STAR: Steiner Tree Approximation in Relationship-Graphs

Gjergji Kasneci, Maya Ramanath,
Mauro Sozio, Fabian M. Suchanek,
Gerhard Weikum

MPI–I–2008–5–001 May 2008

Authors' Addresses

Max-Planck-Institut für Informatik
Campus E1 4
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Acknowledgements

We would like to thank Gerard de Melo for the thorough proof reading of this report and his helpful comments.

Abstract

Large-scale 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 graphs that do not fit into memory. We prove that for n query entities, STAR yields an $O(\log(n))$-approximation of the optimal Steiner tree, and show that in practical cases the results returned by STAR are qualitatively 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 STAR outperforms the best state-of-the art database methods by a large margin, and also returns qualitatively better results.

Keywords

Steiner Tree, Algorithm, Ontology

Contents

1 Introduction **2**
- 1.1 Motivation and Problem 2
- 1.2 Contributions and Outline 4

2 Related Work **5**

3 The STAR algorithm **8**
- 3.1 First phase 8
- 3.2 Second Phase 9
  - 3.2.1 Fixed Nodes and Loose Paths 9
  - 3.2.2 Observations 10
  - 3.2.3 Finding an approximate Steiner tree 10
- 3.3 Approximation Guarantee 13
- 3.4 Time complexity 17
- 3.5 Approximate Top-k Interconnections 19

4 Evaluation **21**
- 4.1 Comparison of STAR and DNH 21
- 4.2 Comparison of STAR and BLINKS 23
- 4.3 Comparison of STAR and BANKS 24
- 4.4 Summary of results 26

5 Conclusion **28**

1 Introduction

1.1 Motivation and Problem

Many data-intensive applications need to query and analyze large graphs and networks that do not fit into main memory. 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. Often the data 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. An example of such a graph is the YAGO knowledge base [24], 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/ontology [8]. Figure 1.1 shows an excerpt; the entire YAGO graph consists of more than 1.7 million nodes (entities and entity classes) and 14 million 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.

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*

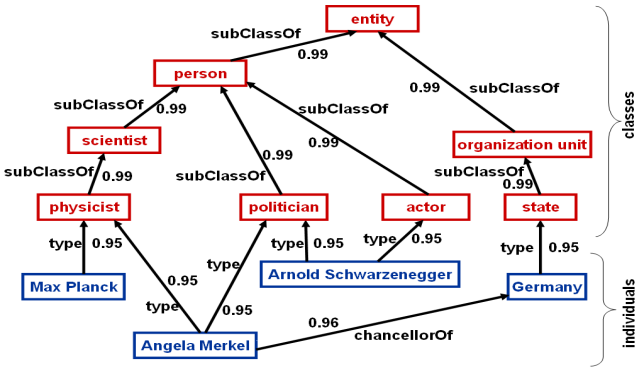

Figure 1.1: Example of an entity relationship graph

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 [18, 23, 25]. 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 [4, 14, 2, 12]; 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

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

on fixed-parameter tractability [5], as well as the tractability results on the approximate enumeration of the *top-k* approximate results [16], most prior works have resorted to heuristics, and, in fact, have typically modified the ranking model for the sake of efficiency. For example, instead of minimizing the Steiner tree cost, they minimize the sum of shortest paths between all pairs of input nodes (e.g., [17, 10]). This is unsatisfying as it mixes arguments about query and ranking semantics with arguments about efficiency.

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.

1.2 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 generalize STAR to an algorithm that is capable of computing approximate *top-k* relation trees for a given set of 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 [5]). In our experiments on real-life datasets, STAR achieves better results (i.e. trees of lower weight) than the ones returned by the 2-approximation algorithm presented in [17].
- We compare STAR to the best state-of-the-art database methods in comprehensive experiments with large graphs. STAR outperforms the opponents by an order of magnitude and sometimes even more.

The remainder of this paper is organized as follows. Section 2 gives an overview on related work. In Section 3, we present our algorithm and a detailed analysis of its runtime complexity and approximation ratio. Furthermore, we generalize our algorithm to a *top-k* approximation algorithm. The evaluation of our approach is presented in Section 4.

2 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 [2, 11, 12, 4, 14, 5, 10], graph search over ER, RDF and other knowledge bases [3, 1, 15], entity relationship queries on the Web [19, 9], etc. Such applications have to deal with large graphs (sometimes with millions of nodes and edges) in general, and require not only qualitatively superior 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. And so, 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 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.

In this paper we describe and evaluate a top-k Steiner tree algorithm for large graphs stored on disk. 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, introduced in [17], 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-approximation to the optimal Steiner tree. This heuristic was refined by Mehlhorn [21] for a fast 2-approximation solution to the Steiner tree problem, and has been emulated by further approaches for the top-k Steiner tree computation [4, 14].
Span and cleanup: This heuristic (see [13]) 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. [20] 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).
Dynamic programming: The first dynamic programming approach to the Steiner tree problem was introduced by Dreyfus and Wagner [6]. It proceeds by computing optimal results for all subsets of terminals. Then the optimal result is computed for all the terminals. In [5], this heuristic is modified to a faster method 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.
Local search: This heuristic has been used in the realm of the Euclidean Steiner tree problem [22, 7] and of the parallel Steiner tree computation. 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 [4, 14] and BLINKS [10]). We briefly describe these two algorithms and then compare them to STAR.

The first BANKS paper [4] (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 follow the edges backwards. As soon as the iterators meet, a result is produced. This technique is improved in BANKS II [14] 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.

The recently proposed BLINKS [10] 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 the block is reached, new cursors are created to explore the remaining blocks connected to this portal node. Experiments show that BLINKS performs an order of magnitude better than BANKS II.

In contrast to the backward expansion approach of BANKS and BLINKS, STAR's first step is to quickly construct an initial tree using a taxonomic DAG when available. This tree is then iteratively refined to improve the cost until no more improvements can be made. As we show in the experimental evaluation, the construction of the initial tree improves the running time by an order of magnitude. Furthermore, STAR returns trees, while 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 and construct the indexes, while our graph can be stored in a database and only database indexes are 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 each block that it has to traverse. Hence for dense graphs, the performance of BLINKS suffers because of the large number of blocks that have a portal in common.

3 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 $w : E \rightarrow \mathbb{R}_+$, 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 *top-k* such trees in the order of increasing weights.

Many real world graphs come along 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 substructures, 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.

3.1 First phase

In order to build a first interconnecting tree, STAR relies on a similar strategy as BANKS I [4]. It runs single source shortest path iterators from each node of V'. 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 3.1). This way, STAR can quickly find a taxonomic ancestor of all nodes from V'. Consider the sample graph of Figure 1.1. Suppose that V'={**Max Planck, Arnold Schwarzenegger, Germany**}. In the first phase, STAR would construct the tree depicted in Figure 3.1.

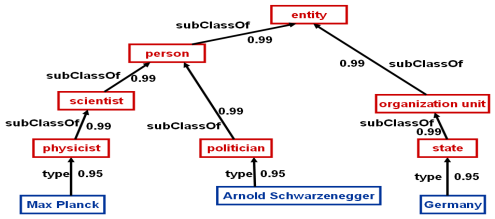

Figure 3.1: Taxonomic interconnection

In case that no taxonomic information about the underlying graph is available, STAR's strategy is the same as that of BANKS I.

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

3.2 Second Phase

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

3.2.1 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 fixed node is a terminal node or a node v that has degree $deg(v) \geq 3$ in T.*

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 iff its end nodes are fixed nodes and all intermediate nodes (if any) are Steiner nodes of 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.

3.2.2 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 3.2, 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 3.2, the removal of the loose path that connects a and b turns b into an unfixed node and d into a fixed node. The loose paths connected to b 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 3.2, the loose path that went through d is split into two loose paths. Hence, the number $|LP(T')|$ of loose paths in an improved tree T' is $|LP(T)| - 2 \leq |LP(T')| \leq |LP(T)| + 2$. It can be shown that the number of loose paths in a given tree T is $|V'| - 1 \leq |LP(T)| \leq 2|V'| - 3$.

3.2.3 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 of 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 3.2, 3.3, and 3.4). 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.

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

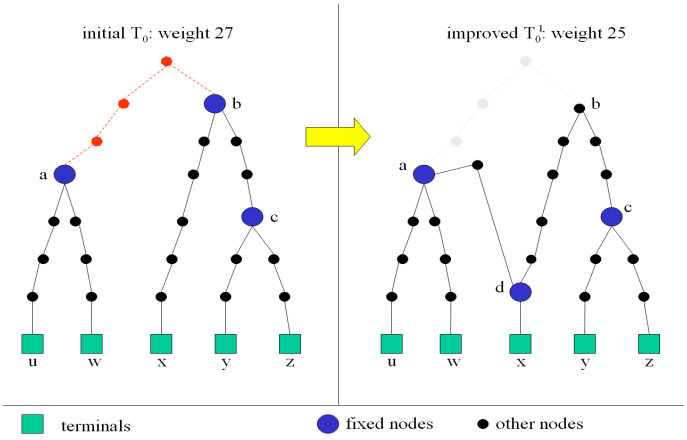

Figure 3.2: After first iteration

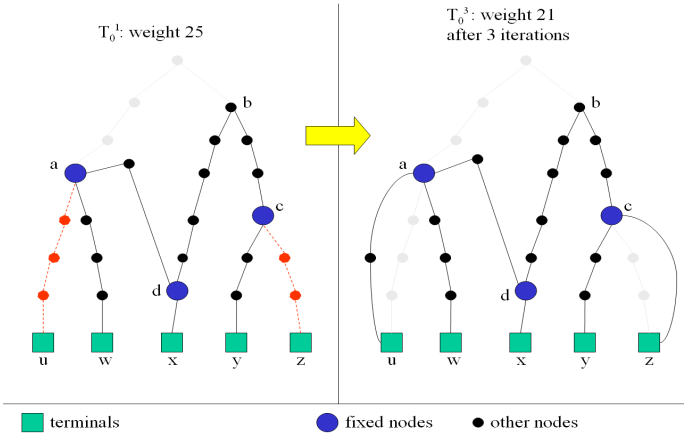

Figure 3.3: After third iteration

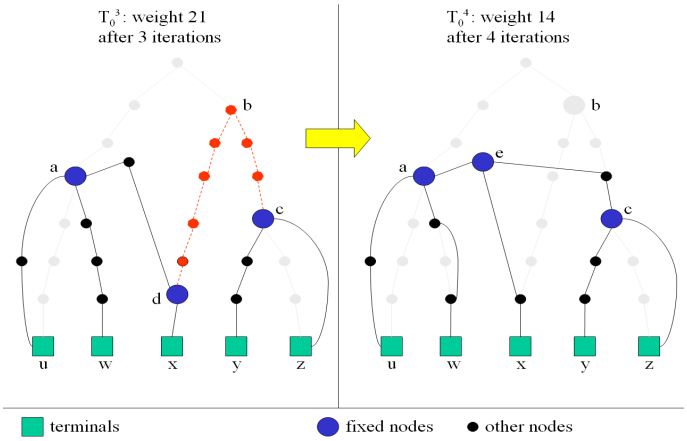

Figure 3.4: After fourth iteration

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
```

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 3.1). In the first iteration the algorithm would remove the loose path from the fixed node labeled with **Germany** to the fixed node labeled with **person**. The improved tree is depicted in Figure 3.5. Note that since STAR aims to find closest relations between entities, it views the edges in Figures 3.5 and 3.6 as undirected.

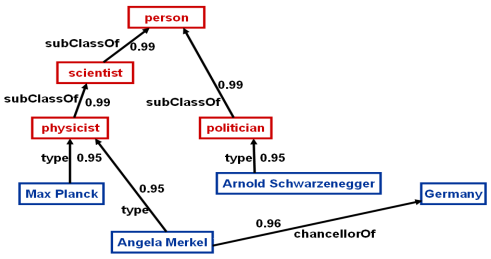

Figure 3.5: Result of the first iteration.

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 is at the same time the final tree, since no loose path can be improved.

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.

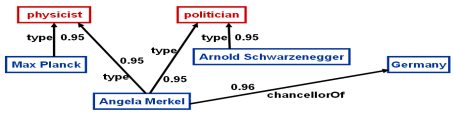

Figure 3.6: Result of the second iteration.

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 total sum of node degrees. Intuitively, $Q_{current}$ represents the iterator that is currently expanded. This expansion heuristic is similar to the one used by BANKS II [14], 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' from 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 to rebuild the path from v' to the source. However, the iterator does not expand a node v (line 14) 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.

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

3.3 Approximation Guarantee

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

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** $degree(Q_{other}) < degree(Q_{current})$ **then**
11: $swap(current, other)$
12: **end if**
13: $v = Q_{current}.dequeue()$
14: **if** $d_{current}(v) \geq w(lp)$ **then**
15: **continue**
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 of* v

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 1 deals with this non-trivial task.

Before diving into the proof, we need some auxiliary notations. We shall denote an ordered pair with (i, j) (this means that $(i, j) \neq (j, i)$), while an

unordered pair will be denoted with $\{i, j\}$. For any graph G, $d_G(u, v)$ denotes the distance between u and v in G. In a tree, we denote with 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 1 *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 $\mu : \mathcal{L}(T_A) \rightarrow V' \times V'$ 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 into $\{v_i, v_{i+1}\}$.*

Proof For convenience of presentation, we root T_A at any arbitrary terminal node and direct edges from the root towards the leaves. Then, we denote with $u \rightarrow v$ a path where u is closer to the root than v. Furthermore, for any subtree T of T_A we shall denote with $\tau(T)$ 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 $P = u \rightarrow v$ 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 with 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 $j_{h+1} \leq 2j_h - 1$. Let $P = u \rightarrow v$ be the loose path closest to the root, between the loose paths

labeled with $(i, i + j_h)$ and $(i, i + j_{h+1})$. 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 $j_{h+1} - j_h \leq j_h$. Hence, P would have been labeled $(i + j_{h+1}, i + j_h)$. In the latter case, $v_{i+j_{h+1}} \notin \tau(T_v)$ and $j_{h+1} - j_h \leq j_h$, which implies that P would have been labeled $(i + j_h, i + j_{h+1})$. 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

$$\sum_{k=1}^{N} d_{T_O}(v_k, v_{k+1}) = 2w(T_O). \tag{3.1}$$

Using Lemma 1, 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

$$d_{T_A}(u, v) \leq d_{T_O}(\mu(uv)), \tag{3.2}$$

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

$$
\begin{aligned}
w(T_A) &= \sum_{uv \in LP(T_A)} d_{T_A}(u,v) && (3.3)\\
&\leq \sum_{uv \in LP(T_A)} d_{T_O}\mu(uv) && (3.4)\\
&\leq \sum_{k=1}^{N} \left(2\lceil \log N + 2 \rceil\right) d_{T_O}(v_k, v_{k+1}) && (3.5)\\
&\leq \left(4\lceil \log N \rceil + 4\right) w(T_O). && (3.6)
\end{aligned}
$$

□

where inequality (3.4) follows from Equation 3.2, inequality (3.5) follows from property 3 of the mapping μ, and inequality (3.6) follows from Equation 3.1.

3.4 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, respectively, 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, respectively, the number of vertices, the number of edges, and the number of terminals in the input graph.

Lemma 2 *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

$$w(P) - w(P') \geq (1+\epsilon)w(P') - w(P') \geq \epsilon w_{\min}. \tag{3.7}$$

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

$$mw_{\max} - k\epsilon w_{min} \geq 0 \Leftrightarrow k \leq \frac{1}{\epsilon}\frac{w_{\max}}{w_{\min}}m. \tag{3.8}$$

□

The next theorem shows a tradeoff 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 $O(\frac{1}{\epsilon}\frac{w_{\max}}{w_{\min}}mN(n\log n + m))$.*

Proof The time-complexity bound follows from Lemma 2 and from the fact that at each step the STAR algorithm might invoke Dijkstra's algorithm at most $2N$ times (one for each loose path). To prove the approximation ratio, it suffices to replace Equation 3.2 in Theorem 1 with

$$d_{T_A}(u,v) \leq (1+\epsilon)d_{T_O}(\mu(uv)), \tag{3.9}$$

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

$$\begin{aligned} w(T_A) &= \sum_{uv \in \mathcal{L}(T_A)} d_{T_A}(u,v) && (3.10)\\ &\leq \sum_{uv \in \mathcal{L}(T_A)} (1+\epsilon) d_{T_O}\mu(uv) && (3.11)\\ &\leq \sum_{k=1}^{N} (1+\epsilon)\left(2\lceil \log N \rceil + 2\right) d_{T_O}(v_k, v_{k+1}) && (3.12)\\ &\leq (1+\epsilon)\left(4\lceil \log N \rceil + 4\right) w(T_O). && (3.13) \end{aligned}$$

□

3.5 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 edge-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'

4 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 BLINKS [10], which is the newest and experimentally best algorithm in this field. We compared the algorithms both in terms of the quality of the returned results and in terms of their performance. The third algorithm is BANKS [4] and its improved version BANKS II [14], which are state-of-the-art algorithms for keyword proximity search on relational data. 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.

4.1 Comparison of STAR and DNH

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 $4\log(n) + 4$. These bounds, however, are theoretical bounds for the worst case. Therefore, we studied how the two algorithms perform in practice.

Datasets We use DBLP[1] for our experiments. The DBLP data can be viewed as a graph, whose nodes represent entities (like *author*, *publication*, *conference* etc.), etc. and whose edges represent relations (like *cited_by*, *author_of*, etc.). Since the DNH algorithm is designed to deal only with graphs that can be loaded into main memory, we extracted a subgraph of the DBLP graph with 10,000 nodes and 300,000 edges (dataset DBLP-1). Since the

[1]Data downloadable from `http://dblp.uni-trier.de/xml`

Method	# teminals	Avg. result weight	Avg. time (ms)
STAR	3	**7.3**	**67.7**
DNH		7.5	3729.3
STAR	5	**11.95**	**303.9**
DNH		12.45	16137.4
STAR	7	**16.67**	**742.15**
DNH		17.0	56421.6

Table 4.1: DBLP-1: Quality of answers and efficiency for STAR and DNH

original DBLP does not provide any edge weights, we used uniform weights for DBLP-1. As the qualitative performance of DNH can be different in weighted and unweighted graphs, we constructed a second dataset (DBLP-2) with the same nodes and edges as DBLP-1 and with randomly chosen edge weights between 0 and 1.

Queries We constructed three query sets with 3,5 and 7 terminals, respectively. Each query set consists of 20 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) to the weight of the tree returned by DNH. We also measured the running times of the algorithms.

Results Table 4.1 shows the results of our experiments on DBLP-1. Column 3 shows the average weight of the result over the 20 queries in the query sets returned by STAR and DNH. 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 thus consistently better. We validated the statistical significance of the superiority of STAR using a *t-test* at level $\alpha = 0.05$ and conclude that despite its worse approximation ratio, STAR returns better results than DNH for these practical cases. Column 4 shows the average runtime of the algorithms in milliseconds.

Table 4.2 shows that STAR outperforms DNH also in the case of weighted edges.

1	2	3	4
Method	# terminals	Avg. result weight	Avg. time (ms)
STAR	3	**4.29**	**83.2**
DNH		4.798	3854.05
STAR	5	**6.18**	**256.1**
DNH		6.65	13022.2
STAR	7	**8.10**	**783.35**
DNH		8.84	58151.0

Table 4.2: DBLP-2: Quality of answers and efficiency for STAR and DNH

4.2 Comparison of STAR and BLINKS

BLINKS uses indexes in order to significantly speed up the query processing time. However, in order to build these indexes and to subsequently use them during run time, BLINKS requires the entire graph in main memory. For this reason, we used again the DBLP dataset for the comparison. We experimented with different block sizes and chose a block size of 100 nodes, since this gave the best results. We used an implementation of BLINKS that was kindly provided to us by the authors.

Metrics Since BLINKS uses a different weight metrics (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 the runtime of the algorithms.

Queries Unlike DNH, BLINKS computes the top-k results for a query – like the STAR algorithm. Hence, we compared the algorithms for different values of k. Again, we constructed 3 query sets, each containing 30 random queries with different numbers of terminal nodes (3,5,7). For each of the query sets, we report the average runtime for retrieving the top 1, top 5 and top 10 results.

Results Table 4.3 presents the runtime performance of STAR and BLINKS on the DBLP-1 dataset. The results show that STAR outperforms BLINKS in all cases by an order of magnitude. The reason for this is that the DBLP dataset is relatively dense with an edge to node ratio of 30. This means

# terminals	Avg. time STAR (ms)	Avg. time BLINKS (ms)
top-1		
3	**69.2**	870.53
5	**314.53**	2007.8
7	**727.43**	5189.37
top-5		
3	**787.67**	5347.9
5	**1808.03**	13315.9
7	**4419.9**	32277.8
top-10		
3	**8363.6**	36501.03
5	**21463.0**	89719.33
7	**55013.2**	257040.5

Table 4.3: DBLP-1: Efficiency of STAR and BLINKS

that the index structure of BLINKS may have a large number of partitions with the same portal p. Whenever such a portal is reached, the number of new cursors required to continue the search is equal to the number of blocks for which p is a portal. Hence, if the dataset is very dense, it is likely that a large number of cursors are required to complete the query processing. The overhead of maintaining these cursors negatively affects the overall performance.

By 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. Hence, a combination of a tight upper bound to prune the exploration as well as limited overhead in iterators allows STAR to outperform BLINKS by such a large margin.

4.3 Comparison of STAR and BANKS

BANKS [4, 14] is a state-of-the-art algorithm in the field of proximity search on relational data. Unlike the DNH algorithm and BLINKS, BANKS can be applied efficiently and directly to graphs that do not fit into main memory. Since such large graphs are also realistic scenarios for the Steiner tree problem, we decided to use a disk-resident dataset for the comparison of BANKS and STAR.

3 terminals					6 terminals			
top-1	STAR	STAR(BI)	BANKS I	BANKS II	STAR	STAR(BI)	BANKS I	BANKS II
Avg. score	**0.22**	0.27	0.260	0.234	0.337	**0.324**	0.385	0.368
Avg. # acc. edges	**6981**	85931	84171	81462	**9559**	375523	372634	365004
Avg. run time (ms)	**12440.6**	133917.2	131313.6	104148.5	**15733.1**	394794.4	391601.0	385401.5
top-3	STAR	STAR(BI)	BANKS I	BANKS II	STAR	STAR(BI)	BANKS I	BANKS II
Avg. score	**0.428**	0.431	0.488	0.454	**1.085**	1.131	1.193	1.255
Avg. #acc. edges	**18027**	124566	153078	132141	**27085**	397004	460521	409414
Avg. run time (ms)	**34814.7**	139462.3	190547.7	156535.3	**41187.3**	417303.1	483328.4	427276.3
top-6	STAR	STAR(BI)	BANKS I	BANKS II	STAR	STAR(BI)	BANKS I	BANKS II
Avg. score	**2.102**	2.309	2.453	2.441	**3.315**	3.541	4.148	4.031
Avg. # acc. edges	**43474**	138852	159130	175045	**76259**	447813	503054	491786
Avg. run time (ms)	**71058.2**	159571.3	197543.7	205359.6	**91157.2**	475242.5	511811.0	491785.5

Table 4.4: YAGO: Quality of results and efficiency of STAR, STAR(BI), BANKS I & II

Dataset We chose the graph of the YAGO knowledge base [24]. 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. In order to convert confidence scores into distance measures[2], we weighted each edge with $-log(c)$ where c is the confidence of the edge. We store the graph in an Oracle database with the simple schema

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

To the best of our knowledge, this is the largest graph ever studied for the Steiner tree problem.

YAGO contains a DAG-shaped taxonomy of *type* and *subClassOf* edges (see Figure 1.1), which can be used by STAR in its first phase to construct the initial tree. To give BANKS a fair chance, we ran STAR both *with* this taxonomic information and *without* (STAR(BI)).

We implemented both BANKS I [4] and its improved version BANKS II [14] in Java following their descriptions for main-memory procedures. Whenever the algorithm explores 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

[2]That is, an edge with high confidence should have correspondingly low weight since the result trees returned are trees with low weights.

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.

Results Table 4.4 shows the results for the performance of STAR, STAR (BI) with BANKS I heuristics for initialization, BANKS I, and BANKS II. Concerning the quality of the output trees, STAR and STAR(BI) return 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 the other algorithms. This is also reflected directly in the number of edges accessed by each algorithm: STAR access 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.

But even if the taxonomic structure is not exploited (STAR(BI)), our algorithm accesses far fewer edges than both BANKS I and BANKS II for $k > 1$. For k=1, STAR(BI) is essentially BANKS I. However, in all other cases (i.e. $k > 1$), STAR(BI) is faster than its competitors. The main reason for the drastic increase of the number of visited edges by BANKS I and BANKS II with growing number of terminals is that the single-source-shortest-path iterators of BANKS I and BANKS II visit all possible edges in a breadth-first manner. There is no upper bound on which the number of visited edges can be pruned. In the case of STAR and STAR(BI), the weight of the loose path upon which the current tree is being improved serves as a tight upper bound for new interconnecting paths, and prunes the number of visited edges effectively.

4.4 Summary of results

We compared STAR to 3 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 metrics and hence cannot give an approximation guarantee. In all experiments, STAR outperforms its competitors.

In summary, we have shown that the results returned by STAR are not only better than those returned by BANKS I and II, but also better than the results returned by a 2-approximation algorithm. We have also shown that STAR has superior runtime performance in memory compared to BLINKS. The reason for this efficient performance is three-fold: 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 length of the paths and prunes the possible paths that can be included in the result tree.

5 Conclusion

This paper has addressed the problem of efficiently answering relationship queries over large 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 all-pairs shortest-path algorithm for subtrees defined by the notion of loose paths.

We proved that STAR is 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 [4, 14]. 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 all database methods or better on practically relevant datasets.

As for run-time, STAR outperformed all opponents by a large margin, often by an order of magnitude. This holds for both the in-memory case and on-disk graphs with a size that could be handled only by a subset of the prior methods.

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 [15]. Our future work will look into more complex search patterns over this kind of rich relationship-graphs, using STAR as a key building block.

Bibliography

[1] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ado.net entity framework. In *SIGMOD Conference*, pages 877–888, 2007.

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

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

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

[5] 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.

[6] S. Dreyfus and R. Wagner. The steiner problem in graphs. In *Networks*, 1972.

[7] O. Faroe, D. Pisinger, and M. Zachariasen. Local search for final placement in vlsi design. In *ICCAD*, pages 565–572, 2001.

[8] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[9] J. Graupmann. *The SphereSearch Engine for Graph-based Search on heterogeneous semi-structured data*. PhD thesis, Universität des Saarlandes, May 2006.

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. volume 15, pages 141–145, June 1981.

[18] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, 2005.

[19] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing web pages by "information unit". In *WWW*, pages 230–244, 2001.

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

[21] K. Mehlhorn. A faster approximation algorithm for the steiner problem in graphs. *Inf. Process. Lett.*, 27(3):125–128, 1988.

[22] R. B. Muhammad. A parallel local search algorithm for euclidean steiner tree problem. In *SNPD-SAWN '06: Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 157–164, Washington, DC, USA, 2006. IEEE Computer Society.

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

[24] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *Proc. of WWW*, 2007.

[25] S. Trissl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD*, 2007.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

.

MPI-I-2008-5-001	G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, G. Weikum	STAR: Steiner Tree Approximation in Relationship-Graphs
MPI-I-2007-RG1-002	T. Hillenbrand, C. Weidenbach	Superposition for Finite Domains
MPI-I-2007-5-003	F.M. Suchanek, G. Kasneci, G. Weikum	Yago : A Large Ontology from Wikipedia and WordNet
MPI-I-2007-5-002	K. Berberich, S. Bedathur, T. Neumann, G. Weikum	A Time Machine for Text Search
MPI-I-2007-5-001	G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, G. Weikum	NAGA: Searching and Ranking Knowledge
MPI-I-2007-4-008	J. Gall, T. Brox, B. Rosenhahn, H. Seidel	Global Stochastic Optimization for Robust and Accurate Human Motion Capture
MPI-I-2007-4-007	R. Herzog, V. Havran, K. Myszkowski, H. Seidel	Global Illumination using Photon Ray Splatting
MPI-I-2007-4-006	C. Dyken, G. Ziegler, C. Theobalt, H. Seidel	GPU Marching Cubes on Shader Model 3.0 and 4.0
MPI-I-2007-4-005	T. Schultz, J. Weickert, H. Seidel	A Higher-Order Structure Tensor
MPI-I-2007-4-004	C. Stoll	A Volumetric Approach to Interactive Shape Editing
MPI-I-2007-4-003	R. Bargmann, V. Blanz, H. Seidel	A Nonlinear Viseme Model for Triphone-Based Speech Synthesis
MPI-I-2007-4-002	T. Langer, H. Seidel	Construction of Smooth Maps with Mean Value Coordinates
MPI-I-2007-4-001	J. Gall, B. Rosenhahn, H. Seidel	Clustered Stochastic Optimization for Object Recognition and Pose Estimation
MPI-I-2007-2-001	A. Podelski, S. Wagner	A Method and a Tool for Automatic Veriication of Region Stability for Hybrid Systems
MPI-I-2007-1-002	E. Althaus, S. Canzar	A Lagrangian relaxation approach for the multiple sequence alignment problem
MPI-I-2007-1-001	E. Berberich, L. Kettner	Linear-Time Reordering in a Sweep-line Algorithm for Algebraic Curves Intersecting in a Common Point
MPI-I-2006-5-006	G. Kasnec, F.M. Suchanek, G. Weikum	Yago - A Core of Semantic Knowledge
MPI-I-2006-5-005	R. Angelova, S. Siersdorfer	A Neighborhood-Based Approach for Clustering of Linked Document Collections
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining Linguistic and Statistical Analysis to Extract Relations from Web Documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment Texture Editing in Monocular Video Sequences based on Color-Coded Printing Patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: Index-access Optimized Top-k Query Processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems

MPI-I-2006-4-010	A. Belyaev, T. Langer, H. Seidel	Mean Value Coordinates for Arbitrary Spherical Polygons and Polyhedra in R^3
MPI-I-2006-4-009	J. Gall, J. Potthoff, B. Rosenhahn, C. Schnoerr, H. Seidel	Interacting and Annealing Particle Filters: Mathematics and a Recipe for Applications
MPI-I-2006-4-008	I. Albrecht, M. Kipp, M. Neff, H. Seidel	Gesture Modeling and Animation by Imitation
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving Non-local Denoising of Static and Time-varying Range Data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced Dynamic Reflectometry for Relightable Free-Viewpoint Video
MPI-I-2006-4-005	A. Belyaev, H. Seidel, S. Yoshizawa	Skeleton-driven Laplacian Mesh Deformations
MPI-I-2006-4-004	V. Havran, R. Herzog, H. Seidel	On Fast Construction of Spatial Hierarchies for Ray Tracing
MPI-I-2006-4-003	E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel	A Framework for Natural Animation of Digitized Models
MPI-I-2006-4-002	G. Ziegler, A. Tevs, C. Theobalt, H. Seidel	GPU Point List Generation through Histogram Pyramids
MPI-I-2006-4-001	A. Efremov, R. Mantiuk, K. Myszkowski, H. Seidel	Design and Evaluation of Backward Compatible High Dynamic Range Video Compression
MPI-I-2006-2-001	T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard	On Verifying Complex Properties using Symbolic Shape Analysis
MPI-I-2006-1-007	H. Bast, I. Weber, C.W. Mortensen	Output-Sensitive Autocompletion Search
MPI-I-2006-1-006	M. Kerber	Division-Free Computation of Subresultants Using Bezout Matrices
MPI-I-2006-1-005	A. Eigenwillig, L. Kettner, N. Wolpert	Snap Rounding of Bézier Curves
MPI-I-2006-1-004	S. Funke, S. Laue, R. Naujoks, L. Zvi	Power Assignment Problems in Wireless Communication
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects
MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-004	Y. Kazakov	A Framework of Refutational Theorem Proving for Saturation-Based Decision Procedures
MPI-I-2005-2-003	H.d. Nivelle	Using Resolution as a Decision Procedure
MPI-I-2005-2-002	P. Maier, W. Charatonik, L. Georgieva	Bounded Model Checking of Pointer Programs
MPI-I-2005-2-001	J. Hoffmann, C. Gomes, B. Selman	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-008	C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga	Cycle Bases of Graphs and Sampled Manifolds
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-003	S. Baswana, K. Telikepalli	Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae

MPI-I-2004-NWG1-001 B. Blanchet

Automatic Proof of Strong Secrecy for Security Protocols