

From: Coward, Jeffery

Sent: 10/29/2013 4:48:43 PM

To: TTAB E Filing

CC:

Subject: U.S. TRADEMARK APPLICATION NO. 85561168 - DEEP WEB INTELLIGENCE - 4335.14US01 - Request for Reconsideration Denied - Return to TTAB - Message 3 of 5

\*\*\*\*\*

Attachment Information:

Count: 3

Files: RDDW2-08.jpg, RDDW2-09.jpg, RDDW2-10.jpg

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

$n$ '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  $\beta$  for each node which is going to be put into  $FN$  or  $BN$  queues. The decay factor  $\beta$  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  $1 \times N \text{Score}(n)$ . 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, NSY NSNP, MOLA, ORT H 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

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  $v_1, \dots, v_n$ , we mark QueryBitMap[u][v] = 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

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

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 allelotype frequency by four F type relations.

between the number of output attributes of D which finally reach to biological concept term  $n$  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  $n$ , 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  $n$ .

## 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