Fast Approximation of Steiner Trees in Large Graphs

Andrey Gubichev Thomas Neumann
Technische Universität München
Garching, Germany
{gubichev, neumann}@in.tum.de

ABSTRACT

Finding the minimum connected subtree of a graph that contains a given set of nodes (i.e., the Steiner tree problem) is a fundamental operation in keyword search in graphs, yet it is known to be NP-hard. Existing approximation techniques either make use of the heavy indexing of the graph, or entirely rely on online heuristics.
In this paper we bridge the gap between these two extremes and present a scalable landmark-based index structure that, combined with a few lightweight online heuristics, yields a fast and accurate approximation of the Steiner tree.
Our solution handles real-world graphs with millions of nodes and provides an approximation error of less than 5% on average.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and Networks*; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Experimentation, Theory, Performance

Keywords

Graph Databases, Keyword Search, Steiner Trees

1. INTRODUCTION

1.1 Motivation

The *Steiner tree problem*, that is, the problem of connecting a given set of nodes (also called keywords, or terminals) in a graph such that the total length is minimized with respect to some predefined cost function, is a problem with long academic history. Its importance is based on a variety of applications ranging from VLSI design to the study of phylogenetic trees. In this paper we will concentrate on three application scenarios in the area of knowledge management:
Keyword search in graphs. Keyword search is the most popular information discovery method because it does not require the knowledge of the query language or the underlying schema. Consider the entity-relationship graph where nodes are entities (e.g., extracted from Wikipedia) and edges represent relationships between entities. The keyword search problem in this setting can be formulated as: Given a few input entries, discover the relationships between them. Due to the proliferation of large web-scale knowledge bases like YAGO, DBPedia or Freebase, the keyword search in graphs receives great attention for information discovery beyond traditional relational databases.
Keyword search in relational databases. Since tuples can be treated as nodes connected with foreign-key relationships, the keyword search on this type of data can be again modelled as finding the Steiner trees in graphs.
Social networks. In the Social Network setting, it is important to identify familiar strangers (Stanley Milgram, 1972) – i.e. the individuals who do not know each other, but share some attributes or properties in common. Here, the Steiner tree that spans the individual and its familiar strangers consists of the minimal number of edges that one needs to traverse in order to discover all of the familiar strangers.

1.2 Problem difficulty and our contributions

The Steiner tree problem is one of Karp's 21 $\mathcal{NP}$-complete problems [8]. Moreover, it is in the $\mathcal{APX}$ class, i.e. an arbitrarily good approximation can not be achieved in polynomial time. We have to, therefore, consider approximation heuristics with rather poor theoretical guarantees.
Most of the practical approximation algorithms for this problem estimate the Steiner tree using the shortest paths between the input keyword nodes. Doing this, existing approaches usually follow one of these extreme lines:
No index: Use the graph as is, and do not perform any precomputation on it. In this case, one has to follow a Breadth-First Search (or similar, e.g. Bidirectional search) expansion strategy starting from every input node of the query. While many effective heuristics on guiding and speeding up the Breadth-First Search in this setting have been proposed (BANKS [1], Bidirectional [7], STAR [9]), their performance is still poor on very large instances of graph data.
Index only: Perform extensive indexing of the graph, and after that do not use the original graph at all. This, for example, includes precomputation of all keyword-to-node distances in the original graph (or, within every partition of the graph [6]). Another proposed index is based on the computing of high powers of the adjacency matrix [12]. As we see, these techniques can not easily be applied for graphs with millions of nodes.

In this paper we propose a mixed approach of *no index* and *only index* that allows for efficient approximation of Steiner trees for both in-memory and disk-resident graphs. The idea is to use a landmark-based index for the fast approximation of distances between the keywords, and then to run the (very limited) local search on both the index and the graph in order to minimize the approximation error.
We build upon our recently proposed path-sketch index for shortest path approximation [5], which in turn is based on the classical concept of landmark-based distance oracles [14]. The goal is to generalize the approximation scheme from the case of just 2 keyword nodes (*shortest path problem*) to the general case of k nodes (*Steiner tree problem*).
To summarize, the contributions of the paper are the following:

1. We build a generalized version of the sketch-based path estimation algorithm for the Steiner tree estimation (the `Sketch` algorithm)

2. We devise an algorithm that utilizes the sketch index and performs (limited) local search on the graph (the `SketchLS` algorithm)

3. We perform an extensive evaluation study on large real-world graphs that contain up to three million nodes.

In all of our benchmark datasets, the estimates returned by our algorithms are better than those delivered by state-of-the-art keyword search algorithms, and we achieve runtimes that are an order of magnitude faster than in other approaches.
The rest of the paper is organized as follows. After the related work overview (Section 2), we describe the previously devised sketch algorithm (Das Sarma et al.[2] and Gubichev et al.[5]) and show the way to use it for Steiner tree approximation (Section 3). Subsequently, we describe the new algorithm combining the sketch index with the online search on the graph. We proceed to Section 4, which contains extensive experiments on the real-world graphs and comparisons with other approaches. Section 5 concludes our findings.

2. RELATED WORK

The problem has a very rich history, here we will briefly review the work done in three major directions: (i) exact methods and approximation bounds, (ii) no-index (exploration) heuristics, (iii) index-based heuristics.

Exact methods and theoretical approximations: Dreyfus and Wagner [4] and recently Ding et al. [3] exploit the dynamic programming approach to the Steiner tree problem by computing optimal results for all subsets of terminals (DPBF algorithm). Both methods are naturally applicable only to moderate size graphs.
One of the first approximation algorithms is the minimum-spanning tree (MST) heuristic [10]. This heuristic builds a complete graph (a *distance network*) on terminals, where edges are attributed with the shortest distances between corresponding terminals. At the second step, the minimum spanning tree of the distance network is computed and returned as an approximation of the Steiner tree, which is guaranteed to be at most 2 times worse than the exact Steiner tree.
Exploration heuristics: The MST heuristic has been emulated by BANKS [1] and Bidirectional [7]. BANKS operates with k iterators (one per input keyword) which are expanded in a breadth-first manner along incoming edges (i.e., in backward direction) until they meet, and then the result subtree is constructed. Bidirectional [7] improves on this method by adding the forward-directed traversal, reducing the number of iterators, and prioritizing nodes with low degrees for expansion. STAR [9] follows the intuition of heuristic local search. Initially, a candidate tree is constructed by similar breadth-first expansions. Then, this candidate tree is improved by replacing the longest path in the tree with a shorter one. The algorithm terminates when no further replacement is possible.
Index-based heuristics: BLINKS [6] operates on two indexes that capture keywords reachable from nodes, and vice versa. The input graph is partitioned into blocks, and the top level block index as well as the intra-block indexes for each block are maintained. The keyword search starts with backward expansion within multiple blocks. If the boundary of a block is reached, new iterators are created to explore the adjacent blocks. BLINKS greately depends on the partitioning of the graph, which is quite an expensive operation in itself. Moreover, the performance of BLINKS suffers in case of relatively dense graphs (such as social networks), where two adjacent blocks may have many boundary nodes in common, and one boundary node may be shared by many blocks.

We note that some of the techniques mentioned above were designed with goals broader or different than just Steiner tree computation: BANKS, BANKS II and DPBF can also deliver the solution of the Group Steiner Tree; many of the algorithms return top-k results and rank them.
We also briefly mention here that the database community has considered keyword queries over relational databases using the schema-based approaches. A survey of this line of work can be found in [15].

3. ALGORITHM DESCRIPTION

In this section we explain our algorithms for Steiner tree approximation in detail. We first provide a concise description of the *sketch* index devised by Das Sarma et al. [2] and modified by Gubichev et al. [5], which lays the foundation for our approach. We proceed with two novel algorithms for the Steiner tree approximation.

3.1 Preliminaries

Let $G = (V, E)$ denote an undirected and unweighted graph with vertexes V and edges E. Note that our algorithms can be easily extended for directed and weighted graphs, we leave out the details due to the lack of space. We assume G is connected.
Steiner Tree. A tree T is a connected subgraph of G without cycles. In other words, if T is a tree, there exists a vertex $r \in T$ (called the *root*) such that every node in T is connected to r via a simple path. We write $|T|$ to denote the size of the tree, i.e. the number of distinct edges in T. The Steiner tree for the node set $Q = \{q_1, \dots q_k\} \subseteq V$ is the tree of minimal size that contains all the nodes from Q (also called terminals or *keywords*). Note two extreme cases: when $k = 2$, the Steiner tree is the shortest path between q_1 and q_2; when $k = |V|$, the Steiner tree is the minimum spanning tree of G.
Steiner Tree Approximation. Given the terminal set $Q \subseteq V$, let T_s denote a Steiner tree (note that there could be several) for Q. Furthermore, let T_a be an arbitrary tree

containing all the nodes from Q. By definition, $|T_s| \leq |T_a|$, and if we consider T_a as an *approximation* of T_s, then the error ratio of this tree is defined as $\text{ratio}(T_a) := \frac{|T_a|}{|T_s|} \in [1, \infty)$. The goal of our heuristical approximation algorithms is to keep $\text{ratio}(T_a)$ as small as possible while achieving low running time.

3.2 Sketch Index

The classical distance approximation algorithm has two stages: indexing of the graph in the offline part, and fast approximation of distances using the index in the online part. For the indexing part, a small subset of nodes (*landmarks*) is selected, and distances between landmarks and all other nodes are computed with the standard Dijkstra's algorithm. The online part employs the triangle inequality: if u and v are the input nodes and l is the landmark, then $dist(u, v) \leq dist(u, l) + dist(l, v)$. We can, therefore, estimate $dist(u, v)$ with the (known from the index) $dist(u, l)$ and $dist(l, v)$.

The sketch index on which we build our approximation algorithms (proposed in [2] and modified in [5]) is computed with the following precomputation procedure:

1. **Seed sampling**. Sample uniformly at random the sets of *seed nodes* $S_1, S_2, \ldots S_m$ of sizes $1, 2, \ldots, 2^{m-1}$, where $m = \log |V|$.
2. **Shortest path tree**. For every seed set S_i perform the Dijkstra's algorithm expansion from it and compute the complete shortest path tree SPT_i.
3. **Sketch** For every node $v \in V$ and for every seed set S_i we get the node $l \in S_i$ which is the closest to v, and the corresponding path between v and S_i. All this information is extracted from SPT_i computed in the previous step. The node $l \in S_i$ is called the *landmark* for v in S_i.

Note that for every seed set the computation of SPT_i is performed independently from other sets. We can therefore easily run m instances of Dijkstra's algorithm in parallel, which leads to a very moderate indexing time.

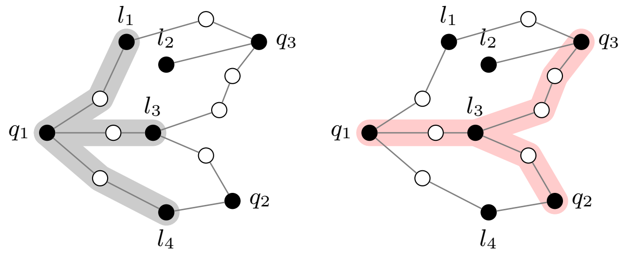
After indexing, the distance estimation algorithm can approximate the path (and therefore the distance) between u and v by loading $\mathsf{Sketch}(u), \mathsf{Sketch}(v)$, finding common landmarks in the sketches along with the corresponding paths, and concatenating those paths.

3.3 Sketch Algorithm

Our first contribution, the `Sketch` algorithm for the Steiner tree problem, is a generalization of the sketch-based shortest path approximation to the case of k input nodes. It is depicted in Algorithm 1. There, after loading the sketches and finding common landmarks between them, we construct the Steiner tree approximation by merging the paths from the input nodes to the landmarks. Figure 1 illustrates this idea: l_3 is the common landmark for three sketches, so we yield the tree consisting of three paths-branches: $(q_1, \ldots, l_3)$, $(q_2, \ldots, l_3)$ and $(q_3, \ldots, l_3)$

Since the diameters and thus the path lengths in the considered graphs are bounded (small world phenomena in social networks [13]), we can assume the constant time of adding a path to the tree T (line 8 of Algorithm 1). Furthermore, the number of iterations (lines 4-8) does not exceed the number of different seed sets used for the sketch computation. We get $|Res| \leq k \log |V|$, and therefore the complexity of the `Sketch` algorithm is $\mathcal{O}(k \log |V|)$.

Figure 1: SKETCH Algorithm Example



(a) Sketches of q_1 (highlighted), q_2 and q_3 are loaded **(b)** Common landmark is l_3. The result tree is highlighted

Algorithm 1: SKETCH($q_1, \ldots, q_k$)

```
Input: $Q = \{q_1, \ldots, q_k\} \in V$
Result: $Res$ – priority queue of trees containing $q_1, \ldots, q_k$
        ordered by tree size
1 begin
2     Load sketches $\mathsf{Sketch}(q_1), \ldots, \mathsf{Sketch}(q_k)$
3     $L \leftarrow$ common landmarks of $\mathsf{Sketch}(q_1), \ldots, \mathsf{Sketch}(q_k)$
4     foreach $l \in L$ do
5         $T \leftarrow \emptyset$
6         foreach $q_i \in Q$ do
7             Add path between $q_i$ and $l$ to $T$
8         Add $T$ to queue $Res$
9     return $Res$
```

3.4 SketchLS Algorithm

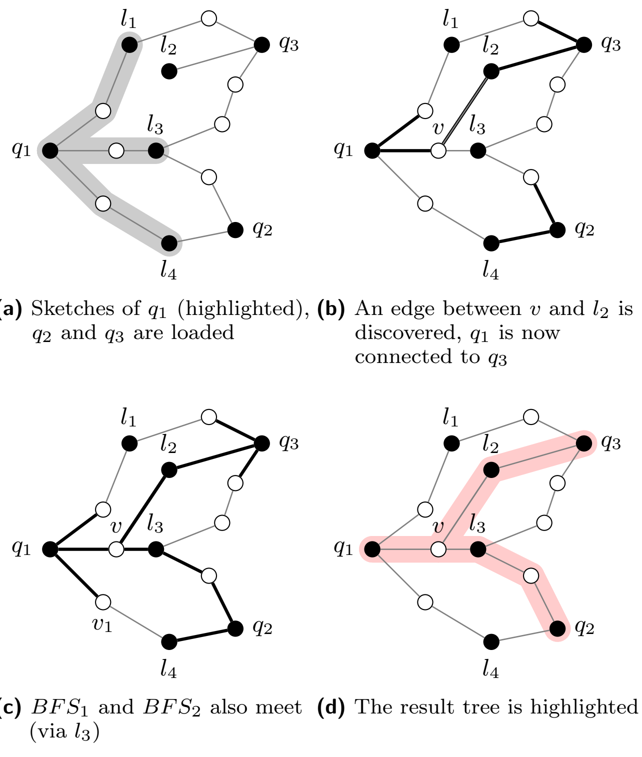
In this section we describe a new algorithm, coined `SketchLS`, for the Steiner tree approximation based on sketches and local search (LS) on the original graph.

The $\mathsf{Sketch}(v)$ consists of paths from v to the landmarks. This collection can be viewed as a tree with the root in v and landmarks as leaves. Every inner node in this tree belongs to some shortest path obtained during the precomputation stage, and the whole tree therefore is a subgraph of G.

The algorithm, depicted in Algorithm 1, starts with loading sketches and initializing k instances of Breadth-First Search that will run on sketches (lines 6-7). For every BFS process we keep the set of *frontier* nodes, that is, the nodes that are currently at the maximum distance from the source node. The BFS instances are called in a round-robbin manner (lines 8-22), and the current process makes one step in the sketch (line 9). The currently visited node v may be connected in the original graph G with the nodes visited by other processes. We check whether this is the case, i.e. whether the node v is in fact a neighbor of any previously visited node (lines 12-18), by looking up the neighbors of v in the original graph G, and if this is the case, we construct the path between q_i and q_j (line 16). This is the "local search" part in a sense that it involves browsing neighbors of a node in the original graph. We keep track of the connected pairs of nodes in S_{cover}, and as soon as all the input nodes are covered, stop the procedure. The set S_{cover} can be viewed as an edge list of the graph with nodes $q_1, \ldots, q_k$, since at every step we add to it an edge of a form (q_i, q_j). The condition in the lines 13-14 makes sure that this graph does not have cycles and that the result T is thus a tree.

The Figure 2 illustrates this algorithm. In the beginning, we load the sketches (Figure 2a) and initialize three BFS

Figure 2: SketchLS Algorithm Example



(a) Sketches of q_1 (highlighted), q_2 and q_3 are loaded **(b)** An edge between v and l_2 is discovered, q_1 is now connected to q_3

(c) BFS_1 and BFS_2 also meet (via l_3) **(d)** The result tree is highlighted

Algorithm 2: SketchLS$(q_1, \dots, q_k)$

```
   Input: $Q = \{q_1, \dots, q_k\} \in V$
   Result: $T$ –the tree containing $q_1, \dots, q_k$
1  begin
2      Load sketches Sketch$(q_1), \dots,$ Sketch$(q_k)$
3      $BFS \leftarrow$ new vector(k)                ▷ vector of BFS processes
4      $F \leftarrow$ new vector(k)                  ▷ frontiers of processes
5      $S_{cover} \leftarrow \emptyset$                          ▷ set of covered nodes
6      foreach $q_i \in Q$ do
7          $BFS[i] \leftarrow$ Breadth First Search from $q_i$
8      foreach $BFS_i \in BFS$ do
           ▷ round-robin iteration
9          $v \leftarrow BFS_i$.next()
10         $F[i]$.insert$(v)$
11         foreach $F[j] \in F, j \neq i$ do
12             if $\mathcal{N}(v) \cap F[j] \neq \emptyset$ then ▷ $\mathcal{N}(v)$ = neighbors of $v \in G$
13                 if $\{q_i, q_j\}$ creates a cycle in $S_{cover}$ then
14                     continue
15                 $n \leftarrow$ any node in $\mathcal{N}(v) \cap F[j]$
16                 $p \leftarrow (q_i, \dots, v, n, \dots, q_j)$          ▷ new path
17                 $T$.insert$(p)$
18                 $S_{cover}$.insert$(\{q_i, q_j\})$;
19         if $\neg BFS_i$.hasNext then
20             $BFS$.remove$(BFS_i)$
21         if $S_{cover}$ covers all $q_1, \dots, q_k$ then
22             break
23     return $T$
```

processes. After few iterations, the bold edges (Figure 2b) denote the edges already visited by Breadth-First iterators. The current node v is in the frontier of the first process, and it is connected to the node l_2 from the frontier of the third process. We immediately get the path (q_1, v, l_2, q_3). At the next step (Figure 2c), the remaining processes BFS_1 and BFS_2 meet via the edge (v, l_3). In this case, we also discover that v_1 and l_4 are neighbors, but that would create a cycle in S_{cover}: (q_1, q_2) and (q_2, q_1), so we skip this step and conclude: now S_{cover} covers all three input nodes.
Let us estimate the complexity of this algorithm. The size of the subgraph on which we perform our BFS processes, is the sum of k sketch sizes, i.e. $\mathcal{O}(k \log |V|)$ (We assume that the diameter of the graph is bounded, this holds for most of the real-world graphs except the road networks [13].) So, we perform $\mathcal{O}(k \log |V|)$ requests for neighbors in the graph G, and $\mathcal{O}(k^2 \log^2 |V|)$ set intersections for every pair of visited nodes in the worst case (line 12). If the maximal degree of G is D_{max}, then the hash-intersection of $F[j]$ and $\mathcal{N}(v)$ has the complexity of $\mathcal{O}(D_{max} + k \log |V|)$, and the overall complexity of `SketchLS` is $\mathcal{O}(k^2 \log^2 |V|(D_{max} + k \log |V|))$. As we see, the asymptotic behaviour of `SketchLS` is worse than that of `Sketch`. However, in the evaluation section we will show that it is compensated by exceptional quality of solutions found by `SketchLS`, and that both algorithms are still orders of magnitude faster than their state-of-the-art competitors.

4. EVALUATION

We describe the competing algorithms and the datasets used for the evaluation in Subsection 4.1. The approximation quality of the different approaches is assessed in Subsection 4.2.1, query runtime measurements are carried out in Subsection 4.2.2.

4.1 Systems and Datasets

We implement our algorithms in C++ uning the GNU-C++ STL library. We use the C++ implementation of DPBF [3] provided by the authors, and re-implement the Bidirectional algorithm [7] in C++ following the reference implementation in Java that was kindly provided by Heo He [6]. We also implement the minimum spanning tree heuristic (MST). We do not consider BANKS, since its successor Bidirectional outperforms it [7], and BLINKS, because it optimizes a different objective function and we are not able to measure the approximation quality.
We examined our approach in comparison with other systems for two classes of real-world graphs:

Relational databases. We experiment with **IMDb** [6], a dataset derived from the popular website. It contains tuples from different tables such as *Movie*, *Person*, *Role*, connected via the foreign-key relationships. The resulting graph has 44,345 nodes and 393,228 edges.

Social networks. We consider (partial) crawls of the following networks: Slashdot [11], Youtube, Flickr and Orkut [13]. In all datasets, users and friendship relationship between them form the graph structure. The datasets consist of 77,360 nodes, 1,138,499 nodes, 1,715,255 nodes, and 3,072,441 nodes, respectively.

All the algorithms were run on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 8 Mb Cache, 64 Gb RAM, running Redhat Enterprise Linux with 2.5.37 kernel.

Table 1: Approximation Error

Dataset	Sketch	SketchLS	MST	Bidirect.	STAR
Slashdot	12%	4.4%	12.2%	23%	11%
IMDb	10.5%	4.1%	12%	37.9%	6.9%

4.2 Experimental results

4.2.1 Approximation quality

We assess our algorithms by using them for 1000 randomly generated queries per dataset. Each query has 3 to 7 keywords sampled from the graph uniformly at random. In case of smaller graphs (Slashdot, IMDb) we were able to compute an exact Steiner tree T_s using the DPBF algorithm. Then, the relative approximation error is computed as $\text{error}(T_a) = \frac{|T_a|-|T_s|}{|T_s|} = \text{ratio}(T_a) - 1$, where T_a is the size of the approximate Steiner tree returned by different heuristics in use, and $\text{ratio}(T_a)$ is the approximation ratio as defined in Section 3.1.

For bigger datasets, however, the exact algorithm does not scale. The metrics of comparison is the relative difference $\text{diff}(T_a) = \frac{|T_a|-|T_{ls}|}{T_{ls}}$, where T_{ls} stands for the tree yielded by the `SketchLS` algorithm, and T_a is the size of the tree returned by the heuristic in use. Theoretically speaking, this value can be negative. However, as in our experiments the resulting trees returned by the `SketchLS` algorithm are the smallest among competing algorithms, it remains non-negative for all the test cases. The obtained approximation errors are given in Table 1, the relative difference between `SketchLS` and competing algorithms is plotted in Figure 3a.

4.2.2 Time

Another important factor that we measure is the number of nodes visited (touched) by the algorithms and, consequently, their running times. Since Bidirectional and STAR are designed to return top-K results, we set that K parameter to 1 for them, so that only one result is required. We observed that the Local Search done on the original graph G is limited in all cases to exploring of few hundreds of nodes. On the other hand, the *no-index* strategy of Bidirectional, STAR and MST leads to extremely large number of nodes that they visit during the query processing. This, in turn, explains their prohibitedly large running times for our graphs, plotted in Figure 3b using a logarithmic scale on the vertical axis.

As the numbers demonstrate, our algorithms' running times are clearly superior to all existing approaches: the simple `Sketch` algorithm is up to 3 orders of magnitude faster than any other existing algorithm under consideration, the new `SketchLS` algorithm is up to 2 orders of magnitude faster than competitors.

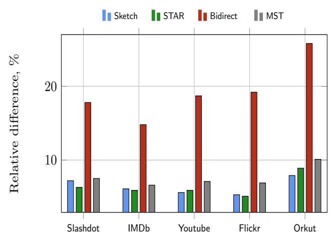
5. CONCLUSIONS

In this paper we present two novel algorithms for the Steiner tree approximation. They are based on the existing work done in the area of shortest path approximation, and combine the indexing with graph search strategies – the idea that has not yet been considered for the Steiner tree problem.

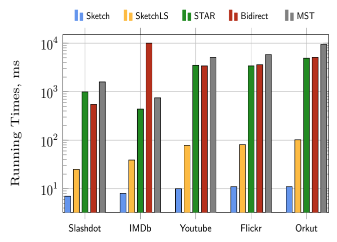
We evaluate the quality and the speed of our approximations by conducting a number of experiments on large real-world graphs. We demonstrate that our heuristics outperform the existing algorithms in quality, and can significantly improve the running times of the keyword-searh query processing.

Figure 3: Evaluation results: (a) Relative difference between `SketchLS` and others, (b) Running time of algorithms

(a) Relative Difference



(b) Running Time



6. REFERENCES

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, pages 431–440.

[2] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A Sketch-Based Distance Oracle for Web-Scale Graphs. In *WSDM'10*, pages 401–410, 2010.

[3] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE'07*, pages 836–845, 2007.

[4] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[5] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM'10*, pages 499–508, 2010.

[6] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD'07*, pages 305–316, 2007.

[7] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB'05*.

[8] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[9] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-tree approximation in relationship graphs. In *ICDE'09*, pages 868–879, 2009.

[10] L. T. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Inf.*, 15:141–145, 1981.

[11] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting Positive and Negative Links in Online Social Networks. In *WWW'10*, pages 641–650, 2010.

[12] 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'08*.

[13] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM'07*.

[14] M. Thorup and U. Zwick. Approximate Distance Oracles. In *STOC'01*.

[15] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.