Bidirectional Expansion For Keyword Search on Graph Databases

Varun Kacholia * Shashank Pandit * Soumen Chakrabarti S. Sudarshan
Rushi Desai * Hrishikesh Karambelkar *

Indian Institute of Technology, Bombay
varunk@acm.org shashank+@cs.cmu.edu {soumen,sudarsha,hrishi}@cse.iitb.ac.in rushi@desai.name

Abstract

Relational, XML and HTML data can be represented as graphs with entities as nodes and relationships as edges. Text is associated with nodes and possibly edges. Keyword search on such graphs has received much attention lately. A central problem in this scenario is to efficiently extract from the data graph a small number of the "best" answer trees. A *Backward Expanding* search, starting at nodes matching keywords and working up toward confluent roots, is commonly used for predominantly text-driven queries. But it can perform poorly if some keywords match many nodes, or some node has very large degree.

In this paper we propose a new search algorithm, *Bidirectional Search*, which improves on Backward Expanding search by allowing forward search from potential roots towards leaves. To exploit this flexibility, we devise a novel search frontier prioritization technique based on spreading activation. We present a performance study on real data, establishing that Bidirectional Search significantly outperforms Backward Expanding search.

1 Introduction

Keyword search over graph-structured textual data has attracted quite some interest in the database community lately. Graphs where nodes (and possibly edges) have associated text are a convenient "common denominator" representation for relational data (where nodes are tuples and links are induced by foreign keys), semistructured XML data (where nodes are elements and links represent element containment, keyrefs, and IDREFs), and Web data (where nodes can be whole pages or DOM elements and links can represent HREFs or DOM element containment). This common representation enables novel systems for heterogeneous data integration and search.

Systems for "schema-agnostic" keyword search on databases, such as DBXplorer [1], BANKS [3] and Discover [9], model a response as a tree connecting nodes (tuples) that contain the different keywords in a query (or more generally, nodes that satisfy specified conditions). Here "schema-agnostic" means that the queries need not use any schema information (although the evaluation system can exploit schema information). For example, the query "`Gray transaction`" on a graph derived from DBLP may find `Gray` matching an author node, `transaction` matching a paper node, and an answer would be the connecting path; with more than two keywords, the answer would be a connecting tree. The tree model has also been used to find connected Web pages, that together contain the keywords in a query [10].

As in Web search, graph search systems must define a measure of relevance or merit for each response. Responses must be presented in relevance order. Ideally, the query processor must efficiently generate only a few responses that have the greatest relevance scores. A variety of notions of response, relevance measures and search algorithms have been proposed for graph search.

DBXplorer and Discover use the number of edges in a tree as a measure of its quality, preferring trees with fewer edges. However, this measure is coarse grained: for example, it cannot favor a highly cited paper on transactions by Gray over a less-known paper. Both DBXplorer and Discover can be extended to associate weights with edges based only on the database schema, but do not consider node weights or edge weights that are not determined by the schema alone, which are required to fully exploit the generality of graph search.

XRank [7] assigns a specific Pagerank-like [4]

[*] Work done while at IIT Bombay. Current affiliations of first two and last two authors are U. C. Berkeley, C.M.U., Microsoft and Oracle, Bangalore, respectively

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

prestige to individual nodes, but depends on tree-structured XML for efficient query execution. There are many scenarios where data forms arbitrary graphs, and cannot be meaningfully modeled by tree structures or even DAG structures; the XRank index structure (and other common XML index structures) cannot be used in such scenarios. ObjectRank [2] also assigns a Pagerank-like [4] score to each node, but does not score answer subgraphs as response units.

In contrast, the Backward expanding strategy used in BANKS [3] can deal with the general model. In brief, it does a best-first search from each node matching a keyword; whenever it finds a node that has been reached from each keyword, it outputs an answer tree. However, Backward expanding search may perform poorly w.r.t. both time and space in case a query keyword matches a very large number of nodes (e.g. if it matches a "metadata node" such as a table or column name in the original relational data), or if it encounters a node with a very large fan-in (e.g. the "paper appeared in conference" relation in DBLP leads to "conference" nodes with large degree).

Our Contributions: We introduce a new search algorithm, which we call **Bidirectional Search**, for schema-agnostic text search on graphs. Note that the graph on which search occurs may be explicitly represented (as in BANKS) or may be implicitly present in the database, as in DBXplorer and DISCOVER. Our search algorithm can be used in either case.

Unlike backward expanding strategies, which can only explore paths backward from keyword nodes toward the roots of answer trees, the Bidirectional algorithm can explore paths forward from nodes that are potential roots of answer trees, toward other keyword nodes. For example, if `transaction` matches a large number of nodes, while `Gray` matches fewer nodes, it may be profitable to search backward from nodes matching `Gray` toward potential answer roots, and then forward to find nodes that match `transaction`. This is equivalent to a join order (with indexed nested loops join) which favors starting from the relation with fewer tuples and probing the one with more tuples.

In information retrieval, it is standard to intersect inverted lists starting with the smallest one [15]. This can be regarded as a special case of our algorithm.

Spreading activation: A key innovation needed to contain and exploit the flexibility of simultaneous backward and forward expansion is a novel frontier prioritization scheme based on spreading activation (roughly speaking, Pagerank with decay). The prioritization technique allows preferential expansion of paths that have less branching; our experiments show that this can provide large benefits in many cases.

Our prioritization mechanism can be extended to implement other useful features. For example, we can enforce constraints using edge types to restrict search to specified search paths, or to prioritize certain paths over others.

If we visualize answer tree generation as computing a join, the Bidirectional search algorithm, in effect, chooses a join order dynamically and incrementally on a per-tuple basis. In contrast, the Backward expanding search algorithm can be visualized as having a fixed join order.

Experiments: We have implemented the Bidirectional search algorithm as part of the BANKS system, and present a performance study comparing it with the Backward expanding search [3].

Our prototype, tested on DBLP[1], IMDB[2], and a subset of US Patents[3], shows that Bidirectional search can execute at interactive speeds over large graphs with over 20 million nodes and edges, on an ordinary desktop PC. Our study shows that Bidirectional Search outperforms Backward Expanding Search by a large margin across a variety of scenarios.

The rest of this paper is organized as follows. Section 2 outlines our data, query and response models. Section 3 outlines the Backward expanding search algorithm presented in [3]. Section 4 describes our Bidirectional search algorithm. Section 5 describes our performance study. Section 6 describes related work. We make concluding remarks in Section 7.

2 Data, Query and Responses

In this section, we briefly outline the graph model of data, and the answer tree model that our graph search algorithms focus on.

2.1 Graph Data Model

We model the database as a weighted directed graph in which nodes are entities and edges are relationships. A node may represent a tuple or row in a database, or an XML element. An edge may represent a foreign key/primary key relationship, element containment, or IDREF links in XML.

For example, in the graph model used in several systems (such as DBXplorer, BANKS, DISCOVER and ObjectRank) for each row r in a database that we need to represent, the data graph has a corresponding node u_r. We will speak interchangeably of a tuple and the corresponding node in the graph. For each pair of tuples r_1 and r_2 such that there is a foreign key from r_1 to r_2, the graph contains an edge from u_{r_1} to u_{r_2}. In DBXplorer and DISCOVER, the edges are undirected.

Directionality is natural in many applications: the strength of connections between two nodes is not necessarily symmetric. In BANKS [3], edge directionality was introduced avoid meaningless short paths through "hubs". Consider the data graph of DBLP, which has a (metadata) node called *conference*, connected to a node for each conference, which are then connected

[1] `http://www.informatik.uni-trier.de/~ley/db/`

[2] `http://www.imdb.com/`

[3] `http://www.uspto.gov/main/patents.htm`

to papers published in those conferences. The path through the *conference* node is a relatively meaningless (compared to an authored-by edge from paper to author, say) "shortcut" path which can make papers look more similar than they are. To deal with this, edges are treated as directed; the paper has a reference to the conference node, so a directed edge is created from the paper to the conference node.

On the other hand, to admit interesting response graphs, the model must allow paths to traverse edges "backwards". For example, if paper u co-cites v and w, there is no directed path between v and w, but we often wish to report such subgraphs. To handle this situation, given an edge $u \rightarrow v$ in the original database graph, BANKS creates a "backward edge" $v \rightarrow u$, with a weight dependent on the number of edges incident on v. Backward edges from "hubs" with many incident edges would have a high weight, thereby resulting in a low relevance score for meaningless shortcuts through such hubs. In the conference example, given that many papers refer to a particular conference, and many conferences refer to the *conference* node, the backward edges from these nodes would have large weight, and thus a low relevance.

2.2 Query and Response Models

In its simplest form, a query is a set of keywords. Let the query consist of n terms $t_1, t_2, \ldots, t_n$. For each search term t_i in the query, let S_i be the set of nodes that match the search term t_i. A node matches a term if the corresponding tuple contains the term; if a term matches a relation name, all tuples in the relation are assumed to match the term. In the directed graph model of [3], a *response* or *answer* to a keyword query is a minimal rooted directed tree, embedded in the data graph, and containing at least one node from each S_i. In undirected graph models such as DBXplorer and DISCOVER, the answer tree is not directed.

Intuitively, the paths connecting the keyword nodes (i.e., nodes with keyword(s)) *explain* how the keywords are related in the database.

2.3 Response Ranking

In addition to edge weights, the ranking of an answer may also depend on a notion of node prestige. As with Pagerank, not all nodes are equal in status; for example, users expect the query `recovery` on DBLP to rank first the most popular papers about recovery, as judged by their link neighborhood, including citations. A method of computing node prestige based on indegree is defined in [3], while [2] defines global and per-keyword node prestige scores for each node. Node prestige scores can be assumed to be precomputed for our purpose, although they could potentially be computed on-the-fly. We do not address the issue of how to compute node prestige here.

The overall score of an answer must then be defined by (a) a specification of the overall edge-score of the tree based on individual edge weights, and (b) a specification of the overall node-prestige-score of the tree, obtained by combining individual node prestige scores, and finally, (c) a specification for combining the tree edge-score with the node-prestige-score to get the overall score for the answer tree.

The focus of this paper is on the search algorithms, rather than on the ranking technique. For concreteness, we outline the specification used in our BANKS code; see [3] for details.

- The weights of forward edges (in the direction of foreign keys, etc.) are defined by the schema, and default to 1.
- If the graph had a forward edge $u \rightarrow v$ with weight w_{uv}, we create a backward edge $v \rightarrow u$ with weight $w_{vu} = w_{uv} \log_2(1+\text{indegree}(v))$. This discourages spurious shortcuts.
- We define a score $s(T, t_i)$ for an answer tree T with respect to keyword t_i as the sum of the edge weights on the path from the root of T to the leaf containing keyword t_i.
- We define the aggregate edge-score E of an answer tree T as[4] $\sum_i s(T, t_i)$.
- The prestige of each node is determined using a biased version of the Pagerank [4] random walk, similar to the computation of global ObjectRank [2], except that, in our case, the probability of following an edge is inversely proportional to its edge weight taken from the data graph instead of the schema graph.
- We define the tree node prestige N as the sum of the node prestiges of the leaf nodes and the answer root.
- We define the overall tree score as EN^λ where λ helps adjust the importance of edge and node scores. As a default, we use $\lambda = 0.2$, a choice found to work well [3].

We have chosen a response-ranking specification guided by recent literature, but clearly, other definitions of answer tree scores are possible. Comparison of alternatives must involve large-scale user studies, which is scanty in current literature and would be valuable as future work.

Our search algorithm works on arbitrary directed weighted graphs, and is not affected by how the edges and edge weights are defined, or on the exact technique for computing answer scores. However, our prioritization functions do need to take these into account, as do functions that compute upper bounds on the score of future answers (so that answers with a higher score can be output without waiting further). We address this issue later, in Sections 4.3 and 4.5.

[4] This step differs from the formula in [3], which adds up the scores of all edges in a tree. The current version is simpler to deal with and we found it gives results of equivalent quality.

3 Backward Expanding Search

Before we present our Bidirectional search algorithm, we present the Backward expanding search algorithm of [3] (which we sometimes call Backward search, for brevity). Backward expanding search is based on Dijkstra's single source shortest path algorithm. Given a set of keywords, it first finds, for each keyword term t_i, the set of nodes S_i that match t_i; the nodes in S_i are called keyword nodes. To facilitate this a single index is built on values from selected string-valued attributes from multiple tables. The index maps from keywords to (table-name, tuple-id) pairs.

Let the query be given by a set of keywords $K = (t_1, t_2, ...t_n)$ and set S_i denote the set of nodes matching keyword t_i. Thus $S = S_1 \cup S_2 \cup \ldots \cup S_n$ is the set of relevant nodes for the query. The Backward search creates $|S|$ iterators; each iterator executes a the single source shortest path algorithm starting with one of the nodes in S as source. When a getnext() function is called on an iterator, it restarts from the state saved on the last getnext() call, and runs a step of the Dijkstra algorithm; that is, it finds the minimum distance node m on the frontier, expands the frontier to include all nodes connected to m, saves the fronter in its state, and returns m.

Unlike the normal shortest path algorithm, each iterator traverses the graph edges in "reverse direction", using edges pointing toward the node being expanded rather that edges pointing away from it. The idea is to find a common vertex from which a forward path exists to at least one node in each set S_i. Such paths will define a rooted directed tree with the common vertex as the root and containing all the keyword nodes[5]. The tree thus formed is an answer tree. Since the iterators traverse edges in the "backward" direction, the algorithm is called the Backward expanding search algorithm.

At each iteration of the algorithm, one of the iterators is picked for further expansion. The iterator picked is the one whose next vertex to be visited has the shortest path to the source vertex of the iterator (the distance measure can be extended to include node weights of the nodes matching keywords). A list of all the vertices visited is maintained for each iterator. Consider a set of iterators containing one iterator from each set S_i. If the intersection of their visited vertex lists is non-empty, then each vertex in the intersection defines a tree rooted at the vertex, with a path to at least one node from each set S_i. A resulting tree is an answer tree only if the root of the tree has more than one child. If the root of a tree T has only one child, and all the keywords are present in the non-root nodes, then the tree formed by removing the root node is also present in the result set and has a higher relevance score. Such a non-minimal tree T is therefore discarded.

4 Bidirectional Expanding Search

In this section we motivate and describe our new algorithm for generating answer responses to keyword queries on graphs, which we call *Bidirectional expanding search* (or Bidirectional search, for brevity).

Note that the problem of finding answer trees is NP-hard, since the well known Steiner tree problem for undirected graphs can be easily reduced to the problem of finding answer trees [3]. In fact, even polynomial (approximation) algorithms that require an examination of the entire graph would not be desirable for finding answer trees, since the overall graph may be very large, whereas answer trees can potentially be found by examining a small part of the graph. The goal of Backward expanding search as well as of our Bidirectional search is to generate answers while examining as small a fraction of the graph as possible.

4.1 Motivation for Bidirectional Search

The Backward search algorithm would explore an unnecessarily large number of graph nodes in the following scenarios:

- The query contains a frequently occurring term: In the Backward algorithm one iterator is associated with every keyword node. The algorithm would generate a large number of iterators if a keyword matches a large number of nodes. This could happen in case of frequently occurring terms (e.g.. `database` in the DBLP database, `John` in the IMDB database) or if the keyword matches a relation name (which selects all tuples belonging to the relation).
- An iterator reaches a node with large fan-in: An iterator may need to explore a large number of nodes if it hits a node with a very large fan-in (e.g.. a *department* node in a university database which has a large number of *student* nodes connected to it).

We call the nodes that are already hit by an iterator but whose neighbors are yet to be explored by the iterator as the *fringe nodes* and the set of fringe nodes of an iterator as the *iterator frontier*. A node with a large fan-in results in a large increase in the frontier size.

In the above scenarios, the Backward search algorithm may explore a large portion of the graph before finding the relevant answers and this may result in a long search time; we give some empirical evidence of this in the experimental evaluation section (Section 5).

Our first approach to the problem was to create iterators only for keywords not "frequently occurring", and additionally explore forward paths from potential roots to the "frequent" keywords (see Figure 1). Every node reached by an iterator in the Backward search

[5] Each leaf node must contain at least one keyword and the non-leaf nodes may contain keywords.

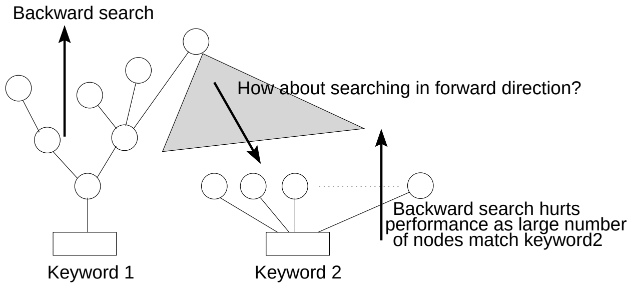

Backward search
How about searching in forward direction?
Backward search hurts performance as large number of nodes match keyword2
Keyword 1
Keyword 2

Figure 1: Need for forward search

algorithm is a potential root. If we follow forward paths from them, we may hit the frequent keyword nodes faster and hence find answers quickly. However, it is hard to decide on cutoffs that define which keywords are "frequent." Moreover, if the iterator reaches a node v with a large indegree, it would still explore a large number of nodes that have an edge to v.

Keyword search engines always intersect inverted lists starting with the rarest word [15] and some other pruning strategies based on word rareness are also used. While inverted lists are "flat" sets of document IDs, our situation is more complex because we can transitively expand neighbors to neighbors of neighbors, etc.

The two key ideas behind our Bidirectional expanding search algorithm are:

- Starting forward searches from potential roots
- A spreading activation model to prioritize nodes on the fringe, whereby nodes on an iterator with a small fringe would get a higher priority, and among nodes within a single iterator, those in less bushy subtrees would get a higher priority. Prioritization additionally takes answer relevance into account, and is described in Section 4.3, after describing the search algorithm in Section 4.2.

Discouraging backward search from large fringes avoids potentially wasteful expansion, yet we are able to connect up to their corresponding keyword nodes by means of forward search from potential roots with higher activation. Our performance results (Section 5) demonstrate the benefits of this approach.

4.2 The Bidirectional Search Algorithm

Let the query be given by a set of keywords $K = (t_1, t_2, ...t_n)$ and set S_i denote the set of nodes matching keyword t_i. The Bidirectional search algorithm attempts to find best rooted directed trees connecting at least one node from each S_i.

Before we present the Bidirectional search algorithm, we outline key differences from the Backward search algorithm:

- We merge all the single source shortest path iterators from the Backward search algorithm into a single iterator and call it the *incoming iterator*.
- We use *spreading activation* (Section 4.3) to prioritize the search. For the incoming iterator, the next node to be expanded is the one with the highest activation. Activation is a kind of "scent" spread from keyword nodes, and edge weights are taken into account when spreading the activation, so the activation score reflects the edge weight as well as the spreading of the search fringe.
- We concurrently run another iterator which we call the *outgoing iterator*. This iterator follows forward edges starting from all the nodes explored by the incoming iterator.

K	Set of Keywords t_1, t_2, ... t_n
S_i	Set of nodes matching keyword t_i
S	$\bigcup_i S_i$
Q_{in}	A priority queue of nodes in backward expanding fringe
Q_{out}	A priority queue of nodes in forward expanding fringe
X_{in}	Set of nodes expanded for incoming paths
X_{out}	Set of nodes expanded for outgoing paths
P_v	Set of nodes u such that edge (u, v) has been explored
$sp_{u,i}$	Node to follow from u for best path to t_i
$dist_{u,i}$	Length of best known path from u to a node in S_i
$a_{u,i}$	Activation at node u from keyword t_i
a_u	Overall activation of node u
$depth_u$	depth of node u from keyword nodes

Figure 2: Notation used in Bidirectional Algorithm

Figure 2 shows the data structures used by the algorithm. Each iterator has two queues, one for the set of nodes to be expanded further and one for the set of already expanded nodes. For the incoming iterator, these are Q_{in} and X_{in} respectively. For the outgoing iterator, these are Q_{out} and X_{out} respectively. For every node u explored so far, either in outgoing or in incoming search, we keep track of the best known path from u to any node in S_i. Specifically, for every keyword term t_i we maintain the child node $sp_{u,i}$ that u should follow to reach a node in S_i in the best known path. We also maintain $dist_{u,i}$, the length of the best known path from u to a node in S_i, and the activation $a_{u,i}$ that t_i spreads to u (explained later).

The term *reached-ancestor* of a node u refers to ancestors (i.e. nodes that have a path to u) that have been reached earlier (i.e. are in one of Q_{in}, Q_{out}, X_{in} or X_{out}).

The pseudo-code for the algorithm is shown in Figure 3. The two key data structures used by the algorithm are the incoming iterator Q_{in} and the outgoing iterator Q_{out}; we describe the intuition behind the two iterators below.

At each step of the algorithm, among the incoming and outgoing iterators, the one having the node with highest priority is scheduled for exploration. Ex-

```
BIDIR-EXP-SEARCH()
  $Q_{in} \leftarrow S; Q_{out} \leftarrow \phi; X_{in} = \phi; X_{out} \leftarrow \phi;$
  $\forall u \in S : P_u \leftarrow \phi, depth_u = 0; \forall i, \forall u \in S : sp_{u,i} \leftarrow \infty;$
  $\forall i, \forall u \in S :$ if $u \in S_i, dist_{u,i} \leftarrow 0$ else $dist_{u,i} \leftarrow \infty$
  while $Q_{in}$ or $Q_{out}$ are non-empty
    switch
      case $Q_{in}$ has node with highest activation :
            Pop best $v$ from $Q_{in}$ and insert in $X_{in}$
            if IS-COMPLETE(v) then EMIT(v)
           if $depth_v < d_{max}$ then
             $\forall u \in incoming[v]$
               EXPLOREEDGE($u, v$)
                if $u \notin X_{in}$ insert it into $Q_{in}$
                  with depth $depth_v + 1$
              if $v \notin X_{out}$ insert it into $Q_{out}$
      case $Q_{out}$ has node with highest activation :
            Pop best $u$ from $Q_{out}$ and
              insert in $X_{out}$
            if IS-COMPLETE(u) then EMIT(u)
           if $depth_u < d_{max}$ then
             $\forall v \in outgoing[u]$
               EXPLOREEDGE($u, v$)
                if $v \notin X_{out}$ insert it into $Q_{out}$
                with depth $depth(u) + 1$

EXPLOREEDGE($u, v$)
  for each keyword $i$
    if $u$ has a better path to $t_i$ via $v$ then
      $sp_{u,i} \leftarrow v$; update $dist_{u,i}$ with this new dist
      ATTACH($u, i$)
       if IS-COMPLETE(u) then EMIT(u)
    if $v$ spreads more activation to $u$ from $t_i$ then
       update $a_{u,i}$ with this new activation
      ACTIVATE($u, i$)

ATTACH($v, k$)
  update priority of $v$ if it is present in $Q_{in}$
  Propagate change in cost $dist_{vk}$
    to all its reached-ancestors in best first manner

ACTIVATE($v, k$)
  update priority of $v$ if it is present in $Q_{in}$
  Propagate change in activation $a_{vk}$
    to all its reached-ancestors in best first manner

EMIT($u$)
  construct result tree rooted at u
  and add it to result heap

IS-COMPLETE($u$)
  for i = 1 to N
    if $dist_{u,i} == \infty$ return false; /* No path to $t_i$*/
  return true;
```

Figure 3: Bidirectional expanding search

ploring a node v in Q_{in} (resp. Q_{out}) is done as follows: incoming (resp. outgoing) edges are traversed to propagate keyword-distance information from v to adjacent nodes, and the node moved from Q_{in} to X_{in} (resp. Q_{out} to X_{out}). Additionally, if the node is found to have been reached from all keywords it is "emitted" to an output queue. Answers are output from the output queue when the algorithm decides that no answers with higher scores will be generated in future.

The distance of a node u from the keyword nodes (i.e. the number of edges from the nearest keyword node, as determined when it is first inserted into Q_{in} or Q_{out}) is stored in $depth_u$. A depth value cutoff d_{max} is used to prevent generation of answers that would be unintuitive due to excessive path lengths, and ensures termination; in our experiments, we used a generous default of $d_{max} = 8$.

4.2.1 Incoming Iterator

Unlike Backward expanding search, which runs multiple iterators, one from each node matching a keyword, in Bidirectional search we run only one iterator to explore backward paths from the keyword nodes. Note that, unlike the Backward search algorithm, the iterator is not a shortest path iterator since we do not order the nodes to be expanded solely on the basis of the distance from the origin; the nodes are ordered by a prioritization mechanism described later. Each node is present only once in the single backward path iterator, and popped from it only once; in contrast, in Backward expanding search, each node may be present in multiple iterators, since there is an iterator for each node matching each keyword. The benefit of having a single iterator is that the amount of information to be maintained is sharply reduced.

However, there are two issues to be addressed. It is possible for a node v to be popped after it is reached from one keyword node, say t_1; when a node is popped, its incoming edges from other nodes u are explored. Later on we may find that the node is reached from another keyword node t_2. At this stage, we have to traverse its incoming edges again to update distances to t_2 from all nodes u that have an incoming edge to v; in fact, this has to be done recursively, to update distances to t_2 from already reached ancestors of u. Procedure ATTACH(u, i) carries out this task (in our example, $i = 2$). In fact, since we prioritize using factors other than distance from the keyword node, it is possible that after finding one path from v to a keyword t_i, we may later find a shorter path; the distance update propagation has to be done each time a shorter path is found. Although such repeated propagation could potentially increase execution time, our performance results show that the benefits of prioritization outweigh the costs.

The second issue is that, in order to minimize space, we store only the shortest path sp_u, and distance $dist_{u,i}$ from each node u to the closest node among those that match keyword t_i. In contrast, Backward expanding search keeps shortest paths to each term that matches t_i. This optimization reduces space and time cost, but at the potential cost of changing the answer set slightly, although in practice this effect appears to be negligible – we revisit this issue in Section 4.6.

The nodes u in Q_{in} are the nodes on the frontier of the incoming iterator, and Q_{in} is ordered on the activation a_u of these nodes; the higher the activation of a node, the higher its scheduling priority. The Bidirectional search procedure starts by inserting all keyword nodes $u \in S$ where $S = \bigcup_{\forall i} S_i$ into the incoming iterator Q_{in} with an initial activation. For each keyword node, this seed activation is spread to the nodes that are reached in backward direction starting from this keyword node. The exact formulae for calculating the seed activation and for propagating activation can be decided depending on the answer ranking technique, and we discuss them further in Section 4.3.

4.2.2 Outgoing iterator

The outgoing iterator expands nodes in the forward direction from potential answer roots. Every node reached by the incoming iterator is treated as a potential answer root. For each root, the outgoing iterator maintains shortest forward paths to each keyword; some of these would have been found earlier by backward search on the incoming iterator, others may be found during forward search on the outgoing iterator.

When we explore forward paths from node u, for every adjacent node v such that there is an edge from u to v, we check for each keyword term t_i if there is a better path from u to t_i through v. If it exists, we update $sp_{u,i}$, $dist_{u,i}$ and $a_{u,i}$. A change in these values must be propagated to the ancestors of u. For this purpose, with every node u we maintain a list of its *explored* parent nodes P_u and update it every time an edge is explored. After exploring all edges, we check if v has been previously expanded in forward search (i.e. is in Q_{out} or was in Q_{out}; "was" can be checked by seeing if it is in X_{out}); if not we insert it in Q_{out}.

As in the incoming iterator, nodes in Q_{out} are on the frontier of the outgoing iterator and the queue is ordered on the total activation a_u of each node u. Activation is spread from each node u to all nodes v such that there is a forward edge from u to v.

4.2.3 Generating Results

Each time we update the path lengths from a node to a keyword t_i, we check if the node has paths to all the other keywords. In that case, we build the corresponding answer tree and insert it into the OutputHeap. The OutputHeap buffers and reorders answers since they may not be generated in relevance score order. In addition to the fact that some answers with a lower edge score may be generated after answers with a higher edge score, the node prestige of the root and leaf nodes also affects the relevance answer score. Results are output from the OutputHeap when we determine that no better result can be generated, as described in Section 4.5. It is also possible for the same tree to appear in more than one result, but with different roots. Such duplicates with lower score are discarded when they are inserted into the OutputHeap.

4.3 Activation Initialization and Spreading

As mentioned earlier, the Bidirectional search algorithm can work with different ways of defining the initial activation of nodes as well as with different ways of spreading activation. For concreteness we specify here formulas that are tailored to the answer ranking model described in Section 2.3. The overall tree score depends on both an edge score and on the node prestige, and both need to be taken into account when defining activation to prioritize search.

Nodes matching keywords, are added to the incoming iterator Q_{in} with initial activation computed as:

$$a_{u,i} = \frac{nodePrestige(u)}{|S_i|}, \forall u \in S_i \qquad (1)$$

where S_i is the nest of nodes that match keyword t_i. Thus, if the keyword node has high prestige, that node will have a higher priority for expansion. But if a keyword matches a large number of nodes, the nodes will have a lower priority.

The activations from different keywords are computed separately to separate the priority contribution from each keyword. When we spread activation from a node, we use an attenuation factor μ; each node v spreads a fraction μ of the received activation to its neighbours, and retains the remaining $1 - \mu$ fraction. As a default we set $\mu = 0.5$. The fraction μ of the received activation is divided amongst the neighbors as described below.

For the incoming iterator, the activation from keyword t_i is spread to nodes u_j such that there is an edge $u_j \rightarrow v$, Amongst these nodes, activation is divided in inverse proportion to the weight of the edge $u_j \rightarrow v$ (respectively, $v \rightarrow u_j$). This ensures that the activation priority reflects the path length from u_j to the keyword node; trees containing nodes that are farther away are likely to have a lower score.

For the outgoing iterator, activation from keyword t_i is spread to nodes u_j such that there is an edge $v \rightarrow u_j$, again divided in inverse proportion to the edge weights $v \rightarrow u_j$. This ensures that nodes that are closer to the potential root get higher activation, since tree scores will be worse if they include nodes that are farther away.

When a node u receives activation from a keyword t_i from multiple edges, we define $a_{u,i}$ as the maximum of the received activations. This reflects the fact that trees are scored by the shortest path from the root to each keyword. With scoring models that aggregate scores along multiple paths (as is done in [2]), we could use other ways of combining the activation, such as adding them up[6].

[6] This extension is implemented in the BANKS system and supports a form of queries which we call "near queries". For lack of space we cannot describe it further here, but the interested reader can try it out on the BANKS web site `http://www.cse.iitb.ac.in/banks`.

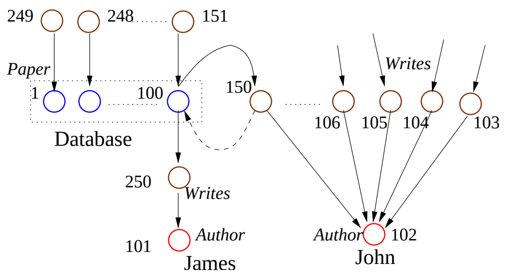


Figure 4: Bidirectional Search Example

The overall of activation a node is then defined as the sum of its activations from each keyword. Specifically, for a node u the overall activation a_u is defined as:

$$a_u = \sum_{i=0}^{n} a_{u,i}$$

This reflects the fact that if a node is close to multiple keywords, it is worth prioritizing the node, since connections to fewer keywords are left to be found.

4.4 Bidirectional Search Example

A sample graph and query is shown in Figure 4. The user wants to find out "Database" papers co-authored by "James" and "John". "Database" is a frequent keyword and has a large origin set while "James" and "John" match singleton nodes. It should be noted that node #102 has a large fan-in as "John" has authored many papers. For simplicity lets assume all node prestiges and edge weights to be unity. Backward expanding search would explore at least 151 nodes (and touch 250 nodes) before generating the result rooted at node #100.

Bidirectional search would start from nodes #101 and #102 (as nodes #1..100 have a lower activation due to large origin set). Expanding #101 would add #250 (with approximately the same activation) to the incoming queue while #102 would add 48 nodes (#103 .. #150) with a lower activation ($\sim$ $ActivationOf$(#101)/48). This would result in the exploration of #250 followed by #100. Finally, #100 would be expanded by the outgoing iterator to hit #150 producing the desired result. Hence, Bidirectional search would explore only 4 nodes (and touch about 150 nodes) before generating the result rooted at 100.

4.5 Producing Top-k Result Trees

As mentioned in Section 4.2.3, answers are placed in an output buffer when they are generated since they may not be generated exactly in relevance score order. At each iteration of Bidirectional search, we compute an upper bound on relevance of the the next result that can be generated, and use the bound to output all buffered answers with a higher relevance.

The upper bound is computed as follows. For each keyword, we maintain the minimum path length m_i among all nodes in the backward search trees for keyword i; that is, all nodes whose path length to a node containing keyword k_i is less than m_i have already been generated. As a coarser approximation, we can use the minimum path length among all nodes in Q_{in} as a lower bound for all the m_is. The best possible tree edge score for any answer node not yet seen would be defined by an edge score aggregation function $h(m_1, m_2, \ldots, m_k)$ where k is the number of keywords; for the ranking function in Section 2.3, the edge score aggregation function h simply adds up its inputs.[7]

However, every node that we have already seen is also a potential answer node; for each such node, we already know the distance to some of the keywords k_i; we use these scores along with the bound m_i for the remaining keywords to compute their best possible score. Combining these bounds with the maximum node prestige, we can get an upper bound u_b on relevance score of any answer that has not yet been generated. (The upper bound computation is similar to that used in the NRA algorithm of [5].) Any answer with relevance score greater than or equal to u_b can then be output.

As a looser heuristic, we can output any answers whose tree edge score is greater than the score $h(m_1, m_2, \ldots, m_k)$ described above; if there are multiple such answers they are sorted by their relevance score and output. This may output some answers out of order, since (a) some nodes seen already (but which have not yet been reached from all keywords) may yet have a score higher than this bound, and (b) the above heuristic ignores node prestige. However, the heuristic is cheaper to implement, and outputs answers faster, and the recall/precision measurements (Section 5.7) show that answers were output in the correct order on almost all queries we tested.

4.6 Single Iterator vs. Multiple Iterators

Bidirectional search maintains a single iterator across all keywords, recording for each node n the shortest path from n to a node containing keyword t_i, for each i. In contrast, Backward search maintains multiple iterators, recording the shortest path from each node n to each node n_j containing keyword t_i. Using a single iterator reduces the cost of search significantly. However, because of using a single iterator as above, Bidirectional search does not generate multiple trees with the same root,unlike Backward search. Even if a tree T_k rooted at n cannot be generated as a result, a rotation of T_k would be generated, albeit with a differ-

[7] The edge score combination function used in [3] is of a slightly different form, since it adds up the weights of answer tree edges, rather than combining scores with respect to each keyword. For such an edge scoring function, we can still heuristically use a bounding function that adds up the m_i's, although the result would not be an accurate bound.

ent score (the rotation would be rooted at one of the other nodes of T_k, with edges pointing from the root toward the leaves). In our experiments, we found that such alternative trees were indeed generated.

To separate the effect of using a single iterator from the other effects of Bidirectional search, we created a version of backward search which we call *single iterator backward search* or *SI-backward search*. This is identical to Backward search except that it uses only one merged backward iterator, just like Bidirectional search. However, it does not use a forward iterator, and its backward iterator is prioritized only by distance from the keyword, as in the original backward search, without any spreading activation component. To avoid ambiguity, we shall call the original version of Backward search as multiple-iterator Backward search (*MI-backward search*).

5 Experimental Evaluation

In this section we study the quality of answers and the execution time of Bidirectional search with respect to two versions of Backward expanding search, and the Sparse algorithm from [8]. Differences from other keyword search algorithms, which make them incomparable, are explained in Section 6.

We used a single processor 2.4 GHz P4 HT machine with 1GB of RAM, running Linux, for our experiments. We have experimented with three datasets - the complete DBLP database, IMDB (the Internet Movie Database), and a subset of the US Patent database. The complete DBLP database has about 300,000 authors and 500,000 papers resulting in about 2 million nodes and 9 million edges. IMDB database has a similar size while the US Patent databases is even larger and has 4 million nodes and 15 million edges.

5.1 Implementation Details

We used an in-memory representation of the graph, and wrote all our code using Java, using JDBC to communicate with the database. Note that the in-memory graph structure is really only an index, and does not contain actual data. Only small node identifiers are used in the in-memory graph, while attribute values are present only in the tuples that remain resident on the disk. With our compact array-based representation, the overall space requirement is $16 \times |V| + 8 \times |E|$ bytes for a graph with V vertices and E edges. Graphs with tens of millions of edges/vertices can thus easily fit in memory. We believe that the requirement that the graph fit in (virtual) memory will not be an issue for all but a few extreme applications.[8]

The node prestige calculation for the data sizes we tested takes about a minute. Our implementation currently does not handle updates, but it is straightforward to maintain the graph skeleton, given a log of updates to the source data. Node prestige can be recomputed periodically, since it is not essential to keep it up to date.

We used the default values noted earlier in the paper for all parameters (such as μ, λ and d_{max}).

5.2 Measures of Performance

We use three metrics for comparison: the nodes *explored* (i.e. popped from Q_{in} or Q_{out} and processed) and the nodes *touched* by the two algorithms, (i.e. inserted in Q_{in} or Q_{out}), and the time taken. The nodes explored indicate how effective the algorithm was in deciding what paths to explore, whereas the nodes touched indicates how many fringe (neighbour) nodes were seen from nodes that were explored.

For all the performance metrics, we use the last relevant result (or the tenth relevant result in case there are more than ten relevant results) as the point of measurement. Answer relevance was judged manually; the top 20 to 30 results of Backward and Bidirectional search were examined to find relevant answers (for the queries in Section 5.4, we executed SQL queries to find relevant answers).

By default, the time taken indicates the time to output the last relevant result. In addition to the time to output, another metric is the time to generate the last relevant answer. The answer may be output significantly later than when it was generated, since the system has to wait till it decides that no answer with higher score can be generated later. Thus the generation time ratio tells us the effectiveness of our prioritization techniques, whereas the output time ratios also take into account secondary effects that affect the score upper bounds.

For comparison with the Sparse algorithm of [8] we manually generated all "candidate networks" (i.e., join trees) smaller than the relevant ones, and ran each query several times to get a warm cache before measuring time taken, so IO time does not distort the results. Indices were created on all join columns. The time shown is a lower bound since we do not evaluate larger candidate networks, whereas Bidirectional search has to try longer paths to get answer bounds that allow it to output answers.

5.3 Results on Sample Queries

Figure 5 shows some of our queries, along with the number of nodes matching each keyword, the number of relevant answers generated and the size of the relevant answer trees. The queries labeled DQi were executed on DBLP, the IQi queries were executed on IMDB and the UQi queries on the US patent database.

We present three sets of numbers: the first set gives performance measure ratios comparing MI-Backward with SI-Backward, while the second set gives performance measure ratios comparing SI-Backward with Bidirectional search. The third set gives absolute

[8] We note that ObjectRank [2] also uses a similar in-memory structure to create its indices.

	Query	#Keyword nodes	Rel Ans	Ans Size	$\frac{\text{MI-Bkwd}}{\text{SI-Bkwd}}$ Time	$\frac{\text{SI-Bkwd}}{\text{Bidir}}$ Ratios				Time (secs)		
						Nodes expl.	Nodes touched	Gen time	Out time	SI Bkwd	Bidir	Sparse-LB (#CN)
DQ1	"David Fernandez" parametric	(2, 584)	18	3	3.48	16.78	3.45	7.0	1.25	0.07	0.05	0.12 (1)
DQ3	Giora Fernandez	(5, 188)	13	5	4.12	5.30	4.65	1.03	7.6	0.59	0.08	0.75 (2)
DQ5	Krishnamurthy parametric query optimization	(51, 584, 3236, 3874)	2	3	16.7	24.51	3.94	23.5	8.6	1.30	0.15	1.40 (1)
DQ7	Naughton Dewitt query processing	(5, 8, 3236, 4986)	1	5	12.45	4.39	1.39	59.95	1.35	3.54	2.63	3.70 (3)
DQ9	Divesh Jignesh Jagadish Timber Querying XML	(1, 4, 4, 7, 595, 1450)	1	7	4.07	6.83	1.89	23.13	2.29	1.39	0.61	6.20 (6)
IQ1	Keanu Matrix Thomas	(4, 430, 3670)	4	3	7.49	13.99	7.77	3.14	18.5	0.77	0.04	3.20 (2)
IQ2	Zellweger Jude Nicole	(3, 119, 1085)	1	7	3.64	6.38	1.10	2.039	4.34	2.2	1.63	6.80 (8)
UQ1	Microsoft recovery	(1, 1138)	4	2	4.91	2.93	1.17	4.14	1.22	0.22	0.19	0.50 (2)
UQ3	Cindy Joshua	(86, 207)	1	3	2.74	1.53	1.19	4.0	1.36	0.85	0.63	1.30 (4)
UQ5	Chawathe Philip	(28, 2773)	5	3	7.06	6.82	1.95	1.86	2.11	0.77	0.36	1.20 (4)

Figure 5: Bidirectional vs. Backward search on some sample queries

times taken by SI-Backward, Bidirectional, and a lower bound for the Sparse algorithm of [8].

When comparing SI-Backward with Bidirectional, the column with heading Gen Time gives the generation time ratio, i.e. the ratio of the times when the last relevant answer was generated.

Column 6 shows that SI-backward takes significantly less time that MI-Backward; the nodes explored (and touched) ratios for are similar to the time ratios, and are not shown.

The next 4 columns show that Bidirectional beats SI-Backward by a very large margin: in terms of the number of nodes explored, sometimes by nearly two orders of magnitude, and by a smaller but large margin in terms of nodes touched. It can be seen that the output time ratio lies in-between the nodes explored and nodes touched ratios (except when exploration time is very small and dominated by fixed overheads) The number of nodes touched by Bidirectional search ranges from under 1000 for DQ1 (which took about 50 msec to execute), to around 143,000 for DQ7 (which took 2.63 seconds).

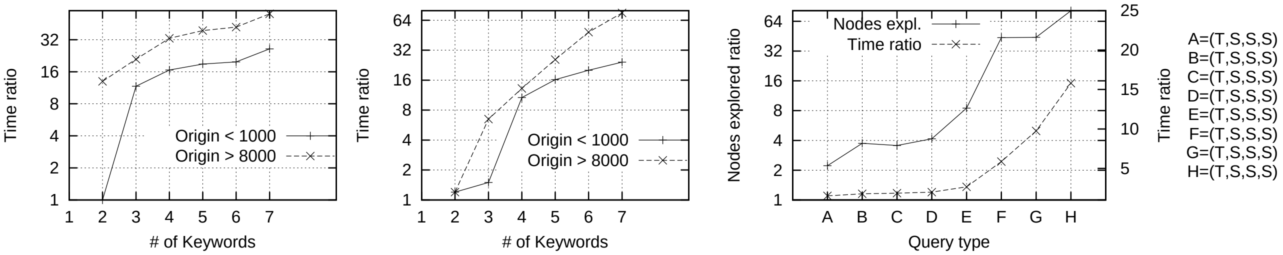
Figure 5 shows that the generation time ratio is significantly higher than the output time ratio. In almost all cases where some of the keywords match a large number of nodes and others matched only a few nodes, Bidirectional found the relevant answers quickly, but in several cases it could not output them till much later, based on the lower bound estimation. For example, in the case of *DQ*7 Bidirectional found the relevant answer very fast, in 47 msec (vs. 2.7 sec taken by Backward), but was able to output it only much later (2.63 sec). The reason why an answer generated early on cannot be output may be because some paths of low weight have a low activation, and do not get explored, but pull up the value of the upper bound estimate for answer scores.

The most important part of Figure 5 are the last 3 columns of the table; these show that in terms of the time taken to output the last relevant (or tenth) result, Bidirectional search is the clear winner, often by an order of magnitude, over SI-backward as well as the Sparse lower bound (labeled Sparse LB). The Sparse lower bound time actually corresponds to Gen time of Bidirectional, and Bidirectional beats Sparse by an even larger margin on the Gen time measure (e.g. 47 msec versus 3.7 seconds for DQ7). It can also be seen that Sparse does progressively worse as the number of candidate networks (shown in parantheses) increases. For example, for the 9th and 10th results of DQ7 (which Bidirectional output in 2.3 seconds), Sparse took 15 min. and 10 min. respectively; these networks used citation and co-authorship links (respectively) requiring 6 and 8 joins respectively. Although these results were not judged as highly relevant they are small and intuitive.

5.4 Multi-Iterator vs Single-Iterator

Table 5 showed SI-Backward search significantly outperforms MI-Backward search on the sample queries. We now compare SI-Backward search with the MI-Backward search on a larger workload of 200 queries consisting of 2-7 keywords. For each of the queries the size of the most relevant results is fixed to 5 i.e 4 joins would be required to produce the relevant results. To generate the workload we used the DBLP database and executed SQL queries having a join network of size 5. Keywords were selected at random from each tuple in the result set. Small origin size queries denote those where less than 1000 records matched at least one of the query keywords, while large origin queries denote those where more than 8000 records matched at least one of the keywords.

We ran a large number of queries to compare



(a) MI-Bkwd/SI-Bkwd time ratio (b) SI-Bkwd/Bidirec time ratio (c) SI-Bkwd/Bidirec time & node ratio

Figure 6: Comparison of search algorithms

the result sets of Bidirectional, SI-Backward and MI-Backward. In all cases we found that Bidirectional, SI-Backward and MI-Backward return the same sets of relevant answers. Recall/precision results are presented in Section 5.7.

Figure 6(a) shows the average ratio of $\frac{Time(\text{MI-Backward})}{Time(\text{SI-Backward})}$ for small as well as large origin sizes. SI-Backward wins over MI-Backward by at least an order for almost all the queries. For the 2 keyword case with small origin size SI-Backward wins only marginally over MI-Backward due to the relatively low overhead required by the MI-Backward algorithm for creating multiple iterators. We also observed that the ratio of $\frac{Nodes\text{-}Explored(\text{MI-Backward})}{Nodes\text{-}Explored(\text{SI-Backward})}$ was identical to the time ratio as both the algorithms explore the graph in a similar fashion.

5.5 Bidirectional vs SI-Backward

Figure 6(b) shows the ratio of average time taken to answer a query using the Bidirectional and SI-Backward search algorithms on the workload described in Section 5.4. We observe that Bidirectional wins over SI-Backward by a large margin. We also measured the ratio of nodes explored, which followed roughly the same pattern as the time taken, although the ratios were higher by a factor of about 2; we omit details for lack of space.

Although Bidirectional search wins over SI-Backward search by an order of magnitude in most cases, across a large number of queries that we tried, we found that Bidirectional performed worse on just a few queries. For example Bidirectional search took about 1.5 times more time than the SI-Backward search to answer the query “"C. Mohan" Rothermel”. The reason is that both authors have an extremely low initial origin size (1 and 5 respectively) and a large fan-in (as both have written many papers). Bidirectional search initiates backward as well as forward search, but forward search does not help much in such a symmetric case.

5.6 Join order comparison

We now describe an experiment to test our claim that Bidirectional search chooses a better join order than the Backward search. We fix the number of keywords to 4 and size of the most relevant result to 3. Keywords in our queries are divided into four categories: tiny(T) (keyword matches 1 to 500 tuples), small(S) (keyword matches 1000-2000 tuples), medium(M) (keyword matches 2500-5000 tuples) and large(L) (keyword matches over 7000 tuples). We generate a workload of 400 queries from the DBLP database using techniques outlined in Section 5.4.

Figure 6(c) shows the ratio of time taken by Bidirectional and Backward along with the nodes explored ratio for selected combinations of keyword categories in the query. We omit the other combinations for lack of space. We observe that Bidirectional outperforms SI-Backward in all the cases and the speedup increases as the difference between the origin sizes of keyword increases. Thus Bidirectional beats SI-Backward by a large margin when the keywords belong to the category (*Tiny, Tiny, Tiny, Large*), whereas the win is much smaller for (*Medium, Medium, Medium, Medium*) and (*Medium, Large, Large, Large*).

5.7 Recall/Precision Experiments

We used the queries generated in Section 5.4 and measure the recall and precision ratios for each algorithm. We find the set of relevant results by executing SQL queries on the database while generating the workload, as described in Section 5.4. Our results indicated that both MI-Backward and Bidirectional performed equally well on the recall and precision ratios. The recall was found to be close to 100% for all the cases with an equally high precision at near full recall; in other words, almost all relevant answers were found before any irrelevant answer. Note that such high recall/precision are not unreasonable, given that the relevant answers are well defined, and our weighting schemes discourage irrelevant shortcut answers.

6 Related Work

Among other search algorithms, DBXplorer [1] and Discover [9] do not perform result ranking, and are therefore not directly comparable. [8] presents an extension of the Discover algorithm, which they call the sparse algorithm, which adds a simple ranking measure, generates the top-K answers for each candidate network (with some pruning), and merges the results. The other algorithms presented in [8] work only on a class of ranking functions that is monotonic on leaf node scores, and as a result cannot handle edge weights, and Sparse worked best for the AND semantics (which we use). Unlike our algorithms, the search algorithms in [1, 9] and [8] cannot be used on schemaless graphs, and suffer from a problem of common subexpressions.

The ObjectRank algorithm [2] uses a different answer model, making it incomparable, The ObjectRank score of a node with respect to a keyword is designed to reward nodes that are connected to keyword nodes with short and/or many paths. Their answer model does not generate trees that explain answers, and their evaluation algorithm requires expensive precomputation (costing 6 million seconds for a 3 million node database).

Work on finding connecting paths in XML data, such as XRank [7] or Schema-Free XQuery [11], apply only to tree structured data. Real world data is most definitely not tree structured, and not even DAG structured; cycles are common.

Li et al. [10] propose algorithms to find a linked set of Web pages that together contain the set of keywords in a query. Their algorithm is similar to Backward expanding search, but restricted to undirected graphs.

Work on finding paths in semistructured data, such as [12, 13], is related, but their queries supply schema labels whereas our do not, and more importantly they do not rank or prioritize answers. Their hybrid algorithm which combines top-down (forward) search from roots with bottom-up (backward) search from leaves has some resemblance to Bidirectional search, but unlike Bidirectional it cannot go backward from some leaf and then forward to other leaves.

There has been work on prioritization of search when finding shortest connecting paths: e.g. [14] in the context of searching in a state space, and [6] in the context of finding routes on a road network. This body of work is not applicable to the more general problem of finding connecting trees which we address.

7 Conclusions and Future Work

Keyword search on textual data graphs is an emerging application that unifies Web, text, XML, and RDBMS systems and has important applications in heterogeneous data integration and management. We pointed out limitations of earlier keyword search techniques on graphs. We introduced the Bidirectional search algorithm and the novel frontier prioritization technique (based on spreading activation) that guides it. We presented extensive experiments that clearly demonstrate the benefits of Bidirectional search over earlier algorithms.

We have extended our answer ranking model as well as the Bidirectional search algorithm to handle partial specification of schema and structure using tree patterns with approximate matching. Implementing these extensions is part of future work. Other areas of future work include (a) improved "lookahead" techniques for Bidirectional search which can reduce the number of nodes touched, (b) more sophisticated activation spreading techniques that will help reduce the upper bounds on answers not yet computed, allowing faster output of already computed answers, (c) using schema information to reduce search priority on paths that are unlikely to give good answers, and (d) a performance study of alternative activation spreading and prioritization techniques on different data sources, such as Web data with hyperlinks.

Acknowledgments: We would like to thank Arvind Hulgeri and Govind Kabra for their help with the code, and the referees for their feedback.

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*, 2004.

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

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7), 1998.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[6] A. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004.

[7] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRank: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

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

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

[10] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing Web pages by "information unit". In *WWW*, 2001.

[11] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.

[12] J. McHugh and J. Widom. Optimizing branching path expressions. Technical Report 1999-49, Stanford University, 1999.

[13] J. McHugh and J. Widom. Query optimization for XML. *The VLDB Journal*, pages 315–326, 1999.

[14] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[15] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, 1999.