A System for Relational Keyword Search Over Deep Web Data Sources

Fan Wang* Gagan Agrawal* Ruoming Jin†
*Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{wangfa,agrawal}@cse.ohio-state.edu
† Department of Computer Science
Kent State University, Kent OH 44242
{jin}@cs.kent.edu

ABSTRACT

Increasingly, many data sources appear as online databases, hidden behind query forms, thus forming what is referred to as the *deep web*. It is desirable to have a tool that can provide keyword search functionality on such data sources. However, to provide such functionality, we need to address the following challenges. First, we only know query schemas of deep web data sources and the real content of the back-end databases is hidden in web servers. Second, in most cases, no single database can provide all desired data, and many relationships between keywords of interest can only be derived by querying across multiple deep web data sources. Third, deep web data sources are often inter-dependent on each other. This implies that multiple data sources need to be queried in an intelligent order. Fourth, unlike most traditional databases, there is much data redundancy in deep web data sources. On one hand, we can take advantage of such data redundancy to generate multiple valid query plans for a single query. But, at the same time, data source selection and ranking become challenging problems.

This paper considers answering keyword search queries in the context of a deep web integration system. We have developed a bidirectional query planning algorithm which is can generate multiple valid query answering plans based on a *multi-source inter-dependence hyper-graph* model. We also have designed a domain ontology to support data source selection and query answering plan ranking. To the best of our knowledge, our work is the first to address the problem of answering such queries based on a dependence model, while also considering data source selection, on deep web data sources.

Our experiments show that our bidirectional query planning algorithm can generate query answering plans with high relevance score and low execution time and our ontology based data source ranking strategy is effective. For most cases, our algorithm can also generate query answering plans that are as good as the optimal plans generated by an exhaustive algorithm, while taking significantly less time. The quality of results produced by our implementation were evaluated by a collaborating biologist, who found that the answer extracted to be correct and complete.

1. INTRODUCTION

Keyword search is a very popular information discovery method and much recent research has focused on it. Recently, there have been several efforts on developing keyword search tools in traditional databases [14, 13, 17, 9, 22, 1, 2]. Most of this work represents traditional databases as graphs, with nodes in the graphs being the data tuples in each relational table, and edges being the *foreign key relationship* between the tuples. With this representation, graph search algorithms can be used to perform the keyword search.

A parallel trend in data dissemination involves online data sources that are hidden behind query forms, thus forming what is referred to as the *deep web* [10]. As compared to the *surface web*, where the HTML pages are static and data is stored as document files, deep web data is stored in databases. *Dynamic* HTML pages are generated only after a user submits a query by filling an online form. Thus, standard search engines like Google are not able to crawl to these web-sites. At the same time, manually submitting online queries to numerous query forms, keeping track of the obtained results, and combining them together is a tedious and error-prone process.

Many challenges are associated in querying the deep web. First, for deep web data sources, we only know the query schemas, and not the contents of the back-end databases. Second, most deep web data sources are *inter-dependent*, furthermore, many of the inter-dependencies are *multi-source*, i.e., the output results from multiple data sources are needed for querying a particular data source. Thus, for a given user query, a set of data sources may need to be queried in an *intelligent* order to retrieve all the desired information. Third, there can be data redundancy across deep web data sources. A data source selection and ranking strategy is essential for the system.

In this paper, we consider the following scenario. We have multiple correlated online data sources, each of which has one or more query forms. A submitted query using these forms triggers a query on the back-end database. We want our system to support two types of keyword search queries: 1) *Keyword-Attributes Search*, where a user may submit an entity name and one or more attributes, and would like to search based on attributes of interest for the entity, and 2) *Keyword-Keyword Relationship Search*, where a user submits multiple entity names from a domain, and wants to know possible relationships among these names.

We will use the following two motivating examples to explain these types of queries:

Motivating Example 1: Keyword-Attributes Search: Suppose we have a keyword query, Q1={ERCC6,NSYNSNP, ORTH_BLAST}. This keyword query has the following intention: given a gene name ERCC6, we want to find all the

non-synonymous SNPs located in this gene and the BLAST results between this gene and its orthologous genes of non-human mammals. (We will address the semantic issues of the query in Section 2) To answer this query, we need to first query on an SNP database such as dbSNP to find out all non-synonymous SNPs. Then, we use a gene database, such as Entrez Gene, to obtain the encoded protein in human species and other orthologous species. After that, we search a sequence database to find the sequences. Finally, we use the sequences to do an alignment using an alignment database such as Entrez BLAST. From this example, we see that there is a clear query path guiding the search. The query path is determined by the multi-source dependence among the data sources.

Motivating Example 2: Keyword-Keyword Relationship Search Suppose we have a keyword query, Q2={MSMB, RET}. This query means that given two gene names MSMB and RET, a user wants to know what kind of a relation is present between these two genes. Our system needs to determine that the two genes can be connected by a chromosome using database SNP500Cancer. i.e., it turns out that the two genes are located in the same chromosome 10q11.2.

Overall, keyword queries over deep web data sources have specific features that are very distinct from keyword searches over relational databases. First, the number of deep web data sources in a domain can be substantially larger than the number of data tables in a relational database. For example, there are about 500 deep web data sources about SNP (Single Nucleotide Polymorphism) data, which is only a small branch of biology. Second, in many domains, it is common that a keyword query contains as many as 20 keywords, which rarely occurs in keyword queries over relational databases. Third, many relationships of interest can be derived only by querying across multiple data sources.

Our system integrates online biological deep web data sources and represents them as a *multi-source inter-dependence hyper-graph*. A novel bidirectional query planning algorithm generates multiple valid plans for answering keyword search queries. To address data redundancy, our system uses a domain ontology and a novel ranking strategy to select the most relevant set of data sources.

The rest of the paper is organized as follows. We describe our data model for keyword search in Section 2. In Section 3, we introduce our bidirectional query planning algorithm. The domain ontology and dependence graph ranking strategy are introduced in Section 4 and Section 5, respectively. In Section 6, we evaluate our system. We compare our work with related efforts in Section 7 and conclude in Section 8.

2. QUERY AND DATA SOURCE MODEL

In this section, we will introduce the query and data source model we consider in this paper.

2.1 Query Model

A query consists of n, $n > 1$, search terms $t_1, t_2, \ldots, t_n$. The search terms are of two types, which are defined as follows.

Attribute Set AS**:** AS contains all attributes in the studied domain. An attribute is a part of the metadata (column name of the hidden databases) of a deep web data source. In other words, an attribute corresponds to an attribute of an entity in the ER diagram of the deep web data source's hidden databases. For example, suppose data source A has a hidden data table containing a column named Gene_Name, then Gene_Name is an attribute for our discussion.

Entity Set ES**:** ES contains all entity names in the studied domain. An entity name is an instantiation of an attribute. For example, ERCC6 is the name of a particular gene, and we know that Gene_Name is an attribute, so ERCC6 is an entity name.

A query Q is formally defined as $Q = \{t_1, t_2, \ldots, t_n\}$, where $\{t_1, t_2, \ldots, t_m\} \in ES$, and $\{t_{m+1}, t_{m+2}, \ldots, t_n\} \in AS$, $n \geq m$. If $m = 1$ and $n - m > 0$, query Q has one entity name and multiple attributes of interest. This type of query is the keyword-attributes query. If $m > 0$ and $n = m$, query Q has multiple entity names. This type of query is the keyword-keyword relationship query. In our current system, we only support these two types of keyword queries.

It should be noted that semantics is an important issue in keyword search. Semantics are used to determine the scope or type of a keyword and the intention of the query. In our problem, the type of a keyword is determined by a domain ontology, which we will introduce in Section 4. Because the keywords involved in our searches are technical terms from a specific domain, it is relatively easy to decide the intention of each query. In generalizing our system in the future, we will incorporate the existing work [18, 26, 29] on this topic.

2.2 Data Model for a Single Deep Web Data Source

Each online deep web data source has a query interface and an output format. A user can construct a query by specifying some input attributes and constraints. Our model captures the above features of a deep web data source.

In our data source model, we view all the deep web data sources belonging to one sub-domain of biology (such as SNP) as a *virtual relational table*. In this *virtual table*, each data tuple is the query schema of a deep web data source. We call such a data tuple as a *virtual data tuple*. The *virtual data tuples* are connected by the inter-dependence relationship between deep web data sources.

Furthermore, if a single deep web data source has only one query interface (query schema), we model it as one virtual data tuple. If it has multiple query interfaces, we use multiple virtual data tuples to represent it. Each virtual data tuple is modeled as a record with three types of attributes, the *input* attributes, which are required in the query form, the *output* attributes, which are the attributes returned for the corresponding query, and the *inherent constraints*, which are the attribute conditions imposed on the data source by its designer. In addition, we separate the input attributes into two categories, the *must-fill* ones which have to be provided to get the query results and the *optional* ones which can be omitted and only provide extra constraint conditions to narrow down the search space. If the optional attributes are not provided in a query, we assume they will appear in the output attributes.

We denote a virtual data tuple formally as $R(MI, OI, O, C)$, where MI, OI, O and C correspond to the sets of must-fill attributes, optional attributes, output attributes and inherent constraints. For example, data source SeattleSNP has two query schemas, so it is modeled as two virtual data tuples as in Table 1. The first schema takes Gene Name as a must-fill input attribute, Up Base and Down Base as the optional attributes, and SNP_Function and Frequency information as the outputs. It also has an inherent con-

Table 1: Data Model for SeattleSNP Data Source

Data Source	MI	OI	O	C
Seattle	Gene Name	Up_Base Down_Base	SNP_Function Frequency	Organism= Human
Seattle	SNPID	Up_Base Down_Base	Alleles Dis-equilibrium	Organism= Human

straint which means that all data in this data source are from human species. We notice that the model only contain high-level terms, not the real content of the database. For example, we only know that SeattleSNP can return SNP function information, but we do not know which SNP is included in this data source.

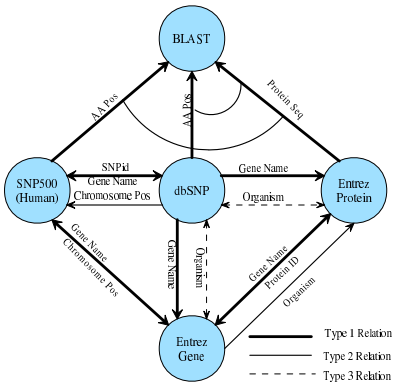


Figure 1: Dependence Relations between Five Data Sources

2.3 Data Model for Inter-Dependent Data Sources

Data sources are connected by the inter-dependence between them. For a certain query, a data source may need to be queried prior to another data source, so as to obtain the necessary input attributes of the second data source.

Consider a group of n deep web data sources,

$$R_1(MI_1, OI_1, O_1, C_1), \ldots, R_n(MI_n, OI_n, O_n, C_n)$$

We assume all their attributes belong to a universal set of attributes. For two deep web sources R_1 and R_2, we define three types of dependence relationships: Type 1) The query output of R_1 can be applied to R_2's must-fill input if $O_1 \cap MI_2 \neq \Phi$; Type 2) the query output of R_1 can be applied to R_2's optional input if $O_1 \cap OI_2 \neq \Phi$; Type 3) the optional input of R_1, which is also part of its output, can be applied to R_2's must-fill input or optional input attribute, i.e., $OI_1 \cap (MI_2 \cup OI_2) \neq \Phi$. The first type of relation shows that R_1 has to be queried before R_2 in order to obtain the necessary input attributes of R_2. The second type of relation shows that if R_1 is queried before R_2, using the output from R_1, we can narrow down the searching scope of R_2 or make the query on R_2 more accurate. The third type of relation is the combination of the first two. These three types of dependencies play different roles when we are generating query answering plans. Figure 1 shows the inter-dependence among five deep web data sources for SNP data. Nodes are virtual data tuples, and three different types of arrows represent the three types of dependence relations above. We can see that dbSNP and Entrez protein form a hyper node for its descendant BLAST, which means that to be able to query BLAST, one needs to query both of dbSNP and Entren Protein first.

Besides the dependence relationship between two data sources, R_1 and R_2 can also share common must-fill input attributes or output attributes. The first case implies that they share the same predecessor in the dependency graph. The second case implies that they can be the predecessors of the same descendant. We call this *data redundancy*.

3. BIDIRECTIONAL PLANNING ALGORITHM FOR KEYWORD SEARCH

We now consider the problem of keyword search over deep web data sources. For keyword search in traditional databases, we have the direct access to the data. As a result, research in literature on keyword search with relational databases takes the set of data tuples in every relational table as nodes, and the foreign key relation between tuples as edges. Relational tables are also connected by foreign key relations forming a *schema graph*. Keyword query answering is done by using graph searching algorithms [14, 13, 17, 9, 22, 1, 2].

Algorithm 3.1: Bidirectional-Query-Planning$(Q, Nodes)$

```
Initialize IL, ∀u∈IL, add u to FQ, FE=Φ, FN=Φ
Initialize BQ, BE=Φ, BN=Φ
Create a pseudo-starting node PS
∀u∈Nodes, ∀j∈Keywords
if j∈ O_u
  distance_{u,j} = 0
  else distance_{u,j} = ∞
∀ui∈IL, ∀j∈Keywords
if j∈ O_u
  decedents_{u,j} = u
  else decedents_{u,j} = null
∀u∈Nodes, ∀j∈Keywords, sibling_{u,j} = null
∀u,v∈Nodes, BitMapGraph_{u,u} = 0
if v∈IL
  BitMapGraph_{PS,v} = 0
  else BitMapGraph_{u,v} = ∞
ContinueSearch=true
while (FQ≠ Φ or BQ≠ Φ) and continueSearch=true
  select the node with the highest priority score from FQ and BQ
  call it nextnode
  if nextnode∈IL
    Explore(PS,nextnode)
  if Find_Answer(PS)
    Plan_Constructor()
    if doesn't need to generate more plans
      continuesearch=false
      else Refresh()
  if nextnode comes from FQ
    Forward-Explore(nextnode)
    else
    if nextnode∉IL
      Backward-Explore(nextnode)
      else Explore(PS,nextnode)
```

Algorithm 3.2: Forward-Explore($node$)

```
foreach v∈Neighbors(node)
  if v∈IL
    Explore(PS,node)
    else
    U=Partners(node,v)
    if U is a single node
      Explore(U,v)
    if U is a hyper-node
      if for all n∈U, n∈FE or n∈BE
        Explore(U,v)
        else put unexplored partners into BQ
    if v∉FQ  and v∉FE
      add v to FQ
    if node∉FQ
      decay the priority score of node
      add node to FN
```

Algorithm 3.3: Backward-Explore($node$)

```
foreach U∈Parents(node)
  if U is a single node
    Explore(U,node)
  if U is a hyper-node
    if for all n∈U, n∈FE or n∈BE
      Explore(U,node)
      else put unexplored partners into BQ
  if node∉FQ  and node∉FE
    add node to FQ
  if node∉FQ
    decay the priority score of node
    add node to FN
  foreach n∈U
    if n∉BQ  and n∉ BE
      add n to BQ
    if n∉BQ
      decay the priority score of n
      add n to BN
```

Algorithm 3.4: Explore(U, v)

```
foreach keyword k
  if U is a single node
    if there is a shorter path from U to k via v
      nextnode_{U,k} = newdis, decedents_{U,k} = v
      propagate the newdis to all reached ancestors of U
  if  U is a hyper-node
    compute an olddis which is the current shortest distance
    from any of the nodes in U to keyword k
    if there is a shorter path to k
      foreach n∈U
        nextnode_{n,k} = newdis, decedents_{n,k} = v
      compute CA=CommonAncestor(U)
      foreach n∈U
        foreach anc∈Ancestor(n)
          if anc∉CA
            propagate newdis to anc
      foreach ca∈CA
        if ca is not an ancestor of another common
         ancestor in CA
          newdis = Max{BitMapGraph_{ca,n}
          +distance_{n,k}}, n ∈ U
          olddis = distance_{ca,k}
          if newdis<olddis
            update the distance
            propagate the distance to ca's ancestor
            decedents_{ca,k} = U
            add sibling information to any node in U
```

Algorithm 3.5: FindAnswer(PS)

```
find=true
foreach keyword k
  if distance_{PS,k} = ∞
    find=false
return (find)
```

For deep web data sources, we do not know the data inside the hidden databases. So, keyword search cannot be directly performed on the data tuple granularity level. Instead, we need to address the keyword search problem at a higher-level (schema-level). Recall that in our data source model, we view all the deep web data sources belonging to one sub-domain (such as SNP) as a *virtual relational table*. In this virtual table, each data tuple is the query schema of a deep web data source, which we had earlier referred to as a *virtual data tuple*.

These virtual data tuples are connected by the inter-dependence relationship between deep web data sources. The number of deep web data sources of a sub-domain is substantially large in most cases, as we pointed out in Section 1, and many sources have multiple query interfaces corresponding to different query schemas. Thus, scale of the graph composed by virtual data tuples is comparable to the scale of the graph composed by real data tuples in traditional keyword search. By making this analogy, we want to adapt graph algorithms onto a higher granularity, in order to solve the keyword searching problem over deep web data sources.

Systems for keyword search on relational database, such as DBXplorer [1] and DISCOVER [14], model the query answering plan as a tree and have the direct access to the actual data. In a well defined relational database, there is not much data redundancy. As a result, the above systems do not consider data table ranking. The number of tuple sets for generating the candidate network used in DISCOVER system is proportional to the size of the power set of the keyword set. It is reasonable for keyword queries on relational database. In our scenario, we do not know the actual data inside the deep web data sources, and because some data sources can be queried in parallel (there is no dependency between them), therefore, a tree is not sufficient for us, instead, we need a forest structure to represent the query answering plan. Furthermore, we need to take the data redundancy into account to perform data source ranking. Finally, it is common to have keyword queries with more than 10 or even 20 keywords. Due to these reasons, we need to propose a new algorithm to solve our problem.

One of the keyword searching algorithms is the bidirectional searching algorithm addressed in [17]. Since our data model is also a graph model, which is analogous with the model used in [17], we can address our problem by building on this bidirectional search algorithm. However, the characteristics of our dependence graph model give us many new challenges. First, unlike the foreign key relation in a relational database, our dependence relation is directed. This requires that our bidirectional algorithm is direction sensitive. Second, since the dependence relation is a multi-source relation, in our new algorithm, we not only need to keep track of the sequential order of node exploration, but also keep track of the parallel sibling relation between multi-source predecessors. Furthermore, we need to come up with a new edge exploration function which can deal with hyper-nodes. Finally, since in our problem, the query answering plan can be a forest with disconnected component, we need

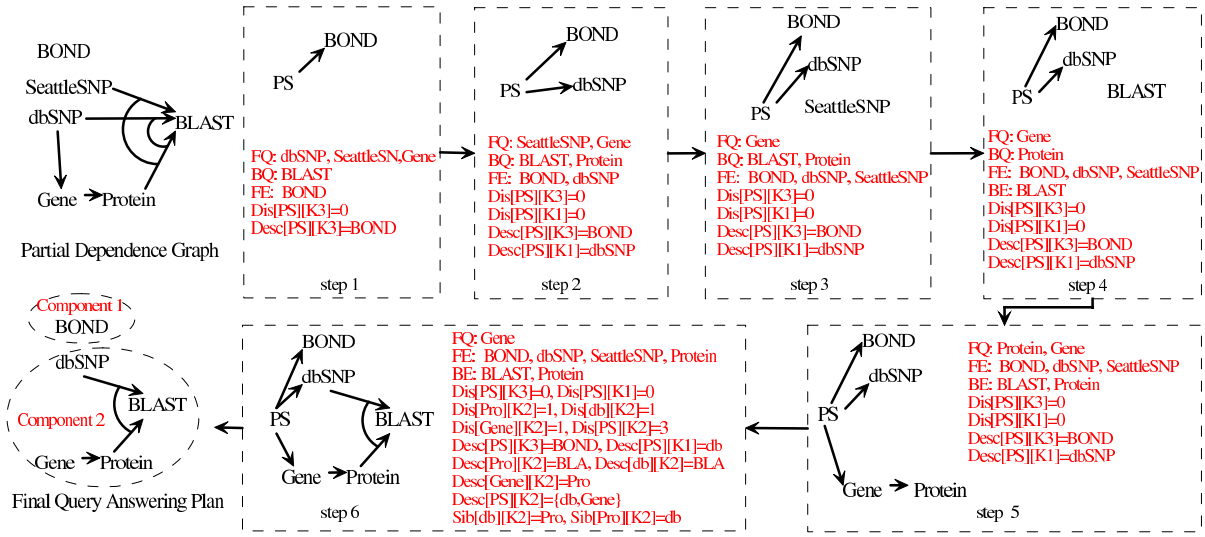


Figure 2: Running Example of a Sample Query

multiple starting nodes to initiate the search on different component.

3.1 Algorithm Overview and Assumptions

Given a keyword query with n keywords and the dependency hyper-graph data model introduced in Section 2.3, we need to first map the keywords onto the nodes in the dependency graph. Next, we want to traverse the graph using some algorithm to find multiple connected components which connect the mapped nodes to form a forest structure. Though we will discuss an example later, an example of the query answering plan forest can be seen in Figure 2. The traversing of the graph is done in a bidirectional manner. A forward exploration touches as many executable data sources as possible from a current data source. But, if one data source has many dependent data sources, it is likely to explore some unnecessary sources as well. If we are sure that some data sources should certainly be queried to answer the given query, we can view them as targets and perform a backward exploration.

To present the algorithm, the following concepts are defined.

Starting Node: From Section 2.1, we know that a query Q in our system must contain at least one entity name. The entity names are served as the *triggering keywords* which initiate the query. The answering of the keyword query must start from the data sources which can take the triggering keywords as their input. We call these data sources the *starting nodes*.

Data Source Necessity: Each data source has a set of output attributes. If an attribute can only be provided by a single data source, that data source should have a higher priority to be selected. Conversely, if the attribute can be provided by multiple data sources, a lower node score can be assigned to these data sources with respect to this attribute. Based on this idea, each term is associated with a *Data Source Necessity* value. Formally, for a term k, if R data sources can provide it as output, the data source necessity value for k is $DSN_k = \frac{1}{R}$. Then, the data source necessity value for a data source d is defined as follows $DSN_d = Max\{DSN_k\}, k \in O_d$.

It should be noted that our algorithm can be used on any domain. We assume that we have a ranking strategy for ranking each data source schema (node in graph) and a dependence relation (edge in graph). Furthermore, we assume that a domain ontology is built which can support the ranking strategy. The usability of the algorithm is independent of the ranking strategy and domain ontology proposed for this specific domain.

3.2 Algorithm Details

In this section, we first introduce the data structures we use in our algorithm, followed by the detail algorithm description. Algorithms 3.1, 3.2, 3.3, 3.4 and 3.5 show the pseudo-code.

3.2.1 *Data Structures*

Suppose there are N nodes in the dependence graph and K keywords in the query. Our bidirectional query planning algorithm uses the following data structures (some of these are based on [17]).

InitialNodeList IL: A list containing all the starting nodes of the search.

ForwardQueue FQ: A priority queue containing all nodes which are ready to be explored in the forward fashion.

ForwardExplored FE: A queue containing all nodes that have already been explored in the the forward manner.

ForwardNextRound FN: A priority queue containing all nodes which have already been explored in the forward fashion in the current round, and ready to be explored in the forward manner in the next round.

BackwardQueue BQ: A priority queue containing all nodes which are ready to be explored in the backward manner.

BackwardExplored BE: A queue containing all nodes that have already been explored in the the backward manner.

BackwardNextRound FN: A priority queue containing all nodes which have already been explored in the backward manner in the current round, and ready to be explored in the backward manner in the next round.

distance[N][K]: An array contains the shortest distance from any node to any keyword. The shortest distance is computed in terms of dependence graph edge score, which we will de-

fine later.
descendants[N][K]: An array maintains the next set of nodes that need to be visited in order to obtain the shortest distance from any node to any keyword.
sibling[N][K]: An array maintains a set of siblings (partners) needed to obtain the shortest distance from any node to any keyword. This data structure is designed for taking care of hyper-node predecessors.
BitMapGraph[N][N]: An array maintains the shortest distance for every pair of nodes in the graph.

3.2.2 Algorithm Initialization

Because of the nature of the deep web queries we are considering, a valid query answering plan can be a forest with disconnected components, as in the example shown in Figure 2. This feature requires that we need a starting node for each disconnected component which results in multiple starting nodes. The original Bidirectional search algorithm uses the idea of Dijkstra algorithm, which solves the single-source shortest-paths problem on a weighted directed graph [7]. In order to solve our multi-source problem, we need to convert a multi-source shortest path problem to a single source shortest path problem. Based on this idea, we add a pseudo-starting node (PS) into our graph. PS serves as single entering point for the search. It is connected by a pseudo-dependent relation edge with each data source node in the initial list with edge score of zero. This means that there is no cost to travel from PS to any of the actual starting nodes. We can use PS to check whether an answer has been found or not for a query. If the distance from PS to any keyword is a finite number, an answer is found.

The data structures of the algorithm are initialized as follows: We add the starting nodes to the initial list and the forward queue. Then, we find all data source nodes whose output attributes cover any of the non-triggering keywords into the backward queue. All other queues are initialized as empty. If a data source covers a certain keyword, the distance from this source to the keyword in the distance array is set as 0, otherwise the value is set as infinity. Other arrays are initialized similarly.

3.2.3 Edge Exploration

The bidirectional search is performed within a loop until the top k query answering plans are found. At each round, we select the node with the highest node score from the forward and backward queue. Edge exploration is conducted based on the queue from which the highest scored node is selected. In forward exploration, all out-going neighbors of the current node will be explored. In backward manner, all in-coming parents of the current node will be explored. In edge exploration, we take two nodes. One is the predecessor denoted as U, the other one is the descendant denoted as v. Forward edge exploration is performed from the predecessor (U) to the descendant (v) and backward edge exploration is performed from v to U.

If U is not a hyper-node, i.e., it is a single node, we call the *Explore* function directly. If U is a hyper-node (U is composed of multiple nodes), first, we check whether all the predecessors in U have been explored or not. If any of the predecessors have not been explored, we will skip the exploration in this round and add the unexplored predecessors into backward queue. This is because the accessibility of the dependent node depends on the accessibility of all its predecessors, as a result, if any one of U is not accessible currently, we cannot access v. Only when all predecessors are explored, we can call the *Explore* function to do edge exploration.

We perform edge exploration between U and v as follows. If U is not a hyper-node, the *Explore* function just updates the shortest distance information from U to any keyword via v and propagates the updated information to U's ancestors if necessary. If U is a hyper-node, we use a different propagation strategy. We obtain all common ancestors of U to form a common ancestor set CA. A common ancestor ca is a node which is an ancestor of all nodes in U. For any node n in U, we propagate the distance information to n's ancestors as normal except the ancestors in CA. This is because the shortest distance from any common ancestor to a keyword depends on all nodes in the set U. The distance from a common ancestor ca to a keyword is the longest distance from ca to a keyword via any of the node in U. If this distance is smaller, we update the distance information on ca. Then we add U as the descendants of ca. The sibling information between nodes in U is also updated. Finally, we propagate the distance information to any ancestor of ca. The example in Section 3.3 shows the idea.

After the exploration of edge (U, v), we need to update FQ and BQ. In the forward manner, v is added to FQ because we want to continue to explore from v. In the backward manner, we first add U to BQ, because we want to continue to explore from U backwardly; second, we add v to FQ in order to explore the frontier of v.

In order to detect an answer as soon as it has been generated using PS. We need to obtain the distance information from PS to any keyword as soon as possible. As a result, we do the pseudo-dependent edge exploration frequently. This is the reason we invoke the edge exploration function whenever we detect the current being explored node is in the initial list.

3.2.4 Kernel Nodes and Decay Factor

An important difference between our algorithm and the algorithm in [17] is that in their algorithm, as long as one node is explored, it will be put into the explored queue. Then, it can never be used again for the current query when generating other query answering paths. But in our problem, some keywords can only be provided by a single data source, which has DSN value of 1. It must be reused for generating other valid query answering plans. We call the data sources of this kind the *kernel nodes*.

Kernel nodes are the nodes we want to reuse. There are also nodes we do not want to reuse when generating the next query answering plan for a query. If a keyword can be provided by multiple data sources with a similar score, we want to change the score of the used source and give other sources a chance to be selected while generating the next plan. This is to avoid missing possible new query answering plans. We introduce a *decay factor* β for each node which is going to be put into FN or BN queues. The decay factor β will decrease the score of the node according to the node's possibility of being a kernel node. The decayed score of node n would be $\beta \times NScore(n)$. In future exploration, node n will use the decayed score, not the original score. Formally, we define the decay factor of a node n to be $\beta_n = DSN_n$, where DSN_n is the data source necessity value introduced in Section 3.1. Here, if the DSN value of the node n is large, for example $DSN_n = 1$, this means that this node is a kernel node, we do not want to decay this node's score,

as we can see the decayed score would be $1 \times NScore(n)$. But if a node has a very small DSN value, it means many other data sources have the same attributes as this node, as a result, the score of this node will be severely decreased. In this way, other similar nodes can be used in later query plans.

3.3 Example

We give a simple example of the algorithm proposed above to illustrate the idea. We focus on the general idea in our description, and the actual execution of the algorithm is much more complex than what we discuss here.

We have a keyword query

$$Q = \{ERCC6, NSYNSNP, MOLA, ORTH\_BLAST\}$$

and six data sources dbSNP, SeattleSNP, Gene, Protein, BOND and BLAST. ERCC6 is a gene name. NSYNSNP (K1) is covered by dbSNP and SeattleSNP, ORTH_BLAST (K2) is covered by BLAST and MOLA (K3) is covered by BOND. For simplicity, we assume the score of each edge in the dependence graph is 1. We run the bidirectional planning algorithm to find a query answering plan for this query. Initially, the IL list contains data source nodes dbSNP, SeattleSNP, Gene and BOND, because these four data sources have Gene_Name as their input attributes. FQ is initialized the same as IL and BQ contains BLAST at the beginning. A partial dependence graph is constructed and shown in Figure 2. The order of the data sources according to their node scores is BOND, dbSNP, SeattleSNP, BLAST, Protein, Gene (from high to low). Figure 2 shows the six steps used in this example. At each step, we display the state of main data structures after the execution of the step.

At the first step, BOND is selected and because BOND is in IL, we explore the edge (PS,BOND). Keyword $K3$ is reached, and the corresponding values in the distance and descendant array are updated. At step 2, dbSNP is selected to be explored. Because dbSNP is also in IL, the edge (PS,dbSNP) is explored and keyword $K1$ is reached. We try to forwardly explore the edge ({dbSNP,Protein},BLAST), in which {dbSNP,Protein} forms a hyper-node predecessor for BLAST. Since Protein has not been explored yet, we skip this exploration but add Protein to BQ to make it as a target want to be explored in backward fashion. At step 3, we try to explore using SeattleSNP, suppose the distance from PS to $K1$ via SeattleSNP is no shorter than the distance using dbSNP, the exploration of SeattleSNP will not change the state of the data structure. At step 4, we try to explore the edge ({dbSNP,Protein},BLAST) in a backward manner. Since Protein is not explored yet, we skip this exploration. At step 5, Protein is selected from BQ to do a backward exploration using the edge (Gene,Protein). After this exploration, the nodes Gene and PS are connected. But $K2$ is still not reachable. We add Protein into FQ wishing a forward exploration from it. At step 6, Protein is selected from FQ, and a forward exploration on edge ({dbSNP,Protein},BLAST) is performed. Keyword $K2$ is reached, and we update the distance and descendant information. Because PS is a common ancestor of dbSNP and Protein, we cannot directly update the distance from PS to $K2$ via dbSNP and Protein separately. The distance from PS to $K2$ is the longest distance between the two possible paths. The first one is passing through dbSNP, and the second one is passing through Gene and Protein. The final query answering plan is shown in Figure 2, and we can see that it is a forest with two disconnected components.

Then, the data sources dbSNP, Gene, Protein, BLAST and BOND will be put into FN and BN respectively and their scores will be decayed properly. In the next round, SeattleSNP is likely to be selected instead of dbSNP to form a new query answering plan. Because BLAST has decay factor of 1 (only BLAST can cover $K2$), it is served as a kernel node, and it will be included in the next query plan again.

3.4 Query Plan Construction

We use a two dimensional, $N \times N$, array $QueryBitMap$ to store a query answering plan, where N is the total number of nodes in the dependence graph, excluding the pseudo-starting point. Taking the pseudo-starting point as the starting iterator, we follow the *descendants* data structure. If the descendants of a node u is a set of nodes $v_1, \ldots, v_m$, we mark $QueryBitMap[u][v_i] = 1$. If we meet a node with siblings, we obtain its sibling and create another iterator starting from that sibling, continuing to complete query construction. The $QueryBitMap$ array will serve as the unique identification of a query answering plan.

Given the $QueryBitMap$ of a query answering plan, we use graph topological sorting algorithm to obtain the query order. The query order will have several levels, with each level containing all the data sources which can be executed in parallel.

4. DOMAIN ONTOLOGY

In this section, we will introduce the design of our domain ontology and explain how we use ontology to support node ranking.

4.1 Ontology Design

Our domain ontology is designed with the following goals. First, we want to identify the relationship between domain terms, which could appear as keywords, to understand the intent behind a query. For example, if a query contains keyword human and from the ontology we know human is a type of organism which is used to categorize genes, we would know the query uses keyword human to impose a constraint on search scope. Second, a user may not use the exact scientific terms which are used by the data sources. She may use some aliases or synonyms or even some words with a fuzzy meaning instead of the exact scientific terms in a query. We need to map these terms into a form in which all terms are recognizable by the data sources. For example, if a query contains keyword SNP_Frequency, we need to map SNP_Frequency which is an abstract term into the exact terms genotype frequency and alleletype frequency. Third, as we had mentioned earlier, there is data redundancy in deep web data sources. We want to obtain a data source quality score for each keyword by using domain ontology.

Our ontology contains attribute terms (and a few concept terms from the same domain). The ontology does not contain entity names. For example, Gene_Name, which is an attribute term, is included in the ontology, but ERCC6 or other actual gene names are not included in the ontology. In this sense, our ontology is a *schema level* ontology. Because the number of attribute terms is likely quite limited in a domain, our ontology remains small and scalable to a large number of data sources. As a query can also contain names such as ERCC6, we use heuristics to map it to the attribute term Gene_Name in the ontology.

Our ontology is a connected directed graph $OG = (ON, OE)$. ON is the set of nodes in the ontology graph, and OE is set of edges. The nodes in ontology graph are the domain terms and edges are relations between these terms.

Domain Term: There are three types of domain terms in the ontology: biological concept terms, attribute terms, and a single special Root term. Biological concept terms are high level conceptual terms such as Chromosome and SNP which are not among the input or output attributes of any deep web data sources. Attribute terms were introduced in Section 2.1 as the set AS. An attribute term can also be a synonym or high-level abstract term of other attribute terms. For example, Human is a synonym of Homo Sapiens and SNP_Frequency is a high level attribute term which covers four lower-level attribute terms.

Ontology Relation: We define four types of ontology relation. 1). *A type relation.* It connects a term with its synonym. An A type link comes from a term within the vocabulary of a data source to its alias or synonym which is not within that scope. For example, there is an A type link pointing from Organism to Species. 2). *B type relation.* It connects two biological concept terms, which are related in the domain of ontology. For example, biological concept terms Gene and SNP are connected by a B type relation because a SNP is located in a specific gene. B type link is undirected. 3). *C type relation.* It captures the class-subclass relationship. A C type link points from a biological concept term to an attribute term by which the concept term can be categorized into several sub-classes. For example, SNP and SNP_Function are connected by a C type relation, because SNP can be divided into several different classes by its SNP_Function attribute. 4). *F type relation.* It connects a biological concept term with its attribute terms or connects a high-level abstract attribute term with its low level attribute terms. For example, SNP_Frequency is connected to its four low level attribute terms population, sample, genotype frequency, and alleletype frequency by four F type relations.

4.2 Data Source Representatives

We introduce a new concept, *representative of a data source.* Each data source schema has a list of output attributes which characterize the main focus or interest of the data source. If we know the main focus of a data source, we can use this information to rank the data source's relevance with respect to a keyword. For example, data source A has 20 output attributes. We find that 15 output attributes come from the concept term SNP, and the rest 5 comes from the concept term Protein. With this information, we roughly know that data source A is about SNP and protein data, and the main focus of A is providing SNP information. Data source B has 10 output attributes, and we find that all the 10 terms come from the Protein node. Data source B is built solely for providing protein information. Given the above information, if a user keyword can be mapped to the protein node in the ontology graph, we can have a high confidence in concluding that the data source B can provide more relevant information than A.

We define the *representatives* of a data source to be all biological concept terms in the ontology which can be reached by reverse traversing F or C type links starting from the data source's output attribute. Suppose a data source D has n representatives $r_1, r_2, \ldots, r_n$, and each representative r_i is associated with a weight w_i. The weight w_i is the ratio between the number of output attributes of D which finally reach to biological concept term r_i and the total number of output attributes of D. In other words, if a greater number of output attributes of D can be mapped to a representative r_i, this representative will have a higher weight. This is because we believe that the data source D is more likely to be focusing on providing information about r_i.

5. RANKING STRATEGY

In this section, we will introduce the ranking strategy used in our current implementation. Our ranking strategy has three components, which are node ranking, edge ranking, and query answering plan ranking. We first outline the desired properties for ranking functions, then we define our specific ranking function.

5.1 Desired Properties and Main Ideas

For our bidirectional search to be efficient, the node ranking function should give a data source higher node score if the node has the following properties: 1) it can cover more keywords, 2) it is easier to reach from the set of starting nodes, 3) it provides the data with higher quality, and 4) it satisfies the constraints which are specified in the query.

Similarly, an edge should be given higher priority if the exploration of this edge can help to narrow down the search space or provide more accurate answer to a query. For example, if an edge e between node u and v contains two types of dependence relations, which are the first type and the second type introduced in Section 2.3, we know the second type of dependence relation can be considered as providing supplemental information, which may shrink the search space. In this case, the edge e should be ranked higher than other edges which only have the first type of dependence relation.

Finally, a query answering plan should be ranked higher if its nodes and edges are highly ranked.

5.2 Node Ranking Strategy

Suppose a query contains a set of keywords $Q = \{k_1, \ldots, k_m\}$. A node corresponds to a data source schema. Given a node n_i, we first define the node score of n_i with respect to query keyword k_j as $NScore(n_i, k_j) = c_{ij} \times q_{ij} \times DSN_{k_j}$. Here, c_{ij} is the node coverage score of the node n_i with respect to k_j, and q_{ij} is the node quality score of node n_i with respect to k_j. If multiple data source can provide this keyword k_j, i.e. DSN_{k_j} is small, the score of this node n_i is decreased.

Node Coverage Score: The node coverage score of node n_i with respect to k_j is defined as $c_{ij} = \frac{1}{Level(i)+1} \times Is\_Contain(j)$. $Level(i)$ is the shortest distance in terms of the number of edges from any of the starting points to the current node n_i. $Is\_Contain(j)$ is a function which returns 0 if data source n_i does not contain k_j as its output, returns 1 otherwise. Note that the $Level(i)$ value can vary across queries, because different queries have different starting points. This ranking function gives a node higher score if it covers more keywords and is easier to reach.

Node Quality Score: The node quality score of a node n_i with respect to k_j is defined as $q_{ij} = OntoScore(i, j)$. The function $OntoScore(i, j)$ returns a relevance score of node n_i with respect to k_j. The intuition behind this function is as follows. We obtain the representatives of a data source which illustrate the focus of the source, then we try to compute a kind of distance between the keyword and the representatives of the source. The shorter the distance, the

closer the keyword to the focus of the data source, and the higher the relevance.

To compute $OntoScore(i, j)$, first, we find the representatives of data source n, as $r_1, r_2, \ldots, r_m$ with weights $w_1, w_2, \ldots, w_m$. For each representative r_i, we compute the least common superclass of r_i with respect to the keyword k following the general definition of Learning Accuracy from Cimiano et al. [6]: $lcs(k, r_i) = argmin_{c \in Ontology}\{\delta(k, c) + \delta(r_i, c) + \delta(Root, c)\}$, where $\delta(a, b)$ is the shortest distance between node a and b in the ontology. Then, we compute the similarity score between r_i and k as follows:

$$Sim(r_i, k) = \frac{\delta(Root, f) + 1}{\delta(Root, f) + 1 + \delta(r_i, f) + \delta(k, f)}$$

where $f = lcs(r_i, k)$. We define the ontology quality score of data source n_i with respect to keyword k_j as follows: $ontoScore(i, j) = \sum_{y=1}^{m} w_y \times Sim(r_y, k_j)$, where m is the total number of representatives of node n_i and w_i is the weight of representative r_i. The node quality score $q_{ij} = ontoScore(i, j)$.

Score of Matched Constraints: Some keywords represent constraints that a user sets on the query. For example, the keyword "Human" implies that the user wants to find data about humans. Each data source has a C attribute which represents the inherent constraints of the data source. We prefer to use a data source which has inherent constraints matching with the user specified constraints. The higher the number of matched constraints, the higher the node score. We use function $CountConstraints(n_i, Q)$ to compute the score of matched constraints of a data source with respect to a query. If a query has a constraint set UC, and the node n_i has inherent constraints set NC, We compute the score of matched constraints based on the following cases: 1). $UC = \Phi$ and $NC = \Phi$. In this case, both the query and the data source do not have any constraint, we simply simply return zero. 2). $UC = \Phi$ and $NC \neq \Phi$. In this case, user does not set any constraints, but the current data source has inherent constraints. The data source inherent constraints will shrink the answers to a narrower range, as a result, the data source should receive a penalty. We return $-|NC|$. 3). $UC \neq \Phi$ and $NC = \Phi$. In this case, user sets constraints, but data source does not have constraints. We return zero. 4). $UC \neq \Phi$ and $NC \neq \Phi$ and $HasConfliction(UC, NC) = true$. $HasConfliction()$ is a function to detect whether there is any confliction between UC and NC. For example, user sets constraint on "Organism=Human", but the data source has constraint "Organism=Mouse". In this case, this data source definitely cannot be chosen for this query. So we will return a special value *null*. 5). $UC \neq \Phi$ and $NC \neq \Phi$ and $HasConfliction(UC, NC) = false$. In this case, we return $|UC \cap NC| - |NC - UC|$. $|UC \cap NC|$ is the number of user specified constraints which are also matched in data source constraint set. $|NC - UC|$ is the number of data source constraints which are not requested by the user.

Node Score for A Single Node: Now we define the node score for a node n_i as

$$NScore(n_i) = \begin{cases} null & \text{if } \varphi, \\ \frac{\sum_{k_j \in Q} NScore(n_i, k_j)}{m \times (|NCM|+1)} & \text{if } \chi, \\ \frac{\sum_{k_j \in Q} NScore(n_i, k_j) \times (|NCM|+1)}{m} & \text{if } \psi. \end{cases}$$

φ represents $CountConstraints(n_i, Q) = null$. χ represents $CountConstraints(n_i, Q) = NCM$ and $NCM < 0$. ψ represents $CountConstraints(n_i, Q) = NCM$ and $NCM >= 0$. m is the number of keywords in query Q. The above node score function considers the effect of node coverage score, node quality score, and score of matched constraints.

Node Score for Query Answering Plan: The node score of the query answering plan QAP is defined as

$$NScore(QAP) = \frac{\sum_{\forall n_i \in QAP} NScore(n_i)}{N}$$

where N is the number of nodes in QAP. We can see that the node score of query answering plan QAP is roughly the average node score for all nodes in the plan. This function gives penalty to a plan with many nodes (longer plans with redundant nodes).

5.3 Edge Ranking Strategy

The edge score is considered as a cost one needs to pay to traverse from one node U to its descendant V. Therefore, a higher ranked edge has a lower score. An edge can be built between U and V and a constant score is assigned to it only when there is a first type dependence relation between them. According to the desired properties in Section 5.1, if there exists second and/or third type dependence relation, we reduce the edge score, essentially giving it a bonus. Along this line, the edge score of e connecting U and V can be defined as

$$EScore(U, V) = \begin{cases} \infty & \text{if no first type dependence relation between } U \text{ and } V, \\ 4 - T1 - T2 - T3 & \text{if has first type dependence relation between } U \text{ and } V. \end{cases}$$

$$T1 = \begin{cases} 0 & \text{if no second type dependence relation between } U \text{ and } V, \\ 1 & \text{if has second type dependence relation between } U \text{ and } V. \end{cases}$$

$$T2 = \begin{cases} 0 & \text{if no third type dependence relation between } U \text{ and } V, \\ 1 & \text{if has third type dependence relation between } U \text{ and } V \\ & \text{and it is pointing from optional input to must-fill input.} \end{cases}$$

$$T3 = \begin{cases} 0 & \text{if no third type dependence relation between } U \text{ and } V, \\ 1 & \text{if has third type dependence relation between } U \text{ and } V \\ & \text{and it is pointing from optional input to optional input.} \end{cases}$$

The total edge score for query answering plan QAP is defined as:

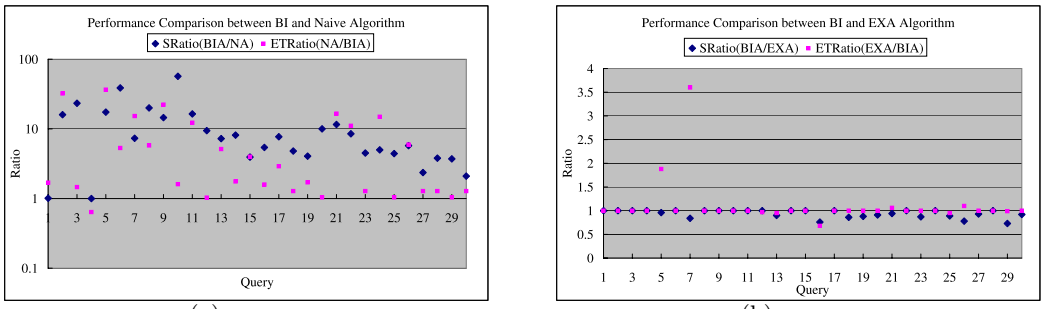
$$EScore(QAP) = \frac{1}{\sum_{\forall e \in QAP} EScore(U, V)}$$

5.4 Query Answering Plan Ranking Strategy

Combining the node and edge ranking functions above, the query answering plan ranking function is a linear combination of its node score and edge score, which is defined as :

$$Score(QAP) = \lambda \times NScore(QAP) + (1 - \lambda) \times EScore(QAP)$$

In our current system, we set the parameter λ to be 0.5. We prefer query answering plans that have a higher score.



(a) (b)

Figure 3: Comparison among BIA, NA and EXA: (a) Comparison between BIA and NA;(b) Comparison between BIA and EXA.

Table 2: Query Statistics

Query ID	Number of Terms
1-10	2-5
11-18	8-12
19-24	17-23
25-28	27-33
29,30	37-43

6. EVALUATION

This section describes the experiments we conducted to evaluate our algorithm.

6.1 Experiment Setup

Our evaluation was done using 14 different biological deep web databases we have integrated. We created 30 queries for our evaluation. Among these 30 queries, 10 are real queries specified by a domain expert we have collaborated with. The remaining 20 queries were generated by randomly selecting query keywords from the ontology to form queries. We create two types of queries, keyword-attributes queries and keyword-keyword relation queries. Among the 30 queries, 22 queries are of the first type, and 8 queries are of the second type. We also vary the number of terms in each query in order to evaluate the scalability of our algorithm. Table 2 summarizes the statistics for the 30 queries.

Among our queries, we have several queries with a large number of keywords. We choose these queries for the following reasons. Unlike the traditional relational database queries in commercial domains, a biological domain query is very likely to have a large number, i.e., 20 or more, keywords. This is because users tend to use high-level abstract terms in their keyword search. For example, SNP_Frequency, a very common query keyword, is a high-level abstract term that corresponds to four low-level keywords. Another reason for creating long queries was that we wanted to test our algorithm in extreme cases.

In the evaluation, we compare our BIdirectional Algorithm (BIA) with three other algorithms, which are the Naive Algorithm (NA), the EXhaustive Algorithm (EXA) and the Backward Algorithm (BA).

Naive Algorithm: As the name suggests, this algorithm does query planning in a naive way. The algorithm selects all data sources which can be queried at each round, until all keywords are covered. This algorithm can quickly find a query answering plan, but it is likely to have a very low score and a long execution time.

Exhaustive Algorithm: This algorithm searches the entire space in a recursive manner, and compares every possible query answering plan. Then, it selects the plan with the highest score. This algorithm always finds the optimal answering plan, but has high time and space requirements.

Backward Algorithm: This algorithm uses exactly the same data structures as the bidirectional planning algorithm. The only difference is that backward algorithm can only search from the backward direction.

6.2 Evaluation Metrics

Query Answering Plan Score: We use the ranking function introduced in Section 5 to compute a score for each query answering plan. We prefer the plan with a higher score, because higher score implies that a smaller number of data sources are involved, and they have higher relevance.

Query Answering Plan Estimated Execution Time: The query execution time of a deep web data source is estimated by issuing multiple randomly selected sample queries. Since the query answering plan has a disconnected component and some sources can be executed in parallel, the estimated execution time of the entire plan is computed based on the parallel execution model. The longest path (in terms of time) in the plan determines the execution time.

Query Planning Time: Efficiency of query planning algorithm is an important consideration. We record the query planning algorithm's running time.

Planning Time for Generating the First Query Plan: BIA and BA algorithms can generate multiple query answering plans. We record the time used to generate the first (possibly the best) plan. We prefer the algorithm which can generate the first plan quickly.

Average Query Planning Time: For BIA and BA, we compute the average query planning time by dividing the total query planning time by the number of query plans generated.

In addition to comparing different algorithms using the above metrics, we have also evaluated the scalability of the bidirectional planning algorithm with respect to the number of data sources involved in the query.

One important observation is that a query plan with a

Table 3: Query Planning Time Comparison Between BIA, NA and EXA

Selected Query	NA Planning Time (ms)	EXA Planning Time (ms)	BIA Planning Time (ms)
1	6	1385	30
2	3	774	29
3	5	1020	131
4	15	3090	62
5	4	615	48

high score does not necessarily have a low estimated execution time, and vice versa. The reason is that a very highly ranked data source may have longer execution time. As a result, it is possible to see a query answering plan with high score but a large execution time.

6.3 Experiment Results

6.3.1 Comparing BIA against NA and EXA

In Figure 3, sub-figure (a) shows the comparison between BIA and NA. It is plotted in logarithmic scale. The SRatio (Diamond) is the ratio between BIA's generated query answering plan's score and NA's generated query answering plan's score. The ETRatio (Rectangle) is the ratio between NA's generated plan's estimated execution time and BIA's generated plan's estimated execution time. Figure (b) shows the same comparison between BIA and EXA. In both the sub-figures (a) and (b), the x-axis is the query ID, and the y-axis is the ratio value.

From sub-figure (a), we can see that except for queries 1 and 4, the plans generated by BIA always have much higher (more than 5 times) score than the plans generated by NA. Similarly, the execution time of the plans generated by BIA are always lower than the execution time of plans generated by NA. For queries 1 and 4, NA can also obtain very good results. This is because these two queries are very simple, and only need a single data source. For the query 4, the execution time of NA's query plan is even shorter than that of BIA's query plan. This is reasonable, because a data source with higher relevance may have long execution time.

From sub-figure (b), we can observe that the score of plans generated by BIA are almost the same as the score of the plans generated by EXA. This shows that the quality of the plans generated by BIA is very close to the quality of the optimal plans generated by EXA. The execution time has the same pattern as the score, except for queries 5 and 7. For these two queries, we can see that the estimated execution time for query plans of BIA are even shorter than the optimal query plan's execution time. This is reasonable because our optimal query plan is generated based on query answering plan score and BIA algorithm, for this two queries, selects the data source with a lower execution time.

In Table 3 we show the comparison of query planning time between BIA, NA, and EXA. We can clearly see that BIA takes much less time than EXA. Compared to the time of NA, BIA's running time is still modest.

In summary, BIA outperforms NA in terms of generated plan's quality. BIA can generate nearly optimal query answering plans while take much less time than EXA.

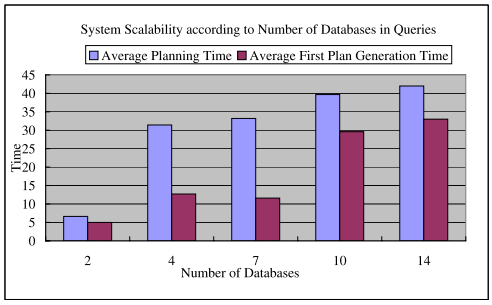


Figure 5: The Scalability of BIA according to Number of Data Sources Involved in Queries.

6.3.2 Comparing BIA against BA

In Figure 4, sub-figure (a) shows the SRatio and ETRatio between BIA and BA. Because BIA and BA use the same data structure, we expect both the ratios should be near 1. We can observe that this is actually the case. We can also note that for some queries (10 out of 30), BIA obtains answers with higher scores than BA. This shows that in terms of the quality of query answering plan, BIA and BA have nearly the same performance, with BIA outperforming BA in some cases.

Because BA searches only from one direction, we expect that BIA will beat BA in terms of query planning time. Sub-figure (b) and (c) are plotted in logarithmic scale. In Figure 4, sub-figure (b) shows the ratio between BA's average query planning time and BIA's average query planning time. We can observe that for most queries, BIA takes smaller amount of time to generate a query plan. The sub-figure (c) shows the time ratio for the first generated query plan between BA and BIA. We have the same observation, i.e. BIA can generate the first query plan much faster than BA.

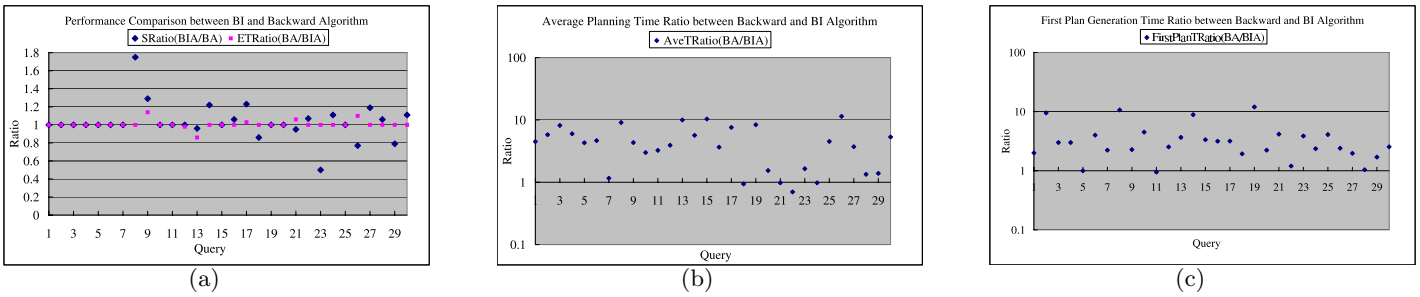
6.3.3 The Scalability of BIA:

From Figure 5, we can observe that in terms of the average planning time, there is a sharp increase when the number of data sources increases from 2 to 4. Then, the planning time increases moderately with respect to the number of data sources. In terms of the first plan generation time, there is a sharp increase when the number of data sources increases from 7 to 10, otherwise, the increase is moderate. This shows that our system has good scalability.

6.3.4 Actual Query Result Evaluation of BIA

The answers retrieved by our system were checked by a biologist collaborating with us. Currently, we have wrappers for 8 deep web data sources, out of the 14 we used for query planning. For the plans that only extract data from these 8 sources, the plans are automatically executed and answers are retrieved and tabulated. For all other plans, the answers were extracted manually. Both automatically and manually retrieved answers were presented to the biologist.

From the feedback from the domain scientist, all answers to the 30 experimental queries are correct and sufficient, with the exception for one query. The query is "rs7412, rs12982192", which is intended to find the relationship between the two SNPs. The expected answer was that the two SNPs are located in the gene APOE and the chromosome 19. Our system can correctly find the first relation-



(a) (b) (c)

Figure 4: Comparison between BIA and BA: (a) Query Plan Comparison between BIA and BA;(b) Average Planning Time Ratio between BA and BIA;(c) First Plan Generating Time Ratio between BA and BIA.

ship. For the second relationship, our system indeed finds a correct query answering plan, but because the data source our system chose has incomplete data (rs12982192 are not in its database), the relationship is not discovered. This limitation can be addressed in the future by having additional knowledge about the data sources. Our system can also enable execution of another query plan on request from the domain scientist, which may retrieve data from different sources.

7. RELATED WORK

We now compare our work with existing work on a number of related topics, including query planning, keyword search on relational databases, database ranking, deep web mining, and semantic search.

Query Planning: Query planning has been extensively studied in databases. Raschid and co-workers have developed a navigational-based query planning strategy for mining biological data sources [3, 19].They build a source graph representing integrated biological databases and an object graph representing biological objects in each database. The key difference in our work is we focus on deep web data source dependencies, not the physical links between the database objects

Much work on query planning is based on the well known *Bucket Algorithm* [9, 8, 16, 24, 21, 20, 25]. In this work, it is assumed that the user query specifies the databases or relations that need to be queried. The task of the work is to find a query order among the specified relations or databases. Based on user specified relations or sub-goals, a bucket is built containing all the databases which can answer the corresponding sub-goal. In comparison, in our work, the user query only contains keyword and does not specify any databases or relations of interest. Our system takes advantage of domain ontology and data redundancy to select the best data sources automatically. In other words, our system figures out sub-goals by itself. At the same time, query planning is also performed by the system.

In [27], a query planning algorithm is presented which minimizes the query's total running time by optimally exploits parallelism among web services. The main differences between our work and theirs are: first, they assume that one attribute can only be provided by one data source, and second, they assume all available databases should be used in the query plan, and their focus is ordering them to minimize the query plan running time. We use a domain ontology to rank databases and select the most relevant data sources to do query planning and for answering the query.

Keyword Search on Relational Databases: Recently, performing keyword based search over relational databases has attracted a lot of attention [14, 13, 17, 22, 1, 2, 23]. In relational database keyword search, databases are represented as graphs. Each row in relation table is represented as a node, and foreign keys are represented as edges. The graph stores the actual data in the databases. The critical difference in our algorithm are as follows. We also represent the entire deep web as a graph. The nodes in our graph are deep web data source query schemas, and as a result, our graph model only contains the high level abstract information for each data source. We do not know the actual content of the data sources. The edges in our graph are inserted based on multi-source inter-dependencies, which is a directed relationship. This is different from the foreign key relationship, which is single-source and undirected. The multi-sources dependence relationship determines that our graph is a hyper-graph with multiple hyper-nodes. As a result, our search algorithm is adapted to this special feature of the graph. Finally, data redundancy is an important consideration. We use ontology to rank data sources according to their relevance to user query.

Database Ranking: There has been some efforts on database ranking. In [28], the authors rank relational databases' relevance with respect to a set of keywords. The number of joins needs to be made to connect two keywords within one database is used as the measurment. In [15], a search result ranking strategy in the presence of overlapping data sources is addressed. A surprisingness and confidence score are computed for each search result. The results are then ranking by these two scores. In our work, we are considering data redundancy in deep web data sources, not in relational databases. We use domain ontology and similarity measure to select the most relevant databases according to user query.

Deep Web Mining: Lately, there has also been a lot of work on mining useful information from the deep web [11, 5, 4, 12]. However, the above work is generally driven by e-commerce domain applications, focusing on database integration and schema matching. Dependencies between data sources have not been considered by these.

Semantic Search: The semantic issues in keyword search and web services have motivated much research [26, 18, 29]. Two main issues have been addressed by the existing research. The first one is using ontology to determine the type

or scope of a keyword, and the second is using ontology relations to obtain the intention of a query. In our work, we are also considering the semantics issue. A domain ontology is used to decide the type and relation between keywords. But in our case, the keywords involved are technical terms from a specific domain, which are different from some random keywords in a relational text database. Therefore, it is relatively easy to decide the intention underlying a given query.

8. CONCLUSIONS

In this paper, we have considered keyword search over online deep web data sources. Our algorithm is able to support two type of queries, which are keyword-attributes query and keyword-keyword relation query, and derive interesting relationships between the keywords from the deep web. We use a hyper-graph model to capture the multi-sources interdependencies between data sources. Since there is data redundancy across deep web data sources, we have also designed a domain ontology based data source ranking and selection strategy. We have developed a bidirectional query planning algorithm which is based on bidirectional searching algorithm for traditional database keyword search, but addresses several new challenges.

Our query planning algorithm is capable of finding an query answering plan for a keyword query with a high relevance score and a low execution time. We compared our algorithm with the naive algorithm, the exhaustive algorithm, and a backward planning algorithm. We find that our algorithm can generate optimal results for most queries, with much smaller query planning time, and also runs much faster than the backward algorithm.

9. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and autam Das. Dbxplore: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering*, page 5, 2002.

[2] B.Aditya, Gaurav Bhalotia, Soumen Chakrabarti, Arvind Hulgeri, Charuta Nakhe, Parag Parag, and S.Sudarshan. Banks: Browsing and keyword searching in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, volume 28, pages 1083–1086, 2002.

[3] Jens Bleiholder, Samir Khuller, Felix Naumann, Louiqa Raschid, and Yao Wu. Query planning in the presence of overlapping sources. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 811–828, 2006.

[4] K. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web, 2005.

[5] Kevin Chen-Chuan Chang and Junghoo Cho. Accessing the web: From search to integration. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of Data*, pages 804–805, 2006.

[6] Philipp Cimiano, Gunter Ladwig, and Steffen Staab. Gimme' the context: Context-driven automatic semantic annotation with c-pankow. In *Proceedings of the 14th international conference on World Wide Web*, pages 332–341, 2005.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[8] AnHai Doan and Alon Halevy. Efficiently ordering query plans for data integration. In *Proceedings of the 18th International Conference on Data Engineering*, page 393, 2002.

[9] Daniela Florescu, Alon Levy, and Ioana Manolescu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conferenc on Management of Data*, pages 311–322, 1999.

[10] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep web: A survey. *Communications of ACM*, 50:94–101, 2007.

[11] Bin He, Zhen Zhang, and Kevin Chen-Chuan Chang. Knocking the door to the deep web: Integrating web query interfaces. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of Data*, pages 913–914, 2004.

[12] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of web search interfaces with wise_integrator. *The international Journal on Very Large Data Bases*, 12:256–273, 2004.

[13] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over reltional databases. In *Proceedings of the 29th international conference on Very Large Data Bases*, 2003.

[14] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 67–681, 2002.

[15] Philipp Hussels, Silke Tribl, and Ulf Feser. What's new? what's certain? - scoring search results in the presence of overlapping data sources. *Data Integration in the Life Sciences*, 4544:231–246, 2007.

[16] Azchary G. Ives, Daniela Florescu, Marc Friedman, and Alon Levy. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 299–310, 1999.

[17] Varum Kacholia, Shashank Pandit, Soumen Chakrabarti, S.Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very Large Data Bases*, pages 505–516, 2005.

[18] Eser Kandogan, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and Huaiyu Zhu. Avatar semantic search: A database approach to information retrieval. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 790–792, 2006.

[19] Zoe Lacroix, Louiqa Raschid, and Maria-Esther Vidal. Efficient techniques to explore and rank paths in life science data sources. In *Proceedings of the 1st International Workshop on Data Integration in the Life Sciences*, pages 187–202, 2004.

[20] Ulf Leser and Felix Naumann. Query planning with information quality bounds. In *Proceedings of the 4th International Conference on Flexible Query Answering*, pages 85–94, 2000.

[21] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 251–262, 1996.

[22] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in retional databases. In *Proceedings of the 2006 ACM SIGMOD internatinal conference on Management of data*, pages 563–574, 2006.

[23] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: Top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of Data*, pages 115–126, 2007.

[24] Felix Naumann, Ulf Leser, and Johann Christoph Freytag. Quality-driven integration of heterogeneous information systems. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 447–458, 1999.

[25] Z. Nie and S. Kambhampati. Joint optimization of cost and coverage of information gathering plans.

[26] R.Guha, Rob McCool, and Eric Miller. Semantic search. In *Proceedings of the 12th International Conference on World Wide Web*, pages 700–709, 2003.

[27] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query optimization over web services. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 355–366, 2006.

[28] Bei Yu, Guoliang Li, Karen Sollins, and Anthony K.H. Tung. Effective keyword-based selection of relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of Data*, pages 139–150, 2007.

[29] Qi Zhou, Chong Wang, Miao Xiong, Haofen Wang, and Yong Yu. Spark: Adapting keyword query to semantic search. In *Proceedings of the 6th International Semantic Web Conference*, pages 694–707, 2007.