

Figure 11(a)–(c) demonstrates the performance of PRO-HEAPS under different degree distribution settings. Our first observation is that as the degree distribution gets skewed, the running times of all approaches become smaller. A likely cause for this observation is that with the skew, the search space for prioritized k lightest paths between two randomly chosen vertices is smaller on average. On the other hand, graphs with less skewed degree distribution (e.g., $\alpha = 1$) have a larger search space and hence, the advantages of pruning are also considerably higher. As a result, PRO-HEAPS is significantly faster for such graphs. For instance, on graphs with $\alpha = 1$ and for patterns of length 7 and 9, the other baselines do not even finish within the allocated time.

Finally, Figure 11(d) and (e) demonstrates the performance of different methods on HINS with different clustering coefficients. As shown in the figures, the performance of PRO-HEAPS is robust with respect to this fundamental structural attribute.

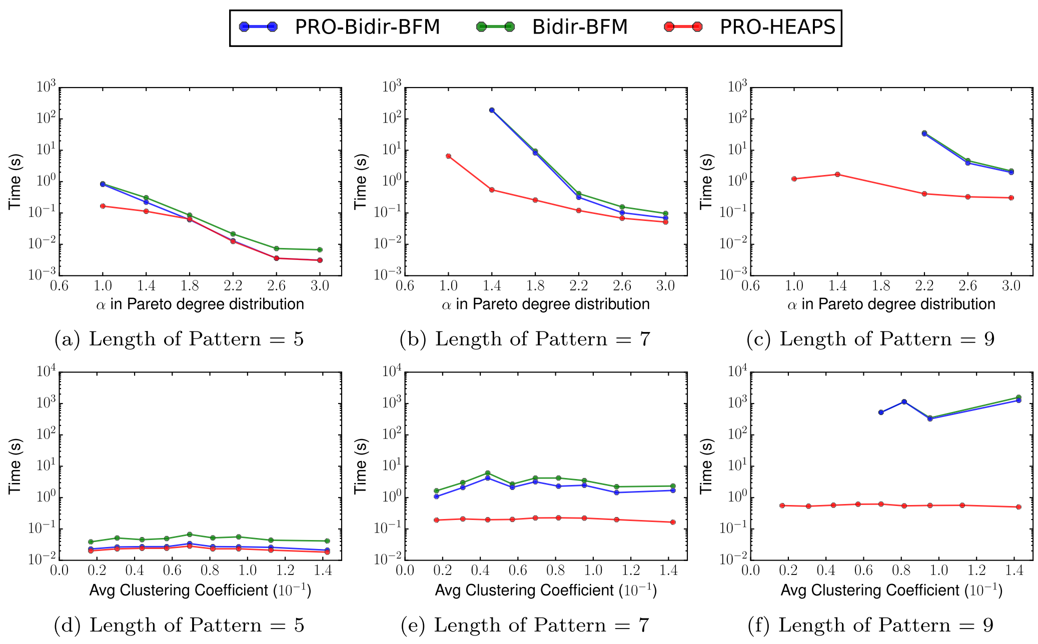


Fig. 11. Effect of degree distribution and average clustering coefficient. (a)–(c) show different α values for Pareto distributions on the x-axis. The higher α is, the more skewed is the degree distribution. The x axes of (d)–(f) show the average clustering coefficient of the graph. For both two cases, we adapted the Enron dataset by maintaining the same number of edges and nodes while rewinding the edges to vary the degree distribution and clustering coefficient. Missing parts of the curve indicate that the method cannot finish within the allocated time.

6 EXAMPLE USE CASES

In this section, we show how an analyst can use our solution, PRO-HEAPS, to study (e.g., to verify the expertise of people) publicly available online data such as Stack Overflow and DBLP. In particular, we present use cases to show how one can formulate queries to discover interesting relationships and how the dynamics of the network as well as the users' interest will change the outputted results.

6.1 Relations between Users on Dynamic 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 Jan 2014, PRO-HEAPS finds a question on MySQL answered by both in December 2013[7] as the best match. For a later query in June 2014, PRO-HEAPS finds a relationship instance where the two users answered a question on Oracle database[8] in April 2014. This example demonstrates the efficacy of path patterns and the effectiveness of PRO-HEAPS in understanding the intent of an analyst and returning the relevant relationship

[7]http://stackoverflow.com/questions/20828174.
[8]http://stackoverflow.com/questions/23298310.

instance. It also shows how PRO-HEAPS can effectively deal with dynamic networks by considering the changes of structure and weights.

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, 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.[9] This indicates that their interests are still on the same general subject.

6.2 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 $\mathcal{P}'$ = (Jiawei Han, Ion Stoica, $author \rightarrow paper \rightarrow venue \rightarrow paper \rightarrow author$) 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[10] and Ion Stoica has a paper in ICDCS'08.[11] 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[12]. Again, this illustrates the efficacy 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.

7 RELATED WORK

There is a considerable body of related work on graph mining techniques, spreading across many domains—Web Science, Social Network, Semantics, Bioinformatics, Databases, Algorithms, Distributed Systems, High-Performance Computing, and so on. We briefly review the papers closest to our work.

7.1 *k*-Shortest Path and Minimum-Weight Path

The k-shortest path problem seeks to find the top-k shortest path between two vertices in a graph (Yen 1971; Shih and Parthasarathy 2012). There are two different variants—loopy and loop-less—depending on whether the shortest paths are allowed to have cycles or not. Our research is closer to the loop-less variant, which is more computationally expensive and cannot be solved in real-time on large graphs. The complexity of the best algorithm for this problem on directed graphs is $O(k \cdot n \cdot SSSP(n, m))$ (Yen 1971) and for undirected graphs, it is $O(k \cdot SSSP(n, m))$ (Katoh et al. 1982), where $SSSP(n, m)$ is the complexity of a single-source shortest path variant ($O(m + n \log n)$ if Dijkstra's algorithm with Fibonacci heap is used). Even the most efficient implementation (Hadjiconstantinou and Christofides 1999) of the sequential undirected k-shortest path algorithm is not considered fast enough for many applications, so researchers have looked into parallelism (Ullrich and Forst 2009) and approximation (Hershberger et al. 2007; Shih and Parthasarathy 2012).

[9]http://stackoverflow.com/questions/18744391, http://stackoverflow.com/questions/17299581.
[10]Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems.
[11]Adaptive distributed time-slot based scheduling for fairness in multi-hop wireless networks.
[12]Jiawei Han published a paper called *mining frequent patterns without candidate generation* in SIGMOD'00 and Ion Stoica published a paper called *Declarative networking: language, execution and optimization* in SIGMOD'06.

Another related problem, the minimum-weight path problem, aims to find the minimum weight simple path of a user-defined length. A powerful randomized algorithm, called color coding, is proposed to effectively solve this problem (Alon et al. 1995). It also has been widely used to detect signaling pathways in protein interaction networks (Scott et al. 2006).

In contrast to these techniques that primarily focus on homogeneous graphs, we work on heterogeneous graphs with notions of labels on nodes and edges. While the existing methods can not be directly applied to our problem, 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.

7.2 Generalized Pattern Matching and Path Searching in Graphs

The classic graph pattern match problem, known as sub-graph isomorphism, aims to find matches for a given pattern in a graph database. It has been proven to be NP-complete (Garey and Johnson 2002). A basic approach is to enumerate all possible mappings of the pattern graph to the data graph and to check whether each mapping is legitimate. It is usually done using *state-space representation tree*, where each node represents a pair of matched nodes and a path from the root to the leaf node indicates a mapping. To reduce the search-space, Ullmann proposed a refinement procedure to prune unpromising sub-trees (Ullmann 1976). Pruning strategies are based on vertex degree, one-to-one mapping of vertices and adjacency checking. Due to the exponential time complexity of these exact algorithms, inexact algorithms are also studied. One representative algorithm is SUBDUE (Cook and Holder 1994), which performs graph pattern matching in the process of graph mining. The matching algorithm also constructs state-space representation tree, but allowing inexact matching by introducing the cost of the mismatch. An algorithm based on branch-and-bound search is proposed to find the least cost matching subgraph. For a detailed introduction of algorithms for this problem, refer to the survey (Shasha et al. 2002).

Beyond focusing just on structure, there is another category of works on semantic matching, where pattern graph and data graph contain labels on nodes and/or edges. GraphGrep(Giugno and Shasha 2002) 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 mappings. Apparently, the complexity of this algorithm is exponential and only works for small graphs. Even approximate algorithms (Aleman-Meza et al. 2005; Coffman et al. 2004; Wolverton et al. 2003; Khan et al. 2011, 2013) and distributed variants (Bai et al. 2014) often do not scale to large graphs.

While the above semantic graph pattern matching algorithms focus on general pattern matching in the 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.

7.3 Metapaths in Heterogeneous Information Networks Mining

Metapath, essentially a labeled path within a HIN (Sun and Han 2013), has been shown to be a powerful tool for mining HIN. It has found significant use as a mechanism to quantify the similarity between a pair of nodes within HIN (Sun et al. 2011, 2012; Lao and Cohen 2010; Shi et al. 2014). Sun et al. (2011) utilize metapath for similarity search, where the goal is to find the target nodes with high similarity given a source node and a metapath. The similarity is defined by the number of paths following the metapath and matrix multiplication is used for this purpose. Follow-up work (Sun et al. 2012) tries to predict the relationship in HINs by using metapath-based topology as parts of input features for the generalized linear model. Lao and Cohen (2010) use path constraint random walk to quantify the similarity of nodes, while (Shi et al. 2014) define a symmetric relevance measurement based on pair-wise random walks. Weighted HINs are studied for personalized recommendation by considering the values on the links and conducting semantic path matching (Shi et al. 2015). Refer to the survey papers (Sun and Han 2013; Shi et al. 2017) for more details about using metapath for HINs mining.

A more recent work (Meng et al. 2015) studies how to discover the most relevant metapaths given pairs of related nodes, where the authors define the problem in the supervised learning context and leverage a greedy algorithm similar to feature selection to find out the most potential metapaths. They design a tree-based data structure to efficiently generate metapath that maximize the gain of adding it as a new feature to the supervised learning.

These works are distinct from ours with respect to applications (e.g., similarity search, clustering, link prediction, and pattern learning) and the fact that they do not leverage users' preference and are limited to unweighted graphs.

7.4 Relationship Mining in Graphs

Keyword searching in relational databases is an important problem in web-based search where vertex represent tuples and edges represent the foreign-key relationships. Solutions seek tree- or subgraph- patterns to explain the relationships among a given set of keywords. BANKS (Bhalotia et al. 2002) designs a heuristic algorithm to compute the Steiner trees covering the given keywords and runs Dijkstra algorithm concurrently from each keyword node. It stores the common vertices, serving as the roots of subtrees, and eventually merge paths into a tree. Follow-up work (Kacholia et al. 2005) conducts the similar procedures but with more sophisticated strategies on expanding the nodes during candidate path generation and enumeration.

Related to the above is the notion of center-piece subgraphs where the authors (Faloutsos et al. 2004; Tong and Faloutsos 2006) 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. (2011) work on explaining the relationship of two individual entities in heterogeneous graphs. Their method runs the bidirectional single source shortest path algorithm to obtain paths and union them to enumerates subgraph patterns. Patterns are then ranked based on the interestingness function defined on the number of paths and rarity of patterns.

To sum up, our work is different from the existing works in the following perspectives: (1) We work on weighted heterogeneous graphs where graph structure and weights of edges can be dynamic. (2) The path pattern is provided as a query to provide the preference of relationship. (3) We are only interested in top-k lightest paths that follow the path pattern.

8 CONCLUSIONS

In this article, we solve the problem of prioritized relationship analysis 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.

REFERENCES

Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 349–360.

Boanerges Aleman-Meza, Christian Halaschek-Wiener, Satya Sanket Sahoo, Amit Sheth, and I. Budak Arpinar. 2005. Template based semantic similarity for security applications. In *Proceedings of the Intelligence and Security Informatics*. Springer, 621–622.

Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *Journal of the ACM* 42, 4 (1995), 844–856.

Yiyuan Bai, Chaokun Wang, Xiang Ying, Meng Wang, and Yunqing Gong. 2014. Path pattern query processing on large graphs. In *Proceedings of the IEEE 4th International Conference on Big Data and Cloud Computing (BdCloud).* IEEE.

Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. 2002. Keyword searching and browsing in databases using BANKS. In *ICDE*. IEEE.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.

Thayne Coffman, Seth Greenblatt, and Sherry Marcus. 2004. Graph-based technologies for intelligence analysis. *Communications of the ACM* 47, 3 (2004), 45–47.

Diane J. Cook and Lawrence B. Holder. 1994. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research* 1 (1994), 231–255.

Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. 2010. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*. ACM, 401–410.

Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. 2004. Fast discovery of connection subgraphs. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 118–127.

Lujun Fang, Anish Das Sarma, Cong Yu, and Philip Bohannon. 2011. Rex: Explaining relationships between entity pairs. *Proceedings of the VLDB Endowment* 5, 3 (2011), 241–252.

Michael R. Garey and David S. Johnson. 2002. *Computers and Intractability*, Vol. 29. WH Freeman.

Rosalba Giugno and Dennis Shasha. 2002. Graphgrep: A fast and universal method for querying graphs. In *Proceedings of 16th International Conference on Pattern Recognition*, Vol. 2. IEEE, 112–115.

Eleni Hadjiconstantinou and Nicos Christofides. 1999. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks* 34.2 (1999), 88–101.

John Hershberger, Matthew Maxel, and Subhash Suri. 2007. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms* 3, 4 (2007), 45.

Petter Holme and Beom Jun Kim. 2002. Growing scale-free networks with tunable clustering. *Physical Review E* 65, 2 (2002), 026107.

Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. 2005. Bidirectional expansion for keyword search on graph databases. *Proceedings of the VLDB Endowment* (2005), 505–516.

Naoki Katoh, Ibaraki Toshihide, and Mine Hisashi. 1982. An efficient algorithm for k shortest simple paths. *Networks* 12, 4 (1982), 411–427.

Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. 2011. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data.* ACM, 901–912.

Arijit Khan, Yinghui Wu, Charu C. Aggarwal, and Xifeng Yan. 2013. Nema: Fast graph search with label similarity. *Proceedings of the VLDB Endowment* 6, 3 (2013), 181–192.

Ni Lao and William W. Cohen. 2010. Relational retrieval using a combination of path-constrained random walks. *Machine Learning* 81, 1 (2010), 53–67.

Jiongqian Liang, Deepak Ajwani, Patrick K. Nicholson, Alessandra Sala, and Srinivasan Parthasarathy. 2016. What links alice and bob? matching and ranking semantic patterns in heterogeneous networks. In *Proceedings of the 25th International Conference on World Wide Web.* ACM, 879–889.

Jiongqian Liang, Peter Jacobs, Jiankai Sun, and Srinivasan Parthasarathy. 2018. SEANO: Semi-supervised embedding in attributed networks with outliers. In *Proceedings of the 2018 SIAM International Conference on Data Mining*. SIAM.

Changping Meng, Reynold Cheng, Silviu Maniu, Pierre Senellart, and Wangda Zhang. 2015. Discovering meta-paths in large heterogeneous information networks. In *Proceedings of the 24th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, 754–764.

Judea Pearl. 1984. Heuristics: Intelligent search strategies for computer problem solving. Addison-Wesley (1984).

Stuart Russell and Peter Norvig. 1995. Artificial Intelligence: A modern approach. *Pearson Education* 25 (1995), 97–104.

Jacob Scott, Trey Ideker, Richard M. Karp, and Roded Sharan. 2006. Efficient algorithms for detecting signaling pathways in protein interaction networks. *Journal of Computational Biology* 13, 2 (2006), 133–144.

Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. 2002. Algorithmics and applications of tree and graph searching. In *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems.* ACM, 39–52.

Chuan Shi, Xiangnan Kong, Yue Huang, S. Yu Philip, and Bin Wu. 2014. Hetesim: A general framework for relevance measure in heterogeneous networks. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2479–2492.

Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S. Yu Philip. 2017. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering* 29, 1 (2017), 17–37.

Chuan Shi, Zhiqiang Zhang, Ping Luo, Philip S. Yu, Yading Yue, and Bin Wu. 2015. Semantic path based personalized recommendation on weighted heterogeneous information networks. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 453–462.

Yu-Keng Shih and Srinivasan Parthasarathy. 2012. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics* 28, 12 (2012), i49–i58.

Christian Sommer. 2014. Shortest-path queries in static networks. *ACM Computing Surveys* 46, 4 (2014), 45.

Yizhou Sun and Jiawei Han. 2013. Mining heterogeneous information networks: A structural analysis approach. *ACM SIGKDD Explorations Newsletter* 14, 2 (2013), 20–28.

Yizhou Sun, Jiawei Han, Charu C. Aggarwal, and Nitesh V. Chawla. 2012. When will it happen? Relationship prediction in heterogeneous information networks. In *Proceedings of the 5th ACM international conference on Web search and data mining*. ACM, 663–672.

Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment* 4, 11 (2011), 992–1003.

Hanghang Tong and Christos Faloutsos. 2006. Center-piece subgraphs: Problem definition and fast solutions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 404–413.

Julian R. Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM* 23, 1 (1976), 31–42.

Alexander Ullrich and Christian V. Forst. 2009. k-PathA: K-shortest path algorithm. In *Proceedings of the IEEE International Workshop on High Performance Computational Systems Biology*.

Michael Wolverton, Pauline Berry, Ian W. Harrison, John D. Lowrance, David N. Morley, Andres C. Rodriguez, Enrique H. Ruspini, and Jerome Thomere. 2003. LAW: A workbench for approximate pattern matching in relational data. In *Proceedings of the 5th Innovative Applications of Artificial Intelligence Conference*, Vol. 3. 143–150.

Jin Y. Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17, 11 (1971), 712–716.

Received January 2017; revised September 2017; accepted October 2017