STAR: Steiner-Tree Approximation in Relationship Graphs

Gjergji Kasneci #1, Maya Ramanath #2, Mauro Sozio #3, Fabian M. Suchanek #4, Gerhard Weikum #5

#*Max-Planck Institute for Informatics,*
Database and Information Systems,
Saarbrücken, Germany
1kasneci@mpi-inf.mpg.de
2ramanath@mpi-inf.mpg.de
3msozio@mpi-inf.mpg.de
4suchanek@mpi-inf.mpg.de
5weikum@mpi-inf.mpg.de

***Abstract*— Large graphs and networks are abundant in modern information systems: entity-relationship graphs over relational data or Web-extracted entities, biological networks, social online communities, knowledge bases, and many more. Often such data comes with expressive node and edge labels that allow an interpretation as a semantic graph, and edge weights that reflect the strengths of semantic relations between entities. Finding close relationships between a given set of two, three, or more entities is an important building block for many search, ranking, and analysis tasks. From an algorithmic point of view, this translates into computing the best Steiner trees between the given nodes, a classical NP-hard problem. In this paper, we present a new approximation algorithm, coined STAR, for relationship queries over large relationship graphs. We prove that for n query entities, STAR yields an $O(\log(n))$-approximation of the optimal Steiner tree in pseudopolynomial run-time, and show that in practical cases the results returned by STAR are qualitatively comparable to or even better than the results returned by a classical 2-approximation algorithm. We then describe an extension to our algorithm to return the *top-k* Steiner trees. Finally, we evaluate our algorithm over both main-memory as well as completely disk-resident graphs containing millions of nodes. Our experiments show that in terms of efficiency STAR outperforms the best state-of-the-art database methods by a large margin, and also returns qualitatively better results.**

I. Introduction

A. Motivation and Problem

Many modern applications need to deal with graph-based knowledge representations. Such applications include business and customer networks managed in relational databases, entity-relationship (ER) graphs over products, people, organizations, and events that are automatically extracted from Web pages, metabolic and regulatory networks in biology, social networks and social-tagging communities, knowledge bases and ontologies in RDF or ER-flavored models, and many more. The data of such graphs exhibits semantics-bearing labels for nodes and edges and can thus be seen as a *semantic graph*, with nodes corresponding to entities and edge weights capturing the strengths of semantic relationships. Often, these graphs are too large to fit into main memory, such that the task of querying and analyzing them in an efficient way becomes non-trivial. An example of such a graph is the YAGO knowledge base [1], which has been constructed by systematically harvesting semi-structured elements (e.g., infoboxes, categories, lists) from Wikipedia. The resulting entities and relation instances have been integrated with the WordNet thesaurus [2]. Figure 1 shows an excerpt; the entire YAGO graph consists of millions of nodes (entities and entity classes) and tens of millions of edges (facts that connect two entities or classes). Another well-known graph with a simpler structure is the IMDB movie database with movies, actors, producers, and other entities as nodes and the movie cast (information about directors, producers, composers, etc.) as edges.

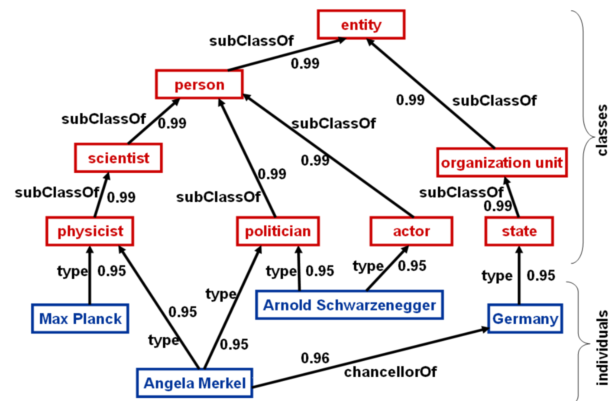

Fig. 1. Example of an entity relationship graph

Such graphs can be represented in relational or ER models, XML with XLinks, or in the form of RDF triples. Correspondingly, they can be queried using languages like SQL, XQuery, or SPARQL. An important class of queries is *relationship search*: Given a set of two, three, or more entities (nodes), find their closest relationships (edges or paths) that connect the entities in the strongest possible way. Here, a "strong" interconnection should reflect the informativeness of the answer. For example, when asking *"How are Germany's chancellor Angela Merkel, the mathematician Richard Courant, Turing-Award winner Jim Gray, and the Dalai Lama related?"*,

a compact and informative answer would be that all four have a doctoral degree from a German university (honorary doctorates in the last two cases). On movie/actor graphs, the game *"six degrees of Kevin Bacon"*[1] entails similar search patterns. On biological networks such as the KEGG pathway repository[2], the closest relationships between the two specific enzymes and a particular gene would be of interest [3], [4], [5]. Similar queries are needed to analyze business networks between companies, their executive VPs, board members, and customers, or to discover connections in intelligence and criminalistic applications.

All the above scenarios aim at information discovery (as opposed to mere lookup), so queries should return multiple answers ranked by a meaningful criterion. Each answer can be naturally defined as a tree that is embedded in the underlying graph and connects all given input nodes. A reasonable scoring model then is some aggregation of node and edge weights over this tree. This query and ranking model has originally been proposed for schema-agnostic keyword queries over relational databases [6], [7], [8], [9]; a number of variations have appeared in the literature (see Section 2). The formal problem that underlies these models is to compute the k *lowest-cost Steiner trees*: Given a graph $G(V, E)$, with a set of nodes V and a set of edges E, let $w : E \rightarrow \mathbb{R}_+$ denote a non-negative weight function. For a given node set $V' \subseteq V$, the task is to find the *top-k* minimum-cost subtrees of G that contain all query nodes of V', where the cost of a subtree T with nodes $V(T)$ and edges $E(T)$ is defined as $\sum_{e \in E(T)} w(e)$.

Given the NP-hardness of the problem and notwithstanding the results on fixed-parameter tractability [10], as well as the tractability results on the approximate enumeration of the *top-k* approximate results [11], most prior works have resorted to heuristics, and, in fact, have typically modified the ranking model for the sake of efficiency (e.g., [12], [13], [14]). This is unsatisfying as it mixes arguments about query and ranking semantics with arguments about efficiency. Furthermore, many of the leading database methods lack approximation or run-time guarantees (e.g., [15], [7], [6], [12]). A theoretical study conducted by the authors of [10] shows that the methods presented in [15], [7], [6], [16] turn out to have an approximation ratio of $O(n)$ where n is the number of query terms.

This paper overcomes these problems by staying with the original, most natural semantics while computing near-optimal Steiner trees with practically viable run-times. In fact, the approximation algorithm developed in this paper even outperforms those prior methods that have worked with relaxed semantics.

B. Contributions and Outline

The main contributions of this paper are the following.

- We present STAR, a new efficient algorithm to the Steiner tree problem, which exploits taxonomic schema information when available to quickly produce results for n given query entities.
- We prove that STAR has a worst case approximation ratio of $O(\log(n))$. This improves the previously best-known approximation guarantees of $O(\sqrt{n})$ or even $O(n)$ for practically leading database methods (see [10]). In our experiments on real-life datasets, STAR achieves better results (i.e. trees of lower weight) than the ones returned by the $2(1 - \frac{1}{n})$-approximation algorithm presented in [17].
- We analyze the time complexity of the algorithm and prove that it has a pseudo-polynomial run-time (i.e., polynomial under the realistic assumption that the ratio of the maximum edge weight to the minimum edge weight is polynomial in the size of the graph.)
- We generalize STAR to an algorithm that is capable of computing approximate *top-k* relation trees for a given set of query entities.
- We compare STAR with the best state-of-the-art database methods in comprehensive main memory and on-disc experiments. STAR outperforms all opponents, often by an order of magnitude and sometimes even more.

The remainder of this paper is organized as follows. Section II gives an overview on related work. In Section III, we present our algorithm and in Sections IV and V we give a detailed analysis of its approximation ratio and run-time complexity. Furthermore, we generalize our algorithm to a *top-k* approximation algorithm in Section VI. The evaluation of our approach is presented in Section VII.

II. Related Work

Relationship queries – queries which ask for relationships between two or more entities – occur in many different applications. For example, keyword proximity search over relational databases [8], [18], [9], [6], [7], [10], [12], graph search over ER, RDF and other knowledge bases [19], [20], [21], [22], [23], entity relationship queries on the Web [24], [25], etc. Such applications have to deal with large graphs (sometimes with millions of nodes and edges) in general, and require not only qualitatively good solutions, but also implementations that are efficient. Our focus in this paper is on a particular kind of relationship query which requires the system to find *top-k* connections between two or more entities. Formally, the problem of determining the closest interconnections between two, three, or more nodes in a graph is the Steiner tree problem.

The Steiner tree problem can be stated as follows. Given a weighted graph $G = (V, E)$ and a set of nodes $V' \subseteq V$, called *terminals*, find a tree in G of minimal weight such that it contains all the terminals. It has been shown that the Steiner tree problem is NP-hard. Consequently, there has been a lot of research on finding approximate solutions to this problem. The quality of an approximation algorithm is measured by the *approximation ratio*. That is, the ratio between the weight of the tree output by the algorithm and the optimal Steiner tree. The Steiner tree problem can be generalized to the Group

[1] http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon
[2] http://www.genome.ad.jp/kegg/pathway.html

Steiner tree problem (GST): Given a weighted graph $G = (V, E)$ and a set of groups $V_1, \ldots, V_k$ where each V_i contains nodes from V, find a tree in G of minimal weight such that it contains *at least* one node from each group.

We give a brief overview of the related literature in the following and compare it with our work.

Algorithms for Steiner Tree Computation: Existing approaches can be categorized according to their strategies: i) distance network heuristic (DNH), ii) span and cleanup, iii) dynamic programming, and iv) local search.

DNH: This heuristic [17], [26] builds a complete graph on the terminals. The edge weights reflect the shortest distance between two terminals in the underlying graph. By a minimum spanning tree (MST) heuristic this complete graph can be used to construct a $2(1-\frac{1}{n})$-approximation to the optimal Steiner tree. This heuristic is applicable to graphs of moderate size, which can fit into main memory. It has been emulated by other approaches for the *top-k* Group Steiner tree computation [6], [7]. The latter two approaches, however, turn out to have an approximation ratio of $O(n)$, where n is the number of query terms (see [10]).

Span and cleanup: This heuristic [16] aims at constructing the MST on the terminals by starting from an arbitrary terminal and spanning the tree stepwise until it covers all terminals. Redundant nodes are deleted in a cleanup phase. [15] exploited this heuristic by means of two different spanning strategies. In contrast to the original heuristic, each terminal is a starting point for a tree yielding a possible MST. While the first spanning strategy chooses the edge with a minimum weight to span a tree (minimum edge-based spanning), the second strategy chooses the tree the spanning of which results in a minimum cost tree (balanced MST spanning). Both methods turn out to have an approximation ratio of $O(n)$ (see [10]).

Dynamic programming and DPBF: The first dynamic programming approach to the Steiner tree problem was introduced by Dreyfus and Wagner [27]. It proceeds by computing optimal results for all subsets of terminals. Then the optimal result is computed for all the terminals. In [10], this heuristic is modified to a faster method coined DPBF for the optimal solution in the GST case. While the former work proved the fixed parameter tractability of the Steiner tree problem, the latter proved it for the GST variant. However, both methods are applicable to graphs of moderate size.

Local search: This heuristic has been used in the realm of the Euclidean Steiner tree problem and of the parallel Steiner tree computation [28], [29]. In the first phase an interconnecting tree is built based on the distance network heuristic introduced by [17]. In the second phase the current tree is iteratively improved by considering different nodes in the underlying graph that may improve the cost of the current tree.

Algorithms for Top-k Steiner Tree Computation: Top-k Steiner tree computation has been previously studied in the context of keyword search on relational databases (see BANKS [6], [7] and BLINKS [12]).

The first BANKS paper [6] (referred to as BANKS I), addresses the GST problem on directed graphs. It emulates the DNH heuristic by running single source shortest paths iterators from each node in each of the V_is, where V_i is the set of nodes which contain the keyword k_i. The iterators are expanded in a best-first strategy and follow the edges backwards. As soon as the iterators meet, a result is produced. This technique is improved in BANKS II [7] by (1) reducing the number of iterators, (2) allowing forward expansion on edges in addition to backward expansion, (3) using a heuristic of spreading activation which prioritizes nodes with low degrees, and edges with low weights during the expansion of iterators. However, the performance of both BANKS I and BANKS II can significantly degrade in the presence of high-degree nodes during the expansion process.

[13] makes use of the approaches of BANKS I and BANKS II to generate a first minimal-height tree that contains the query keywords. The authors show that with respect to the tree heights the *top-k* answers can be efficiently generated with provable guarantees.

DPBF [10] can be extended to a *top-k* algorithm by using the intermediate subtrees generated during the dynamic programming process to compute approximate *top-k* results.

Based on the notion of r-radius Steiner graphs, [14] exploits graph partitioning and subgraph indexing along similar lines as [12] for keyword proximity search over heterogeneous data. The presence of the modified ranking model and subgraph indexes make theoretical implications on the run-time or approximation ratio of the approach impossible.

The recently proposed BLINKS [12] makes use of the backward search strategy of BANKS, but based on cost-based expansion. The authors prove that this expansion strategy, which picks the cluster with the smallest cardinality to expand next, has a bound on the worst case performance. Two kinds of indexes are built to speed up the search. First, a keyword-node index is built which stores, for each keyword w, a list of nodes that can reach w along with the distance of each node from w. Second, a node-keyword index is built which stores, for each node, the set of keywords reachable from it and its distance to each keyword. However, since the proposed indexes can be too large to store and too expensive to compute, the graph is partitioned into *blocks*. The blocks are formed by partitioning the graph using node separators, also called *portals*. A high level keyword-block index is built, and more detailed indexes are built at the block level. Multiple cursors are used to perform the backward search within blocks. Whenever a portal of a block is reached, new cursors are created to explore the remaining blocks connected to this portal node. Instead of trees, BLINKS returns $(r, \{n_i\})$ pairs, where r is the root of the result tree and n_i is a set of nodes containing the query keywords. Hence, it is difficult to reconstruct the result trees. Moreover, BLINKS needs to have the graph in memory to partition it and to construct the indexes, while in our approach the graph can be stored in a database and only database indexes need to be used. Finally, the performance of BLINKS is dependent on the number of portals (i.e. nodes that belong to more than one block) and the strategy for choosing them. This is because BLINKS needs

to use separate cursors not just for each keyword cluster, but also for each block that it has to traverse. Hence for a high number of portals, the performance of BLINKS suffers because of the large number of blocks that have portals in common. Although BLINKS lacks approximation and runtime guarantees, experiments show that it performs up to an order of magnitude faster than BANKS II.

III. The STAR algorithm

As described in the introduction, we are given an undirected graph $G(V, E)$ with a set of nodes V and a set of edges E, and a non-negative weight function $w : E \to \mathbb{R}_+$, intuitively representing the connection strength between the two nodes of an edge. For any subgraph G' of G we denote the set of nodes of G' by $V(G')$, and the set of edges of G' by $E(G')$. Furthermore, we extend the weight function w on G' by $w(G') = \sum_{e \in E(G')} w(e)$.

Given a set $V' \subseteq V$, we are interested in finding a subgraph T of G that contains all nodes from V', such that the weight of T is minimal among all possible subgraphs of G that contain all nodes from V'. Note that inevitably, such a subgraph T has to be a tree. Furthermore, we are interested in finding the *top-k* such trees in the order of increasing weights.

Many real world graphs come with semantic annotations such as node labels, representing entities, and edge labels, representing relations. Furthermore, these graphs may have taxonomic substructures indicated by the labels of the corresponding edges. Our local search algorithm STAR can exploit such taxonomic backbones, when available, to efficiently find an approximate solution to the above problem. It runs in two phases. In the first phase, it tries to quickly build a first tree that interconnects all nodes from V'. In the second phase it aims to iteratively improve the current tree by scanning and pruning its neighborhood. In the following, we present both phases in detail.

A. First phase

In order to build a first interconnecting tree, STAR relies on a similar strategy as BANKS I [6]. But, instead of running single source shortest path iterators from each node of V' (as BANKS I does), STAR runs simple breadth-first-search iterators from each terminal. The iterators are called in a round-robin manner. As soon as the iterators meet, a result is constructed.

Unlike BANKS I, in this phase, STAR may exploit taxonomic information (when available) to quickly build a first tree, by allowing the iterators to follow only taxonomic edges, i.e. edges labeled by taxonomic relations such as *type* or *subClassOf* (see Figure 2). This way, STAR can quickly find a taxonomic ancestor of all nodes from V'. Consider the sample graph of Figure 1. Suppose that V'={**Max Planck, Arnold Schwarzenegger, Germany**}. In the first phase, STAR would construct the tree depicted in Figure 2.

In the following, we describe how we gradually improve the tree returned by the first phase of our algorithm.

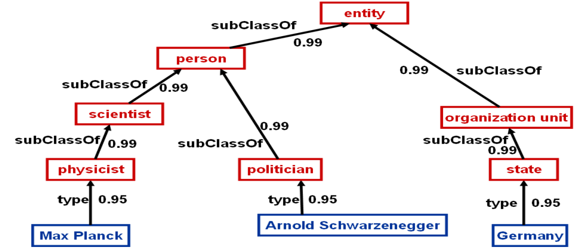

Fig. 2. Taxonomic interconnection

B. Second Phase

In the second phase, STAR aims at improving the current tree iteratively by replacing certain paths in the tree by new paths of lower weight from the underlying graph. In the following we define which paths can be replaced.

1) Fixed Nodes and Loose Paths: Let T be a tree interconnecting all nodes of V'. We denote the degree of a node v in T by $deg(v)$. A node $v \in V'$ is called a *terminal node*, all other nodes of T are called *Steiner nodes*.

Definition 1 (fixed node) *A node in T is a fixed node if it is either a terminal node or a Steiner node that has degree $deg(v) \geq 3$.*

Intuitively, a fixed node is a node that should not be removed from T during the improvement process.

Definition 2 (loose path) *A path p in T is a loose path if it has minimal length with respect to the following property: its end nodes are fixed nodes.*

From the definition above, it follows immediately that every intermediate node in a loose path must be a Steiner node with degree two.

Intuitively, a loose path is a path that can be replaced in T during the improvement process.

It follows immediately that a minimal Steiner tree with respect to V' is a tree in which all loose paths represent shortest paths between fixed nodes.

2) Observations: In the following, for a tree T, we denote the set of loose paths of a tree T by $LP(T)$. Removing a loose path lp from T splits T into two subtrees T_1 and T_2. In Figure 3, the removal of the loose path that connects the nodes a and b from T_0 would return two subtrees interconnecting the terminals u, w and x, y, z, respectively. Replacing a loose path lp by a new, shorter path, means computing the shortest path between any node of T_1 to any node of T_2. Note that since the end nodes of the loose path lp are fixed nodes, they are not removed when lp is removed. This means that removing a loose path that ends into a fixed node v of degree three turns v into an unfixed node, and the two remaining loose paths that had v as an end node are merged into one single loose path. In Figure 3, the removal of the loose path that connects a and b turns a and b into unfixed nodes. The loose paths that were connected to b (or to a, respectively) are merged into a

single loose path. On the other hand, inserting a loose path that ends into an unfixed node v turns v into a fixed node, and the loose path that passes through v is split into two loose paths. In Figure 3, connecting a and d by a new path turns a and d into fixed nodes. The loose path that went through d (or through a, respectively) is split into two loose paths. Hence, the number $|LP(T')|$ of loose paths in an improved tree T' is $|LP(T)| - 2 \leq |LP(T')| \leq |LP(T)| + 2$.

Lemma 1 *A tree T with terminal set $V', |V'| \geq 2$, has at least $|V'| - 1$ and at most $2|V'| - 3$ loose paths.*

Proof The proof is by induction on the number of terminals. Obviously, for a tree T with two terminals $|V'| - 1 \leq |LP(T)| \leq 2|V'| - 3$ holds. Let T be a tree with $|V'| > 2$. Let lp be a loose path in T. Removing lp from T splits T into two subtrees T_1 with a terminal set V_1' and T_2 with a terminal set V_2'. By induction, our claim holds for T_1 and T_2. With the above discussion, connecting T_1 and T_2 again through lp may lead in each of the trees T_1 and T_2 to one more loose path. Hence, the overall number of loose paths in T is upperbounded by $|LP(T)| \leq |LP(T_1)| + |LP(T_2)| + 2 + 1$. On the other hand, the connection through lp may leave the number of loose paths in T_1 and T_2 unchanged, resulting in $|LP(T)| \geq |LP(T_1)| + LP|(T_2)| + 1$. Assuming that $|LP(T_1)| = 2|V_1'| - 3$ and $|LP(T_2)| = 2|V_2'| - 3$ leads to $|LP(T)| \leq (2|V_1'| - 3) + (2|V_2'| - 3) + 2 + 1 = 2|V'| - 3$. Assuming that $|LP(T_1)| = |V_1'| - 1$ and $|LP(T_2)| = |V_2'| - 1$ leads to $|LP(T)| \geq (|V_1'| - 1) + (|V_2'| - 1) + 1 = |V'| - 1$ $\square$

3) Finding an approximate Steiner tree: In the second phase, STAR keeps on iteratively improving the current tree T. In each iteration our algorithm removes a loose path lp from the current tree T. Consequently, in each iteration T is decomposed into two components T_1 and T_2. The new tree T is obtained by connecting T_1 and T_2 through a path that is shorter than lp (see Figures 3, 4, and 5). Hence, the inherently difficult Steiner tree problem is reduced to the problem of finding shortest paths between subsets of nodes. Heuristically, in each iteration we remove the loose path with the maximum weight in T. A high-level overview is given in Algorithm 1.

Algorithm 1 *improveTree*(T, V')

```
1:  priorityQueue Q = LP(T) //ordered by decreasing weight
2:  while Q.notEmpty() do
3:      lp = Q.dequeue()
4:      T' ← Replace(lp, T)
5:      if w(T') < w(T) then
6:          T = T'
7:          Q = LP(T) //ordered by decreasing weight
8:      end if
9:  end while
10: return T
```

Speaking abstractly, the above algorithm greedily scans and prunes the neighborhood of T for better trees. Paths that

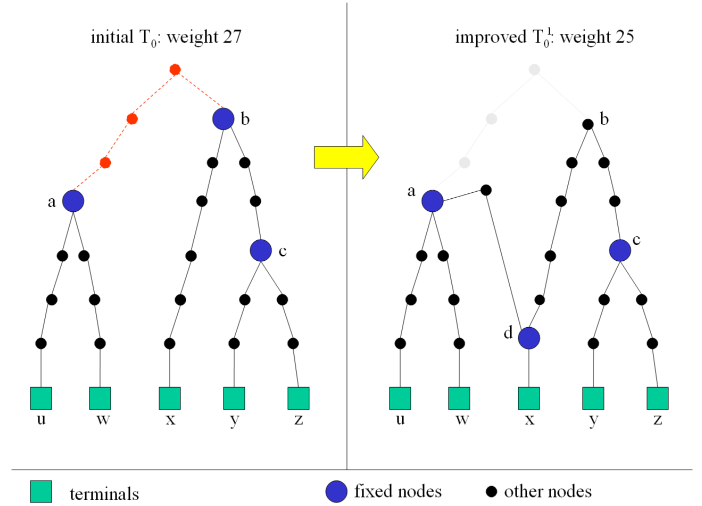

Fig. 3. After first iteration

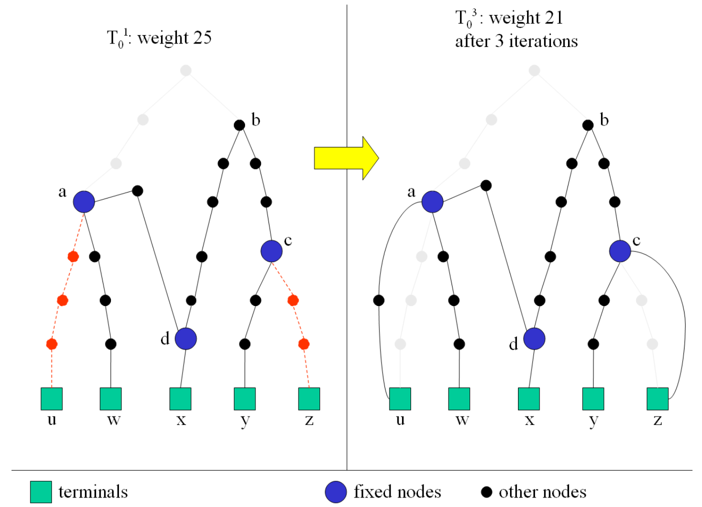

Fig. 4. After third iteration

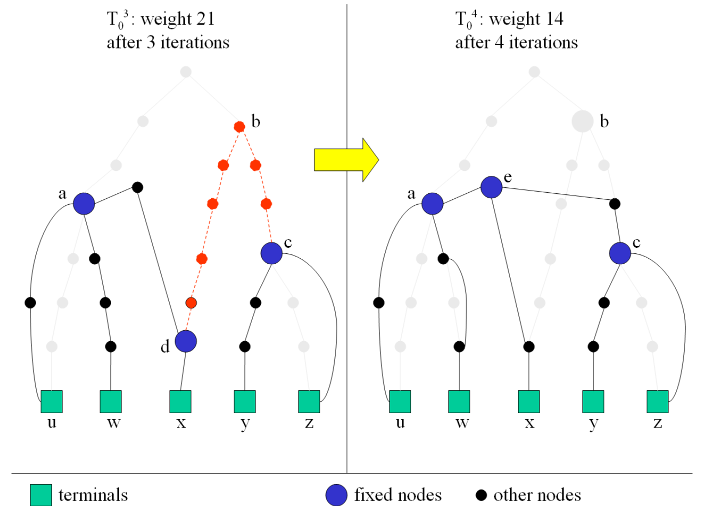

Fig. 5. After fourth iteration

exceed the weight of the loose path upon which the current tree is being improved are pruned. Note that this method leads only to a local optimum. However, we show in Theorem 1 that this local optimum is relatively close to the global optimum.

As an example, we show how STAR would improve the taxonomic tree returned by the first phase of the algorithm (see Figure 2). In the first iteration the algorithm would remove the loose path that connects the fixed node labeled with **Germany** to the fixed node labeled with **person**. The improved tree is depicted in Figure 6. Note that since STAR aims to find closest relations between entities, it views the edges in Figures 6 and 7 as undirected.

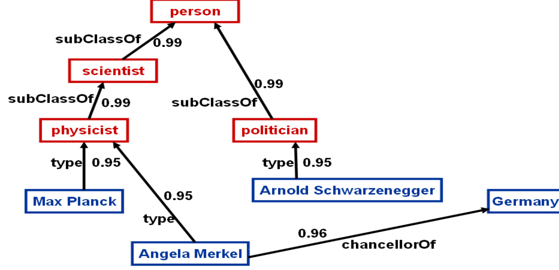

Fig. 6. Result of the first iteration.

In the second iteration the path connecting the fixed node labeled with **Arnold Schwarzenegger** to the fixed node labeled with **physicist** is removed. The improved tree (depicted in Figure 7) is at the same time the final tree, since no loose path can be improved. Another example is depicted in Figures 3-5.

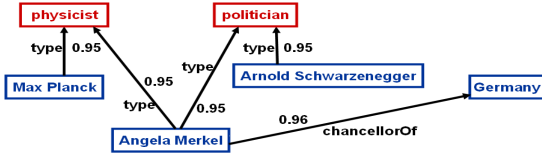

Fig. 7. Result of the second iteration.

The method *Replace*(lp, T) (line 4 of Algorithm 1) removes the loose path lp from T. This removal splits T into two subtrees T_1 and T_2. Then the shortest path in G that connects any node of T_1 to any node of T_2 is determined and combined with T_1 and T_2 into a new tree T' of lower weight. For this purpose, *Replace*(lp, T) calls another method, called *findShortestPath*$(V(T_1), V(T_2), lp)$, which runs one single source shortest path iterator from each of the node sets $V(T_1)$ and $V(T_2)$. This method is presented in Algorithm 2. In the beginning, each of the iterators Q_1, Q_2 contains all the nodes from $V(T_1)$ and $V(T_2)$, respectively (lines 5, 6). The variables $current$ and $other$ (lines 7 and 8) represent the subscript indices of Q_1 and Q_2. As presented in lines 10 to 12, $Q_{current}$ points to the iterator that has minimal number of fringe nodes. Intuitively, $Q_{current}$ represents the iterator that is currently expanded. This expansion heuristic is similar to the cost-balanced expansion used by BLINKS [12], which attempts to balance the number of accessed nodes (i.e., the search cost) for each iterator. It is also similar to the expansion heuristic used by BANKS II [7], which prioritizes nodes with low degrees during the expansion. However, the difference is that we consider the whole node collection in an iterator as a single node. Each iterator aims at reaching a node from the starting set (source) of the other iterator, represented by $V(T_{other})$ in line 27. Hence, in case that $Q_{current}$ points to the iterator that started from $V(T_1)$, the set $V(T_{other})$ points to $V(T_2)$ and vice versa. During the expansion, for each node v' visited by the current iterator, we maintain its current predecessor, that is, the node v from which the iterator reached v' (line 23). Again the predecessor is dependent on the current iterator. The current predecessor of v' is chosen such that the distance $d_{current}$ of v' to the source of the current iterator is minimized (lines 21-23). We maintain this distance for each visited node v' (line 22). Maintaining the predecessor of a visited node v', helps us rebuild the path from v' to the source. However, as soon as the iterator $Q_{current}$ encounters a node v that has a distance greater than or equal to the weight of the loose path lp upon which we are aiming to improve the current tree, the expansion stops (lines 14, 15). The reason for this is that all other nodes in $Q_{current}$ have a greater distance to the source than v, since the nodes in the iterators are ordered by increasing distance from the sources.

Algorithm 2 *findShortestPath*$(V(T_1), V(T_2), lp)$

1: **for all** $v \in V$ **do**
2:    **if** $v \in V(T_1)$ **then** $d_1(v) = 0$ **else** $d_1(v) = \infty$
3:    **if** $v \in V(T_2)$ **then** $d_2(v) = 0$ **else** $d_2(v) = \infty$
4: **end for**
5: *PriorityQueue* $Q_1 = V(T_1)$ //ordered by inc. distance d_1
6: *PriorityQueue* $Q_2 = V(T_2)$ //ordered by inc. distance d_2
7: $current$=1
8: $other$=2
9: **repeat**
10:    **if** *fringe*(Q_{other})<*fringe*$(Q_{current})$ **then**
11:       *swap*$(current, other)$
12:    **end if**
13:    $v = Q_{current}$.*dequeue*()
14:    **if** $d_{current}(v) \geq w(lp)$ **then**
15:       **break**
16:    **end if**
17:    **for all** $(v, v') \in E$ **do**
18:       **if** v' *has been dequeued from* $Q_{current}$ **then**
19:          **continue**
20:       **end if**
21:       **if** $d_{current}(v') > d_{current}(v) + w(v, v')$ **then**
22:          $d_{current}(v') = d_{current}(v) + w(v, v')$
23:          $v'.predecessor_{current} = v$
24:       **end if**
25:       $Q_{current}$.*enqueue*(v')
26:    **end for**
27: **until** $Q_1 = \emptyset \vee Q_2 = \emptyset \vee v \in V(T_{other})$
28: **return** *path connecting* T_1 *and* T_2

Now we move on to proving the approximation guarantee of the STAR algorithm.

IV. Approximation Guarantee

The following proof has a very important implication. It entails that the approximation ratio for the cost of the final tree returned by STAR is independent of the tree constructed in the first phase.

The proof proceeds as follows. We define a mapping between each loose path in the tree returned by the algorithm, and a more expensive path in the optimum solution. Such a mapping has the property that at most $2\lceil\log N\rceil + 2$ loose paths are mapped onto a same path. Moreover, each edge in the optimum solution occurs in the range of the mapping at most twice. Hence, summing over all paths in the range of the mapping gives an upper bound (of $4\lceil\log N\rceil + 4$) on the cost of the tree yielded by the algorithm.

The process of finding such a mapping consists of two phases. First, we identify a collection of paths in the optimum tree that do not overlap too much. Then, we go back to the tree yielded by the algorithm, trying not to assign too many loose paths to the same path in the optimal tree. Lemma 2 deals with this non-trivial task.

Before diving into the proof, we need some auxiliary notations. We shall denote an ordered pair by (i, j) (this means that $(i, j) \neq (j, i)$), while an unordered pair will be denoted by $\{i, j\}$. For any graph G, $d_G(u, v)$ denotes the shortest distance between u and v in G. In a tree, we denote by uv the (unique) path between u and v.

Our input is an undirected graph $G = (V, E)$ and a set of terminals $V' \subseteq V$ that are to be connected. Let $N = |V'|$ (in what follows we assume $N > 2$). Let T_O be an optimal Steiner tree with respect to the set V' of terminals in the input. Let T_A be the Steiner tree yielded by the STAR algorithm.

Lemma 2 *Let $\mathcal{L}(T_A)$ be the set of* loose paths *in T_A. For any circular ordering $v_1, \ldots, v_N$ of the terminals in T_A, there is a mapping $\mu : \mathcal{L}(T_A) \to V' \times V'$ such that:*

1) *μ is defined for all loose paths in T_A;*
2) *for each loose path P with end points u and v, let T_1 and T_2 be the two trees obtained by removing from T_A all nodes in P (and their edges), except u and v; then, $\mu(P) = \{v_i, v_{i+1}\}$ for some $i = 1, \ldots, N$ and one of the nodes v_i, v_{i+1} belongs to T_1, while the other one belongs to T_2;*
3) *for each pair of terminals $\{v_i, v_{i+1}\}$ there are at most $2\lceil\log N\rceil + 2$ loose paths mapped to $\{v_i, v_{i+1}\}$.*

Proof For ease of presentation, we assume T_A is rooted at any arbitrary terminal node and its edges are directed from the root towards the leaves. Then, we denote by $u \to v$ a path where u is closer to the root than v. Furthermore, for any subtree T of T_A we shall denote by $\tau(T)$ the set of terminals belonging to T. The first step in defining the mapping is to find a labeling with good properties, as follows.

For each loose path $P = u \to v$ let T_u and T_v be the subtrees of T_A rooted at u and v, respectively. Let v_i and v_j be the two terminals having the minimum absolute difference $|i - j|$ among all pairs v_i, v_j, satisfying the constraints $v_i \in \tau(T_v)$ and $v_j \in \tau(T_u) \setminus \tau(T_v)$. Label P with the ordered pair (i, j). Iterate this procedure for all loose paths.

We now study some properties of this labeling. Let v_i be any terminal and let $\mathcal{P}_i$ be the path connecting the root to v_i. Consider the set of labels occurring in $\mathcal{P}_i$ of the kind (i, j), where $j > i$; let $(i, i + j_1), \ldots, (i, i + j_k)$ be the sequence of such pairs, ordered by non-decreasing j_h's. We prove that $j_{h+1} \geq 2j_h$, $h = 1, \ldots, k - 1$, which together with the fact that j_h's are not larger than N implies $k \leq \lceil\log N\rceil + 1$.

Suppose by contradiction that there is h such that $j_{h+1} < 2j_h$. Consider the two loose paths labeled with $(i, i + j_h)$ and $(i, i + j_{h+1})$. Let $P = u \to v$ be the one of the two that is closest to the root.

By the definition of the labeling, $\{v_i, v_{i+j_h}, v_{i+j_{h+1}}\} \subseteq \tau(T_u)$. There are two cases, either P is labeled with $(i, i + j_h)$ or P is labeled with $(i, i + j_{h+1})$. In the former case, $v_{i+j_h} \notin \tau(T_v)$ and $j_{h+1} - j_h < j_h$. Hence, P would have been labeled with $(i + j_{h+1}, i + j_h)$. In the latter case, $v_{i+j_{h+1}} \notin \tau(T_v)$ and $j_{h+1} - j_h < j_h$, which implies that P would have been labeled with $(i + j_h, i + j_{h+1})$. Therefore, in both cases we obtain a contradiction.

In other words, we just proved that in the path between the root and any terminal v_i, the number of labels of the kind (i, j), where $j > i$, is at most $\lceil\log N\rceil + 1$. From the way the labeling has been defined, as well as from the fact that there is exactly one path between the root and any terminal, it follows that in the whole tree T_A such labels can occur at most $\lceil\log N\rceil + 1$ times. Symmetrically, we can show that the number of labels of the kind (i, j) where $j < i$, is bounded by the same quantity.

In order to obtain the desired mapping the labeling is refined in the following way. Replace each label (i, j) with $(i, i + 1)$ if $j > i$ and with $(i, i - 1)$ otherwise. Now, drop the ordering of the pairs, that is, turn each label $(i, i + 1)$ into $\{i, i + 1\}$. This implies that each label can occur at most $2\lceil\log N\rceil + 2$ times. Finally, for each loose path P, define $\mu(P) = \{v_i, v_j\}$ where $\{i, j\}$ is the label of P. It is straightforward to see that the claimed three properties are satisfied. □

Theorem 1 *The STAR algorithm is a $(4\lceil\log N\rceil + 4)$-approximation algorithm for the Steiner Tree problem.*

Proof Consider a walk on T_O that uses each edge exactly twice and that visits all nodes in T_O. Such a walk gives a circular ordering $v_1, \ldots, v_N$ of the terminals, ordered according to their first occurrence in such a walk. We have that

$$\sum_{k=1}^{N} d_{T_O}(v_k, v_{k+1}) = 2w(T_O). \tag{1}$$

Using Lemma 2, we define a mapping μ with respect to the circular ordering $v_1, \ldots, v_N$. From property 2 of the mapping μ and from the termination condition of the STAR algorithm, it follows that for any loose path $P = uv$ in T_A

$$d_{T_A}(u, v) \leq d_{T_O}(\mu(uv)), \tag{2}$$

where $d_{T_O}(\mu(uv))$ is the distance, in the optimum solution, between the two entries of $\mu(uv)$. Finally, we can write

$$
\begin{aligned}
w(T_A) &= \sum_{uv \in LP(T_A)} d_{T_A}(u,v) && (3)\\
&\leq \sum_{uv \in LP(T_A)} d_{T_O}\mu(uv) && (4)\\
&\leq \sum_{k=1}^{N} (2\lceil \log N + 2 \rceil)\, d_{T_O}(v_k, v_{k+1}) && (5)\\
&\leq (4\lceil \log N \rceil + 4)\, w(T_O). && (6)
\end{aligned}
$$

where inequality (4) follows from Equation 2, inequality (5) follows from property 3 of the mapping μ, and inequality (6) follows from Equation 1. □

V. TIME COMPLEXITY

The algorithm as it has been presented might have exponential running time. In fact, the cost of the tree might decrease at each step by an infinitesimally small amount. Fortunately, this can be solved by using a relatively simple "trick", which guarantees that at each step a significant improvement on the cost of the current tree is made.

Given $\epsilon > 0$, we introduce the *improvement-guarantee rule*, which is defined as follows. Let P be a loose path, and let P' be the path selected by the algorithm to replace P; replace P if and only if $w(P') \leq \frac{w(P)}{1+\epsilon}$. The algorithm is then iterated until no loose path can be improved.

Let $w_{\max}$ and $w_{\min}$ be the maximum and minimum cost of the edges in the input graph. The following theorem shows that the STAR algorithm with the improvement-guarantee rule is a pseudopolynomial algorithm, namely its running time is polynomial if the ratio $\frac{w_{\max}}{w_{\min}}$ is polynomial in the size of the input. Let n, m, N denote the number of vertices, edges, and terminals of the input graph, respectively.

Lemma 3 *Given $\epsilon > 0$, the STAR algorithm with the improvement-guarantee rule is guaranteed to terminate in $O\left(\frac{1}{\epsilon}\frac{w_{\max}}{w_{\min}} m\right)$ steps.*

Proof Let $\bar{T}$ be the initial tree. We have that $w(\bar{T}) \leq m w_{\max}$. At any step of our algorithm, let P be a loose path and let P' be the path selected by the algorithm to replace P. By the improvement-guarantee rule, it follows that

$$w(P) - w(P') \geq (1+\epsilon) w(P') - w(P') \geq \epsilon w_{\min}. \quad (7)$$

Hence, the cost of the tree decreases at each step by at least $\epsilon w_{\min}$. This gives a bound on the number of steps k, as follows

$$m w_{\max} - k \epsilon w_{min} \geq 0 \Leftrightarrow k \leq \frac{1}{\epsilon}\frac{w_{\max}}{w_{\min}} m. \quad (8)$$

□

The next theorem shows a trade-off between the approximation guarantee of the STAR algorithm and its running time.

Theorem 2 *Given $\epsilon > 0$, the STAR algorithm with the improvement-guarantee rule is a $(1+\epsilon)(4\lceil \log N \rceil + 4)$-approximation algorithm for the Steiner Tree problem. Its running time is $O(\frac{1}{\epsilon}\frac{w_{\max}}{w_{\min}} m N (n \log n + m))$.*

Proof The time-complexity bound follows from Lemma 3 and from the fact that at each step the STAR algorithm might invoke Dijkstra's algorithm at most $(2N-3)$ times (one for each loose path, see Lemma 1). To prove the approximation ratio, it suffices to replace Equation 2 in Theorem 1 with

$$d_{T_A}(u,v) \leq (1+\epsilon) d_{T_O}(\mu(uv)), \quad (9)$$

and change the remaining equations accordingly. We include all steps for completeness. We have that

$$
\begin{aligned}
w(T_A) &= \sum_{uv \in \mathcal{L}(T_A)} d_{T_A}(u,v) && (10)\\
&\leq \sum_{uv \in \mathcal{L}(T_A)} (1+\epsilon) d_{T_O}\mu(uv) && (11)\\
&\leq \sum_{k=1}^{N} (1+\epsilon)\,(2\lceil \log N \rceil + 2)\, d_{T_O}(v_k, v_{k+1}) && (12)\\
&\leq (1+\epsilon)\,(4\lceil \log N \rceil + 4)\, w(T_O). && (13)
\end{aligned}
$$

□

VI. APPROXIMATE TOP-K INTERCONNECTIONS

As demonstrated in Algorithm 2, the weight of the loose path lp upon which the current tree T is being improved serves as an upper bound for the weights of new interconnecting paths between the subtrees of T that result from the removal of lp from T. The final result of the STAR algorithm, as given by Algorithm 1, is a tree T in which there is no loose path upon which T can be improved.

In order to generalize STAR to an algorithm that can compute approximate *top-k* interconnections, we start from the final tree T returned by the original STAR algorithm, which is stored in a priority queue (see lines 1-3 of Algorithm 3). While the size of this priority queue is smaller than k, we keep on generating new trees from an artificial relaxation of the loose path weights of the current tree.

Algorithm 3 *getTopK*(T, k)

```
1: Q : priority queue of trees
2: T = improveTree(T,V')
3: Q.enqueue(T)
4: while Q.size < k do
5:     T' = relax(T, ε)
6:     T' = improveTree'(T', V')
7:     T = reweight(T')
8:     Q.enqueue(T)
9: end while
```

As shown in Algorithm 4, we artificially relax the weights of each loose path lp in the current T by adding a tunable value

$\epsilon > 0$. We denote the tree with the relaxed loose path weights by T'. We use these artificial loose path weights as upper bounds for the weights of new interconnecting paths between subtrees of the current tree T' that result from the removal of the corresponding loose path from T'. Then, in line 6 of Algorithm 3, we call a modification of the method *improveTree* (see Algorithm 1) on the input (T', V'). This modification takes care that during the improvement of T' upon one of its loose paths lp the new interconnecting path is not the same as lp. Note that this would always happen since the weight of lp was artificially increased, and in the underlying graph G the path lp would still be the shortest path connecting the two corresponding subtrees of T'. For this purpose, we consider only interconnecting paths that are node-disjoint to lp.

The method *reweight* (line 7) reweights the result of *improveTree*$'$. That is, the weight of loose paths of T' which were also loose paths in the previous tree T is set back to its original value.

Algorithm 4 *relax*(T, ϵ)

1: $T' = T.copy$
2: **for all** $lp \in LP(T')$ **do**
3: $\quad w'(lp) = w(lp) + \epsilon$
4: **end for**
5: *return* T'

VII. Evaluation

We compare the STAR algorithm with the most well-known algorithms for Steiner tree approximation. The algorithm [17] was the first to achieve a 2-approximation of the optimal Steiner tree. We refer to it as DNH (for "distance network heuristic"). The second algorithm is DPBF [10], a dynamic programming approach which can compute an optimal tree and performs best on a small number of terminals. The third algorithm is BLINKS [12], which is the newest and experimentally best algorithm in this field. The fourth algorithm is BANKS [6] and its improved version BANKS II [7], which are state-of-the-art algorithms for keyword proximity search on relational data. We compared the algorithms both in terms of the quality of the returned results and in terms of their performance.

All experiments were performed on a 1.8 GHz Pentium machine with 1 GB of main memory and an Oracle Database (version 9.1) as the underlying persistent storage for all on-disk experiments. All implementations are in Java.

In this study we focus on efficiency and the goodness of Steiner trees (i.e., their weights). We do not consider the "semantic quality" or user perceived relevance of results. This aspect is orthogonal to the algorithmic focus of the current paper; our earlier work [21] has shown that a Steiner-tree-based scoring function does indeed contribute to high relevance from a user's viewpoint.

A. Top-1 comparison of STAR, DNH, DPBF, BANKS I, and BANKS II

The goal of the DNH algorithm is to compute a good approximation to the optimal Steiner tree for a given graph and given terminal nodes. The algorithm has an approximation ratio of $2(1 - \frac{1}{n})$, where n is the number of terminal nodes. STAR, by contrast, has an approximation ratio of $4\log(n)+4$. BANKS I and BANKS II have an approximation ratio of $O(n)$. These bounds, however, are theoretical bounds for the worst case. Therefore, we studied how the above algorithms perform in practice. To compare to optimal tree weights, we also ran DPBF. To have comparable runtimes we reimplemented DPBF in Java[3].

Datasets: We use subsets of DBLP[4] and IMDB[5] for our experiments. DBLP and IMDB can be viewed as graphs in which nodes represent entities (like *author*, *publication*, *conference*, *actor*, *movie*, *year*, etc.), and edges represent relations (like *cited_by*, *author_of*, *acted_in*, etc.). Since the DNH and the DPBF algorithms are designed to deal with graphs that can be completely loaded into main memory, we extracted from DBLP a subgraph with 15,000 nodes and 150,000 edges (dataset DBLP). As the qualitative performance of the algorithms can be influenced by different graph topologies, a second graph consisting of 30,000 nodes and 80,000 edges was extracted from IMDB (dataset IMDB). Since the original DBLP and IMDB do not provide any edge weights, we used random weights between 0 and 1 for both graphs. Note that since these datasets do not have any kind of taxonomic backbone, STAR uses its breadth-first heuristic for the initialization phase.

Queries: We constructed three query sets with 3, 5 and 7 terminals, respectively. Each query set consists of 60 queries with the same number of terminals. The terminals were chosen randomly from the graph.

Metrics: We compare the weight of the *top-1* tree returned by STAR (without taxonomic information) with the weight of the tree returned by DNH, BANKS I, and BANKS II on the basis of optimal scores returned by DPBF. We also measured the running times of all algorithms.

Results: Table I shows the results of our experiments on DBLP. The best values across the competitors are in boldface. Column 3 shows the average weight of the result over the 60 queries in the query sets returned by each algorithm. The average weight of the tree returned by the STAR algorithm is consistently below the average weight of the tree returned by DNH (for the same number of terminals) and also better than the scores returned by BANKS I and BANKS II. We validated the statistical significance of the superiority of STAR using a *t-test* at level $\alpha = 0.05$. In particular, STAR returns better results than DNH for this practical case, even though DNH has a better approximation ratio. Column 4 shows the average runtime of the algorithms in milliseconds. STAR

[3]The original C++ code was kindly provided to us by the authors of [10].
[4]Data downloadable from `http://dblp.uni-trier.de/xml`
[5]http://www.imdb.com/

Method	# terminals	avg. weight	avg. runtime (ms)
STAR	3	0.61	**604.2**
DNH		0.7	5402.9
DPBF		**0.58**	33096.7
BANKS I		1.22	2096.3
BANKS II		1.81	3214.1
STAR	5	0.86	**960.2**
DNH		0.98	9166.7
DPBF		**0.81**	432361.5
BANKS I		1.87	3617.3
BANKS II		2.46	5797.5
STAR	7	**1.12**	**1579.6**
DNH		1.22	17430.9
DPBF		?	?
BANKS I		2.37	5945.5
BANKS II		3.42	9435.5

TABLE I

TOP-1 TREE COMPARISON ON DBLP

Method	# terminals	avg. weight	avg. runtime (ms)
STAR	3	3.42	**1044.5**
DNH		3.37	9110.1
DPBF		**2.93**	18014.7
BANKS I		3.85	7153.4
BANKS II		5.31	4153.2
STAR	5	4.35	**1353.5**
DNH		4.33	12912.7
DPBF		**4.14**	121863.3
BANKS I		5.52	9671.4
BANKS II		7.17	5429.1
STAR	7	**5.31**	**1732.9**
DNH		**5.31**	18317.3
DPBF		?	?
BANKS I		7.47	11681.8
BANKS II		9.12	6953.7

TABLE II

TOP-1 TREE COMPARISON ON IMDB

determines the *top-1* tree much faster than all its competitors. The dynamic programming approach of DPBF and the spreading activation heuristic of BANKS II seem to be less adequate for the topology of the DBLP subgraph. The question marks in row 13 of the table reflect the fact that DPBF did not return a single result within 30 minutes. Table II shows that BANKS II significantly improves its performance relatively to its competitors on the IMDB subgraph, but is still outperformed by STAR.

Table II shows that for the IMDB subgraph, the scores of STAR and DNH lie very close to each other. We hypothesize that the higher edge-to-node ratio of the DBLP subgraph allows STAR to return clearly better scores than DNH on the DBLP subgraph. In a denser graph STAR has more possibilities to improve the current tree.

B. Top-k comparison of STAR, BANKS I, BANKS II, and BLINKS

Unlike the DNH algorithm, BANKS I, BANKS II and BLINKS can compute the *top-k* results for a query – like the STAR algorithm. In this comparison we analyze the *top-k* performance of BANKS I, BANKS II, BLINKS, and STAR. We used a Java implementation of BLINKS that was kindly provided to us by the authors. BLINKS uses indexes in order to speed up the query processing time. However, in order to build these indexes and to subsequently use them during runtime, BLINKS requires the entire graph in main memory. For this reason, we used again the DBLP and IMDB dataset for the comparison. As for the partitioning strategy of BLINKS, we experimented with different block sizes and chose a block size of 100 nodes for DBLP and a block size of 5 nodes for IMDB, since these block sizes gave the best results.

Method	*top-k*	avg. weight	avg. runtime (ms)
STAR	top 10	**1.57**	**1206.3**
BANKS I		2.43	5851.8
BANKS II		3.78	7895.9
BLINKS		n/a	19051.4
STAR	top 50	**2.23**	**3118.3**
BANKS I		3.12	7335.1
BANKS II		5.31	8928.3
BLINKS		n/a	21837.9
STAR	top 100	**3.01**	**4705.1**
BANKS I		4.15	9640.8
BANKS II		6.81	11071.3
BLINKS		n/a	24632.3

TABLE III

TOP-K TREE COMPARISON ON DBLP

Method	*top-k*	avg. weight	avg. runtime (ms)
STAR	top 10	**5.21**	**1587.2**
BANKS I		6.13	10611.3
BANKS II		8.25	6619.4
BLINKS		n/a	2848.97
STAR	top 50	**6.32**	**1936.8**
BANKS I		7.21	12049.3
BANKS II		10.04	7892.2
BLINKS		n/a	3708.6
STAR	top 100	**8.07**	**2503.2**
BANKS I		9.92	13694.1
BANKS II		14.98	8873.3
BLINKS		n/a	4917.7

TABLE IV

TOP-K TREE COMPARISON ON IMDB

Metrics: Since BLINKS uses a different weight metric (the *match-distributive* semantics) and returns only the root nodes of the output trees, we could not compare STAR and BLINKS by the weight of the output trees. Hence, our comparison with BLINKS is only with respect to the runtime. For BANKS I, BANKS II and STAR we also report the average scores of the output trees.

Queries: We compared the algorithms for $k = 10, k = 50$ and $k = 100$ on the same Steiner tree problem instances. For the comparison, we constructed for each dataset (DBLP and IMDB) 60 random queries with five terminals each.

Results: We computed the average runtime and the average score for the retrieved *top-10*, *top-50* and *top-100* results. Table III and Table IV present the runtime performance of STAR, BANKS I, BANKS II and BLINKS on the DBLP and

IMDB datasets, respectively. Note that in this comparison we have discounted the times needed by BLINKS to construct the indexes. The results show that STAR outperforms its competitors in all cases. It is interesting to see that BANKS II and BLINKS perform better on the sparser IMDB graph. During search, BLINKS has to cope with a large number of cursors resulting from a large number of partitions. Whenever BLINKS reaches a portal p which belongs to multiple partitions, it has to construct a new cursor for each partition in which p is a portal. In dense datasets, it is likely that a large number of cursors are required to complete the query processing. The overhead of maintaining these cursors adversely affects the overall performance. An indication for this is given by the worse runtime performance of BLINKS on the DBLP dataset.

In contrast, STAR has to maintain only two iterators per improvement step. Furthermore, these iterators do not visit nodes that have a distance from the source that is higher than the upper bound given by the loose path to be replaced. The combination of tight upper bounds to prune the exploration with low overhead in iterators allows STAR to outperform BLINKS by a large margin.

C. External storage comparison of STAR and BANKS

Unlike DNH and BLINKS, BANKS and STAR can be directly applied to graphs that do not fit into main memory. Since these kinds of scenarios are realistic for the Steiner tree problem, we decided to simulate such a scenario by using a disk-resident dataset for the comparison of BANKS and STAR.

Dataset: We chose the graph of the YAGO knowledge base [1]. It contains 1.7 million nodes and 14 million edges. Each edge corresponds to a fact in YAGO, and has a confidence score between 0 and 1 associated with it. We converted these confidence scores into distance measures. We store the graph in a relational database with the simple schema

$$EDGE(source, target, weight).$$

YAGO contains a DAG-shaped taxonomy of *type* and *subClassOf* edges (see Figure 1), which is exploited by STAR in its first phase to construct the initial tree.

We implemented both BANKS I [6] and its improved version BANKS II [7] in Java following their descriptions for main-memory procedures. Whenever the algorithms explore a new edge, we loaded the edge from the database. This way, BANKS and STAR were treated uniformly as far as the overhead for database calls is concerned.

Queries: We generated 2 sets of queries with 3 and 6 terminals each. Each query set consisted of 30 queries with randomly chosen terminal nodes. We measured the performance of the algorithms for the *top-1*, *top-3* and *top-6* results.

Metrics: We measured both the quality of the output trees and the efficiency of the algorithms. As for the quality of the trees, we report the *average weight* of the *top-k* results. As for efficiency, we report the running times and also the number of edges accessed during the query executions. There were several cases for which BANKS I and BANKS II did not return a result within 30 minutes and we had to stop the process. To be fair, we excluded these cases from our evaluation.

Results: Table V shows the results for the performance of STAR, BANKS I, and BANKS II. Concerning the quality of the output trees, STAR returns better results across all values for k and all sets of queries.

As for the efficiency of the algorithms, we note that STAR is an order of magnitude faster than BANKS. This is also reflected directly in the number of edges accessed by each algorithm: STAR accesses an order of magnitude fewer edges than its competitors. This clearly shows the enormous gains that can be made by exploiting the taxonomic structure of the tree to construct the initial result.

D. Summary of results

We compared STAR to different state-of-the-art algorithms. Some of these algorithms come with specific constraints: The DNH algorithm, for example can only handle graphs that fit into main memory and can produce only *top-1* results. BLINKS uses indexes and a different metric and hence cannot give an approximation guarantee. To be fair, it should be emphasized that some of these methods were designed with broader goals beyond Steiner-tree-like relationship queries. Our comparison focuses on Steiner tree computation and is fair by giving all methods the same inputs, operating conditions and resources. In all experiments, STAR outperforms its competitors.

The reason for the efficient performance of STAR is threefold: i) STAR uses the taxonomic structure of the graph when possible to quickly return an initial result which is then improved, ii) STAR requires only two iterators per improvement step (independent of the number of terminals), and iii) STAR uses fairly tight upper bounds on the lengths of the paths and prunes the possible paths that can be included in the result tree.

VIII. Conclusion

This paper has addressed the problem of efficiently answering relationship queries over entity-relation-style data graphs. The STAR algorithm can exploit taxonomic structures that are inherent in many knowledge-base graphs (e.g., the isA hierarchy) for fast computation of an initial seed solution. However, it does not depend on this option, and can use other initializations as well. Its main power for efficiency and result quality comes from iteratively improving the seed tree by a very fast shortest-path algorithm for subtrees defined by the notion of loose paths.

We proved that STAR achieves an $O(\log n)$ approximation for the optimal Steiner tree, which is significantly better than the worst-case approximation quality given by prior database methods [6], [7]. While the DNH method for in-memory graphs has a much better worst-case approximation guarantee than STAR, our experiments give evidence that STAR achieves at least the same result quality (Steiner tree weight) as DNH and other database methods or better on practically relevant datasets.

3 terminals				6 terminals		
top-1	STAR	BANKS I	BANKS II	STAR	BANKS I	BANKS II
avg. score	**0.22**	0.260	0.234	**0.337**	0.385	0.368
avg. # acc. edges	**6981**	84171	81462	**9559**	372634	365004
avg. run time (ms)	**12440.6**	131313.6	104148.5	**15733.1**	391601.0	385401.5
top-3	STAR	BANKS I	BANKS II	STAR	BANKS I	BANKS II
avg. score	**0.428**	0.488	0.454	**1.085**	1.193	1.255
avg. #acc. edges	**18027**	153078	132141	**27085**	460521	409414
Avg. run time (ms)	**34814.7**	190547.7	156535.3	**41187.3**	483328.4	427276.3
top-6	STAR	BANKS I	BANKS II	STAR	BANKS I	BANKS II
avg. score	**2.102**	2.453	2.441	**3.315**	4.148	4.031
avg. # acc. edges	**43474**	159130	175045	**76259**	503054	491786
avg. run time (ms)	**71058.2**	197543.7	205359.6	**91157.2**	511811.0	491785.5

TABLE V

YAGO: QUALITY OF RESULTS AND EFFICIENCY OF STAR, BANKS I & II

The motivation for this database-algorithmic work has been to support graph-based information retrieval and knowledge queries over large datasets in the spirit of NAGA [21], where STAR closes a big efficiency-oriented gap. Our future work will look into more complex search patterns over this kind of rich relationship-graphs, using STAR as a key building block.

STAR has been implemented as a query answering component of the NAGA system. NAGA is available online at: `http://www.mpi-inf.mpg.de/~kasneci/naga/`

IX. ACKNOWLEDGEMENTS

We thank the authors of [12] and the authors of [6], [7] for providing us with the Java code of BLINKS and BANKS. We also thank the authors of [10] for providing us with the original C++ code of DPBF.

REFERENCES

[1] F. M. Suchanek, G. Kasneci, and G. Weikum, “Yago: A Core of Semantic Knowledge,” in *Proc. of WWW*, 2007.

[2] C. Fellbaum, *WordNet: An Electronic Lexical Database*. MIT Press, 1998. [Online]. Available: citeseer.ist.psu.edu/lin98wordnet.html

[3] U. Leser, “A query language for biological networks,” *Bioinformatics*, vol. 21, no. 2, pp. 33–39, 2005.

[4] C. Plake, T. Schiemann, M. Pankalla, J. Hakenberg, and U. Leser, “Ali baba: Pubmed as a graph,” *Bioinformatics*, vol. 22, 2006.

[5] S. Trissl and U. Leser, “Fast and practical indexing and querying of very large graphs,” in *Proc. of SIGMOD*, 2007.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, “Keyword searching and browsing in databases using BANKS,” in *Proc. of ICDE*, 2002.

[7] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, “Bidirectional expansion for keyword search on graph databases,” in *Proc. of VLDB*, 2005.

[8] S. Agrawal, S. Chaudhuri, and G. Das, “DBXplorer: A system for keyword-based search over relational databases,” in *Proc. of ICDE*, 2002.

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

[10] B. Ding, J. Yu, S. Wang, L. Qing, X. Zhang, and X. Lin, “Finding top-k min-cost connected trees in databases,” in *Proc. of ICDE*, 2007.

[11] B. Kimelfeld and Y. Sagiv, “Finding and approximating top-k answers in keyword proximity search,” in *PODS*, 2006, pp. 173–182.

[12] H. He, H. Wang, J. Yang, and P. Yu, “BLINKS: Ranked keyword searches on graphs,” in *Proc. of SIGMOD*, 2007.

[13] K. Golenberg, B. Kimelfeld, and Y. Sagiv, “Keyword proximity search in complex data graphs,” in *SIGMOD Conference*, 2008, pp. 927–940.

[14] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, “Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data,” in *SIGMOD Conference*, 2008, pp. 903–914.

[15] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal, “Query relaxation by structure and semantics for retrieval of logical web documents,” in *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[16] E. Ihler, “Bounds on the quality of approximate solutions on the group steiner tree problem,” in *16th International Workshop on Graph-Theoretic Concepts in Computer Science*, 1991.

[17] L. Kou, G. Markowsky, and L. Berman, “A fast algorithm for steiner trees,” vol. 15, no. 2, June 1981, pp. 141–145.

[18] V. Hristidis, L. Gravano, and Y. Papakonstantinou, “Efficient ir-style keyword search over relational databases,” in *Proc. of VLDB*, 2003.

[19] K. Anyanwu, A. Maduko, and A. P. Sheth, “Sparq2l: towards support for subgraph extraction queries in rdf databases,” in *WWW*, 2007, pp. 797–806.

[20] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar, “Anatomy of the ado.net entity framework,” in *SIGMOD Conference*, 2007, pp. 877–888.

[21] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, “NAGA: Searching and Ranking Knowledge,” in *24th International Conference on Data Engineering (ICDE 2008)*. IEEE, 2008.

[22] H. Tong and C. Faloutsos, “Center-piece subgraphs: problem definition and fast solutions,” in *KDD ’06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2006, pp. 404–413.

[23] C. Faloutsos, K. S. Mccurley, and A. Tomkins, “Fast discovery of connection subgraphs,” in *KDD ’04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM Press, 2004, pp. 118–127. [Online]. Available: http://dx.doi.org/10.1145/1014052.1014068

[24] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal, “Retrieving and organizing web pages by “information unit”,” in *WWW*, 2001, pp. 230–244.

[25] J. Graupmann, “The spheresearch engine for graph-based search on heterogeneous semi-structured data,” Ph.D. dissertation, Universität des Saarlandes, May 2006.

[26] K. Mehlhorn, “A faster approximation algorithm for the steiner problem in graphs,” *Inf. Process. Lett.*, vol. 27, no. 3, pp. 125–128, 1988.

[27] S. Dreyfus and R. Wagner, “The steiner problem in graphs,” in *Networks*, 1972.

[28] R. B. Muhammad, “A parallel local search algorithm for euclidean steiner tree problem,” in *SNPD-SAWN ’06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 157–164.

[29] O. Faroe, D. Pisinger, and M. Zachariasen, “Local search for final placement in vlsi design,” in *ICCAD*, 2001, pp. 565–572.