BLINKS: Ranked Keyword Searches on Graphs*

Hao He† Haixun Wang‡
† Duke University
Durham, NC 27708

Jun Yang† Philip S. Yu‡
‡ IBM T. J. Watson Research
Hawthorne, NY 10532

ABSTRACT

Query processing over graph-structured data is enjoying a growing number of applications. A top-k keyword search query on a graph finds the top k answers according to some ranking criteria, where each answer is a substructure of the graph containing all query keywords. Current techniques for supporting such queries on general graphs suffer from several drawbacks, e.g., poor worst-case performance, not taking full advantage of indexes, and high memory requirements. To address these problems, we propose *BLINKS*, a bi-level indexing and query processing scheme for top-k keyword search on graphs. BLINKS follows a search strategy with provable performance bounds, while additionally exploiting a bi-level index for pruning and accelerating the search. To reduce the index space, BLINKS partitions a data graph into blocks: The bi-level index stores summary information at the block level to initiate and guide search among blocks, and more detailed information for each block to accelerate search within blocks. Our experiments show that BLINKS offers orders-of-magnitude performance improvement over existing approaches.

Categories and Subject Descriptors: H.3.3 [Information Search and Retrieval]: Search process; H.3.1 [Content Analysis and Indexing]: Indexing methods.

General Terms: Algorithms, Design.

Keywords: keyword search, graphs, ranking, indexing.

1 Introduction

Query processing over graph-structured data has attracted much attention recently, as applications from a variety of areas continue to produce large volumes of graph-structured data. For instance, XML, a popular data representation and exchange format, can be regarded as graphs when considering IDREF/ID links. In Semantic Web, two major W3C standards, RDF and OWL, conform to node-labeled and edge-labeled graph models. In bioinformatics, many well-known projects, e.g., BioCyc (http://biocyc.org), build graph-structured databases. In other applications, raw data might not be graph-structured at the first glance, but there are many implicit connections among data items; restoring these connections often allows more effective and intuitive querying. For example, a number of projects [1, 17, 3, 23, 6] enable keyword search over relational databases, where tuples are treated as graph nodes connected via foreign-key relationships. In personal information management (PIM) systems [8, 4], objects such as emails, documents, and photos are interwoven into a graph using manually or automatically established connections among them. The list of examples of graph-structured data goes on.

Ranked keyword search over tree- and graph-structured data [1, 17, 3, 5, 14, 16, 2, 25, 18, 23, 21, 6] has attracted much attention recently for two reasons. First, this simple, user-friendly query interface does not require users to master a complex query language or understand the underlying data schema. Second, many graph-structured data have no obvious, well-structured schema, so many query languages are not applicable.

In this paper, we focus on implementing *efficient* ranked keyword searches on schemaless node-labeled graphs. On a large data graph, many substructures may contain the query keywords. Following the standard approach taken by other systems, we restrict answers to those connected substructures that are "minimal," and further provide scoring functions that rank answers in decreasing relevance to help users focus on the most interesting answers.

Challenges Ranked keyword searches on schemaless graphs pose many unique challenges. Techniques developed for XML [5, 14, 25], which take advantage of the hierarchical property of trees, no longer apply. Also, lack of schema precludes many optimization opportunities (such as in [1, 17]) at compile-time and makes efficient runtime search much more critical. Previous work in this area suffers from several drawbacks. First, many existing keyword search algorithms employ heuristic graph exploration strategies that lack strong performance guarantees and may lead to poor performance on certain graphs. Second, existing algorithms for general graphs do not take full advantage of indexing. They only use indexes for identifying the set of nodes containing query keywords; finding substructures connecting these nodes relies on graph traversal. For a system supporting a large, ongoing workload of keyword queries, we argue that it is natural and critical to exploit indexes that provide graph connectivity information to speed up searches. Lack of this feature can be attributed in part to the difficulty in indexing connectivity for general graphs, because a naive index would have an unacceptably high (quadratic) storage requirement. We discuss these issues in detail in Sections 3 and 9.

Contributions To overcome these difficulties, we propose ***BLINKS*** (***Bi-Level INdexing for Keyword Search***), an indexing and query processing scheme for ranked keyword search over node-labeled directed graphs. Our main contributions are the following:

*The first and third authors are supported by NSF CAREER award IIS-0238386, an IBM Ph.D. Fellowship, and an IBM Faculty Award.

SIGMOD'07, June 11–14, 2007, Beijing, China.

- **Better search strategy.** BLINKS is based on *cost-balanced expansion*, a new policy for the *backward search* strategy (which explores the graph starting from nodes containing query keywords). We show that this policy is optimal within a factor of m (the number of query keywords) of an "oracle" backward search strategy that magically knows how to determine the top k answers with minimum cost. This new strategy alleviates many problems of the original backward search proposed in [3].
- **Combining indexing with search.** BLINKS augments search with an index, which selectively precomputes and materializes some shortest-path information. This index significantly reduces the runtime cost of implementing the optimal backward search strategy. At the same time, this index enables *forward search* as well, effectively making the search *bidirectional*. Compared with the heuristic prioritization policy for bidirectional search proposed in [18], BLINKS is able to make longer and more directed forward jumps in search. To the best of our knowledge, BLINKS is the first scheme that exploits indexing extensively in accelerating keyword searches on general graphs.
- **Partitioning-based indexing.** A naive realization of an index that keeps all shortest-path information would be too large to store and too expensive to maintain for large graphs. Instead, BLINKS partitions a data graph into multiple subgraphs, or *blocks*: The bi-level index stores summary information at the block level to initiate and guide search among blocks, and more detailed information for each block to accelerate search within the block. This bi-level design allows effective trade-off between space and search efficiency through control of the blocking factor. BLINKS also addresses the problem of finding a graph partitioning that results in an effective bi-level index.

Experiments on real datasets show that BLINKS offers orders-of-magnitude performance improvement over existing algorithms. We also note that BLINKS supports sophisticated, realistic scoring functions based on both graph structure (e.g., node scores reflecting PageRank and edge scores reflecting connection strengths) and content (e.g., IR-style scores for nodes matching keywords).

The rest of the paper is organized as follows. We formally define the problem and describe our scoring function in Section 2. We review existing graph search strategies and propose the new *cost-balanced expansion* policy in Section 3. To help illustrate how indexing helps search, we present a conceptually simple (but practically infeasible) single-level index and the associated search algorithm in Section 4. In Sections 5 and 6, we introduce our full bi-level index and search algorithm. We discuss optimizations in Section 7 and present results of experiments in Section 8. Finally, we survey the related work in Section 9 and conclude in Section 10.

2 Problem Definition

Data and Query Similar to [3, 18], we are concerned with querying a directed graph $G = (V, E)$, where each node $v \in V$ is labeled with some text. For example, in the graph shown in Figure 1(A), node 9 contains two keywords $\{b, g\}$. A keyword search query q consists of a list of *query keywords* $(w_1, \ldots, w_m)$. We formally define an answer to q as follows:

DEFINITION 1. *Given a query $q = (w_1, \ldots, w_m)$ and a directed graph G, an answer to q is a pair $\langle r, (n_1, \ldots, n_m)\rangle$, where r and n_i's are nodes (not necessarily distinct) in G satisfying the following properties:*

(Coverage) *For every i, node n_i contains keyword w_i.*

(Connectivity) *For every i, there exists a directed path in G from r to n_i.*

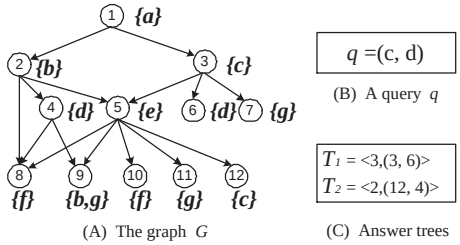

Figure 1: Example of query and answers.

We call r the *root* of the answer and n_i's the *matches* of the answer. The connectivity property requires that an answer must be a subtree whose root reaches all keywords. In Figures 1, for graph G and query $q = (c, d)$, we find two answers T_1 and T_2 shown in Figure 1(C).

Top-k Query In this paper, we are concerned with finding the top-ranked answers to a query. Answer goodness is measured by a *scoring function*, which maps an answer to a numeric score; the higher the score, the "better" the answer. We now give the semantics of a top-k query.

DEFINITION 2. *Given a query and a scoring function S, the* (best) score *of a node r is the maximum $S(T)$ over all answers T rooted at r (or 0 if there are no such answers). An answer rooted at r with the best score is called a* best answer *rooted at r. A top-k query returns the k nodes in the graph with the highest best scores, and, for each node returned, the best score and a best answer rooted at the node.*

Note that in the definition above, the k best answers have distinct roots. We have several reasons for choosing this *distinct-root semantics*. First, this semantics guards against the case where a "hub" node pointing to many nodes containing query keywords becomes the root for a huge number of answers. These answers overlap and each carries very little additional information from the rest. As a concrete example, suppose we search for "privacy," "mining," and "sensor" in a publication data graph with the intention of finding authors who publish in all three areas. Say an author has published n_1 papers with titles containing "privacy," n_2 papers containing "mining," and n_3 papers containing "sensor." This author would be the root of $n_1 \times n_2 \times n_3$ answers; if $n_1 \times n_2 \times n_3$ is close to k, the top k answers would not be very informative. Granted, there might be times when we are actually interested more in the combination of three papers than in the author, but accommodating such cases is not difficult: Given an answer (which is the best, or one of the best, at its root), users can always choose to further examine other answers with this root.

A second reason for the distinct-root semantics is more technical: It enables more effective indexing. We defer more discussion of this point to Section 7.

Scoring Function Many scoring functions have been proposed in the literature (e.g., [3, 5, 14, 16, 18, 13, 23, 21]). Since our primary focus is indexing and query processing, we will not delve into the specifics here. Instead, we provide a general definition for our scoring function, and discuss the features and properties relevant to efficient search and indexing.

Our scoring function considers both graph structure and content, and incorporates several state-of-the-art measures developed by database and IR communities. Formally, we define the score of an answer $T = \langle r, (n_1, \ldots, n_m)\rangle$ for query $(w_1, \ldots, w_m)$ as $S(T) = f(\bar{S}_r(r) + \sum_{i=1}^{m} \bar{S}_n(n_i, w_i) + \sum_{i=1}^{m} \bar{S}_p(r, n_i))$, where $\bar{S}_p(r, n_i)$ denotes the shortest-path distance from root r to match n_i

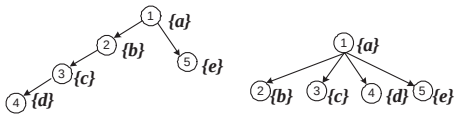

Figure 2: Answers with different tree shapes.

based on some non-negative graph distance measure. The input to $f(\cdot)$ is the sum of three score components, which respectively capture the contribution to the score from (1) the answer root, (2) the matches, and (3) the paths from the answer root to the matches. The component score functions $\bar{S}_r$, $\bar{S}_n$, and $\bar{S}_p$ incorporate measures based on both graph structure (e.g., node scores reflecting PageRank and edge distances reflecting connection strengths) and content (e.g., IR-style TF/IDF scores for matches).

Our scoring function has two properties worth mentioning:

- *Match-distributive semantics.* In the definition of $S(T)$ above, the net contribution of matches and root-match paths to the final score can be computed in a distributive manner by summing over all matches. Consequently, all root-match paths contribute independently to the final score, even if these paths may share some common edges. Our semantics agrees with [18] but contrasts with some of other systems, e.g., [3, 21, 6]. Those systems score an answer by the total edge weight of the answer tree spanning all matches; therefore, each edge weight is counted only once, even if the edge participates in multiple root-match paths. For example, these systems would rank the two answers in Figure 2 equally (assuming identical distances along all edges), whereas our scoring function would prefer the answer on the right, as the connection between its root and matches is intuitively tighter. These two semantics also have very different implications on the complexity of search and indexing, which we discuss further in Section 7.
- *Graph-distance semantics.* In the definition of $S(T)$ above, the score contribution of a root-match path, $\bar{S}_p(r, n_i)$, is defined to be shortest-path distance from the root to the match in the data graph, where edges have non-negative distances. This semantics, also used by [18], for example, is intuitive and clean, and allows us to reduce part of the keyword search problem to the classic shortest-path problem. Most of our algorithms and data structures assume this semantics; we point out some exceptions in Section 4, where this assumption is not required.

An Assumption for Convenience For simplicity of presentation, we ignore for now the root and match components of the score, and focus only on $\sum_{i=1}^{m} \bar{S}_p(r, n_i)$, the contribution from the root-match paths. Given non-negative distance assignments for edges in the data graph, our problem now reduces to that of finding k nodes, where each node can reach all query keywords and the sum of its graph distances to these keywords—which we call the *combined distance* (of the node to query keywords)—is as small as possible.

This assumption is for convenience only and does not affect the generality of our results. We discuss how incorporate the root and match components back into the scoring function in Section 7.

3 Towards Optimal Graph Search Strategies

In this section, we discuss the search strategy of BLINKS on a high level and compare it qualitatively with previous approaches.

Backward Search In the absence of any index that can provide graph connectivity information beyond a single hop, we can answer the query by exploring the graph starting from the nodes containing at least one query keyword—such nodes can be identified easily through an inverted-list index. This approach naturally leads to a *backward search* algorithm, which works as follows.

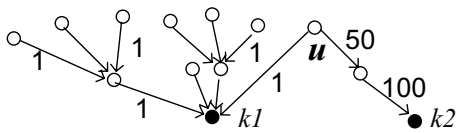

Figure 3: An example where distance-balanced expansion across clusters performs poorly.

1. At any point during the backward search, let E_i denote the set of nodes that we know can reach query keyword k_i; we call E_i the *cluster* for k_i.
2. Initially, E_i starts out as the set of nodes O_i that directly contain k_i; we call this initial set the *cluster origin* and its member nodes *keyword nodes*.
3. In each search step, we choose an incoming edge to one of previously visited nodes (say v), and then follow that edge *backward* to visit its source node (say u); any E_i containing v now expands to include u as well. Once a node is visited, all its incoming edges become known to the search and available for choice by a future step.
4. We have discovered an answer root x if, for each cluster E_i, either $x \in E_i$ or x has an edge to some node in E_i.

The first backward keyword search algorithm was proposed by Bhalotia et al. [3]. Their algorithm uses the following two strategies for choosing what to visit next. For convenience, we define the distance from a node n to a set of nodes N as the shortest distance from n to any node in N.

- ***Equi-distance expansion in each cluster***: This strategy decides which node to visit for expanding a keyword. Intuitively, the algorithm expands a cluster by visiting nodes in order of increasing distance from the cluster origin. Formally, the node u to visit next for cluster E_i (by following edge $u \rightarrow v$ backward, for some $v \in E_i$) is the node with the shortest distance (among all nodes not in E_i) to O_i.
- ***Distance-balanced expansion across clusters***: This strategy decides the frontier of which keyword will be expanded. Intuitively, the algorithm attempts to balance the distance between each cluster's origin to its frontier across all clusters. Specifically, let (u, E_i) be the node-cluster pair such that $u \notin E_i$ and the distance from u to O_i is the shortest possible. The cluster to expand next is E_i.

Bhalotia et al. [3] did not discuss the optimality of the above two strategies. Here, we offer, to the best of our knowledge, the first rigorous investigation of their optimality. First, we establish the optimality of equi-distance expansion within each cluster (the proof can be found in [15]).

THEOREM 1. *An optimal backward search algorithm must follow the strategy of equi-distance expansion in each cluster.*

On the other hand, the second strategy employed in [3], distance-balanced expansion across clusters, may lead to poor performance on certain graphs. Figure 3 shows one such example. Suppose that $\{k_1\}$ and $\{k_2\}$ are the two cluster origins. There are many nodes that can reach k_1 with short paths, but only one edge into k_2 with a large weight (100). With distance-balanced expansion across clusters, we would not expand the k_2 cluster along this edge until we have visited all nodes within distance 100 to k_1. It would have been unnecessary to visit many of these nodes had the algorithm chosen to expand the k_2 cluster earlier.

Bidirectional Search To address the above problem, Kacholia et al. [18] proposed a *bidirectional search* algorithm, which has the option of exploring the graph by following forward edges as well.

The rationale is that, for example, in Figure 3, if the algorithm is allowed to explore forward from node u towards k_2, we can identify u as an answer root much faster. To control the expansion order, Kacholia et al. prioritize nodes by heuristic *activation factors*, which intuitively estimate how likely nodes can be answer roots. While this strategy is shown to perform well in multiple scenarios, it is difficult to provide any worst-case performance guarantee. The reason is that activation factors are heuristic measures derived from general graph topology and parts of the graph already visited; they may not accurately reflect the likelihood of reaching keyword nodes through an unexplored region of the graph within a reasonable distance. Without additional connectivity information, forward expansion may be just as aimless as backward expansion.

Our Approach Is there any hope of having a simple search strategy with good performance guarantees? We answer in the affirmative with a novel approach based on two central ideas: First, we propose a new, *cost-balanced* strategy for controlling expansion across clusters, with a provable bound on its worst-case performance. Second, we use indexing to support forward jumps in search. Indexing allows us to determine whether a node can reach a keyword and what the shortest distance is, thereby eliminating the uncertainty and inefficiency of step-by-step forward expansion. The use of indexing will be discussed in detail in following sections; here, we describe our new cost-balanced expansion strategy and prove its optimality.

- ***Cost-balanced expansion across clusters***: Intuitively, the algorithm attempts to balance the number of accessed nodes (i.e., the search cost) for expanding each cluster. Formally, the cluster E_i to expand next is the cluster with the smallest cardinality.

This strategy is intended to be combined with the equi-distance strategy for expansion within clusters: Once we choose the smallest cluster to expand, we then choose the node with the shortest distance to this cluster's origin.

To establish the optimality of an algorithm A employing these two expansion strategies, we consider an optimal "oracle" backward search algorithm P. As shown in Theorem 1, P must also do equi-distance expansion within each cluster. However, in addition, we assume that P "magically" knows the right amount of expansion for each cluster such that the total number of nodes visited by P is minimized. Obviously, P is better than the best practical backward search algorithm we can hope for. Although A does not have the advantage of the oracle algorithm, we show in the following theorem that A is m-optimal, where m is the number of query keywords (the complete proof can be found in [15]). Since most queries in practice contain very few keywords, the cost of A is usually within a constant factor of the optimal algorithm.

THEOREM 2. *The number of nodes accessed by A is no more than m times the number of nodes accessed by P, where m is the number of query keywords.*

In following sections, we describe the top-k keyword search algorithms that leverage the new search strategy (*equi-distance* plus *cost-balanced* expansions) as well as indexing to achieve good query performance.

4 Searching with a Single-Level Index

Before presenting the full BLINKS, we first describe a conceptually simple scheme to help illustrate benefits of our search strategy and indexing. This scheme works well for small graphs, but is not practical on large graphs. The full BLINKS, presented in Sections 5 and 6, is designed to scale on large graphs.

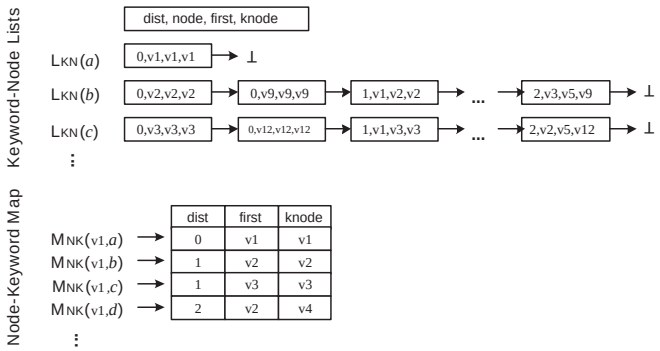

Figure 4: Keyword-node lists and node-keyword map.

4.1 A Single-Level Index

Motivation and Index Structure For each cluster E_i, the standard way of implementing equi-distance backward expansion is to maintain a priority queue of nodes ordered by their distances from keyword k_i. The queue represents a "frontier" in exploring k_i, which may grow exponentially in size even for sparse graphs. The time complexity is also high, as it takes $O(\log n)$ time to find the highest-priority node, where n is the size of the queue. Our goal is to reduce the space and time complexity of search.

A common approach to enhance online performance is to perform some offline computation. We pre-compute, for each keyword, the shortest distances from every node to the keyword (or, more precisely, to any node containing this keyword) in the data graph. The result is a collection of ***keyword-node lists***. For a keyword w, $L_{KN}(w)$ denotes the list of nodes that can reach keyword w, and these nodes are ordered by their distances to w. Each entry in the list has four fields (dist, node, first, knode), where dist is the shortest distance between node and a node containing w; knode is a node containing w for which this shortest distance is realized; first is the first node on the shortest path from node to knode.[1] In Figure 4, we show some parts of the keyword-node lists built for the graph in Figure 1 (assuming all edges have weight 1). As an example, in the list for keyword b, the first entry is $(0, v_2, v_2, v_2)$, which reflects the fact that v_2 can reach the keyword b with distance 0 and first and knode happen to be v_2 itself. The last entry $(2, v_3, v_5, v_9)$ reflects the fact that the shortest path from v_3 to b is $v_3 \rightarrow v_5 \rightarrow v_9$ with distance 2.

Furthermore, as motivated in Section 3, we would like to augment backward search with forward expansion, so that we can find answers faster. In previous approaches, forward expansion follows node-by-node graph exploration with little guidance. Can forward expansion be made faster and more informed?

We pre-compute, for each node u, the shortest graph distance from u to every keyword, and organize this information in a hash table called ***node-keyword map***, denoted M_{NK}. Given a node u and a keyword w, $M_{NK}(u, w)$ returns the shortest distance from u to w, or ∞ if u cannot reach any node that contains w. The hash entry for (u, w) can contain, in addition to dist (the shortest distance), first and knode, which are defined identically as in L_{KN} and used for the same purposes. Figure 4 also shows the node-keyword map. In fact, the information in $M_{NK}(u, w)$ can be derived from $L_{KN}(w)$. However, it takes linear time to search $L_{KN}(w)$ for the shortest distance between u and w, while with $M_{NK}(u, w)$, the operation can be completed in practically $O(1)$ time.

We call the duo of keyword-node lists and node-keyword map a ***single-level index*** because the index is defined over the entire

[1]The knode field is useful in locating matches in answer, while first is useful in reconstructing root-match paths; see Section 7.

data graph (as opposed to the bi-level index to be introduced in Section 5). It is easy to see that both the keyword-node lists and the node-keyword map contain as many as $N \cdot K$ entries, where N is the number of nodes, and K is the number of distinct keywords in the graph. In many applications, K is on the same scale as the number of nodes, so the space complexity of the index comes to $O(N^2)$, which is clearly infeasible for large graphs. The bi-level index we propose in Section 5 addresses this issue.

Index Construction The single-level index can be populated by backward expanding searches starting from keywords. To compute the distances between nodes and keywords, we concurrently run N copies of Dijkstra's single source shortest path algorithm in a backward expanding fashion, one for each of the N nodes in the graph. This process is similar to the keyword query algorithm given by BANKS [3], except that we are creating an index instead of answering online queries. We omit the detailed algorithm here. Note that the time complexity of this algorithm is $O(N^2)$, which is high for large graphs. Our results in Section 5 also reduce this complexity.

The single-level index can be used for any scoring function with distinct-root and match-distributive semantics. However, the index construction algorithm outlined above additionally assumes the graph-distance semantics (cf. Section 2).

4.2 Search Algorithm with Single-Level Index

We present searchSLINKS, the algorithm for searching with single-level index, in Algorithm 1. This algorithm assumes a scoring function with distinct-root and match-distributive semantics; it does not assume the graph-distance semantics (cf. Section 2).

Expanding Backward Given a query $(w_1, \cdots, w_m)$, we use a *cursor* to traverse each keyword-node list $L_{KN}(w_i)$. Cursor c_i advances on list $L_{KN}(w_i)$ by calling next (Line 6), which returns the next node in the list, together with its shortest distance to keyword w_i. By construction, the list gives the equi-distance expansion order in each cluster. Across clusters, we pick a cursor to expand next in a round-robin manner (Line 5), which implements cost-balanced expansion among clusters. These two together ensure optimal backward search.

Expanding Forward In addition, we use the node-keyword map M_{NK} for forward expansion in a direct fashion. As soon as we visit a node, we look up its distance to the other keywords (Line 17). Using this information we can immediately determine if we have found the root of an answer. More specifically, for each node we visit we maintain a structure $\langle \mathsf{root}, \mathsf{dist}_1, \mathsf{dist}_2, \ldots, \mathsf{dist}_m \rangle$, where root is the node visited, and dist_i is the distance from the node to keyword w_i. If any dist_i is ∞, then root cannot possibly be the root of an answer, because it cannot reach w_i. On the other hand, if none of $\mathsf{dist}_1, \cdots, \mathsf{dist}_m$ is ∞, we know we have an answer (Line 18, where $\mathsf{sumDist}(u)$ is the combined distance from u to keywords computed as $\sum_i^m \mathsf{dist}_i$).

Stopping How do we know we have found all top k answers? We maintain a pruning threshold τ_{prune}, which is the current k-th shortest combined distance among all known answer roots (provided that there are at least k answers). For a new answer to be in the top k, its root must have combined distance no greater than τ_{prune}. Meanwhile, due to equi-distance expansion in each cluster, we know that any unvisited node will have combined distance of at least $\sum_{j=1}^m c_i.\mathsf{peekDist}()$, where peekDist() is the next distance to be returned by a cursor. If this lower bound exceeds τ_{prune}, we can stop the search (Line 8).

Algorithm 1: Searching with the single-level index.

```
   Variables: R: nodes visited; initially ∅.
              A: answers found; initially ∅.
              τ_prune: pruning threshold; initially ∞.
 1 searchSLINKS(w_1, ..., w_m) begin
 2     foreach i ∈ [1, m] do
 3         c_i ← new Cursor(L_KN(w_i), 0);
 4     while ∃j ∈ [1, m] : c_j.peekDist() ≠ ∞ do
 5         i ← pick from [1, m] in a round-robin fashion;
 6         ⟨u, d⟩ ← c_i.next();
 7         if ⟨u, d⟩ ≠ ⟨⊥, ∞⟩ then visitNode(i, u, d);
 8         if |A| ≥ k and Σ_{j=1}^m c_i.peekDist() > τ_prune then
 9             exit and output the top k answers in A;
10     output up to top k answers in A;
11 end
12 visitNode(i, u, d) begin
13     if R.contains(u) then return;      // already visited
14     R.add(⟨u, ⊥, ..., ⊥⟩);
15     R[u].dist_i ← d;
16     foreach j ∈ [1, i) ∪ (i, m] do      // expand forward
17         R[u].dist_i ← M_NK(u, w_i);
18     if sumDist(u) < ∞ then             // answer found
19         A.add(R[u]);
20         if |A| ≥ k then
               τ_prune ← the k-th largest of {sumDist(v) | v ∈ A}
21 end
```

Discussion Compared with previous work that does not use index, searchSLINKS finds the top k answers in a time- and space-efficient manner: (1) The current state of graph exploration is managed by m cursors instead of m priority queues. (2) Finding the next node to explore is much faster from a cursor than from a priority queue. (3) Forward expansion using the node-keyword map allows the search to converge on answers faster, which also translates to earlier stopping.

Connection to the Threshold Algorithm Keen readers might have noticed a resemblance between searchSLINKS and the *Threshold Algorithm* (TA) proposed by Fagin et al. [9]. TA arises in the context of finding objects with top overall scores, which are computed over m scores, one for each of m attributes. TA assumes that for each attribute, there is a list of objects and their scores under that attribute, sorted in descending score order. TA finds the top k objects by visiting the m sorted lists in parallel, and performing random accesses on the lists to find scores for other attributes. TA has been proven optimal in terms of number of objects visited, assuming the aggregate function that combines the m scores is *monotone* [9].

With the single-level index, the keyword search problem can be framed as one addressed by TA. Here, each object corresponds to a node in the graph, and an object's score under an attribute corresponds to the shortest distance between the node and a keyword. Algorithm searchSLINKS conducts (1) equi-distance expansion for each keyword, and (2) cost-balanced expansion across keywords. Clearly, (1) is embodied by the problem definition of TA, where lists are sorted, and (2) is embodied by the fact that TA visits the lists in parallel. According to our analysis in Section 3, a keyword search algorithm is optimal if it follows (1) and (2). Although we had arrived at this optimality result for general keyword search without assuming indexing, our conclusion coincides with the optimality of the TA algorithm when a single-level index is used.

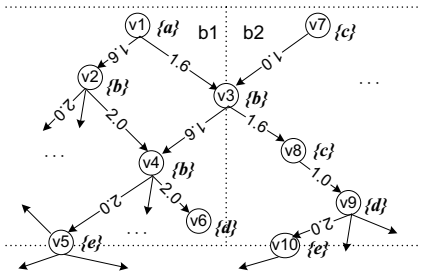

Figure 5: Example of portals and blocks.

5 Bi-Level Indexing in BLINKS

As motivated in Section 4, a naive realization of the single-level index, which includes the *keyword-node lists* and the *node-keyword map*, is impractical for large graphs: the index is too large to store and too expensive to construct. To address this problem, BLINKS uses a divide-and-conquer approach to create a bi-level index.

BLINKS partitions a data graph into multiple subgraphs, or *blocks*. A *bi-level index* consists of a top-level *block index*, which stores the mapping between keywords and nodes to blocks, and an *intra-block index* for each block, which stores more detailed information within a block. We show that the total size of the bi-level index is a fraction of that of a single-level index.

We discuss the intra-block index in Section 5.1, and the block index in Section 5.2. To create the bi-level index, we need to first decide how to partition the graph into blocks; the partitioning strategy is presented in Section 5.3. Before we proceed, however, we need to introduce the concept of *portal* nodes in order to clarify what we mean by partitioning of graph into blocks.

Partitioning by Portal Nodes Graph partitioning has been studied for decades in many fields. One can partition a graph by edge separators or node separators. In either case, we need to maintain the set of separators in order to handle the case where an answer in general may span multiple partitions. We choose node-based partitioning for two reasons. (1) The total number of separators is much smaller for node-based partitioning than edge-based partitioning. Therefore, there is less information to store for separators, and during search, we need to cross fewer separators, which is more efficient. (2) Our keyword search strategy considers nodes as the basic unit of expansion, so using node-based partitioning makes the implementation easier.

DEFINITION 3. *In a node-based partitioning of a graph, we call the node separators* portal nodes *(or* portals *for short). A* block *consists of all nodes in a partition as well as all portals incident to the partition. For a block, a portal can be either "in-portal" or "out-portal" or both.*

- *In-portal: it has at least one incoming edge from another block and at least one outgoing edge in this block.*
- *Out-portal: it has at least one outgoing edge to another block and at least one incoming edge from this block.*

This definition can be illustrated by an example in Figure 5. The dotted line represents the boundary of blocks. Node v_3, v_5 and v_{10} are hence portal nodes, and they appear in the intra-block index of all blocks they belong to. For block b_1, v_5 is an out-portal. Imagine we are doing backward expansion across blocks. Through v_5, we can only expand search from other blocks back into block b_1. However, v_3 is both in-portal and out-portal for both blocks b_1 and b_2; the expansion can go both ways.

5.1 Intra-Block Index

In this section, we describe the intra-block index (IB-index), which indexes information inside a block. For each block b, the IB-index consists of the following data structures:

- ***Intra-block keyword-node lists***: For each keyword w, $L_{KN}(b, w)$ denotes the list of nodes in b that can reach w without leaving b, sorted according to their shortest distances (within b) to w (or more precisely, any node in b containing w).
- ***Intra-block node-keyword map***: Looking up a node $u \in b$ together with a keyword w in this hash map returns $M_{NK}(b, u, w)$, the shortest distance (within b) from u to w (∞ if u cannot reach w in b).
- ***Intra-block portal-node lists***: For each out-portal p of b, $L_{PN}(b, p)$ denotes the list of nodes in b that can reach p without leaving b, sorted according to shortest distances (within b) to p.
- ***Intra-block node-portal distance map***: Looking up a node $u \in b$ in this hash map returns $D_{NP}(b, u)$, the shortest distance (in b) from a node u to the closest out-portal of b (∞ if u cannot reach any out-portal of b).

We next describe these data structures in more detail.

Keyword-Node Lists and Node-Keyword Map Structures L_{KN} and M_{NK} are identical to those introduced in Section 4, with the only difference that they are restricted to a block. Partitioning implies that the shortest distances stored in L_{KN} and M_{NK} are not necessarily globally the shortest. For example, $M_{NK}(b, u, w) = \infty$ means there is no path local to block b from u to a node in b containing w; however, it is still possible for u to reach some keyword node outside b, or even for u to reach some keyword node inside b through some path that leaves b and then comes back. Clearly, the local information about shortest paths cannot be used directly for finding top-k answers.

We can use the same procedure for building the single-level index in Section 4.1 to build the intra-block keyword-node lists and the node-keyword map. Instead of the entire graph, the procedure simply operates on each block. For block b, the data structures are of size $O(N_b \cdot K_b)$, where N_b is the number of nodes in the block, and K_b is the number of keywords that appear in the block. With the assumption $K_b = O(N_b)$, the index size comes to $O(N_b^2)$. In practice, the number of entries is likely to be much smaller than N_b^2, as not every node and every keyword are connected.

Portal-Node Lists and Node-Portal Distance Map The L_{PN} lists are similar to the L_{KN} lists. The difference is that L_{PN} stores the shortest path information between nodes and out-portal nodes, instead of between nodes and keywords. For an out-portal $p \in b$, each entry in $L_{PN}(b, p)$ consists of fields $(\mathsf{dist}, \mathsf{node}, \mathsf{first})$, where dist is the shortest distance from node to p, and first is the first node on the shortest path. For example, in Figure 6, v_3 is an out-portal and can reach another portal v_5 through the shortest path $v_3 \rightarrow v_4 \rightarrow v_5$ with the corresponding entry $[3.6, v_3, v_4]$.

The primary purpose of L_{PN} is to support cross-block backward expansion in an efficient manner, as an answer may span multiple blocks through portals. Since we search mainly in the backward direction, we do not index connectivity between in-portals and nodes.

The D_{NP} map gives the shortest distance between a node and its closest out-portal within a block. This distance is used by the search algorithm (to be discussed in Section 6) in lower bounding node-keyword distances, which are useful in pruning.

L_{PN} can be constructed simply by running a standard single-source shortest-path algorithm from each out-portal of a block. Information in D_{NP} can be easily computed with the results of the shortest-path algorithms. The L_{PN} lists for a block b have a total

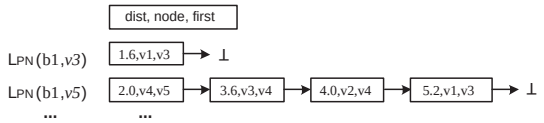

Figure 6: The portal-node lists of b_1.

size of $O(N_b \cdot P_b)$, where N_b is the block size and P_b is the number of out-portal nodes in a block. Since P_b is usually much smaller than N_b, these lists are smaller than keyword-node lists. The size of D_{NP} is only $O(N_b)$.

5.2 Block Index

The block index is a simple data structure consisting of:

- ***Keyword-block lists***: For each keyword w, $L_{KB}(w)$ denotes the list of blocks containing keyword w, i.e., at least one node in the block is labeled with w. In the example of Figure 5, if block b_2 does not contain the keyword a, we have $L_{KB}(a) = \{b_1\}$; keyword d appears in both blocks, so $L_{KB}(d) = \{b_1, b_2\}$; the portal v_3 between b_1 and b_2 contains b, so $L_{KB}(b) = \{b_1, b_2\}$.
- ***Portal-block lists***: For each portal p, $L_{PB}(p)$ denotes the list of blocks with p as an out-portal. In Figure 5, v_3 resides in both b_1 and b_2 as an out-portal, so $L_{PB}(v_3) = \{b_1, b_2\}$. But v_5 is only an out-portal of b_1, so $L_{PB}(v_5) = \{b_1\}$.

The keyword-block lists are used by the search algorithm to start backward expansion in relevant blocks. The portal-block lists are used by the search algorithm to guide backward expansion across blocks. Note that with the portal-block lists, it is not necessary for each node to remember which block it belongs to; during backward expansion it should always be clear what the current block is.

Construction of the block index is straightforward and we omit it for brevity. Let $\bar{N}_b$ be the average block size and $\bar{K}_b$ be the average number of keywords in a block. Since each block will appear in $\bar{K}_b$ linked lists, the space requirement for the L_{KB} lists is $O((N/\bar{N}_b)\bar{K}_b)$. If we assume $\bar{K}_b = O(\bar{N}_b)$, this requirement comes to $O(N)$. Let P be the total number of portals. The space requirement for the L_{PB} lists is $O((N/\bar{N}_b) \cdot P)$, though in practice the space should be much lower because a portal is usually shared by only a handful of blocks.

5.3 Graph Partitioning

Before creating indexes, we first partition the graph into blocks. As we will see in Section 8, the partitioning strategy has an impact on both index size and search performance. In this section, we first discuss guidelines for good partitioning, and then describe two partitioning methods.

Effect of partitioning on index size is captured by the theorem below, which follows directly from the analysis in Section 5:

THEOREM 3. *Suppose a graph with N nodes is partitioned into B blocks. Let N_b denote the size of block b, and assume that the number of keywords in b is $O(N_b)$. The overall size of the two-level index is $O(\sum_b N_b^2 + BP)$.*

On the other hand, exact effect of partitioning on search performance is rather difficult to quantify, because it is heavily influenced by a number of factors such as the graph structure, keyword distribution, query characteristics, etc. Nonetheless, two guidelines generally apply:

- First, we want to keep the number of portals (P) low. In terms of space, according to Theorem 3, P appears as the one of the terms in the space complexity of our index, and N_b also increased with P (since blocks include portals). In terms of search performance, intuitively, the more portals we have, the more often we have to cross block boundaries during search, which hurts performance.
- Second, we want to keep blocks roughly balanced in size, because a more balanced partitioning tends to make index smaller. In Theorem 3, the term $\sum_b N_b^2$ is minimized when N_b's are equal, given that $\sum_b N_b$ is fixed.

Algorithm 2: Node-based partitioning algorithm.

```
1 Partition(G) begin
2     find an edge-based partitioning of G;
3     S ← edge separators of the edge-based partitioning;
4     P ← ∅;
5     foreach (u, v) ∈ S do
6         w ← choosePortal(u, v);
7         P ← P ∪ {w};                    // mark as portal
          S ← S − (edges incident to w in S);
8     return P;
9 end
```

To complicate matters further, finding an optimal graph partitioning is NP-complete [11]. Thus, we instead use a heuristic approach to partitioning based on the two guidelines above. As discussed earlier, we use node-based partitioning. However, the only heuristic partitioning algorithm for node-separators [24] has complexity as high as $O(N^{3.5})$. Thus, we propose two algorithms that first partition a graph using edge-separators and then convert edge-separators into node-separators (i.e., portals).

BFS-Based Partitioning We first propose a simple and fast partitioning method based on breadth-first search (BFS). To identify a new block, we start from an unassigned node and perform BFS; we add to this block any nodes that we visit but have not been previously assigned to any block, until the given block size is reached. In case that BFS ends but the block size is still too small, we pick another unassigned node and repeat the above procedure. At the end, we obtain an edge-based partitioning.

To convert this partitioning into a node-based one, for each edge separator (u_1, u_2), which currently connects two different blocks b_1 and b_2, we shift the block boundary so that one of u_1 and u_2 is on the boundary, which makes this node a portal. The choice of portal is controlled by the following choosePortal logic:

- Let s_1 and s_2 be the numbers of edge separators (in the edge-based partitioning) incident to u_1 and u_2, respectively.
- Choose u_i with the bigger $s_i + \delta|b_i|$ as a portal, where δ is tunable constant.

choosePortal seeks to balance the block sizes and minimize the number of portals, in light of the two partitioning guidelines. Once we make a node portal, it will belong to all blocks in which its neighboring nodes reside, allowing us to remove all of its incident edges separators from consideration. Hence, choosing a node with more incident edge separators heuristically reduces the number of portals we need to choose later. At the same time, we prefer to choose the node in the larger block (which allows the smaller block to grow), in order to balance block sizes. Parameter δ attempts to balance these two sometimes conflicting goals.

The complete partitioning algorithm is presented in Algorithm 2.

METIS-Based Partitioning The problem with the BFS-based partitioning is that it may start with a large and poor set of edge separators in the first place. Hence, we instead try the METIS algorithm [19], which aims to minimize the total weight of edge separators. The overall procedure is still given by Algorithm 2, except that on Line 2 we use METIS instead of BFS for edge partitioning.

Before feeding the graph into METIS, we apply some heuristics to adjust edge weights[2] to encourage subtrees to stay within the same blocks. In experiments, we will show that each method has respective advantage on different graphs.

6 Searching with the Bi-Level Index

We now present searchBLINKS, the algorithm for searching with the bi-level index, in Algorithm 3. At a high level, searchBLINKS is quite similar to searchSLINKS in Section 4.2: We generally follow our optimal backward search strategy, and expand in the forward direction when possible.

However, the bi-level nature of our index introduces an obvious complication: Since the graph has been partitioned, we no longer have the global distance information as with the single-level index. Therefore: (1) A single cursor is no longer sufficient to implement backward expansion from a keyword cluster. Multiple blocks may contain the same keyword, and simultaneous backward expansion is needed in multiple blocks. (2) Backward expansion needs to continue across block boundaries, whenever in-portals are encountered, into possibly many blocks. (3) Distance information in intra-block node-keyword maps can no longer be used as actual node-keyword distances, as shorter paths across blocks may exist.

Fortunately, the bi-level index has been carefully designed with these challenges in mind, and we show how to address these challenges in the remainder of this section.

Backward Expansion with Queues of Cursors To support backward expansion in multiple blocks while still taking advantage of the intra-block keyword-node lists, we use a queue Q_i of cursors for each query keyword w_i. Initially, for each keyword w_i, we use the keyword-block list to find blocks containing w_i (Line 4). A cursor is used to scan each intra-block keyword-node list for w_i; these cursors are all put in queue Q_i (Line 5).

When we reach an in-portal u of the current block, we need to continue backward expansion in all blocks that have u as their out-portal. We can easily identify such blocks by the portal-block list (Line 12). For each such block b, we continue expansion from u using a new cursor, this time to go over the portal-node list in block b for out-portal u (Line 13). Note that we initialize the cursor with a starting distance equal to the shortest distance from u to w_i. The cursor will automatically add this starting distance to the distances that it returns. Thus, the distances returned by the cursor will be the correct node-to-keyword distance instead of the node-to-portal distances in the portal-node list.

It is possible for searchBLINKS to encounter the same portal node u multiple times. There are two possible cases: (1) u can be reached (backwards) by nodes containing the same keyword in different blocks; (2) u can be reached (backwards) by nodes containing different keywords. Interestingly, we note that in Case (1), we only need to expand across u when it is visited for the first time; subsequent visits from the same keyword can be "short-circuited." The rationale behind this optimization is the optimal equi-distance expansion: The first visit to u from keyword w_i must yield the global shortest distance from u to w_i; therefore, any subsequent expansion through u from w_i will always have longer starting distances. We implement this optimization using a bitmap crossed to keep track of whether u has ever been crossed starting from a query keyword (Lines 11 and 14). An immediate consequence of this optimization is the following lemma:

LEMMA 1. *The number of cursors opened by* searchBLINKS *for each query keyword is $O(P)$, where P is the number of portals in the partitioning of the data graph.*

[2]Note that these weights are relevant to graph partitioning only, and are not the same as those used in the scoring function.

Algorithm 3: Search using bi-level indexes.

Variables: R: potential and completed answers, initially $\varnothing$.
A: completed answers; initially $\varnothing$.
τ_{prune}: pruning threshold; initially ∞.
Q_i: a queue of cursors, prioritized by peekDist() (lower peekDist() means higher priority).
crossed(i,u): a bitmap indicating whether backward expansion of w_i has ever crossed portal u (initially all false).

```
1  searchBLINKS(w_1, ..., w_m) begin
2      foreach i ∈ [1, m] do
3          Q_i ← new Queue();
4          foreach b ∈ L_KB(w_i) do
5              Q_i.add(new Cursor(L_KN(b, w_i), 0));
6      while ∃j ∈ [1, m] : Q_j ≠ ∅ do
7          i ← pickKeyword(Q_1, ..., Q_m);
8          c ← Q_i.pop();
9          ⟨u, d⟩ ← c.next();
10         visitNode(i, u, d);
11         if ¬crossed(i, u) and L_PB(u) ≠ ∅ then
12             foreach b ∈ L_PB(u) do        // cross portal
13                 Q_i.add(new Cursor(L_PN(b, u), d));
14             crossed(i, u) ← true;
15         if c.peekDist() ≠ ∞ then Q_i.add(c);
16         if |A| ≥ k and Σ_j Q_j.top().peekDist() > τ_prune and
           ∀v ∈ R − A : sumLBDist(v) > τ_prune then
17             exit and output the top k answers in A;
18     output up to top k answers in A;
19 end
20 visitNode(i, u, d) begin
21     if R[u] = ⊥ then                  // not yet visited
22         R[u] ← ⟨u, ⊥, ..., ⊥⟩;
23         R[u].dist_i ← d;
24         b ← the block containing u;
25         foreach j ∈ [1, i) ∪ (i, m] do   // expand forward
26             if D_NP(b, u) ≥ M_NK(b, u, w_i) then
27                 R[u].dist_i ← M_NK(b, u, w_i);
28     else if sumLBDist(u) > τ_prune then // can be pruned
29         return;
30     else if R[u].dist_i = ⊥ then    // previously visited
31         R[u].dist_i ← d;
32     if sumDist(u) < ∞ then              // answer found
33         A.add(R[u]);
34         if |A| ≥ k then
               τ_prune ← the k-th largest of {sumDist(v) | v ∈ A}
35 end
```

Therefore, even though searchBLINKS can no longer use one cursor per keyword, it only has to use one priority queue of $O(|P|)$ cursors for each keyword, still far better than algorithms that do not use indexes and therefore must use a priority queue of the entire search frontier.

Implementing Optimal Backward Search Strategy We implement the cost-balanced expansion strategy across clusters by the function pickKeyword (Line 7), which selects the keyword with the least number of explored nodes.

For each keyword, the prioritization used by the queue of cursor reflects the optimal equi-distance expansion strategy within clusters: The cursor with the highest priority in the queue is the one whose next distance is the smallest.

Expanding Forward Even though the distance information in intra-block node-keyword maps is not always valid globally, sometimes we can still infer its validity, enabling direct forward expansion from a node u to a keyword w_i. In particular (Lines 26 and 27), we consult the intra-block node-portal distance map for the shortest distance from u to any out-portal in its block. If this distance turns out to be longer than the intra-block distance from u to w_i, we conclude that the shortest path between u to w_i indeed lies within the block.

Pruning and Stopping Before describing the stopping and pruning conditions, we first discuss how to lower bound a node's combined distance to the keywords, using the information available to us in the bi-level index and implied by our search strategy. For a node u that has been visited but not yet determined as an answer root, we compute this lower bound $\mathsf{sumLBDist}(u)$ as the sum of lower bounds for distances to individual keywords, i.e., $\sum_{j=1}^{m} \mathsf{LBDist}_j(u)$. Recall from Section 4.2 that for each node u visited we maintain a structure $R[u] = \langle u, \mathsf{dist}_1, \mathsf{dist}_2, \ldots, \mathsf{dist}_m \rangle$, where dist_i is the distance from the node to keyword w_i. Without the single-level index, we may not know every dist_j. However, we can still derive a lower bound as follows. Let b be the block containing u. $\mathsf{LBDist}_j(u) = \max\{d_1, d_2\}$, where:

- (Bound from search) $d_1 = Q_j.\mathsf{top}().\mathsf{peekDist}()$, or ∞ if Q_j is empty. Intuitively, because of equi-distance expansion, if we have not yet visited u from keyword w_j, then u's distance to w_j must be at least as far as the next node we intend to expand to in the cluster.
- (Bound from index) $d_2 = \min\{M_{NK}(b, u, w_j), D_{NP}(b, u)\}$, where $M_{NK}(b, u, w_j)$ is the distance in b from u to w_j (∞ means u cannot reach w_j in b), and $D_{NP}(b, u)$ is u's distance to the closest out-portal of b (∞ means u has no path to any out-portal of b). Intuitively, if the true shortest path from u to w_i is within the block, the distance is simply $M_{NK}(b, u, w_j)$; otherwise, the path has to go out through an out-portal, which makes the distance at least $D_{NP}(b, u)$.

We maintain a pruning threshold τ_{prune} just as in searchSLINKS. If the lower bound on a node's combined distance is already greater than τ_{prune}, the node cannot be in the top k (Lines 28 and 29). The condition for stopping the search (Line 16) is slightly more complicated: We stop if every unvisited node must have combined distance greater than τ_{prune} (same condition as in searchSLINKS), and every visited non-answer node can be pruned.

7 Optimizations and Other Issues

Evaluating Pruning and Stopping Conditions Calculating the lower bound $\mathsf{sumLBDist}(u)$ used in pruning and stopping conditions is expensive, because the quantity $Q_j.\mathsf{top}().\mathsf{peekDist}()$, which constantly changes during the search, can increase the lower bound for many nodes at the same time. Since a weaker lower-bound does not affect the correctness of our algorithm, we can exploit the trade-off between the cost and accuracy of this calculation. Weaker lower bounds can make pruning and stopping less effective, but they avoid expensive calculations and updates that slow down search generally. Our approach is to lazily compute $\mathsf{sumLBDist}(u)$ only when visiting u. We also maintain a priority queue to keep track of the smallest lower bound among all candidate answers, to facilitate checking of the pruning condition. It would be interesting to use more sophisticated data structures tailored toward maintaining such lower bounds, but it is beyond the scope of this paper. Experiments show that our simple approach above provides a reasonable practical solution.

Batch Expansion One performance issue that arises in a direct implementation of searchBLINKS is that it switches a lot among cursors, and the code exhibits poor locality of access. Hence, we relax the equi-distance expansion strategy by allowing a small (and tunable) number of nodes to be expanded from a cursor as a batch. This optimization slightly complicates the pruning and stopping conditions of the algorithm, and may result in some unnecessary node accesses. However, we found such overhead to be generally small compared with the benefit of improved locality.

Recovering Answer Trees For clarity of presentation, the algorithms return only the roots of top k answers and their distances to each query keyword. In practice, users might want to see the matches and/or root-match paths of an answer. It is straightforward to extend the algorithms to return this additional information. The knode fields in keyword-node lists and node-keyword maps allow our algorithms to produce matches for answers with simple extensions (omitted) and without affecting space and time complexity. The first fields in keyword-node lists, node-keyword maps, and portal-node lists allow root-match paths to be reconstructed. Doing so requires extra time linear in the total length of these paths; in addition, searchBLINKS needs to be extended to record the list of portals crossed to reach an answer root.

Handling the Full Scoring Function Again, for clarity of presentation, our search algorithm has so far focused only on the component of the scoring function that deals with root-match paths. We now briefly discuss how the other score components, namely root and matches (recall Section 2), can be incorporated. Assume that the overall scoring function is $(\alpha \bar{S}_r(r) + \beta \sum_{i=1}^{m} \bar{S}_n(n_i, w_i) + \gamma \sum_{i=1}^{m} \bar{S}_p(r, n_i))^{-1}$, where α, β, and γ are tunable weighting parameters. First, when constructing keyword-node lists and node-keyword maps, we treat each node v containing keyword w as being distance $\frac{\beta}{\gamma}\bar{S}_n(v, w)$ away from w. Second, during the search, when an answer root r is identified, we add $\frac{\alpha}{\gamma}\bar{S}_r(r)$ to its combined distance. The pruning and stopping conditions still work correctly because this quantity is non-negative.

Effect of Ranking Semantics Here, we briefly discuss the effect of ranking semantics on the complexity of search and indexing. First, we note that if the score function is a black-box function defined on a substructure of the graph, there is no feasible method of precomputation and materialization that can avoid exhaustive search. Therefore, we must look for properties of the ranking that can be used to turn the problem tractable. In this paper, we have identified three such properties (cf. Section 2):

- *Distinct-root semantics.* With this semantics, we can devise an index that precomputes and materializes, for each node, some amount of information whose size is independent of k, to support top-k queries. The reason is that this semantics effectively requires us to produce at most one answer rooted at each node. Without this semantics, more information must be materialized or computed on the fly.
- *Match-distributive semantics.* Section 2 cited an example of a scoring function that is not match-distributive: Some systems score an answer by the total edge weight of the answer tree spanning all matches; each edge has its weight counted exactly only once, no matter how many matches it leads to. Multiple-keyword search in this setting is equivalent to the group Steiner tree problem [7, 21], which is NP-hard. In contrast, with the

match-distributive semantics, we can efficiently support multi-keyword queries by an index that precomputes, independently for each node and for each keyword, the best path between the node and any occurrence of the keyword. Without this semantics, it would be impossible to score an answer given such an index, because we cannot combine score components for any overlapping paths.

- *Graph-distance semantics.* Intuitively, with this semantics, we can take a root-to-match path, break it into components, precompute and materialize the component scores independently, and still be able to obtain the overall score by combining the component scores. Taking advantage of this semantics, our bi-level index breaks the graph into more manageable blocks to reduce index size; distance information across blocks can be assembled together for paths spanning multiple blocks. Without the graph-distance semantics, we would have to precompute scores for all possible root-to-match paths. Our single-level index offers this option, but its size disadvantage is obvious compared with our bi-level index.

8 Experimental Results

We implemented BLINKS, and for the purpose of comparison, the Bidirectional search algorithm [18], in Java with the JGraphT Library (http://jgrapht.sourceforge.net/). The experiments are conducted on a SMP machine with four 2.6GHz Intel Xeon processors and 4GB memory. Here we present experimental results and discuss factors affecting the performance of the BLINKS algorithm.

8.1 DBLP dataset

Graph Generation We first generate a node-labeled directed graph from the DBLP XML data (http://dblp.uni-trier.de/xml/). The original XML data is a tree in which each paper is a small subtree. To make it a graph, we add two types of non-tree edges. First, we connect papers through citations. Second, we make the same author under different papers share a common node. The graph is huge, but mostly it is still a tree and not very interesting for graph search. To highlight the purpose of graph search, we make it more graph-like by (1) removing elements in each paper that are not interesting to keyword search, such as url, ee, etc.; (2) removing most papers not referencing other papers, or not being referenced by other papers. Finally, we get a graph containing $50K$ papers, $409K$ nodes, $591K$ edges, and $60K$ distinct lower-cased keywords.

Search Performance We perform ranked keyword search using BLINKS and the Bidirectional algorithms on the DBLP graph. The query workload has various characteristics. Table 1 lists 10 typical queries, and Figure 7 shows the time it takes to find the top 10 answers using Bidirectional and four configurations of BLINKS—with two possible (average) block sizes ($|b| = 1000, 300$) and two possible partitioning algorithms (BFS- and METIS-based). Each bar in Figure 7 shows two values: the time it takes to find any 10 answers[3] (as the height of the lower portion[4]), and the time it takes to find the top 10 answers (as the full height of the bar). The first value exposes how fast a search algorithm can respond without considering answer quality. Note that we do not use the measurement described in [18] as their measurement requires knowing the k-th final answer in advance, which is impossible in practice before a query is executed. To avoid a query taking too much time, we return whatever we have after 90 seconds. Due to the wide range of response times, we plot the vertical axis logarithmically.

[3]Or all answer if there are fewer than 10.

[4]The values for $Q5$ and $Q6$ are too small to show.

	Queries	# Keyword Nodes
Q1	algorithm 1999	(1299, 941)
Q2	michael database	(744,3294)
Q3	kevin statistical	(98,335)
Q4	jagadish optimization	(1,694)
Q5	carrie carrier	(6,6)
Q6	cachera cached	(1,6)
Q7	jeff dynamic optimal	(45,770,579)
Q8	abiteboul adaptive algorithm	(1,450,1299)
Q9	hector jagadish performance improving	(6,1,1349,173)
Q10	kazutsugu johanna software performance	(1,3,993,1349)

Table 1: Query Examples

As shown in Figure 7, BLINKS outperforms the Bidirectional search by at least an order of magnitude in most cases, which demonstrates the effectiveness of the bi-level index. It also shows that the number of keyword nodes (see Table 1) is not the single most important factor affecting response time. For instance, although the two keywords in $Q6$ appear in few nodes, the Bidirectional search uses more time on $Q6$ than $Q1$-$Q5$. There are actually two more fundamental factors, but they cannot be statically quantified by the number of keyword nodes. First is the size of the frontier expanded during the search. The number of keyword nodes only determines the initial frontier. Once a frontier reaches nodes with large in-degrees, the size of the priority queue increases dramatically. $Q6$ belongs to this case. Second is when we can safely stop search. It highly depends on pruning effectiveness, which depends on the quality of the answers obtained so far. For example, $Q6$ has only one answer (the root element db|p), so no pruning bound (the score of the k-th answer) can be established to terminate search early.

The other observation is that although the exact search space of a query depends on many factors, queries containing more keywords tend to have larger search spaces as each keyword has its own frontier. In the experiment, $Q1$-$Q6$ each contain two keywords, $Q7$-$Q8$ three, and $Q9$-$Q10$ four. The results of BLINKS show longer response time for queries with more keywords. We also compare BLINKS using different partitioning algorithms. In most cases, BFS-based partitioning shows better performance than METIS-based one. However, it is not always true, shown in the next experiment on the IMDB dataset. We also observe the impact of block size. Generally speaking, with larger blocks, searches will involve fewer cursors. However, for the same query, the query time of larger block partitioning does not always outperform that of smaller one. Actually, the query time is affected by many other factors, such as the loading of block indexes, the way the search space spans blocks, etc.

Index Performance Now we examine the impact of block size the indexing performance of BLINKS, in terms of indexing time, number of portals, and index size, which are shown in Figure 8(a), (b), and (c), respectively. Each figure displays results under five different configurations. The first four vary in their average block sizes ($|b| = 100$, 300, 600, and 1000) and all use METIS-based partitioning. The last one has $|b| = 1000$ and uses the simpler BFS-based partitioning.

Figure 8(a) shows partitioning time and indexing time for each configuration. The first four configurations have similar partitioning time, which is dominated by the METIS algorithm, while indexing time increases consistently when blocks become larger. This increasing trend is also observed in Figure 8(c), where larger blocks lead to more index entries. The last configuration (1000BFS) applies the simple BFS-based partitioning, so it needs much less partitioning time. But interestingly, it also takes much less indexing

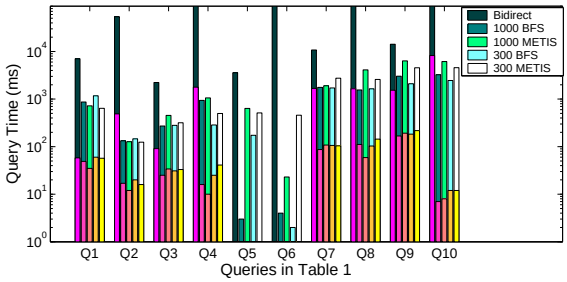

Figure 7: Query performance on the DBLP graph.

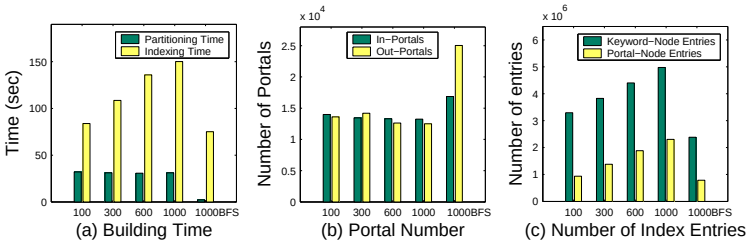

Figure 8: Impact on indexing the DBLP graph with various parameters.

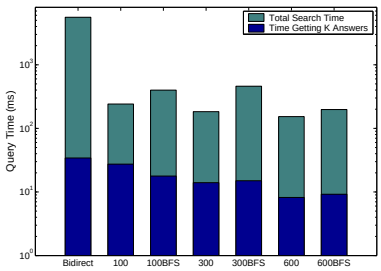

Figure 9: Performance on IMDB

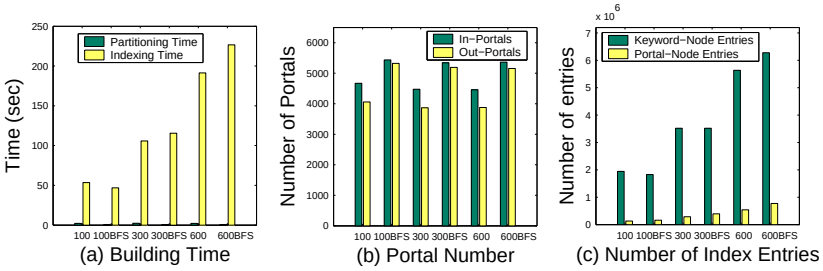

Figure 10: Impact on indexing the IMDB graph with various parameters.

time than the fourth configuration (1000), which uses METIS and has the same average block size. This result can be explained by the DBLP graph topology. As we mentioned earlier, DBLP consists of a very flat and broad tree plus a number of cross-links. Thus, the BFS-based algorithm is likely to produce flat and broad blocks in which paths are usually quite short. The time for building an intra-block index is mostly spent on traversing paths in the block, which is faster on shorter paths. Accordingly, as shown in Figure 8(c), intra-block indexes of such blocks have fewer entries. Note that the BFS-based algorithm does not alway lead to faster indexing and smaller index size; experiments on the IMDB dataset below offer an counter-example. Finally, Figure 8(b) shows that METIS-based partitioning achieves a significant reduction in the number of portal nodes compared with BFS-based partitioning.

8.2 IMDB Dataset

Graph Generation We also generate a graph from the popular IMDB database (http://www.imdb.com). In this experiment, we are interested in highly-connected graphs with few tree components. We generate the graph from the movie-link table using movie titles as nodes and links between movies as edges. The graph contains $68K$ nodes, $248K$ edges, and $38K$ distinct keywords.

Search Performance As different queries may have very different running times, we run a set of queries using Bidirectional and six configurations of BLINKS (with three block sizes $|b| = 100$, 300, and 600, each using BFS-based or METIS-based partitioning). Figure 9 shows the average response time, where each bar contains two readings with the same interpretation as Figure 7. Comparing neighboring bars with the same block size but different partitioning algorithms, we observe that BFS-based partitioning usually leads to worse query performance than METIS-based partitioning. Comparing bars with different block sizes but same partitioning algorithm, we see larger blocks usually lead to better performance.

Index Performance Now we discuss the index performance of BLINKS on highly-connected graphs. Figure 10 again reports three measurements: indexing time, number of portals, and index size under the same configurations as Figure 9. Since the IMDB graph is relatively small, partitioning is fast. However, indexing time on this graph is comparable to the much larger DBLP graph. The reason lies in the IMDB graph topology, where the structure of each block is usually more complex than in the DBLP graph. Therefore, it takes more time to create index for each block, and each intra-block index tends to have more entries, as shown in Figure 10(c). Another issue is that the indexing time and size with the BFS-based algorithm are usually worse than those of the METIS-based algorithm. Comparing Figures 8 and 10, we note that the graph topology plays an important role on BLINKS: BFS works fine on simpler graphs, but experiences difficulty on highly-connected graphs. As future work, it would be interesting to study how to use the characteristics of a graph to automate the choice of the partitioning strategy.

9 Related Work

The most related work, BANKS [3, 18], has been discussed extensively in Section 3. We outline other related work here. Some papers [10, 5, 14, 20, 22] aim to support keyword search on XML data, which is a similar but simpler problem. They are basically constrained to tree structures, where each node only has a single incoming path. This property provides great optimization opportunities [25]. Connectivity information can also be efficiently encoded and indexed. For example, in XRank [14], the Dewey inverted list is used to index paths so that a keyword query can be evaluated without tree traversal. However, in general graphs, tricks on trees cannot be easily applied.

Keyword search on relational databases [1, 3, 17, 16, 23] has attracted much interest. Conceptually, a database is viewed as a labeled graph where tuples in different tables are treated as nodes connected via foreign-key relationships. Note that a graph constructed this way usually has a regular structure because schema restricts node connections. Different from the graph-search approach in BANKS, DBXplorer [1] and DISCOVER [17] construct join expressions and evaluate them, relying heavily on the database schema and query processing techniques in RDBMS.

Because keyword search on graphs takes both node labels and graph structure into account, there are many possible strategies for ranking answers. Different ranking strategies reflect design-

ers' respective concerns. Focusing on search effectiveness, that is, how well the ranking function satisfies users' intention, several papers [16, 2, 13, 23] adopt respective IR-style answer-tree ranking strategies to enhance semantics of answers. Different from our paper, search efficiency is often not the basic concern in their designs. In IR-styled ranking, edge weights are usually query-dependent, which makes it hard to build index in advance.

To improve search efficiency, many systems, such as BANKS, propose ways to reduce the search space. Some of them define the score of an answer as the sum of edge weights. In this case, finding the top-ranked answer is equivalent to the group Steiner tree problem [7], which is NP-hard. Thus, finding the exact top k answers is inherently difficult. Recently, [21] shows that the answers under this scoring definition can be enumerated in ranked order with polynomial delay under data complexity. [6] proposes a method based on dynamic programming that targets the case where the number of keywords in the query is small. BLINKS avoids the inherent difficulty of the group Steiner tree problem by proposing an alternative scoring mechanism, which lowers complexity and enables effective indexing and pruning.

In the sense that distance can be indexed by partitioning a graph, our portal concept is similar to *hub nodes* in [12]. Hub nodes were devised for calculating the distance between any two given nodes, and a global *hub index* is built to store the shortest distance for all of hub node pairs. In BLINKS, we do not precompute such global information, and we search for the best answers by navigation through portals.

10 Conclusion

In this paper, we focus on efficiently implementing ranked keyword searches on graph-structured data. Since it is difficult to directly build indexes for general schemaless graphs, the existing approaches rely heavily on graph traversal at runtime. Lack of more knowledge about the graph structure also leads to less effective pruning during search. To address these problems, we introduce an alternative scoring function that makes the problem more amenable to indexing, and propose a novel bi-level index that uses blocking to control the indexing complexity. We also propose the cost-balanced expansion policy for backward search, which provides good theoretical guarantees on search cost. Our search algorithm implements this policy efficiently with the bi-level index, and further integrates forward expansion and more effective pruning. Results on experiments show that BLINKS improves the query performance by more than an order of magnitude.

Index maintenance is an interesting direction for future work. It includes two aspects. First, when the graph is updated, we need to maintain the indexes. In general, adding or deleting an edge has global impact on shortest distances between nodes. A huge number of distances may need to be updated for a single edge change, which makes storing distances for all pairs infeasible. BLINKS localizes index maintenance to blocks, and we believe this approach will help lower the index maintenance cost. Second, by monitoring performance at runtime, we may dynamically change graph partitions and indexes in order to adapt to changing data and workloads.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

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

[4] Y. Cai, X. Dong, A. Halevy, J. Liu, and J. Madhavan. Personal information management with SEMEX. In *SIGMOD*, 2005.

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.

[6] B. Ding, J.X. Yu, S. Wang, L. Qing, X. Zhang, and X. LIN. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[7] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.

[8] S. Dumais, E. Cutrell, JJ Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *SIGIR*, 2003.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[10] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Comput. Networks*, 33(1-6):119–135, 2000.

[11] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[12] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB*, pages 26–37, 1998.

[13] J. Graupmann, R. Schenkel, and G. Weikum. The spheresearch engine for unified ranked retrieval of heterogeneous XML and web documents. In *VLDB*, pages 529–540, 2005.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.

[15] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. Technical report, Duke CS Department, 2007.

[16] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[17] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

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

[19] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing*, 1995.

[20] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, pages 779–790, 2004.

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

[22] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, pages 72–83, 2004.

[23] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[24] J. Liu. A graph partitioning algorithm by node separators. *ACM Trans. Math. Softw.*, 15(3):198–219, 1989.

[25] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.