PPKWS: An Efficient Framework for Keyword Search on Public-Private Networks

Jiaxin Jiang*, Xin Huang*, Byron Choi*, Jianliang Xu*, Sourav S Bhowmick†, Lyu Xu*
*Department of Computer Science, Hong Kong Baptist University, Hong Kong
† School of Computer Science and Engineering, Nanyang Technological University, Singapore
*{jxjian,xinhuang,bchoi,xujl,cslyuxu}@comp.hkbu.edu.hk, †assourav@ntu.edu.sg

***Abstract*—Due to the unstructuredness and the lack of schemas of graphs, such as knowledge graphs, social networks and RDF graphs, keyword search has been proposed for querying such graphs/networks. In many applications (*e.g.*, social networks), users may prefer to hide parts or all of her/his data graphs (*e.g.*, private friendships) from the public. This leads to a recent graph model, namely the *public-private network* model, in which each user has his/her own network. While there have been studies on public-private network analysis, *keyword search on public-private networks* has not yet been studied. For example, query answers on private networks and on a combination of private and public networks can be different. In this paper, we propose a new keyword search framework, called *public-private keyword search* (PPKWS). PPKWS consists of three major steps: *partial evaluation*, *answer refinement*, and *answer completion*. Since there have been plenty of keyword search semantics, we select three representative ones and show that they can be implemented on the model with minor modifications. We propose indexes and optimizations for PPKWS. We have verified through experiments that, on average, the algorithms implemented on top of PPKWS run 113 times faster than the original algorithms directly running on the public network attached to the private network for retrieving answers that spans through them.**

I. Introduction

Knowledge graphs, social networks and RDF graphs have a wide variety of emerging applications, including semantic query processing [24], information summarization [21], community search [9], collaboration and activities organization [20] and user-friendly query facilities [22]. Such graphs often lack useful schema information for users to formulate their queries. *Keyword search* is a fundamental query paradigm that makes querying such data easy. In a nutshell, a user essentially specifies a set of keywords Q on a data graph G as his/her query. Depending on the search semantics, the answer to Q can be subgraphs that either contain the keywords and/or are top-k subgraphs. For instance, Google's knowledge graph search API[1] facilitates users in finding answers from their knowledge database, and returns the query answers in the form of subtrees. The answers (a) make it easy for users to explore some additional relevant keywords and (b) indicate the relationships of the query keywords.

As reported in a recent study [7], users may have private graphs such as private knowledge bases or social networks. For instance, 52.6% of 1.4 million New York City Facebook users hide their friends lists. Such behavior naturally leads to a new graph model, called *the public-private graph model* [3], [1], and [17]. It consists of a public graph and many private graphs, where the private ones are only accessible to their owners. Generally, each user has his/her own combined graph. This model warrants revisiting the research on keyword search for two reasons. Firstly, the combined graphs can be large. For instance, the latest version of one semantic knowledge base, YAGO, contains 4.5 million entities and 24 million facts. It is not practical to directly apply the existing indexing techniques (*e.g.*, [14] and [10]) to each combined graph for each user. Secondly, there are already several semantics for keyword search. It is desirable to have a unified framework that optimizes their query performance.

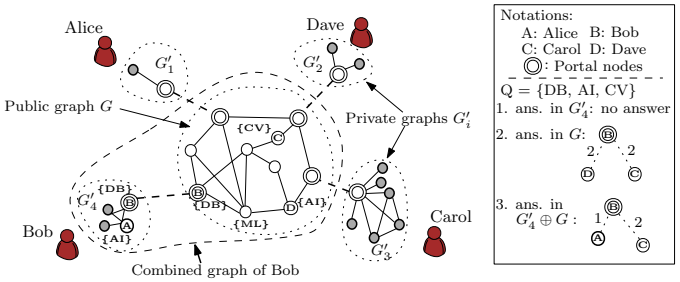


Fig. 1: An example of the public-private graph model (G is a public graph, and G'_1, G'_2, G'_3 and G'_4 are private graphs)

Example I.1. *Consider a public collaboration network G in Fig. 1 (e.g., [11]), where a node is an academic with its labels representing keywords of his/her research interests and an edge is a collaboration in research papers. A professor, Bob, has a private collaboration network G'_4 as shown in Fig. 1 (e.g., for grants, conferences and company organizations). G and G'_4 are visible to Bob. G'_1, G'_2 and G'_3 are not, since they are, respectively, owned by "Alice", "Dave" and "Carol". G and G'_4 are combined by some common nodes (a.k.a. portal nodes, shown as concentric circles in Fig. 1). When Bob proposes a new interdisciplinary project "DB-AI-CV", he first seeks out his close collaborators (say within 2 hops) from his private network G'_4. The query {"DB","AI","CV"} on Bob's network returns "No answer". The answer from the public graph G alone is a subtree rooted at "Bob" whose leaf vertices are {"Dave","Carol"}, but they are not close to each other. From the combined network of G'_4 and G, Bob obtains a subtree rooted at "Bob" whose leaf vertices are {"Alice","Carol"}, which is a closer collaboration.*

[1] https://developers.google.com/knowledge-graph/

TABLE I: Frequently used notations

Notations	Meaning
Q_f / Q	A query of a particular query semantic, such as r-clique, Blinks and k-nk. The subscript is omitted when the context is obvious.
$G/G'/G_c$	the public graph / the private graph / the combined graph
$\mathbb{P}$	portal nodes : the common nodes of public and private graphs
$\mathsf{eval}(G, Q_f, f)$	the evaluation to query Q in G with a keyword search semantic f
$d(u,v)/d'(u,v)$ $/d_c(u,v)$	the shortest distance from u to v in public graph / the private graph / the combined graph
$\mathcal{P}/\mathcal{P}'/\mathcal{P}_c$	a path in the public graph / the private graph / the combined graph

The example above reveals three major challenges for keyword search on the public-private graph model. Firstly, given a query semantic, the query answer on a private graph can be different from the one on the combined graph. Secondly, it is costly to construct and maintain indexes for the combined graph of each user. Thirdly, the constraints, querying and indexing techniques vary by different keyword search semantics. As a consequence, the adaptations of different query semantics for the public-private graph model are different as well. To the best of our knowledge, keyword search on the public-private graph has not been studied yet.

Solution overview and contributions. This paper focuses on the technically interesting case of keyword search on the public-private graph, where the answers span across the public and private graphs, a.k.a. *public-private answers*.[2] The paper makes the following contributions:

- We propose a public-private keyword search framework, called PPKWS. PPKWS consists of three key steps, including (i) *partial evaluation* (PEval), (ii) *answer refinement* (ARefine) and (iii) *answer completion* (AComplete). We show that some representative keyword search algorithms (r-clique [14], Blinks [10], and k-nk [13]) can be implemented on top of PPKWS with small modifications.
- ARefine and AComplete of keyword searches require numerous shortest distance computations. We propose an efficient index on the public graph, namely *PageRank-based all distance sketch* (PADS) and *PageRank-based keyword distance sketch* (KPADS). In a nutshell, PADS and KPADS extend *All Distance Sketches* (ADS) with PageRank in their index construction. PADS and KPADS are much smaller than exact indexes. Importantly, PADS and KPADS exhibit the same theoretical guarantee as ADS in estimating shortest distance with a much higher accuracy in practice.
- Two optimizations for PPKWS are proposed to speed up the three steps, where searches for popular keyword search semantics on PPKWS can be optimized.
- We conduct extensive experiments on PPKWS. The results show that PPKWS can speed up the query performance of some keyword searches on public-private answers such as Blinks on average by 202 times, r-clique on average by 12 times and k-nk on average by 120 times. The accuracy of PADS is 99.7%.

Organization. The rest of this paper is organized as follows: Sec. II presents the background and the problem statement. Sec. III introduces the overview of the framework for public-private keyword search. Sec. IV presents how to implement keyword search semantics on top of PPKWS. Sec. V and Sec. VI present the indexes of PPKWS and the optimization, respectively. Sec. VII reports the experimental evaluation. Sec. VIII discusses the related work. In Sec. IX, we conclude the paper and present the future work.

[2]For query answers that are generated from either the public or the private graph, we can directly apply existing work on keyword search.

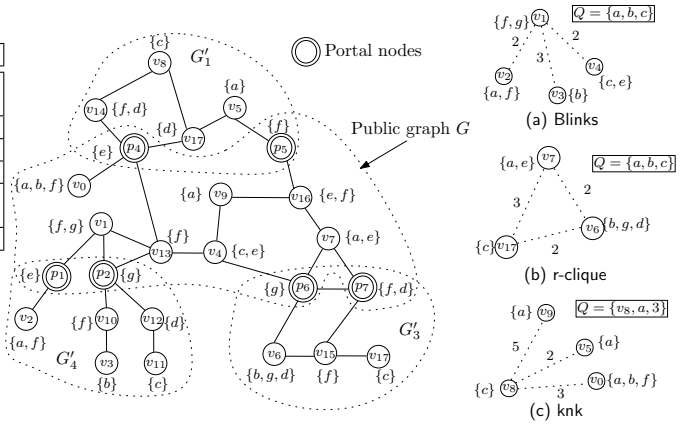


Fig. 2: An example of a public graph, three private graphs, and three popular keyword search semantics

II. Background and Problem Statement

This section presents the background for the technical discussions and then the problem statement. Some frequently used notations are summarized in Tab. I.

Graphs. We consider a *labeled, weighted, undirected graph* $G = (V, E, L, \Sigma)$, where (a) V is a set of vertices; (b) E ($\subseteq V \times V$) is a set of edges; (c) Σ is a set of labels; and (d) L:$V \rightarrow \Sigma$ is a mapping s.t. for each vertex $v \in V$, $L(v)$ maps v to a set of labels in Σ. Each edge $e = (u, v) \in E$ has a positive weight, denoted as $w(e)$. For simplicity, we may omit L and Σ when they are irrelevant to the discussion. With slight abuse of definition, the size of the graph is denoted by $|G|= |V|+|E|$.

Public graphs and private graphs. The majority of previous studies assume that the public (*e.g.*, the query processor) has all the data graphs. As explained in Sec. I, not all of the graphs are accessible to the public. Hence, we distinguish the public graph $G = (V, E, L, \Sigma)$ and a private graph $G' = (V', E', L, \Sigma)$. We define an *attach* operation for attaching the private graph to a public graph via some common nodes (*i.e.*, $V \cap V' \neq \emptyset$) to form a *combined graph* (a.k.a. public-private graph), denoted as $G_c = G \oplus G'$, where $G_c = (V_c, E_c)$, $V_c = V \cup V'$, and $E_c = E \cup E'$. We call the common nodes the *portal nodes*.

Definition II.1. *(Portal node) Given a private graph $G' = (V', E')$ and a public graph $G = (V, E)$, the portal nodes $\mathbb{P}$ are defined as follows: $v \in \mathbb{P}$ iff $v \in V$ and $v \in V'$.*

We remark that G is accessed by all the users. Each G' is possibly disconnected and only accessed by a single user. In addition, $|G'|$ is often relatively smaller than $|G|$.

Keyword search semantics for graphs. Several keyword query semantics have been proposed [10], [14], [2], and [23] (see Fig. 2). We review the following three queries as their

semantics and underlying algorithms are diverse and they are driven by various interesting applications.

Blinks. He et al. [10] propose that a keyword query is a 2-ary tuple (Q, τ) which contains a set of keywords Q=$\{q_1,\ldots,q_n\}$ and a distance bound τ. Given a graph $G = (V, E, L, \Sigma)$, an answer to Q in G is a subgraph of G, denoted as $T = \{r, v_1, \ldots v_n\}$, such that (i) T is a tree rooted at r; (ii) v_i is a leaf vertex of T and $q_i \in L(v_i)$; and (iii) $d(r, v_i) \leq \tau$.

r-clique. Kargar et al. [14] propose r-clique, which determines the subgraph that all pairs of vertices that contain the query keywords are reachable to each other within r hops. That is, $d(v_i, v_j) < r$, where v_i and v_j are a pair of vertices that contain the query keywords in an answer subgraph.

k-nk. Jiang et al. [13] propose k-nk, which determines the top-k vertices R that contain a query keyword which is the nearest to a given query vertex q, *i.e.*, there does not exist $u \notin R$, but u contains the query keyword such that $d(q, u) < \max_{v \in R} d(q, v)$. The semantics have been extended to the conjunction and disjunction of multiple keywords.

Example II.1. *Consider a public graph G and several private graphs (G'_1, G'_3 and G'_4) shown in Fig. 2. All the edge weights are 1. The answer to the query $\{a, b, c\}$ under the* Blinks *semantic on $G \oplus G'_4$ is shown in Fig. 2(a). Fig. 2(b) and 2(c) respectively show the answers of the queries $\{a, b, c\}$ and $\{v_8, a, 3\}$ under the* r-clique *and* k-nk *semantics.*

We remark that the common factor is that the query answers are compact, *i.e.*, the relevant pieces of information are assumed to be located close to each other, either in the public graphs, private graphs, or a combination of the two. It can be observed from Fig. 2 that the answers of all the query semantics involve the *shortest distances* between the nodes of the answer. Also, the shortest distance between two vertices in G and G_c can be different and must be computed. Since each user can have a different combined graph, it is not space efficient to build an index for the combined graph for each user.

The query evaluation of a keyword search algorithm f on the combined graph is denoted as $A = \mathsf{eval}(G \oplus G', Q, f)$. It should also be remarked that an answer obtained from $G \oplus G'$ (a public-private graph) can be either a public answer, a private answer or a public-private answer (see Def. II.2). Existing works can be directly applied on G and G' to tackle the former two cases. As public-private answer is the most technically challenging, this paper focuses on computing them.

Definition II.2. *(Public-private answer) Given an answer $a = (V_a, E_a) \in A$, a is a public-private answer iff i) $\bigcup L(v'_i) \cap Q \neq \emptyset$, where $v'_i \in V_a$ and $v'_i \in G'.V$, and ii) $\bigcup L(v_i) \cap Q \neq \emptyset$, where $v_i \in V_a$ and $v_i \in G.V$.*

Problem statement. Given a public graph G, a private graph G', a keyword query Q of a keyword search algorithm f, we investigate a framework to determine the answer A of Q. □

Algorithm 1: Framework of PPKWS (Sec. III)

Input: A public graph G, a private graph G', a keyword search algorithm f with a query Q
Output: Query answer set A
1 $(A', \mathcal{C}) = \mathsf{PEval}(G', Q, f)$ //partial eval. on G'
2 $A' = \mathsf{ARefine}(A', \mathcal{C}, Q, G \oplus G')$ //refine A' with $\mathcal{C}$
3 $A = \mathsf{AComplete}(A', Q, G)$ //answer completion
4 **return** A

III. Framework of public-private keyword search

We start with an overview of the framework for public-private keyword search (outlined in Algo 1). The details are presented together with specific query semantics in Sec. IV.

Step 1) Partial Evaluation (PEval). The first step of PPKWS is *partial evaluation*, denoted as PEval. PEval is the keyword search algorithm f with a small modifications. Upon receiving $Q = \{q_1, q_2, \ldots, q_n\}$ and the private graph G' (Line 1), PEval computes the partial answers A' and *refinement indicators* $\mathcal{C}$. Each $a' \in A'$ records the query keywords it contains. $\mathcal{C}$ is used to indicate what to be refined in the partial answers A'. (The definition of $\mathcal{C}$ is discussed with query semantics in Sec. IV.) For instance, in this section, we denote $C \in \mathcal{C}$ as $\{(e_1, e_2)\}$, where e_1 and e_2 could be either a vertex or a keyword. Each C records *a set of (e_1, e_2) pairs whose distances need to be further refined in a partial answer $a' \in A'$.*

We remark that existing keyword search algorithms continue to work to retrieve *public or private answers* by using PEval as follows. PEval takes the public graph (resp. the private graph), the keyword query and an algorithm as input. If PEval returns an answer whose $C \in \mathcal{C}$ is $\emptyset$, it is a public answer (resp. private answer). Such PEval simulates the keyword algorithms and does not increase the time complexity. They have the same query performance as the original algorithms.

Step 2) Answers Refinement (ARefine). Instead of rerunning the keyword search algorithms on the combined graph, PPKWS refines and completes the partial answers. The shortest distance between any pair of vertex/keyword can be different after attaching the private graph to the public graph. Hence, ARefine takes the query Q, the partial answers A' and the refinement indicators $\mathcal{C}$ as an input, and refines the distance between each pair in C ($C \in \mathcal{C}$) for each $a' \in A'$.

More specifically, consider any pair $(e_1, e_2) \in C$. Since G' is a subgraph of G_c, the shortest path between e_1 and e_2 in G' is obviously a path in the combined graph $G_c = G \oplus G'$. The shortest distance between e_1 and e_2 in G' (*i.e.*, $d'(e_1, e_2)$) is a trivial upper bound of that in G_c (*i.e.*, $d_c(e_1, e_2)$). Hence, we index the portal distances of G'. When G' is attached to G, the portal distances are refined and then each $d_c(e_1, e_2)$ is refined by comparing the lengths of the paths that cross the portal nodes.

Step 3) Answers Completion (AComplete). For a partial answer $a' \in A'$, PPKWS completes it by using the public graph G. AComplete (a) determines which keywords are missing from the partial answers and (b) completes A' with G to form the final answer set A.

To sum up, a keyword search algorithm f can be implemented on the public-private graph model with the minor modification by following the above three steps. Firstly, PPKWS applies f on the private graph G' to compute partial answers A' and refinement indicators $\mathcal{C}$. Secondly, PPKWS refines each answer $a' \in A'$ according to the indicator $C \in \mathcal{C}$. Lastly, PPKWS completes A' by retrieving the missing keywords on the public graph G to yield A.

IV. Query Processing in PPKWS

In this section, we present how three representative query semantics (r-clique, Blinks and k-nk) are implemented on top of PPKWS. For each semantic, we first summarize its query evaluation and then present its three steps in PPKWS.

A. Distance-based keyword search (r-clique) on PPKWS

We recall that the r-clique keyword search semantic [14] determines the subgraph that all pairs of the vertices that contain the query keywords are reachable to each other within τ hops, where τ is a user-specified parameter. More specifically, the r-clique semantic is as follows:

- **Input:** A graph G, a query $Q = \{q_1, q_2, \ldots, q_n\}$
- **Output:** Answer A, where for each $a \in A$, $a = \{v_1, v_2, \ldots, v_n\}$, s.t. $q_i \in L(v_i)$ and $d(v_i, v_j) \leq \tau$

IV-A.(I) Overview of r-clique

Kargar et al. [14] propose an approximation algorithm to compute the top-k answers in PTIME. We use our notations to present the major steps of r-clique, as follows:

Initialization. The keywords q_is are matched to a set of keyword nodes, denoted as V_{q_i}s. The *search space* is denoted as $SP = (V_{q_1}, \ldots, V_{q_n})$. r-clique inserts a pair $\langle SP, a\rangle$ into a priority queue $\mathcal{S}$, where SP is a search space and $a = \{v_1, \ldots, v_n\}$ is an approximate best answer of SP. The priority queue $\mathcal{S}$ is ordered in ascending order according to the *weight* of a, which is the *total distance between keyword nodes*. Given an SP, to find the best answer a, r-clique computes the shortest distances between $v_i \in V_{q_i}$ and V_{q_j}, where $q_i \neq q_j$ and $q_i, q_j \in Q$. In particular, it computes $a_{v_i} = \{u_1, \ldots v_i, \ldots, u_n\}$ as a candidate best answer, where $u_j = \arg\min_{\forall v_j \in V_{q_j}} d(v_i, v_j)$ (Algo 2, Lines 17-21). The best answer is the best a among all candidate answers obtained from the above method.

Search space decomposition. r-clique recursively decomposes the search space. In each iteration, the pair $\langle SP, a\rangle$ in the front of $\mathcal{S}$ is removed and $a = \{v_1, \ldots, v_n\}$ is added into the answer set. r-clique decomposes the search space SP into n subspaces such that $SP_i = (V_{q_1}, \ldots, V_{q_i} \setminus \{v_i\}, \ldots, V_{q_n})$, $q_i \in Q$ (Algo 2, Line 10). r-clique inserts the search subspaces SP_i into $\mathcal{S}$ together with their respective approximate answers.

Termination. The search procedure terminates when $\mathcal{S}$ is empty or the top-k answers are found.

IV-A.(II) r-clique on PPKWS (PP-r-clique)

Prior to the discussion of r-clique on PPKWS, we present some basic notations.

Algorithm 2: PEval for r-clique

```
Input: Private graph G', 𝕡, keyword query Q
Output: (A', C) = eval(G', Q, r-clique)
1  append portal nodes to possible match V'_{q_i} = V_{q_i} ∪ 𝕡
2  construct a search space SP = (V'_{q_1}, ..., V'_{q_n})
3  initialize two queues A and S and a refinement indicator set C
4  a' = FindTopAnswer(SP)
5  S.add(⟨SP, a'⟩)
6  while S is not empty do
7      ⟨SP, a'⟩ = S.removeTop()
8      A.add(a')
9      C.insert(a'.C)
10     decompose SP and push the subsapces
         ⟨SP_i, FindTopAnswer(SP_i)⟩ into S
11 return (A, C)
12 Function FindTopAnswer(SP)
13     initialize an empty set A'
14     foreach V'_{q_i} ∈ SP do
15         foreach v_i ∈ V'_{q_i} do
16             initialize a' = ⟨v_i, match = ∅⟩, a'.C = ∅
17             foreach V'_{q_j} ∈ SP do
18                 if q_i ≠ q_j then
19                     d_j = d(v_i, V'_{q_j})
20                     u_j = arg min_{∀v_j ∈ V_{q_j}} d(v_i, v_j)
21                     a'.match[q_j] = ⟨u_j, d_j⟩
22                     a'.C.insert((v_i, ⟨u_j, d_j⟩))
23             A'.add(a')
24     return the answer a ∈ A' with the minimum weight
```

Algorithm 3: ARefine for r-clique

```
Input: Partial answers A'=eval(G', Q, r-clique), d_c, Q
Output: Refined partial answers A'
1 foreach a' ∈ A' do
2     foreach (v, ⟨u, d⟩) in a'.C do
3         foreach (p_i, p_j) ∈ 𝕡 × 𝕡 do
4             dist = d'(v, p_i) + d_c(p_i, p_j) + d'(p_j, u)
5             d = min(d, dist)
6 return A'
```

Partial answer $a' \in A'$. A partial answer a' is a tuple $\langle v, \mathsf{match}\rangle$, where v is an answer root, match is a map. $\mathsf{match}[q]$ takes a query keyword q as input and returns two attributes $\langle \mathsf{u}, \mathsf{d}\rangle$, where $\mathsf{match}[q].\mathsf{u}$ is a vertex u such that $q \in L(u)$ or a portal node, and $\mathsf{match}[q].\mathsf{d}$ is the distance between u and v.

(1) PEval. PPKWS takes [14] as PEval to compute all the r-clique on G' (Algo 2). Since it is possible to complete the partial answers with the public graph, we append the portal nodes $\mathbb{P}$ to the search space, *i.e.*, $V'_{q_i} = V_{q_i} \cup \mathbb{P}$. Partial answers are stored in A'. For each partial answer $a' \in A'$, PEval declares a set of vertex pairs to be refined in $C = \{(v, \langle u, d\rangle)\}$, denoted by $a'.C$.

(2) ARefine. ARefine refines $a'.C$ by verifying whether the distance $d_c(v, u)$ is smaller than $d'(v, u)$, after attaching the private graph to the public graph (Algo 3, Lines 3-5). Given any $a' = \langle v, \mathsf{match}\rangle$, the answer can be refined by tightening the distance between the pair v and u of $(v, \langle u, d\rangle) \in a'.C$.

(3) AComplete. Given any refined partial answer $a' = \langle v, \mathsf{match}\rangle$. For any $\langle u, d\rangle = \mathsf{match}[q]$, if u is a portal node and $q \notin L(u)$, the partial answer misses the query keyword q. Therefore, we complete the answer by computing the distance between u and q on the public graph. If $d_c(u, q) + d > \tau$, the

partial answer is pruned, due to r-clique's semantic.

Further, given an answer $a \in A$, we say a is *qualified* to be a public-private answer *iff* 1) $a.\mathsf{match}[q].\mathsf{d} \leq \tau$; and 2) the query keywords in a is located on both public and private graphs. This can be implemented by maintaining a counter for each answer that stores the number of keywords that are matched in the private graph.

Theorem IV.1. *Given an answer of* PP-r-clique, $a = \langle v, \mathsf{match}\rangle$, $a.\mathsf{match}[q].\mathsf{d} \leq (2c-1)d_c(v, a.\mathsf{match}[q].\mathsf{u})$.

Proof. The proof is presented in Appx. E of [12]. □

B. Keyword search with subtree answer (Blinks) *on PPKWS*

A common method to answer keyword query on a data graph without any connectivity index is to traverse the graph starting at the vertices which contain the query keywords. For example, Bhalotia et al. [2] present the first *backward keyword search algorithm*. He et al. [10] propose a search strategy for the backward expansion, namely Blinks. Subtree answers are computed. The query semantic can be described as follows:

- **Input:** A graph G, a query $Q = \{q_1, q_2, \ldots, q_n\}$
- **Output:** Answer A, where for each $a \in A$, $a = \langle r, \{v_1, v_2, \ldots, v_n\}\rangle$ s.t. $q_i \in L(v_i)$ and $d(r, v_i) \leq \tau$

IV-B.(I) Overview of Blinks

We next summarize the major steps of Blinks [2].

Initialization. Consider a keyword query $Q = \{q_1, q_2, \ldots, q_n\}$. We denote the set of vertices that contain the keyword q_i as V_{q_i} (a.k.a. search origin), and the set of vertices that could reach one of the vertices in V_{q_i} as V_i.

Backward expansion. In each search step, the vertex set V_i with the smallest size is processed as follows. The vertex $v \in V_i$ that has the shortest distance to V_{q_i} is chosen for backward expansion. In the expansion, u is added to V_i and it is checked whether u can be an answer root, where (u, v) is an incoming edge of v. Otherwise, the backward expansion continues.

Answer discovery. It discovers an answer root r such that r can reach at least one node that contains q_i, for each $q_i \in Q$.

IV-B.(II) Blinks on PPKWS (PP-Blinks)

We next present how PPKWS can support Blinks. We start with the some notations.

Partial answer $a' \in A'$. A partial answer a' is a tuple $\langle r, \mathsf{match}\rangle$, where r is a candidate answer root, match is a map. $\mathsf{match}[q]$ takes a query keyword q as input and returns two attributes $\langle \mathsf{v}, \mathsf{d}\rangle$, where $\mathsf{match}[q].\mathsf{v}$ is a vertex v such that $q \in L(v)$ or a portal node, and $\mathsf{match}[q].\mathsf{d}$ is the distance between r and v. At the end, PEval also produces $C = \{(r, q)\}$ for each partial answer a' associated with the vertex-keyword pairs to be refined, denoted by $a'.C$. Since the computation of $\mathsf{match}[q].\mathsf{v}$ and that of $\mathsf{match}[q].\mathsf{d}$ are similar, we only show how to compute $\mathsf{match}[q].\mathsf{d}$ below.

(1) PEval. We initialize the search origin with Q in G'. When each vertex $r \in V'$ is traversed, r is stored as a candidate answer a'. We also record the missing keywords in the partial answer a', which require completion on the public graph.

Algorithm 4: ARefine for Blinks

Input: Partial answers A'=eval(G', Q, Blinks), d_c, Q, and bound τ
Output: Refined partial answers A'

```
1 foreach a' ∈ A' do
2     foreach (r, q) in a'.C do
3         foreach (p_i, p_j) ∈ P × P do
4             dist = d'(r, p_i) + d_c(p_i, p_j) + d'(p_j, q)
5             if a'.match[q].d ≥ dist then
6                 a'.match[q].d = dist
7 return A'
```

(2) ARefine. Algo 4 refines $a'.C$ by verifying whether $d_c(r, q)$ is smaller than $d'(r, q)$ when attaching the private graph to the public graph, where $(r, q) \in a'.C$.

Algo 4 shows that each partial answer can be refined with the refined portal distances d_c in $O(|C||\mathbb{P}|^2)$. First, the shortest paths between answer roots and keywords may contain some portal nodes. Second, the distances between portal nodes can be refined in the combined graph. Hence, in Line 5-6, we check whether the refined portal distance refines the distance of an answer root and a keyword.

(3) AComplete. AComplete of Blinks contains three steps.

(a) Backward expansion. The first step is to further backwardly expand on the public graph since the answer root r' can be located on the public graph. For each partial answer whose root $r' \in \mathbb{P}$, AComplete backwardly expands r' by Breadth-First Traversals (denoted by $\mathcal{T}_{r'}$) on the public graph up to x hops from r', where $x = \max\{\tau - \mathsf{match}[q].\mathsf{d}\}$. For the x'-hop vertex u in $\mathcal{T}_p$, if u has been visited by $\mathcal{T}_{p'}$, where $p' \neq p$, PPKWS adopts the same strategy of flooding search (cf. [23]) to update dist of the visited answer (Lines 14-19). Otherwise, PPKWS generates a partial answer rooted at u (Line 8). The shortest distance between u and query keyword q is the sum of x' and the distance between p and q.

(b) Retrieving missing keywords. The second step is to retrieve the missing keywords for each partial answer. For each answer $a \in A$, we compute the distance between $q \in Q$ and $a.r$ in the public graph (Lines 20-23). If $d(a.r, q) \leq a.\mathsf{match}[q].\mathsf{d}$, we set $a.\mathsf{match}[q].\mathsf{d} = d(a.r, q)$.

(c) Answer qualification. The answer qualification is the same as that of PP-r-clique.

Lemma IV.2. *The following quality guarantees of the distances hold for* $a = \langle r, \mathsf{match}\rangle \in \mathsf{eval}(G \oplus G', Q, \mathsf{Blinks})$ *and* $a' = \langle r, \mathsf{match}'\rangle$ *returned by PPKWS:*

- *if* $\mathsf{match}[q].\mathsf{v} \in G'.V$, *then* $\mathsf{match}'[q].\mathsf{v} = \mathsf{match}[q].\mathsf{v}$ *and* $\mathsf{match}'[q].\mathsf{d} = \mathsf{match}[q].\mathsf{d}$; *and*
- *if* $\mathsf{match}[q].\mathsf{v} \notin G'.V$, *then* $\mathsf{match}'[q].\mathsf{d} \leq (2c-1)\mathsf{match}[q].\mathsf{d}$.

Proof. The proof is presented in Appx. B of [12]. □

C. Top-k Nearest Keyword Search (k-nk) *on PPKWS*

A query of k-nk [13] is a triple (v, q, k), where v is a query vertex, q is a query keyword, and it determines the k nearest vertices to v that contain the keyword q.

- **Input:** a query point v, a query keyword q
- **Output:** top k vertices $A = \{a = \{\langle u_i, d_i\rangle\}\}$ ranked by d_i, where $q \in L(u_i)$

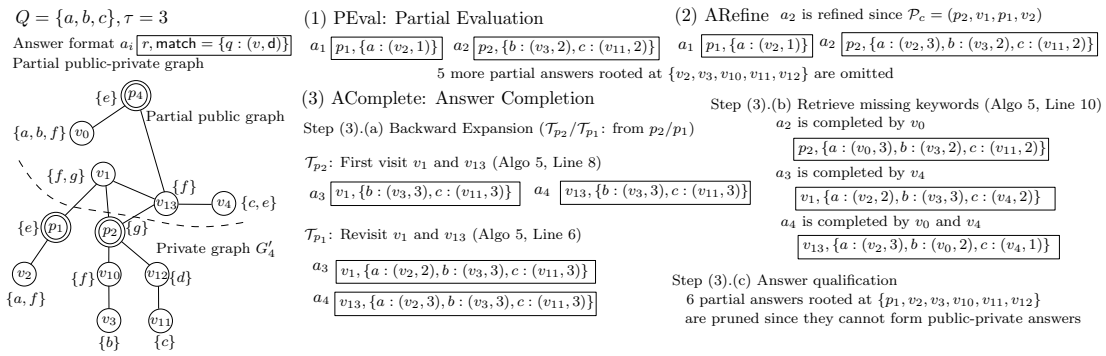


Fig. 3: Example of query execution (PEval, ARefine and AComplete) of PP-Blinks

Algorithm 5: AComplete for Blinks

Input: Public graph G, $\mathbb{P}$, refined answers A', Q, and bound τ
Output: Completed answers A

```
1  initialize an empty map A = {}
2  foreach p ∈ ℙ do
3      if A'.containsKey(p) then
4          foreach u in x'-th hop BF' traversal starting from p on G do
5              if A.containsKey(u) then
6                  A[u] = UpdateAns(A[u], A'[p], x', Q)
7              else
8                  A[u].match = {⟨A'[p].match[q].v, A'[p].match[q].d + x'⟩}
9  foreach a ∈ A do
10     ComplAns(a, Q)
11     if NOT a.isQualified() then
12         A.remove(a)
13 return A
14 Function UpdateAns(a1, a2, x, Q)
15     foreach q ∈ Q do
16         if a2.match[q].d + x < a1.match[q].d then
17             a1.match[q].d = a2.match[q].d + x
18             a1.match[q].v = a2.match[q].v
19     return a1
20 Function CompleteAns(a, Q)
21     foreach q ∈ Q do
22         compute the shortest distance d(a.r, q) on public graph
23         a.match[q].d = min(d(a.r, q), a.match[q].d)
```

IV-C.(I) Overview of k-nk

Any k-nk algorithms can be applied on the PPKWS framework without any modification. Hence, we omit the overview of k-nk, due to space restrictions.

IV-C.(II) k-nk on PPKWS (PP-knk)

PEval of k-nk is the original algorithm [13]. It computes the answers A' from G'. ARefine of k-nk is identical to Sec. IV-A. AComplete completes A' by retrieving $u_i \in G.V$ from the public graph. The details of k-nk are presented in Appx. A of [12].

Complexities. The time complexities of r-clique, Blinks and k-nk on top of PPKWS have not increased. The analysis is presented in Appx. C of the technical report [12].

V. Index for PPKWS

As presented in Sec. II, the definitions of keyword search semantics often involve the shortest distances of nodes, *e.g.*, [10], [14], [2], and [23]. Their query algorithms require numerous shortest distance computations. For example, when applying r-clique [14] on the combined graph $G_c = G \oplus G'_3$ in Fig. 2, finding an answer of $Q = \{a, b, c\}$, as shown in Fig. 2(c), requires 12 shortest distance computations in PEval for r-clique on G' (Algo 2, Line 10), and 8 shortest distance computations on $G \oplus G'$ (Algo 3). Hence, we propose indexes for the public graph G and the private graph G' *respectively*, so as to optimize the query processing on the combined graph $G \oplus G'$.

Firstly, to avoid the exhaustive search for the distances of shortest paths that *cross the public graph and private graph*, we propose PADS and KPADS for estimating the shortest distances between the vertices in the public graph and those between the keywords and vertices in the public graph (Sec. V-A and Sec. V-B). Secondly, we index the portal nodes by precomputing their all-pair shortest distances (Sec. V-C). Thirdly, we introduce a portal-keyword distance map to store the shortest distances between the portal nodes and the keywords (Sec. V-C).

A. PageRank-based All Distance Sketches (PADS)

In this subsection, we review ADS and then propose our index. It is known that ADS is small in size, accurate, and efficient in answering shortest distance queries. Our main idea is to use PageRank to determine the chance of a node to be included in the sketch (*i.e.*, the index).

All-Distances sketches (ADS). Recall that in [4], given a graph $G = (V, E)$, each vertex v is associated with a sketch, which is a set of vertices and their corresponding shortest distances from v. To select the vertices in V and put them as the centers in the sketch of v, each vertex is initially assigned a *random* value in $[0, 1]$. If a vertex $u \in V$ has the k-th largest value among the vertices which have been traversed from v in the Dijkstra order, then u is added to the sketch of v. k is a user-defined parameter set by user. A larger k results in larger and more accurate sketches. The shortest distance between u and v can be estimated by the intersection set of ADS(u) and ADS(v) (a.k.a. the common centers).

A drawback of ADS is that it does not consider the relative importance of the vertices when generating the sketch. We observe that vertices with high PageRanks, which roughly estimates the importance of the vertices in a graph, should be added to the sketch to cover the shortest paths. On the contrary, the vertices with low PageRanks are unlikely to be on many shortest paths and should not be added to the sketch.

Algorithm 6: PADS construction

Input: Graph $G = (V, E)$
Output: PADS

```
1  compute the PageRank pr of the vertices in G
2  initialize PADS(v) = {(v, 0)} for each vertex v ∈ V
3  sorted the vertices V by the descending order of pr(v)
4  for v ∈ V do
5      for u in the Dijkstra's traversal do
6          if |{(w, d) ∈ PADS(u) | d ≤ d(v, u)}| < k then
7              add (v, d(v, u)) into PADS(u)
8          else
9              continue the traversal on the next vertex
10 return PADS
```

PageRank. We employ any efficient algorithms to obtain the PageRank of the vertices of a graph G. We use a function pr: $V \to$ [0,1] to denote the PageRank of a vertex v by $pr(v)$.

Dijkstra rank. We recall that we can efficiently obtain the Dijkstra rank of a vertex v w.r.t a source vertex s as follows. We run the Dijkstra's algorithm starting at s and obtain the order of the visited nodes $[v_1, v_2, \ldots, v_l]$. The Dijkstra rank of v_i w.r.t s is i, denoted as $\pi(s, v_i) = i$.

PageRank-based all-distances sketches (PADS). Given a Dijkstra rank π, the PageRank, a vertex v, and a threshold k, the PADS of v is defined as follows:

$$\mathsf{PADS}(v) = \{(u, d(v, u)) \mid pr(u) \geq k(v, u)\}, \qquad (1)$$

where $k(v, u)$ is the k-th largest PageRank among the nodes from v to u according to π.

Example V.1. (*PADS construction*) *Consider the public graph G in Fig. 4. Assume $k = 1$. We compute the PageRank values for all the vertices in the graph, as shown below the vertices' labels. v_{13} covers 41 out of 156 shortest paths in the graph G in total, which is the largest among all the vertices. This shows that the node having a large PageRank value, $pr(v_{13}) = 0.130$, can be an effective center. To determine the PADS of v_1, we run the Dijkstra's algorithm by taking v_1 as the source vertex to obtain the Dijkstra ranked list $[v_1, p_1, p_2, v_{13}, v_4, \ldots, p_7]$. Since the PageRank value of v_{13} is the highest among the first four vertices in the ranked list, v_{13} is added to PADS(v_1) with its distance to v_1. Similarly, v_1 is added to PADS(v_1).*

Shortest distance estimation. Given a shortest distance query (u, v) and the PADS, $\hat{d}(u, v)$ is computed by the intersection of PADS(u) and PADS(v) as follows:

$$\hat{d}(u, v) = \min\{(d_1 + d_2)\}, \qquad (2)$$

where $(w, d_1) \in \mathsf{PADS}(u), (w, d_2) \in \mathsf{PADS}(v)$.

Space complexity. The expected size of PADS(v) is $O(k \ln n)$, where n is the number of nodes reachable from v, which is bounded by $O(k \ln|V|)$. (The analysis of [4] can be applied to PADS.)

Time complexity. Each iteration of PageRank and Dijkstra rank are both computed in $O(|V|+|E|)$. In Algo 6, the times each edge (v, u) has been traversed is bounded by the size of PADS(v) (Line 6). Since the expected size of PADS(v) is bounded by $k \ln|V|$, the time complexity of Algo 6 is $O(k|E|\ln|V|)$ (cf. [4]).

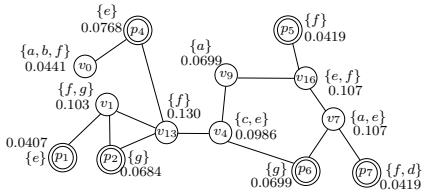


Fig. 4: A public graph (fragment) and the PageRank

TABLE II: An ADS label for the public graph in Fig. 4

Vertex ID	ADS
v_0	$\{(v_0, 0), (p_4, 1), (v_1, 3), (p_1, 4), (p_7, 6)\}$
p_4	$\{(p_4, 0), (v_1, 2), (p_1, 3), (p_7, 5)\}$
v_{13}	$\{(v_{13}, 0), (p_4, 1), (v_1, 1), (p_1, 2), (p_7, 4)\}$
v_1	$\{(v_1, 0), (p_1, 1), (p_7, 5)\}$
p_1	$\{(p_1, 0), (p_7, 6)\}$
p_2	$\{(p_2, 0), (v_1, 1), (p_1, 2), (p_7, 5)\}$
v_4	$\{(v_4, 0), (v_{13}, 1), (v_9, 1), (p_4, 2), (v_1, 2), (p_1, 3), (p_7, 3)\}$
v_9	$\{(v_9, 0), (p_4, 3), (v_1, 3), (p_7, 3)\}$
p_6	$\{(p_6, 0), (v_4, 1), (v_{13}, 2), (v_9, 2), (p_7, 2)\}$
v_{16}	$\{(v_{16}, 0), (v_9, 1), (p_7, 2)\}$
v_7	$\{(v_7, 0), (v_{16}, 1), (p_7, 1)\}$
p_5	$\{(p_5, 0), (v_9, 2), (p_7, 3)\}$
p_7	$\{(p_7, 0)\}$

Consider the graph G in Fig. 4. We set $k = 1$ and compute its ADS shown in Tab. II and its PADS shown in Tab. III. We can see that there are two advantages of PADS. First, the size of PADS is significantly smaller than that of ADS. Second, the estimation of PADS is much more accurate than that of ADS.

Example V.2. (*Shortest distance estimation.*) *Consider the graph G in Fig. 4 and its PADSs in Tab. III. Given two vertices v_9 and v_7, there are two common centers v_{16} and v_{13} in PADS(v_9) and PADS(v_7). The shortest distance is estimated by Eq. 2, i.e., $\hat{d}(v_9, v_7) = 2$ (i.e., 0% error). By ADS, $\hat{d}(v_9, v_7) = 4$ is returned (i.e., 100% error). More specifically, we compare the estimation accuracy of ADS and PADS between all pairs of the vertices in Fig. 4. The average error of PADS (resp. ADS) is around 3% (resp. 38%).*

It is worth noting that PADS exhibits the theoretical guarantee of the shortest path estimation stated below.

Lemma V.1. *The distance between two vertices u and v is estimated using Eq. 2 with an approximation factor $(2c-1)$, where $c = \lceil \frac{\ln|V|}{\ln k} \rceil$ with a constant probability, i.e., $\hat{d}(u, v) \leq (2c-1)d(u, v)$.*

Proof. Let $d = d(u, v)$. Let $N_i(u)$ denote the neighbors of u within id hops. For simple exposition, we denote the intersections of $N_i(u)$ and $N_{i-1}(v)$/$N_{i+1}(v)$ as $I_{2i-1} = N_i(u) \cap N_{i-1}(v)$ and $I_{2i} = N_i(u) \cap N_{i+1}(v)$, respectively. It is worth noting that $I_{i-1} \subseteq I_i$. Consider the ratio of $\frac{|I_{i-1}|}{|I_i|}$ and a ratio threshold $\frac{\alpha}{k}$. Given the vertices with k largest pr values in I_i, if one of them (say w) hits I_{i-1}, w belongs to both PADS(v) and PADS(u). The distance d can be estimated within $(2i-1)d$. The probability of *at least one of the vertices, which has the k largest PageRank values in I_i, hits the I_{i-1}* is $1-(1-\frac{\alpha}{k})^k \approx 1-e^{-\alpha}$. Since there are n vertices in graph G at most, $|I_i| \leq n$. Hence, there exists $i \leq \log_{k/\alpha} n$. □

B. PageRank-based Keyword Distance Sketches (KPADS)

We denote the *shortest distance between a vertex v and a keyword t* by $d(v, t)$, where $d(v, t) = \min\{d(v, u)|t \in$

TABLE III: The PADS label for the public graph in Fig. 4

Vertex ID	PADS
v_0	$\{(v_0,0),(p_4,1),(v_{13},2)\}$
p_4	$\{(p_4,0),(v_{13},1)\}$
v_{13}	$\{(v_{13},0)\}$
v_1	$\{(v_1,0),(v_{13},1)\}$
p_1	$\{(p_1,0),(v_1,1),(v_{13},2)\}$
p_2	$\{(p_2,0),(v_1,1),(v_{13},1)\}$
v_4	$\{(v_4,0),(v_{13},1)\}$
v_9	$\{(v_9,0),(v_4,1),(v_{16},1),(v_{13},2)\}$
p_6	$\{(p_6,0),(v_4,1),(v_7,1),(v_{13},2)\}$
v_{16}	$\{(v_{16},0),(v_7,1),(v_{13},3)\}$
v_7	$\{(v_7,0),(v_{16},1),(v_{13},3)\}$
p_5	$\{(p_5,0),(v_{16},1),(v_7,2),(v_{13},4)\}$
p_7	$\{(p_7,0),(v_7,1),(v_{16},2),(v_{13},4)\}$

TABLE IV: The KPADS label for the public graph in Fig. 4

Terms	KPADS
a	$\{(v_9,0),(v_4,1),(p_4,1),(v_7,0),(v_{13},2),(v_{16},1),(v_0,0)\}$
b	$\{(v_0,0),(v_{13},2),(p_4,1)\}$
c	$\{(v_{13},1),(v_4,0)\}$
d	$\{(v_{13},4),(v_7,1),(p_7,0),(v_{16},2)\}$
e	$\{(v_{13},1),(v_4,0),(v_1,1),(v_7,0),(p_4,0),(v_{16},0),(p_1,0)\}$
f	$\{(p_5,0),(v_1,0),(v_{13},0),(p_4,1),(v_7,1),(v_{16},0),(v_0,0),(p_7,0)\}$
g	$\{(p_6,0),(v_1,0),(v_4,1),(v_{13},1),(v_7,1),(p_2,0)\}$

$L(u), u \in V\}$. To estimate the distance between a given vertex and a keyword, we propose KPADS, which is constructed by PADS-merging: Given any two vertices u and u' where $t \in L(u)$ and $t \in L(u')$, there may exist common centers in PADS(u) and PADS(u'). Hence, we only keep the smaller one between $\hat{d}(v,u')$ and $\hat{d}(v,u)$, since both of them are the upper bound of $d(v,t)$.

Keyword-PADS (KPADS). For each keyword $t \in \Sigma$, we build a sketch KPADS(t). KPADS(t) can be built by merging PADS of those vertices that contain t, *i.e.*, PADS(v) where $t \in L(v)$. More formally, given a center $(w_i, d_i) \in$ PADS(v), $(w_i, d_i) \in$ KPADS(t) *iff* $\forall (w_i, d_i') \in$ PADS(v') and $t \in L(v')$, $d_i' \geq d_i$.

Shortest keyword-vertex distance estimation. Given a vertex v and a keyword t, the shortest distance $\hat{d}(v,t)$ can be computed as follows:

$$\hat{d}(v,t) = \min\{(d_1+d_2) | (w,d_1) \in \mathsf{PADS}(v), (w,d_2) \in \mathsf{KPADS}(t)\} \quad (3)$$

Example V.3. *Consider the graph G in Fig. 4 and its* PADS *in Tab. III. The KPADS is shown in Tab. IV. Consider the shortest distance between a and p_4. The distance can be estimated by the intersection of* KPADS(a) *and* PADS(p_4). *There are two common centers, p_4 and v_{13}. $\hat{d}(a,p_4) = 1$ is returned by the common center p_4.*

Lemma V.2. *The distance between a vertex v and a keyword t derived from Eq. 3 has an approximation factor $(2c-1)$ where $c = \lceil \frac{\ln|V|}{\ln k} \rceil$ with a constant probability, i.e., $\hat{d}(v,t) \leq (2c-1)d(v,t)$.*

Proof. Due to space limitations, the analysis is presented in Appx. B of [12]. □

Time complexity. The time complexity of the shortest distance estimation between a vertex and a keyword (or another vertex) is $O(k \ln|V|)$. The derivation is presented in Appx. C of [12].

Index size. The size of KPADS(t) is bounded by $\sum|\mathsf{PADS}(v_i)|$. Therefore, the total size of KPADS for all the terms is bounded by $\sum_{v_i \in V}|L(v_i)|\mathsf{PADS}(v_i)$. In practice, $|L(v_i)|$ is often small.

Query processing with the indexes. We take Blinks as an example. It takes $O(|E|+|V|\ln|V|)$ to complete an answer of Blinks on the public graph G by Dijkstra's algorithm with Fibonacci heap (Algo 5, Line 22). With KPADS, this procedure can be done in $O(|Q|k\ln|V|)$.

Algorithm 7: Portal distance map construction

Input: All pair portal distance on private graph $d'(p_i,p_j)$, All pair portal distance on public graph $d(p_i,p_j)$
Output: All-Pairs portal distance on the combined graph

```
1  initialize a priority queue Queue
2  for p_i, p_j ∈ ℙ do
3      if d(p_i,p_j) ≥ d'(p_i,p_j) then
4          d(p_i,p_j) = d'(p_i,p_j)
5          Queue.insert(⟨p_i, p_j, d(p_i,p_j)⟩)
6  while Queue is not empty do
7      ⟨p_1, p_2, dist⟩ = Queue.removeTop();
8      for p_i ∈ ℙ do
9          if d(p_i,p_2) ≥ d(p_i,p_1) + dist then
10             d(p_i,p_2) = d(p_i,p_1) + dist
11             Queue.insert(⟨p_i, p_2, d(p_i,p_2)⟩)
12         if d(p_i,p_1) ≥ d(p_i,p_2) + dist then
13             d(p_i,p_1) = d(p_i,p_2) + dist
14             Queue.insert(⟨p_i, p_1, d(p_i,p_1)⟩)
15 return d
```

C. Indexes of portal distances

The shortest distance computations on the combined graphs can be time-consuming. In this subsection, we index the shortest distances of the portal nodes since the number of portal nodes $|\mathbb{P}|$ is often relatively small when compared to $|V|$. We then extend the idea to index the distances of portal and keyword nodes.

Portal distance maps. We call the shortest distance between two portal nodes *the portal distance*. We precompute all the portal distances of $\mathbb{P}$ on the public graph G (denoted as d) and the private graph G' (denoted as d'), respectively. We index the distances in distance maps, d and d', respectively. We can then efficiently index the portal distances of the combined graph G_c as follows.

Step 1. Portal distance refinement. We first refine the portal distance in the private graph in the presence of those in the public graph (shown in Lines 3-5, Algo 7). We use a priority queue $Queue$ to maintain the refined portal distances. Initially, if $d(p_i,p_j) \leq d'(p_i,p_j)$, where $p_i, p_j \in \mathbb{P}$, we refine $d'(p_i,p_j)$ to $d(p_i,p_j)$ (Line 4) and insert the pair with the distance into $Queue$ (Line 5). Next, we pop $\langle p_1, p_2, dist\rangle$ from the head of $Queue$, when $Queue$ is not empty. For each $p_i \in \mathbb{P}$, if the sum of $d(p_i,p_1)$ and the refined portal distance $d(p_1,p_2)$ is smaller than the current portal distance $d(p_i,p_2)$, there is a shorter path between p_i and p_2 via p_1. Then, the portal distance between p_i and p_2 can be refined. Similarly, the distance between p_i and p_1 can be refined by p_2.

Step 2. Shortest distance refinement using portal distance. We next reduce the refinement of shortest distance via the portal distance maps described above. To index the shortest distances of the combined graph, we compare the shortest distance in the private graph and the length of the paths crossing the portal nodes as follows:

$$d_c(v_1,v_2) = \min \begin{cases} d'(v_1,v_2); \\ d'(v_1,p_i)+d'(p_j,v_2)+d_c(p_i,p_j), \text{where } p_i,p_j \in \mathbb{P}. \end{cases} \quad (4)$$

Portal-keyword distance map. We extend the idea to the distances between the portal nodes and the keywords, and index them in a *portal-keyword distance map*, denoted as PKD. More formally, given a portal node $p \in \mathbb{P}$ and $t \in G'.\Sigma$, $\mathsf{PKD}(p,t)$ is a tuple $\langle \mathsf{v}, \mathsf{d} \rangle$, where 1) $\mathsf{PKD}(p,t).\mathsf{v} \in G'.V$ is the nearest vertex v of p such that 1) $t \in L(v)$ and 2) $\mathsf{PKD}(p,t).\mathsf{d} = d'(p,v)$.

Vertex-portal distance map. We also index the distances between the vertex of the private graph and each portal node, denoted by $d'(v,p)$, where $v \in G'.V$ and $p \in \mathbb{P}$. Hence, the refinement between $v \in G'.V$ and $t \in G'.\Sigma$ can be computed by Formula (5).

$$d_c(v,t) = \min \begin{cases} d'(v,t); \\ d'(v,p_i) + d_c(p_i,p_j) + \mathsf{PKD}(p_j,t).\mathsf{d}, \end{cases} \quad (5)$$

where $p_i, p_j \in \mathbb{P}$.

Query processing with the indexes. The indexes proposed in this section significantly improve the performance of answer refinement. For example, the refinement time of each r-clique answer reduces to $O(|Q||\mathbb{P}|^2)$, since the distances (Algo 3, Line 3) have been precomputed. Without the indexes, it takes $O(|Q||\mathbb{P}|^2(|E|+|V|\ln|V|))$ by running the Dijkstra's algorithm on $G \oplus G'$.

VI. Optimization for PPKWS

In this section, we present two optimizations that are applicable to answer refinement and answer completion, regardless of the query semantic implemented on the top of them.

A. Reduced answer refinement

The distance of a vertex pair (v_1, v_2), as presented in Formula (4), is refined only when their portal pairs have been refined. More formally, we state this in the following lemma. It is established by a simple proof by contradiction.

Lemma VI.1. *If $d_c(v_1,v_2) \leq d'(v_1,v_2)$, there exists $p_i \in \mathbb{P}$ and $p_j \in \mathbb{P}$ such that (a) $d_c(p_i,p_j) \leq d'(p_i,p_j)$; and (b) $p_i, p_j \in \mathcal{P}_c(v_1,v_2)$.*

For each private graph, we use a table to record the portal pairs that have been refined during the answer refinement. We maintain it in the main memory during query processing. Before (v_1, v_2) is refined, we check the table and Lemma VI.1 to see if refinement is necessary.

B. Dynamic programming for answer completion

As introduced in Sec. IV, given a query Q, PPKWS first evaluates it on the private graph G' and generates the partial answer set A'. In the worst case, answer completion retrieves missing keywords $|A'||Q|$ times. Combined with the estimation time (elaborated in Sec. V-B), the time complexity of answer completion is then $O(|A'||Q||\mathbb{P}|k\ln|V|)$.

For different partial answers, some missing keywords completion can be shared. Therefore, we estimate the shortest distance between a portal node $p_i \in \mathbb{P}$ and a query keyword $q_j \in Q$ in the public graph and store them in a 2-D array denoted by PKA online such that $\mathsf{PKA}[i][j] = \hat{d}(p_i, q_j)$. Then, the complexity can be reduced to $O(|A'||Q||\mathbb{P}|+|\mathbb{P}||Q|k\ln|V|)$.

Next, we present how dynamic programming is used to cache some intermediate answers, which further reduces the number of retrievals (*i.e.*, $|A'||Q||\mathbb{P}|$) on the public graph.

We use a keyword-portal pair $m = (t,p)$ in a partial answer $a \in A'$ to denote that a expects to retrieve missing keyword t through a portal p. For each a, we denote all such pairs as $\mathcal{M} = \{m = (t_j, p_i)\}$, where $t_j \in Q, p_i \in \mathbb{P}$. The current state is $\mathcal{S}(\mathcal{M})$. $\mathcal{S}(\mathcal{M})$ can be computed by $\mathcal{S}(\mathcal{M}-\{m\}) \cup \mathcal{S}(\{m\})$. $\mathcal{S}(\{m\})$ can be retrieved by looking up $\mathsf{PKD}[i][j]$.

The complexity of incorporating dynamic programming is $O(|A'|+2^{|Q|}|\mathbb{P}|+|\mathbb{P}||Q|k\ln|V|)$.

TABLE V: Statistics of real-world datasets

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	avg. # of keywords	$\lvert V'\rvert$	$\lvert E'\rvert$
YAGO3	2,635,317	5,260,573	3.79	482	501
DBpedia	5,795,123	15,752,299	3.72	538	873
PP-DBLP	2,221,139	5,432,667	10	9.2	27.6

TABLE VI: Characteristics of PADS and ADS

Datasets	Construction time		Size (# of centers)		Approx. ratio	
	ADS	PADS	ADS	PADS	ADS	PADS
YAGO3	5096s	5066s	28.79M	20.57M	1.08452	1.00001
DBpedia	39237.3s	38757s	103.65M	74.21M	1.13194	1.0059
PP-DBLP	3761s	2770s	20.49M	15.15M	1.06178	1.00284

VII. Experimental study

We used real-life datasets to conduct three sets of experiments to evaluate PPKWS for their (1) index characteristics, (2) query performance and (3) optimization performance.

A. Experimental Setup

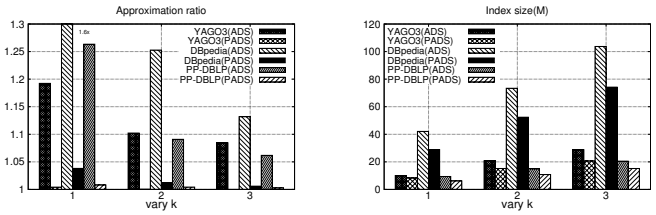
1) Software and hardware: Our experiments were run on a machine with a 2.93GHz CPU and 64GB memory running CentOS 7.4. The implementation was made memory-resident.

2) Algorithms: We implemented Blinks and r-clique in C++ and used the same settings as presented in the original works. For Blinks, we adopted METIS for partitioning. For r-clique, we built the neighbor index with $R = 3$, as in [14]. We obtained the code of k-nk from [13] and used the same setting. We designed the baseline algorithms (Baseline-PPKWS) as follows. 1) For Baseline-Blinks and Baseline-r-clique, we extended Blinks and r-clique with a simple qualification function to verify if an answer is a valid public-private answer and applied them on the combined graph G_c. For Baseline-knk, we directly applied k-nk on the combined graph G_c.

3) Datasets and default indexes: Tab. V summarizes some characteristics of the real-life datasets used.

YAGO3.[3] YAGO3 [16] is a large knowledge base, derived from Wikipedia, WordNet and GeoNames. In the experiment, we extracted the entities (vertices) and the corresponding facts (edges) in specific domains (*e.g.*, chemistry, or movies) to form the private graphs. The rest of the entities and facts formed the public graphs.

[3] http://www.mpi-inf.mpg.de/yago



(a) Apprximation ratio (b) Index size

Fig. 5: Comparison between PADS and ADS

DBpedia.[4] DBpedia (v3.9) is a knowledge graph with 5.8M vertices and 15.8M edges. It extracts structured content from the information created in various Wikimedia projects. Similar to YAGO3, we derived private graphs of DBpedia from specific domains. The rest of the entities and facts form the public graph.

Intuitively, the information from a specific domain can be kept privately by their owners (*e.g.*, private laboratories, or movie investors). Hence, we extracted the entities in a specific domain by retrieving YAGO3's ontology graph, *i.e.*, all the descendant entities of the domain term (*e.g.*, chemicals and moive information etc) will be returned. All these entities consist of V' and the corresponding induced subgraph of the ontology graph form E'. The portal node set $\mathbb{P} = \{v | v \in V'$ and $v \in V\}$. In this section, we only present the performance results of the experiments that used the entities in chemistry and movie domains as private graphs.

PP-DBLP.[5] We used public-private graphs from real-world DBLP records, called PP-DBLP [11]. We set the "current" time as 2013. Existing collaborations made the public graph, while ongoing collaborations formed the private graphs, as they were only known by some authors.

4) Queries: We generated 50 random synthetic keyword queries for the experiments. Some details are given below. For each algorithm, we report the results of 10 queries, including three good, three bad, and four medium cases.

Blinks *and* r-clique. The query keyword $q \in Q$ was randomly picked from the label set $G.\Sigma$ and $G'.\Sigma$. For Blinks, we set the pruning threshold d_{max} (a.k.a. τ_{prune} in [10]) to 5 to ensure keyword nodes were reachable from the root vertex within 5 hops. We remark that if $Q \cap G'.\Sigma = \emptyset$ or $Q \cap G.\Sigma = \emptyset$, Q has no public-private answer. Users obtain the public answers (resp. private answers) by passing the public graph G (resp. private graph G') to PEval as input. But PPKWS does not offer the performance improvement. As a consequence, Q cannot show the performance of AComplete. To make sure the public-private answers exist and investigate the runtimes of the three key steps, we generate Q s.t. $Q \cap G'.\Sigma \neq \emptyset$ and $Q \cap G.\Sigma \neq \emptyset$.

k-nk. We note that the frequency of a keyword in the private graph is smaller than 64. Again, to study public-private answers, we generated the query (v, q, k), where k was set to 64, v was randomly picked from $G'.V$, and q was selected following the keyword distribution of the combined graph.

[4] http://dbpedia.org

[5] https://github.com/samjjx/pp-data

B. Experimental Results

Exp-1: Characteristics of PPKWS. We next report the size of the PADS and the time of constructing KPADS. We also present the efficiency and effectiveness of PPKWS in Tab. VI and Fig. 5.

Index sizes. For comparison, we implemented [5] for the shortest distance estimation. For real-life graphs, PADS is 28.6% (resp. 28.5% and 26.1%) smaller than ADS on YAGO3 (resp. DBpedia and PP-DBLP).

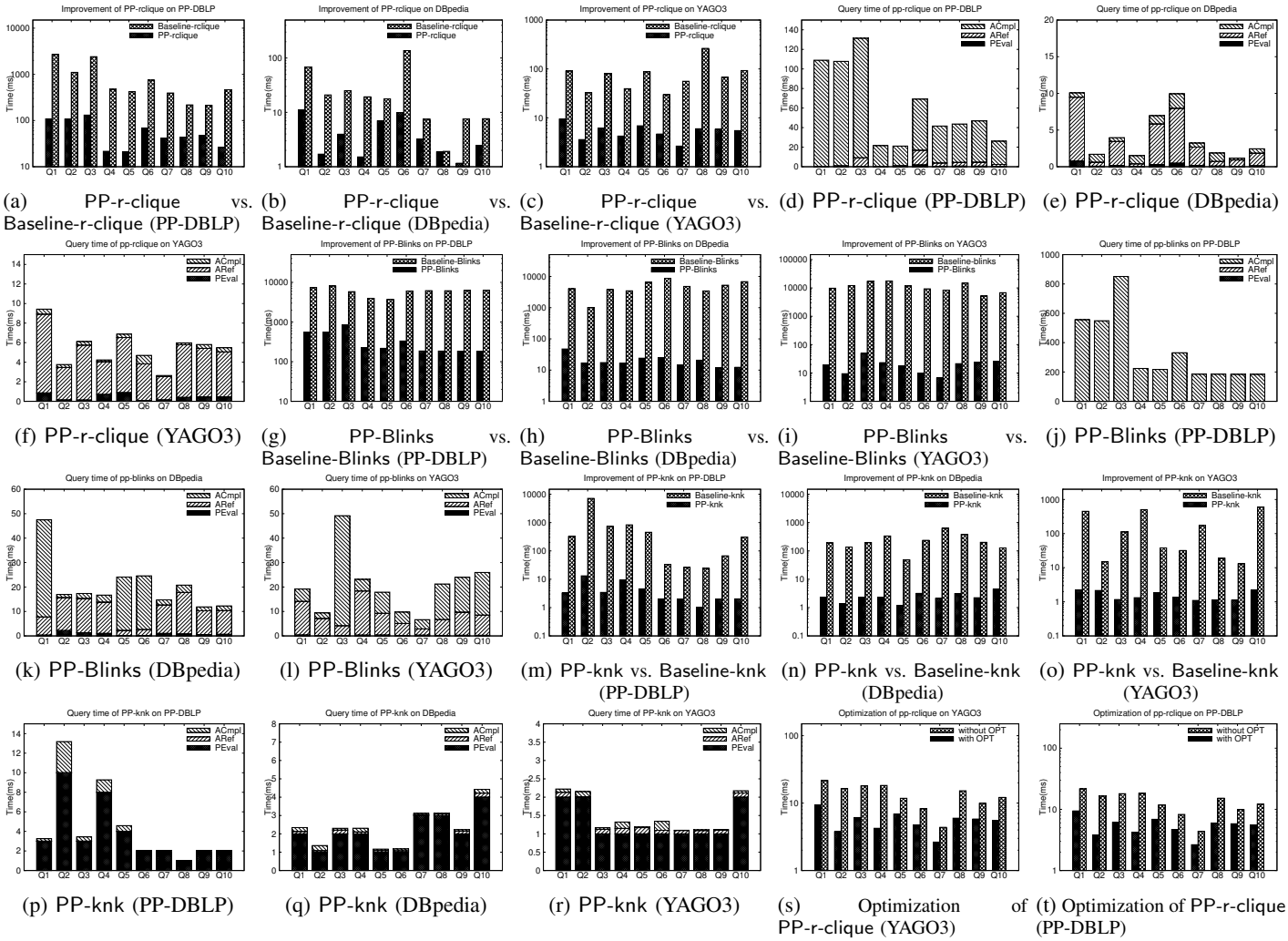
Construction time. We report the construction times in Tab. VI. PPKWS takes 1.41 hours (resp. 10.8 hours and 46 minutes) to construct PADS for YAGO3 (resp. DBpedia and PP-DBLP). The construction time on PADS and ADS is slightly different except PP-DBLP. The construction time of PADS is 26.4% smaller than that of ADS.

Accuracy. We randomly selected a vertex pair (s, t) from $|V|$. We compared the accuracy of PADS with ADS by computing the shortest distances between each vertices pair, denoted as $\hat{d}(s, t)$. The exact distance between s and t was computed using Dijkstra's algorithm [6], denoted as $d(s, t)$. We denoted the error as $\epsilon = \frac{\hat{d}(s,t) - d(s,t)}{d(s,t)}$. We repeated the above procedure 1 million times and got the average error $\bar{\epsilon}$. As we presented in Fig. 5a, we varied the parameter k from 1 to 3. On YAGO3, $\bar{\epsilon}$ of PADS reduces from 4.2×10^{-3} to 1×10^{-5}. Similarly, $\bar{\epsilon}$ of PADS also decreases significantly on DBpedia and PP-DBLP when k increases. We set $k = 3$ for the comparison between PADS and ADS. $\bar{\epsilon}$ of PADS is 99.99% (resp. 96.53% and 95.40%) smaller than that of ADS on YAGO3 (resp. DBpedia and PP-DBLP).

Exp-2: Query performance. To evaluate the efficiency of PPKWS, we have tested the performance of Blinks, r-clique and k-nk with and without PPKWS.

r-clique. The comparison between PP-r-clique and Baseline-r-clique is shown in Fig. 6d to Fig. 6f. In a nutshell, PPKWS is 12.11 times faster on average. (1) On PP-DBLP, the query is at most 24.75 times and at least 4.5 times faster than the baseline algorithm. For all the queries, it is 14.30 times faster on average. (2) On DBpedia, the query is at most 13.79 times faster and at least 2.3 times than the baseline algorithm. For all the queries, it is 6.69 times faster on average. (3) On YAGO3, the query is at most 44.09 times, at least 6.31 times and on average 15.4 times faster than the baseline algorithm. This is because Baseline-r-clique requires exploration of the whole search space derived from the combined graph, even the queries have public-private answers.

We next report the query performance breakdown. Fig. 6d-Fig. 6f show the three major steps of query processing. On PP-DBLP, (a) PPKWS spends a large fraction of the time to completing the partial answers; except for Q_6, AComplete of the queries takes more than 90% of the whole query time, (b) the runtime of PEval is negligible, and (c) the runtime for ARefine is only a small part of the query time. On YAGO3 and DBpedia, the experiment results are similar. On DBpedia,



(a) PP-r-clique vs. Baseline-r-clique (PP-DBLP) (b) PP-r-clique vs. Baseline-r-clique (DBpedia) (c) PP-r-clique vs. Baseline-r-clique (YAGO3) (d) PP-r-clique (PP-DBLP) (e) PP-r-clique (DBpedia)

(f) PP-r-clique (YAGO3) (g) PP-Blinks vs. Baseline-Blinks (PP-DBLP) (h) PP-Blinks vs. Baseline-Blinks (DBpedia) (i) PP-Blinks vs. Baseline-Blinks (YAGO3) (j) PP-Blinks (PP-DBLP)

(k) PP-Blinks (DBpedia) (l) PP-Blinks (YAGO3) (m) PP-knk vs. Baseline-knk (PP-DBLP) (n) PP-knk vs. Baseline-knk (DBpedia) (o) PP-knk vs. Baseline-knk (YAGO3)

(p) PP-knk (PP-DBLP) (q) PP-knk (DBpedia) (r) PP-knk (YAGO3) (s) Optimization of PP-r-clique (YAGO3) (t) Optimization of PP-r-clique (PP-DBLP)

Fig. 6: Performance of three keyword search semantics on three datasets

PPKWS spends 5.02% (resp. 62.51% and 32.47%) on PEval (resp. ARefine and AComplete). On YAGO3, PPKWS spends 7.54% (resp. 85.50% and 6.96%) on PEval (resp. ARefine and AComplete). The differences in the percentages of the datasets are due to portal distance refinement. The more portal distances are refined, the more time ARefine takes.

Blinks. PPKWS runs in 202 times faster on average. The results on YAGO3, DBpedia and PP-DBLP are reported in Fig. 6g-Fig. 6i. (1) On PP-DBLP, the query is at most 33.97 times and at least 6.84 times faster than the baseline algorithm. For all the queries, it is 22 times faster on average. (2) On DBpedia, the query is at most 554 times and at least 60 times faster than the baseline algorithm. For all the queries, it is 268 times faster on average. (3) On YAGO3, the query is at most 890 times, at least 77 times and on average 315 times faster than the baseline algorithm. The reason is that PP-Blinks does not traverse the vertices that are far from the private graph.

We next report the query performance breakdown. Fig. 6j to Fig. 6l show the three major steps of query processing. On PP-DBLP, 99.9% of the query time is spent on AComplete. The time of PEval and ARefine is negligible. On YAGO3 and DBpedia, the experiment results are similar. On DBpedia, PPKWS spends 5% (resp. 59.1% and 35.9%) on PEval (resp. ARefine and AComplete). On YAGO3, PPKWS spends 1.7% (resp. 47.1% and 51.2%) on PEval (resp. ARefine and AComplete). We note that the average number of the nodes within x hops of the portal nodes in PP-DBLP is much larger than those of YAGO3 and DBpedia. The more vertices in the public graph are traversed, the more time AComplete takes.

k-nk. PPKWS runs 120 times faster (on average) than the baseline algorithms. The results on YAGO3, DBpedia and PP-DBLP are reported in Fig. 6m to Fig. 6o. On average, PP-knk is 128 times (resp. 110 times and 120 times) faster than Baseline-knk on PP-DBLP (resp. DBpedia and YAGO3).

We next report the query performance breakdown. Fig. 6p to Fig. 6r show the three major steps of query processing. On PP-DBLP, PPKWS spends 92.2% (resp. 0.2% and 7.6%) of the time on PEval (resp. ARefine and AComplete). On YAGO3 and DBpedia, the experiments results are similar. On DBpedia, PPKWS spends 87.5% (resp. 5.5% and 7%) of the time on PEval (resp. ARefine and AComplete). On YAGO3, PPKWS spends 86.6% (resp. 8.0% and 5.4%) on PEval (resp. ARefine

and AComplete).

Exp-4: Improvement of the optimization. We performed an experiment to investigate the effectiveness of the proposed optimization in Sec. VI. We turned the optimization on and off and ran the query sets on YAGO3 and PP-DBLP. The results are reported in Fig. 6s and Fig. 6t. All the optimizations offer 55.8% (resp. 28.8%) performance improvement on YAGO3 (resp. PP-DBLP) on average. This is because 1) ARefine only needs to refine the answer by the portal distances which have been refined rather than by those of all the portal pairs, and 2) the cost of completing a partial answer on public graph reduces from $|Q||\mathbb{P}|k\ln|V|$ to $|Q||\mathbb{P}|$.

VIII. Related work

Keyword search semantics. Recently, keyword search has attracted a lot of interest from both industry and research communities (*e.g.*, [10], [14], [2]). He et al. [10] propose an index and search strategies for reducing keyword search time. Kargar et al. [14] propose distance restrictions on keyword nodes, (*i.e.*, the shortest distance between each pair of keyword nodes is smaller than r). Ye et al. [23] propose a search strategy based on a compressed signature to avoid exhaustive search. These studies optimize a specific keyword search semantic. This work improves the performance of different existing keyword search semantics in a generic manner. We propose a PPKWS framework for public-private keyword search. Their indexes and search strategies could be adopted in our framework with slight modifications.

Public-Private graph model. Some studies on *public-private graph analysis* have been conducted previously. Chierichetti [3] et al. propose two computational paradigms, sketching and sampling, for some key problems on massive public-private graphs. The sketching and sampling are precomputed offline and the online update algorithms are run on the private graphs. Ebadian [8] et al. propose a classification-based hybrid strategy to compute k-truss on public-private graphs, incrementally. Archer [1] et al. propose an approximation algorithm by seeking a set of seeding nodes to solve the reachability query on the public-private graph model. Huang [11] et al. develop a new model of attributed public-private networks by considering the information of vertices. Our work is different from these previous works as PPKWS is the first work that studies different keyword search semantics on the public-private graph model.

IX. Conclusions and Future Works

In this paper, we propose PPKWS for supporting efficient keyword searches on the public-private graph model. We show that three popular keyword search algorithms can be implemented on PPKWS with minor changes. We verify that PPKWS significantly reduces the runtimes of the keyword searches. The proposed indexes PADS and KPADS offer not only a theoretical guarantee in shortest distance estimation but also high accuracy in practice.

In future work, we plan to investigate PPKWS for other query semantics which are relevant to the shortest distance computation, (*e.g.*, community search). We will extend the PPKWS to support keyword search on dynamic graphs.

Acknowledgements. This work is partly supported by HKRGC GRF 12232716, 12201119, and 12201518.

References

[1] A. Archer, S. Lattanzi, P. Likarish, and S. Vassilvitskii. Indexing public-private graphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1461–1470, 2017.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[3] F. Chierichetti, A. Epasto, R. Kumar, S. Lattanzi, and V. Mirrokni. Efficient algorithms for public-private social networks. In *SIGKDD*, pages 139–148. ACM, 2015.

[4] E. Cohen. All-distances sketches, revisited: Hip estimators for massive graphs analysis. *IEEE Trans. on Knowl. and Data Eng.*, 27(9):2320–2334, 2015.

[5] E. Cohen, D. Delling, F. Fuchs, A. V. Goldberg, M. Goldszmidt, and R. F. Werneck. Scalable similarity estimation in social networks: Closeness, node labels, and random edge lengths. In *Proceedings of the first ACM conference on Online social networks*, pages 131–142. ACM, 2013.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[7] R. Dey, Z. Jelveh, and K. Ross. Facebook users have become much more private: A large-scale study. In *PERCOM Workshops*, pages 346–352, 2012.

[8] S. Ebadian and X. Huang. Fast algorithm for k-truss discovery on public-private graphs. *arXiv preprint arXiv:1906.00140*, 2019.

[9] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *PVLDB*, 9(12):1233–1244, 2016.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[11] X. Huang, J. Jiang, B. Choi, J. Xu, Z. Zhang, and Y. Song. Pp-dblp: Modeling and generating attributed public-private networks with dblp. In *ICDM*, 2018.

[12] J. Jiang, H. Xin, B. Choi, J. Xu, S. S. Bhowmick, and L. Xyu. ppkws: An efficient framework for keyword search on public-private networks. *https://www.comp.hkbu.edu.hk/%7Ejxjian/ppkws.pdf*, 2019.

[13] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404. ACM, 2015.

[14] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[15] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *SIGMOD*, pages 173–182, 2006.

[16] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *Seventh Biennial Conference on Innovative Data Systems Research*, 2014.

[17] B. Mirzasoleiman, M. Zadimoghaddam, and A. Karbasi. Fast distributed submodular cover: Public-private data summarization. In *Advances in Neural Information Processing Systems*, pages 3594–3602, 2016.

[18] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[19] Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, pages 589–600, 2011.

[20] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, 2008.

[21] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan. Summarizing answer graphs induced by keyword queries. *PVLDB*, 6(14):1774–1785, 2013.

[22] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. Autog: A visual query autocompletion framework for graph databases. *PVLDB*, 9(13):1505–1508, 2016.

[23] Y. Yuan, X. Lian, L. Chen, J. X. Yu, G. Wang, and Y. Sun. Keyword search over distributed graphs with compressed signature. *IEEE Trans. Knowl. Data Eng.*, 29(6):1212–1225, 2017.

[24] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao. Semantic SPARQL similarity search over RDF knowledge graphs. *PVLDB*, 9(11):840–851, 2016.

APPENDIX

A. Top-k Nearest Keyword Search (k-nk) on PPKWS

Due to space limitations, we present the k-nk semantic in this appendix, which is consistent with other previous works [15], [18], [13], [19]. A query of k-nk is a triple (v, q, k), where v is a query vertex and q is a query keyword. k-nk aims to find the k nearest vertices from v which contain the keyword q. More specifically, the k-nk semantic can be described as follows:

- **Input:** a query point v, a query keyword q
- **Output:** top k vertices $A = \{a = \{\langle u_i, d_i\rangle\}\}$ ranked by d_i, where $q \in L(u_i)$

IV-C.(I) Overview of k-nk Any k-nk algorithms can be applied on the PPKWS framework without any modification. Hence, we omit the overview of k-nk for simplicity.

IV-C.(II) k-nk on PPKWS (PP-knk)

(1) PEval. PPKWS takes [13] as PEval to compute the k-nk answers on the private graph G'.

Partial answer $a' \in A'$. The partial answer a' is a list match where the i-th element has two attributes $\langle \mathsf{u}, \mathsf{d}\rangle$. $a'.\mathsf{match}[i].\mathsf{u}$ is a vertex u, such that $u \in V'$ and $q \in L(u)$ or $a'.\mathsf{u} \in \mathbb{P}$ and $a'.\mathsf{match}[i].\mathsf{d} = d'(v, u)$. We use a boolean variable to record whether u is a portal. For the partial answer a', PEval declares $C = \{(v, u)\}$ to indicate what to be refined, where v and u are two vertices on the private graph. More specifically, v is the query point of k-nk and u is a candidate matched vertex in the private graph.

(2) ARefine. The refinement of the vertex pair $(v, u) \in C$ is identical to that discussed in Sec. IV-A.

(3) AComplete. Given the refined answer a', PPKWS completes a' in the public graph. For the i-th element of $a'.\mathsf{match}$, *i.e.*, $\langle u, d\rangle$, if $q \in L(u)$, u is a candidate match vertex. Moreover, if u is a portal node, PPKWS estimates the shortest distance between u and the keyword q in the public graph with the intersection of PADS(u) and KPADS(q). $\hat{d}(u, q) + d$ is appended with the keyword vertex u' at the end of $a'.\mathsf{match}$, where u' can be obtained by the inverted index of KPADS(q) (For simplicity, we omit the details of this data structure in this paper). It is worth noting that, we maintain a priority queue with a fixed size k for $a'.\mathsf{match}$ rather than a list in AComplete.

Lemma A.1. *If $u \in V'$ belongs to the answer of a k-nk query (v, q, k) on G_c, where $v \in V'$, then u is returned by* PP-knk.

Proof. The detailed proof is presented in [12]-Appendix B. □

B. Proofs of lemmas

Lemma A.2. *The distance between a vertex v and a keyword t derived from Eq. 3 has an approximation factor $(2c - 1)$, where $c = \lceil \frac{\ln |V|}{\ln k} \rceil$ with a constant probability, and k is a parameter set by user as we introduced in Sec. V-A.*

Proof. Given a vertex v and a keyword t, we denote the vertex which is the closest to v and contains t as u, i.e. for any vertex u' where $t \in L(u')$, $d(v, u') \geq d(v, u)$. And $\hat{d}(v, u)$ can be estimated with the same approximation factor, $(2c - 1)$, by PADS(v) and PADS(u) with the same probability, $1 - e^{-\alpha}$, of Lemma V.1. We denote the common center by w_i.

$$d(v, w_i) + d(w_i, u) \leq (2c - 1)d(v, u) \tag{6}$$

By the definition of PADS-merging (we compress the common centers while keep smallest distance), we have $(w_i, d_i) \in$ KPADS, and $d_i \leq d(w_i, u)$.

$$d(v, w_i) + d_i \leq d(v, w_i) + d(w_i, u) \leq (2c - 1)d(v, u) \tag{7}$$

□

We denote the answer of Blinks by $a = \langle r, \mathsf{match}\rangle$, where r is a candidate answer root, match is a map $\mathsf{match}[q] = \langle v, d\rangle$ such that $q \in G.L(v)$, and d is the shortest distance between r and v where $q \in Q$. We have the following conclusion.

Lemma A.3. *The following quality guarantees of the distances hold for $a = \langle r, \mathsf{match}\rangle \in \mathsf{eval}(G \oplus G', Q, \mathsf{Blinks})$ and $a' = \langle r, \mathsf{match}'\rangle$ returned by PPKWS:*

- *if* $\mathsf{match}[q].\mathsf{v} \in G'.V$, *then* $\mathsf{match}'[q].\mathsf{v} = \mathsf{match}[q].\mathsf{v}$ *and* $\mathsf{match}'[q].\mathsf{d} = \mathsf{match}[q].\mathsf{d}$; *and*
- *if* $\mathsf{match}[q].\mathsf{v} \notin G'.V$, *then* $\mathsf{match}'[q].\mathsf{d} \leq (2c - 1)\mathsf{match}[q].\mathsf{d}$.

Proof. For simplicity, $\mathsf{match}[q].\mathsf{v}$ (resp. $\mathsf{match}'[q].\mathsf{v}$) is denoted by v (resp. v').

Case 1: Suppose $r \in G'.V$.

- ***Case 1.1:*** If $v \in G'.V$, due to the definition of Blinks, $\mathsf{match}[q].\mathsf{d} = d_c(r, q)$. Moreover, ARefine refines the distance between of r and q. Hence, $\mathsf{match}'[q].\mathsf{d} = d_c(r, q)$. Hence, $\mathsf{match}'[q].\mathsf{d} = \mathsf{match}[q].\mathsf{d}$. Similarly, we have $v = v'$.
- ***Case 1.2:*** If $v \in G.V$, the shortest path between r and v is denoted by $\mathcal{P}_c(r, \ldots, v)$. Since $v \in G.V$ and $r \in G'.V$, $\mathcal{P}_c(r, \ldots, v)$ contains at least one portal node. We denote them by $\mathbb{P}^c = \{p_1^c, \ldots, p_n^c\}$. We denote the last portal node in $\mathcal{P}_c(r, \ldots, v)$ by p_{last}^c. It is worth noting that the shortest path between p_{last} and v is located on the public graph. Otherwise, there is at least portal node in $\mathcal{P}_c(p_{last}, \ldots, v)$, denoted by p_i.

  1) If $p_i \notin \mathbb{P}^c$, we have $\mathcal{P}_c(r, \ldots, v) = \mathcal{P}_c(r, \ldots, p_1, \ldots, p_{last}, \ldots, p_i, \ldots, r)$, then p_i is the last portal node rather than p_{last}.
  2) If $p_i \in \mathbb{P}^c$, we have $\mathcal{P}_c(r, \ldots, v) = \mathcal{P}_c(r, \ldots, p_1, \ldots, p_i, \ldots, p_{last}, \ldots, p_i, \ldots, r)$, then there is a cycle $\mathcal{P}_c(p_i, \ldots, p_{last}, \ldots, p_i)$ on $\mathcal{P}_c(r, \ldots, v)$ which is contradicted with that $\mathcal{P}_c(r, \ldots, v)$ is the shortest path between r and v.

  Hence, $d_c(p_{last}, v) = d(p_{last}, v)$. Then $d_c(r, v) = d_c(r, p_{last}) + d_c(p_{last}, v)$. Since $r, p_{last} \in G'.V$, $d_c(r, p_{last})$ is returned by ARefine. Obviously, since the shortest path of p_{last} and v are all in the public graph,

$\hat{d}(p_{last}, v) \leq (2c-1)d(p_{last}, v)$ because of Lemma V.1. As a consequence,

$$\begin{aligned}\mathsf{match}'[q].\mathsf{d} &= d_c(r, p_{last}) + \hat{d}(p_{last}, v)\\ &\leq (2c-1)d_c(r, p_{last}) + (2c-1)d(p_{last}, v)\\ &= (2c-1)d_c(r, v)\\ &= (2c-1)\mathsf{match}[q].\mathsf{d}.\end{aligned}$$

Case 2: Suppose $r \in G.V$.

- ***Case 2.1:*** If $v \in G'.V$, the shortest path between r and v is denoted by $\mathcal{P}_c(r, \ldots, v)$. We denote the first portal node in $\mathcal{P}_c(r, \ldots, v)$ as p_{first}. Then $d_c(r, v) = d_c(r, p_{first}) + d_c(p_{first}, v)$. Since $p_{first}, v \in G'.V$, $d_c(p_{first}, v)$ is returned by ARefine. $d_c(r, p_{first}) = d(r, p_{first})$ since the nodes on the shortest path of p_{last} and v are all in the public graph (otherwise, p_{first} is not the first portal node in $\mathcal{P}_c(r, \ldots, v)$). $d(r, p_{first})$ can be computed by a breadth-first traversal that starts from p_{first} (denoted by $\mathcal{T}$). Consider the breadth-first traversal (denoted by $\mathcal{T}_i$) that starts from the portal node p_i, where, $p_i \neq p_{first}$, which visits r.
  - If $p_i \in \mathcal{P}_c(r, \ldots, v)$, $d(r, p_i) \geq d_c(r, p_i)$. Since $p_{first}, p_i \in \mathcal{P}_c(r, \ldots, v)$, we have $d_c(r, p_i) = d_c(r, p_{first}) + d_c(p_{first}, p_i)$. Hence, returned by $\mathcal{T}_i$, $d_c(r, v) = d(r, p_i) + d_c(p_i, v)$, which is larger than that returned by $\mathcal{T}$, since $d(r, p_i) + d_c(p_i, v) \geq d(r, p_{first}) + d_c(p_{first}, p_i) + d_c(p_i, v)$.
  - If $p_i \notin \mathcal{P}_c(r, \ldots, v)$, the subpath between r an p_i returned by $\mathcal{T}_i$, $\mathcal{P}_c(r, \ldots, p_i)$, is not a shorter path between r and v. Otherwise, it is contradicted with that $\mathcal{P}_c(r, \ldots, v)$ is the shortest path between r and v.

  Similarly, we have $v = v'$.
- ***Case 2.2:*** If $v \in G.V$, the proof is similar to ***Case 1.2***.

□

Lemma A.4. *If $u \in V'$ belongs to the answer of a k-nk query (v, q, k) on G_c, where $v \in V'$, then u is returned by* PP-knk.

Proof. We denote the set of vertices containing q as V_q. Given two vertex $u_1, u_2 \in V_q$. Without loss of generality, we assume that $u_1 \in V'$ and $d_c(v, u_1) \leq d_c(v, u_2)$. Then the ranking of u_1 is higher than u_2. It is worth noting that the exact value of $d_c(v, u_1)$ is returned by PPKWS. Next, we prove that the ranking is hold in PPKWS.

- If $u_2 \in V$, then $d_c(v, u_2) \leq \hat{d}(v, u_2)$ since $\hat{d}(v, u_2)$ is always larger than $d_c(v, u_2)$, returned by PPKWS. Naturally, the ranking of u_1 is still higher than that of u_2 since $d_c(v, u_1) \leq d_c(v, u_2) \leq \hat{d}(v, u_2)$.
- If $u_2 \in V'$, since the exact value of $d_c(v, u_2)$ is also returned in the context of PPKWS, $d_c(v, u_1) \leq d_c(v, u_2)$ is still hold. The ranking of u_1 is still higher than that of u_2.

Hence, $\forall u \in V'$ is an answer in G_c, u is returned by PPKWS since its ranking is hold in the context of PPKWS. □

TABLE VII: Complexity of PPKWS

Algorithms	PEval	ARefine	AComplete
PP-r-clique	Same as [14]	$O(\|A'\|\|Q\|\|\mathbb{P}\|^2)$	$O(\|A\|\|Q\|k\ln\|V\|)$
PP-Blinks	Same as [10]	$O(\|A'\|\|Q\|\|\mathbb{P}\|^2)$	$O(m_1\|\mathbb{P}\|\|Q\|+\|A\|\|Q\|k\ln\|V\|)$
PP-knk	Same as [13]	$O(m_2\|\mathbb{P}\|^2)$	$O(\|\mathbb{P}\|k\ln\|V\|)$

TABLE VIII: Query models with their descriptions

Query models	Description
M_1	Users issue queries on the public graph and private graph individually.
M_2	Users issue queries on *the combined graph* directly.
M_3	Users issue queries by PPKWS.

C. Complexity analysis

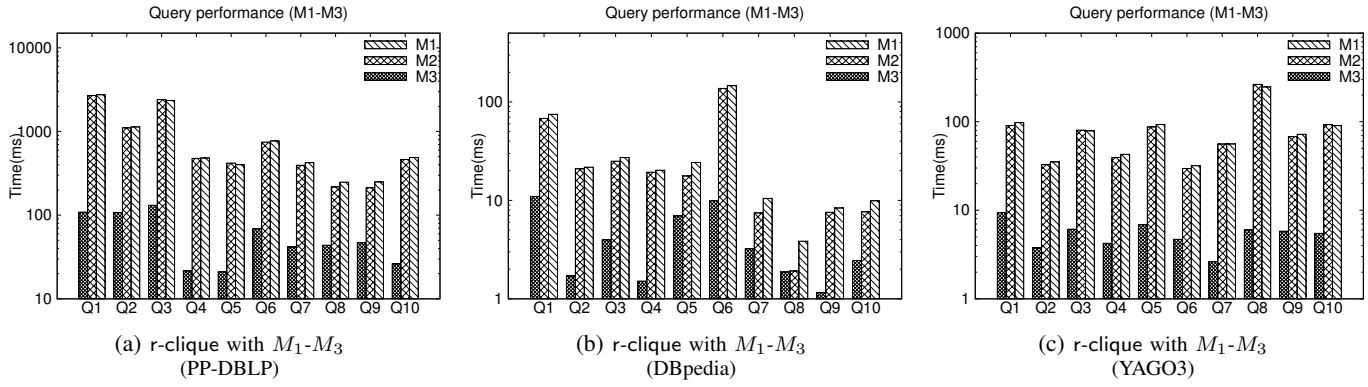
In this section, we analyse the time complexity of estimating the shortest distance in public graph. Moreover, we also present the complexity of each key step of PP-knk, PP-r-clique and PP-knk as shown in Tab. VII.

1) Complexity of the shortest distance estimation.: Given two vertices v_1 and v_2 and they corresponding PADS, the time of shortest distance estimation is $\min\{|\mathsf{PADS}(v_1)|, |\mathsf{PADS}(v_2)|\}$ on average. Since finding a element in a hashset can be finished in a constant time on average, the estimation cost is $O(k\ln|V|)$, where k is a parameter set by user. A larger k will bring larger sketches as well as more accuracy. Similarly, the time of estimating the shortest distance between a keyword t and a vertex v is $O(\min\{|\mathsf{KPADS}(t)|, |\mathsf{PADS}(v)|\})$. In general, the size of $|\mathsf{KPADS}(t)|$ is much larger than $|\mathsf{PADS}(v)|$. Hence, the cost is $O(k\ln|V|)$, too.

2) Complexity of the PP-r-clique*.:* PEval applies the keyword search algorithm of [14] with an answer qualification function which can be finished by a linear scanning, bounded by $O(V')$. Hence, PEval inherits the complexity of r-clique (cf. [14]). ARefine is in $O(|A'||C||\mathbb{P}^2|)$ since refining each partial answer takes $O(|C||\mathbb{P}|^2)$. It is bounded by $O(|A'||Q||\mathbb{P}|^2)$. AComplete is in $O(|A||Q|k\ln|V|)$.

3) Complexity of the PP-Blinks*.:* PEval applies the keyword search algorithm of [10] with an answer qualification function which can be finished by a linear scanning, bounded by $O(V')$. Hence, PEval inherits the complexity of Blinks (cf. [10]). ARefine is in $O(|A'||C||\mathbb{P}^2|)$ since refining each partial answer takes $O(|C||\mathbb{P}|^2)$. It is bounded by $O(|A'||Q||\mathbb{P}|^2)$. AComplete is in $O(m_1|\mathbb{P}||Q|+|A||Q|k\ln|V|)$. The backward expansion on the public graph takes $O(m_1|\mathbb{P}||Q|)$ where m_1 is the average number of the nodes within the x hops of the portals. For each visited node, it takes $O(|Q|)$ to update the distance information (Lines 15-17). AComplete takes $O(|A||Q|k\ln|V|)$ to retrieve the missing keywords of each answer (Lines 20-22).

4) Complexity of the PP-knk*.:* PEval applies the keyword search algorithm of [14] without any changes. Hence, PEval inherits the complexity of k-nk (cf. [13]). ARefine is in $O(m_2|\mathbb{P}^2|)$ where $m_2 = |a'.\mathsf{match}|$ since refining each partial answer takes $O(|\mathbb{P}|^2)$. AComplete is in $O(|\mathbb{P}|k\ln|V|)$.



(a) r-clique with M_1-M_3 (PP-DBLP) (b) r-clique with M_1-M_3 (DBpedia) (c) r-clique with M_1-M_3 (YAGO3)

(d) Blinks with M_1-M_3 (PP-DBLP) (e) Blinks with M_1-M_3 (DBpedia) (f) Blinks with M_1-M_3 (YAGO3)

Fig. 7: Performance under different query models

D. Other query models

In some applications, users might have some other query requirements or be interested not only in the answers spanning both on the public graph and the private graphs. For completeness, we also list some other query models as follows. The summary is presented in the Tab. VIII.

M_1 : $\mathsf{eval}(G, Q, f) \oplus \mathsf{eval}(G', Q, f)$. In this scenario, users issues queries on the public and private graphs individually. The public answers and private answers are obtained. As we mentioned in Sec. III, M_1 query model can be simulated on PPKWS as follows. PEval takes the public graph (resp. the private graph), the keyword query and algorithm as input. For the answers spanning both the public graph and private graphs, the users can simply adopt the M_3 model (below).

M_2 : $\mathsf{eval}(G_c, Q, f)$. In this scenario, queries are directly issued on *the combined graph*. For example, business analysts may focus on the collaborations from a social graph. They may search on *both* the public collaboration graph and their private collaboration graphs. Baseline-Blinks, Baseline-r-clique and Baseline-knk, all belong to this query model.

M_3 is the PPKWS framework. PP-Blinks, PP-r-clique and PP-knk, presented in this paper, all belong to this query model.

Performance analysis. M_1 and M_2 inhert the complexity of the original keyword search algorithms. We report the query performance under different query models in Fig. 7. The query performances of M_1 and M_2 are slightly different. M_3 improves the query time by around 110 times on average. Since the query points of k-nk under M_2 and M_3 are in the private graph, the queries are different with those of k-nk under M_1. Hence, we omit the comparison experiments between M_1 and M_2 or M_3 of k-nk. The average query time of M_1 is closed to that of M_2.

E. The qualities of the query answers of PPKWS

In this section, we show the quality guarantees of the query answers of various query semantics on PPKWS. We show the theoretical bounds of PP-r-clique, PP-Blinks and PP-knk, respectively.

Theorem A.5. PP-r-clique *finds an* r-clique *with* $(l-1)$*-approximation, where* l *is the number of the query keywords, i.e.,* $l = |Q|$.

Proof. We prove PPKWS can return the $(l-1)$-approximate answer of r-clique, $\langle v, \mathsf{match}\rangle$. We denote the optimal r-clique as a_{opt}, and the greedy r-clique as a_{grdy}. And we use $u_i = a_{opt}.\mathsf{match}[q_i].\mathsf{u}$ (resp. $v_i = a_{grdy}.\mathsf{match}[q_i].\mathsf{u}$) to dentoe the keyword nodes in a_{opt} (resp. a_{grdy}). Moreover, we use the symbols $d^{opt}_{i,j} = d_c(u_i, u_j)$ (resp. $d_{i,j} = d_c(v_i, v_j)$) to denote the shortest distance between keyword nodes u_i and u_j (resp. v_i and v_j). We denote the weight of optimal (resp. greedy) r-clique as $W(a_{opt}) = \sum_{i=1}^{l}\sum_{j=1}^{l} d^{opt}_{i,j}$ (resp. $W(a_{grdy}) = \sum_{i=1}^{l}\sum_{j=1}^{l} d_{i,j}$).

Based on the Definition II.2 of the public-private answers, $\exists v_i$ such that $v_i \in G'.V$ and $\exists u_j$ such that $u_j \in G'.V$. We denote them as v_r and u_r, respectively.

Given any two keyword nodes v_i and v_j, the triangle inequality is kept. More specifically, we have the following formula:

$$d_{i,j} \leq d_{r,i} + d_{r,j} \tag{8}$$

Moreover, we have the weight of a_{grdy} as follows:

$$2 \times W(a_{grdy}) = \sum_{i=1}^{l} \sum_{j=1}^{l} d_{i,j} = 2 \times \sum_{i \neq r} d_{r,i} + \sum_{i \neq r} \sum_{j \neq r, j \neq i} d_{i,j}, \tag{9}$$

where $i, j \in (1, l)$.

Consider the worst case, we have:

$$\sum_{i \neq r} \sum_{j \neq r, j \neq i} d_{i,j} \leq \sum_{i \neq r} \sum_{j \neq r, j \neq i} (d_{r,i} + d_{r,j}) = 2 \times (l-2) \sum_{i \neq r} d_{r,i} \tag{10}$$

Consider the Eq. 9 and Eq. 10, we have:

$$2 \times W(a_{grdy}) \leq 2 \times \sum_{i \neq r} d_{r,i} + 2 \times (l-2) \sum_{i \neq r} d_{r,i} = 2 \times (l-1) \sum_{i \neq r} d_{r,i} \tag{11}$$

Next, we consider the weight of the optimal r-clique ans_{opt}:

$$2 \times W(a_{opt}) = \sum_{i=1}^{l} \sum_{j=1}^{l} d_{i,j}^{opt} \geq 2 \times \sum_{i \neq r} d_{r,i}^{opt} \geq 2 \times \sum_{i \neq r} d_{r,i} \tag{12}$$

Therefore, we have:

$$W(a_{opt}) \geq \sum_{i \neq r} d_{r,i} \tag{13}$$

In this case, a_{opt} and a_{grdy} are considered equal. Hence, Formula 13 is established.

Consider the Eq. 11 and Eq. 13, we have:

$$W(a_{grdy}) \leq (l-1) \sum_{i \neq r} d_{r,i} \leq (l-1) \times W(a_{opt}) \tag{14}$$

Consequently, the $(l-1)$ approximation ratio is satisfied.

□

Theorem A.6. *Given an answer of* PP-r-clique, $a = \langle v, \mathsf{match} \rangle$, $a.\mathsf{match}[q].\mathsf{d} \leq (2c-1) d_c(v, a.\mathsf{match}[q].\mathsf{u})$.

Proof. The proof is the same with Lemma A.3, ***Case 1.2***. □

Theorem A.7. PP-Blinks *finds* Blinks *answers with* $(2c-1)$*-approximation.*

Proof. The weight of a Blinks answer $a = \langle r, \mathsf{match} \rangle$ is denoted by $W(a) = \sum \mathsf{match}[q].\mathsf{d}$. We denote the answer returned by PP-Blinks as a' which is rooted at r. We denote the answer rooted at r and returned by applying Blinks on the combined graph as $a \in \mathsf{eval}(G \oplus G', Q, \mathsf{Blinks})$.

Next, we show that $W(a') \leq (2c-1)W(a)$. As we have proved in Lemma A.3, $\mathsf{match}'[q].\mathsf{d} \leq (2c-1)\mathsf{match}[q].\mathsf{d}$. Hence $W(a') = \sum \mathsf{match}'[q].\mathsf{d} \leq (2c-1) \sum \mathsf{match}[q].\mathsf{d} = (2c-1)W(a)$. □

Given a k-nk query (v, q, k), we denote the top k answers returned by PP-knk as $A' = \{a'\}$. And $a'.\mathsf{match}$ is sorted by the ascending order of $a'.\mathsf{match}[i].\mathsf{d}$. And we denote the answers returned by applying k-nk on the combined graph as $A = \{a\} = \mathsf{eval}(G \oplus G', Q, \mathsf{k\text{-}nk})$. And $a.\mathsf{match}$ is sorted by the ascending order of $a.\mathsf{match}[i].\mathsf{d}$.

Theorem A.8. *The distance of k-th element of the answer $a.\mathsf{match}$ returned by* PP-knk *is bounded with $(2c-1)$-approximation, i.e., $a'.\mathsf{match}[k].\mathsf{d} \leq (2c-1) a.\mathsf{match}[k].\mathsf{d}$.*

Proof. We consider the following two cases.

Case 1: Suppose $\forall i, a.\mathsf{match}[i].\mathsf{u} \in G'.V$, $a.\mathsf{match}[i].\mathsf{u}$ will be returned by PP-knk, *i.e.*, $a.\mathsf{match}[i] \in a'.\mathsf{match}$, as we have proved in Lemma A.4. Hence, $a.\mathsf{match} \subseteq a'.\mathsf{match}$. Since $|a'.\mathsf{match}| = |a.\mathsf{match}| = k$, $a'.\mathsf{match} = a.\mathsf{match}$. Hence $a'.\mathsf{match}[k] = a.\mathsf{match}[k]$. $a'.\mathsf{match}[k].\mathsf{d} \leq (2c-1) a.\mathsf{match}[k].\mathsf{d}$ is satisfied.

Case 2: Suppose $\exists i, a.\mathsf{match}[i].\mathsf{u} \notin G'.V$. It is worthing noting that $a.\mathsf{match}[i].\mathsf{d} \leq (2c-1) d_c(v, a.\mathsf{match}[i].\mathsf{u})$ by PP-knk (the proof is the same with Lemma A.3, ***Case 1.2***).

Since $a.\mathsf{match}[i].\mathsf{d} \leq a.\mathsf{match}[k].\mathsf{d}$, we have

$$(2c-1) a.\mathsf{match}[i].\mathsf{d} \leq (2c-1) a.\mathsf{match}[k].\mathsf{d} \tag{15}$$

We prove this theorem by contradiction. If $a'.\mathsf{match}[k].\mathsf{d} > (2c-1) a.\mathsf{match}[k].\mathsf{d}$, then

$$a'.\mathsf{match}[k].\mathsf{u}'.\mathsf{d} > (2c-1) a.\mathsf{match}[i].\mathsf{d} \tag{16}$$

where $i \in (1, k)$.

Hence $a'.\mathsf{match}[k]$ is not among the top-k nearest vertices returned by PP-knk. Therefore $a'.\mathsf{match}[k].\mathsf{d} \leq (2c-1) a.\mathsf{match}[k].\mathsf{d}$ is established. □