

# Bonsai: Growing Interesting Small Trees

Stephan Seufert\*   Srikanta Bedathur\*   Julián Mestre†   Gerhard Weikum\*

\*Max-Planck-Institute for Informatics, Saarbrücken, Germany  
{sseufert|bedathur|weikum}@mpi-inf.mpg.de

†School of Information Technologies, University of Sydney, Sydney, Australia

**Abstract**—Graphs are increasingly used to model a variety of loosely structured data such as biological or social networks and entity-relationships. Given this profusion of large-scale graph data, efficiently discovering interesting substructures buried within is essential. These substructures are typically used in determining subsequent actions, such as conducting visual analytics by humans or designing expensive biomedical experiments. In such settings, it is often desirable to constrain the size of the discovered results in order to directly control the associated costs.

In this paper, we address the problem of finding *cardinality-constrained* connected subtrees in large node-weighted graphs that maximize the sum of weights of selected nodes. We provide an efficient constant-factor approximation algorithm for this strongly NP-hard problem. Our techniques can be applied in a wide variety of application settings, for example in differential analysis of graphs, a problem that frequently arises in bioinformatics but also has applications on the web.

## I. INTRODUCTION

Given a large graph with weighted nodes, how can we efficiently identify a heavy connected subtree within a given size? When each node exhibits an individual interestingness factor, how can we find small but highly interesting subtrees?

The problem of discovering interesting subgraphs from large graphs has for long attracted the attention of researchers from different streams. A variety of measures are used for determining the interestingness of a subgraph, ranging from the sum of scores of selected nodes/edges [15], [16], [24], edge density [13], [17], or the frequency of its (isomorphism class) occurrence in the larger graph [18]. It is striking to see that the proposed methods don't offer any support to directly control the size of the discovered subgraph. As a consequence, the results can be extremely large or their size can vary as an arbitrary function of the parameters of the algorithm. There are many application settings where it is not enough to identify heavy or interesting subgraphs, but it is also essential to keep their size small. A well-known application of this problem arises in the field of bioinformatics [1], [9]. In this setting, we are given the protein-protein interaction (PPI) network of an organism, where each node is annotated with a score signifying its deviation from normal behavior in response to a disease. In order to unearth the biological processes involved and thus aid the design of targeted drugs, it is important to identify not only subnetworks with high score, but also to limit their size so that costs of biomedical trials are kept manageably low.

Similar needs arise in visual analytics applications of large-scale graphs. Due to varying visual fatigue levels (either due to individuals or the device used), it is important to enable users to explicitly control the size of the output graph they are comfortable with for navigation. While substantial progress has been made in visual exploration of large graphs [21], [25], such a control still is not in the hands of the users.

In this paper, we take first steps towards efficiently addressing these

requirements in graph mining. Specifically, we consider solving the following computationally hard problem: Given a large undirected graph, where a weight indicating individual score/relevance is associated with every vertex, identify a maximum-weight connected set of nodes whose size is upper-bounded by a user-specified threshold  $k$ . This set of nodes corresponds to a subtree of  $k$  nodes with maximal weight.

Our main contribution is an efficient constant-factor approximation algorithm for this strongly NP-hard problem. For any given cardinality  $k$ , our algorithm is guaranteed to discover a subtree spanning at most  $k$  vertices that sum to a weight of at least  $\frac{1}{5(1+\epsilon)}$  times the weight of the optimal subtree of this size.

The remainder of this paper is organized as follows: In the next section, we lay out the formal framework for our algorithm and show its relation to another well-known graph mining problem. In Section III we explain our algorithm in detail. In Section IV we provide an experimental evaluation on synthetic and real-world graphs. We address implementation issues in Section V and conclude in Section VI.

## II. PRELIMINARIES

For a given graph  $G = (V, E)$  let  $\mathcal{T}(G)$  denote the set of subtrees of  $G$ . For any integer  $k$ , let  $\mathcal{T}_k(G)$  denote the set of subtrees of  $G$  spanning not more than  $k$  vertices:

$$\mathcal{T}_k(G) := \{T = (V_T, E_T) \in \mathcal{T}(G) \mid |V_T| \leq k\}.$$

Let  $f$  be a function defined on a set  $S$ . By abuse of notation, let  $f(X) := \sum_{x \in X} f(x)$  for a subset  $X \subseteq S$ .

### A. Cardinality-Constrained Weighted Trees

We address the following combinatorial optimization problem in the remainder of this paper:

#### Problem 1: Node-Weighted $k$ -Cardinality Tree (KCT)

Given Undirected graph  $G = (V, E)$ , a non-negative weight function defined on the vertices,  $w : V \rightarrow \mathbb{R}_{\geq 0}$ , and a cardinality  $k \in \mathbb{N}$ .

Goal Identify a subtree  $T = (V_T, E_T)$  of  $G$  with the maximum sum of node weights that satisfies the cardinality constraint  $|V_T| \leq k$ :

$$T := \arg \max_{T \in \mathcal{T}_k(G)} w(V_T). \quad (1)$$

This problem was proven strongly NP-hard by Fischetti et al., using a reduction to the node-weighted Steiner tree problem [12].

Although a large body of literature exists for similar problems (like the variant of KCT with edge costs instead of node weights), the node-weighted KCT problem has not received much attention yet. The existing algorithms rely on:

- (meta-)heuristics that do not provide any guarantees, such as Tabu Search and Genetic Algorithms [4], Variable Neighborhood Search [5], and Ant-Colony Optimization [3],
- Integer Programming via branch-and-bound to obtain exact solutions, however at the expense of worst-case exponential running time, or
- reduction to the related  $k$ -MST problem [20].

### B. Prize-Collecting Steiner Trees

As a subroutine, our algorithm solves carefully-chosen instances of the Prize-Collecting Steiner Tree problem (PCST):

#### Problem 2: Prize-Collecting Steiner Tree (PCST)

Given Undirected graph  $G = (V, E)$ , a non-negative cost function defined on the edges,  $c : E \rightarrow \mathbb{R}_{\geq 0}$ , and a non-negative penalty function defined on the vertices:  $\pi : V \rightarrow \mathbb{R}_{\geq 0}$ .

Goal Identify a subtree  $T = (V_T, E_T)$  of  $G$ , minimizing the sum of costs of the included edges and the penalties of the vertices not included:

$$T := \arg \min_{T \in \mathcal{T}(G)} c(E_T) + \pi(V \setminus V_T). \quad (2)$$

Note that this problem does not include a constraint on the size of  $T$ , rather we assign a penalty for the nodes that are not spanned. The PCST problem, which is known to be NP-hard [22], has been studied intensively in the literature because many real-world problems – like utility network design – can be expressed in its terms. Several good approximation algorithms are known for the PCST. In their seminal work, Goemans and Williamson [14] propose an  $\mathcal{O}(n^3 \log(n))$  clustering algorithm that guarantees an approximation ratio of  $2 - \frac{1}{n-1}$ :

*Theorem 1 (Goemans and Williamson):* There is a polynomial-time algorithm that, given an instance  $(G, c, \pi)$  of PCST, returns a tree  $T \in \mathcal{T}(G)$  such that

$$c(E_T) + 2\pi(V \setminus V_T) \leq 2 \min_{S \in \mathcal{T}(G)} \{c(E_S) + \pi(V \setminus V_S)\}. \quad (3)$$

In this section we give a brief description of the approximation technique of Goemans and Williamson for the PCST, as described in [20]. The algorithm contains two stages: a growth and a pruning phase. In the growth phase, initially every vertex forms a singleton component (cluster). Every component is assigned a growth potential that corresponds to the sum of penalties of all vertices included in the component. A component is called active if it has positive remaining potential and passive otherwise. Additionally, we maintain a residual value  $r(e)$  for every edge  $e$ , that initially corresponds to the edge cost. The active components grow uniformly over time, meaning that for each time increment  $\delta$ , the potential of each active component is reduced by  $\delta$ . At the same time, the residual value of an edge with one active endpoint component is reduced by  $\delta$ , the residual value of an edge with two active endpoint components is reduced by  $2\delta$ .

This growth procedure continues until either

- the potential of an active component reduces to 0 or
- the residual value  $r(e)$  of an edge  $e$  reduces to 0.

In the former case, the endpoint is marked as inactive. In the latter case (we call the edge  $e$  *tight*), we merge both endpoint components of the edge into a new component. The potential of the newly formed component corresponds to the sum of potentials of its constituent two components. The growth phase continues until there are no more active components. The output of the procedure is the set of tight edges (which corresponds to a forest in the graph). In Goemans and Williamson’s algorithm, the growth phase is followed by a pruning phase. As this pruning step is not part of our algorithm, we omit its description and refer to the literature [14], [20].

It is worth noting that in their original paper, Goemans and Williamson reduce PCST to a rooted variant where we are given a designated vertex  $r$ , the root, which must be spanned by the output subtree. In order to obtain the algorithm in Theorem 1 one just runs the algorithm for the rooted version on each possible choice of  $r$ . However, due to the large size of the problem instances, this guessing step would be prohibitively slow for our purposes. Therefore, in our implementation, we use the algorithm of Johnson et al. [20], which is also based on the original work of Goemans and Williamson but works in the unrooted setting and offers an approximation guarantee of 2 while avoiding the guessing step altogether. The time complexity of this algorithm is  $\mathcal{O}(n^2 \log(n))$ . We will denote by  $\text{UnrootedGrowth}(G, c, \pi)$  the output of this algorithm on the PCST instance  $(G, c, \pi)$ .

### III. ALGORITHM

In this section we formally describe our algorithm. Our approach is to solve a number of carefully constructed PCST instances. We will use implicitly the framework of Lagrangian relaxation for approximation algorithms introduced by Jian and Vazirani [19] for location problems and by Chudak et al. [7] for Steiner tree problems. However, we only describe the parts relevant to our analysis. More specifically, we avoid introducing the underlying linear program and its Lagrangian relaxation.

#### A. Main Idea

The key of our algorithm is to construct and solve instances of the PCST problem in such a way that we can guarantee a constant-factor approximation to our original KCT problem.

Throughout the rest of this section, we denote by  $\text{OPT}$  the weight of the optimal solution to the KCT instance at hand. The problem is somewhat easier to solve if  $\text{OPT}$  is known beforehand, therefore we assume for now that the value is known. Indeed this will not be the case in our applications but we can easily guess its value up to an  $\epsilon$  multiplicative error using binary search, as we will show in section III-C. Next, we describe the algorithm in detail.

#### B. Basic Algorithm

Given an instance of KCT and the value  $\text{OPT}$ , we derive several instances of the PCST problem. For this purpose, we identify the node weights with penalties and set the cost of every edge in the graph to 1. By scaling these node penalties (that is, multiplying them with a factor  $\lambda \in \mathbb{R}_{>0}$ ), we can indirectly control the size of the output solution. For instance, if we use a multiplier  $\lambda_1 = 0$ , the optimal solution of the associated PCST instance is given by the empty tree, whereas for a sufficiently large factor, e.g.  $\lambda_2 > n \max_{e \in E} c(e)$ , the optimum is any spanning tree of the graph.

**Algorithm 1: HeavySubtree( $G, w, k, \text{OPT}$ )**


---

**Data:** Graph  $G = (V, E)$ , weight function  $w : V \rightarrow \mathbb{R}_{\geq 0}$ , cardinality  $k \in \mathbb{N}$

```

1 begin
2    $[\lambda_1, \lambda_2] \leftarrow [0, n / (w(V) - \text{OPT})]$   $\triangleright$  initial penalty interval
3    $T_1 \leftarrow (\arg \max_{v \in V} w(v), \emptyset)$   $\triangleright$  tree of heaviest node
4    $T_2 \leftarrow \text{SpanningTree}(G)$   $\triangleright$  any spanning tree of  $G$ 
5   while  $\lambda_2 - \lambda_1 \geq \frac{1}{w(V) - \text{OPT}}$  do
6      $\lambda \leftarrow \frac{\lambda_1 + \lambda_2}{2}$ 
7      $T \leftarrow \text{UnrootedGrowth}(G, \mathbf{1}, \lambda w)$   $\triangleright$  solve PCST with unit edge costs and  $\lambda$ -scaled node penalties
8     if  $w(V_T) \leq \text{OPT}$  then
9        $(\lambda_1, T_1) \leftarrow (\lambda, T)$ 
10    else
11       $(\lambda_2, T_2) \leftarrow (\lambda, T)$ 
12   $T_1 \leftarrow \text{TreeDP}(T_1, w, k)$ 
13   $T_2 \leftarrow \text{TreeDP}(T_2, w, k)$ 
14   $T \leftarrow \arg \max_{w(V)} \{T_1, T_2\}$   $\triangleright$  heaviest subtree of  $T_1$  or  $T_2$  having  $k$  vertices
15  return  $T$ 

```

---

We will use the existing algorithm of Johnson et al. [20] for the PCST (which provides a 2-approximation [11]) to obtain a tree that has a weight of at least  $\text{OPT}$  and is as small as possible.

The idea is to perform binary search over the range of scale factors  $\lambda \in [\lambda_1, \lambda_2]$ . At each step of this binary search procedure, we solve the PCST instance using the  $\lambda$ -scaled penalties. If the returned tree has weight of at least  $\text{OPT}$ , we decrease  $\lambda$ , thus requesting a smaller tree in the next iteration. If the returned tree has weight less than  $\text{OPT}$ , we increase  $\lambda$ , thus allowing for a larger output solution in the next run. This binary search procedure is continued until the final interval is sufficiently small.

As the solution for the original KCT problem, we finally extract the heaviest subtree spanning  $k$  vertices from the tree obtained in the last solved PCST instance. For this purpose, we use a dynamic programming procedure called  $\text{TreeDP}(T, w, k)$ , consisting of the algorithm by Blum [2]. The complete procedure is described in Algorithm 1.

The following theorem ensures the quality of the returned solution:

**Theorem 2:** Given an instance  $(G, w, k, \text{OPT})$  of the KCT problem, the algorithm  $\text{HeavySubtree}$  returns a tree  $T$  of at most  $k$  vertices such that  $w(V_T) \geq \frac{\text{OPT}}{5}$ .

The proof of theorem 2 is omitted due to space constraints. The interested reader is referred to our technical report for details [23].

### C. Guessing the Optimal Weight

The last remaining issue is efficiently guessing  $\text{OPT}$ , the weight of the optimal solution. Let  $w^*$  denote the maximum weight of a node in the graph,  $w^* := \max_{v \in V} w(v)$ . The value  $\text{OPT}$  must then be contained in the interval  $[w^*, kw^*]$ .

Our algorithm performs binary search over this interval. We introduce an additional parameter  $\epsilon > 0$  to our algorithm and terminate the binary search when the final interval  $[w_1, w_2]$  satisfies  $w_2 - w_1 \leq \epsilon w^*$ .

**Algorithm 2: Bonsai( $G, w, k, \epsilon$ )**


---

**Data:** Graph  $G = (V, E)$ , weight function  $w : V \rightarrow \mathbb{R}_{\geq 0}$ , cardinality  $k \in \mathbb{N}$ , error bound  $0 < \epsilon < 1$

```

1 begin
2    $[w_1, w_2] \leftarrow [w^*, kw^*]$   $\triangleright$  initial interval for OPT
3   while  $w_2 - w_1 \geq \epsilon w^*$  do
4      $\text{OPT}_\gamma \leftarrow \frac{w_2 + w_1}{2}$   $\triangleright$  guessed value for OPT
5      $T \leftarrow \text{HeavySubtree}(G, w, k, \text{OPT}_\gamma)$ 
6     if  $w(T) \geq \frac{\text{OPT}_\gamma}{5}$  then
7        $w_1 \leftarrow \text{OPT}_\gamma$ 
8     else
9        $w_2 \leftarrow \text{OPT}_\gamma$ 
10  return  $T$ 

```

---

We then run our algorithm a total of

$$\left\lceil \log \left( \frac{kw^* - w^*}{\epsilon w^*} \right) \right\rceil \leq \left\lceil \log \left( \frac{k}{\epsilon} \right) \right\rceil$$

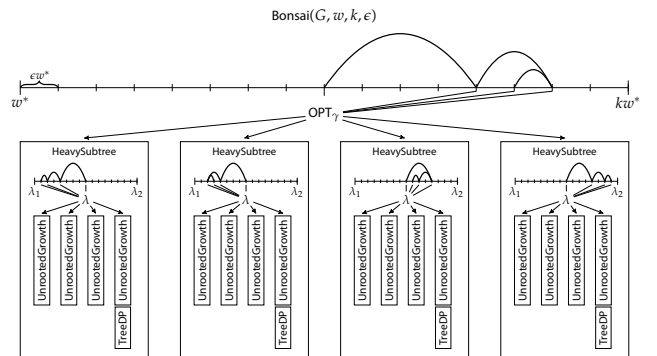
times, thus achieving independence from graph properties like the number of nodes/edges and the maximum node weight  $w^*$ . Using this termination criterion will ensure that the last guessed value,  $\text{OPT}_\gamma$ , differs in the worst-case by a factor of  $\frac{1}{1+\epsilon}$  from the true optimum:

$$\text{OPT}_\gamma(1 + \epsilon) \geq \text{OPT}_\gamma + \epsilon w^* \geq \text{OPT} \quad (4)$$

for the last guessed optimum value  $\text{OPT}_\gamma$  (line 4 in Algorithm Bonsai) and the true optimum  $\text{OPT}$ . Note that the binary search interval can be narrowed down further, for example by computing the greedy solution of the problem and using its weight,  $w_{\text{greedy}}$  as the lower bound. In fact, we use this improvement in our implementation.

### D. Complete Algorithm

We can now combine the existing parts to obtain the complete procedure, given in Algorithm 2:



**Figure 1:** Execution schema of the algorithm

We guess the weight  $\text{OPT}$  of the optimal solution using the outer binary search procedure. For every guess, we run the  $\text{HeavySubtree}$  procedure which performs the inner binary search for the multiplication parameter  $\lambda$ . In each step of the inner binary search we try to obtain a tree with weight of at least  $\text{OPT}$  that is as small

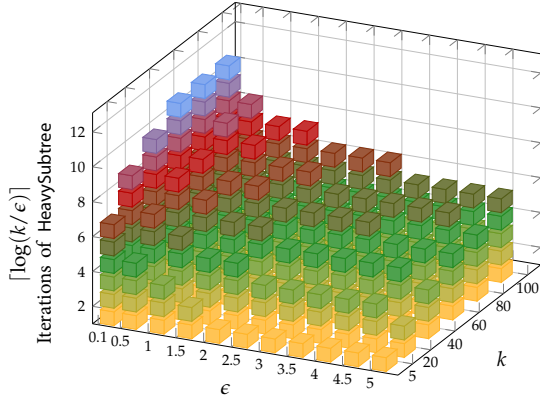


Figure 2: Impact of error bound and cardinality

as possible, getting closer to this goal as the search progresses. For the last obtained trees, we retrieve the heaviest subtree  $T$  that satisfies our cardinality constraint, using the dynamic programming procedure TreeDP. Depending on the weight of tree  $T$  we proceed in the outer binary search procedure, increasing or reducing our guess for the optimum weight until the final interval is small enough. For the resulting tree returned in line 10 of Algorithm 2 we have (using Theorem 2):

$$w(T) \geq \frac{\text{OPT}_\gamma}{5} \stackrel{(4)}{\geq} \frac{\text{OPT}}{5(1+\epsilon)}. \quad (5)$$

A schematic overview of the algorithm is depicted in Figure 1. The impact of the cardinality constraint as well as the error bound on the required number of iterations of HeavySubtree is shown in Figure 2.

#### IV. EXPERIMENTAL EVALUATION

In this section, we provide the experimental evaluation of our algorithm. All experiments were conducted on Dell PowerEdge M610 servers, each of which has two Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and runs Debian GNU/Linux (SMP Kernel 2.6.29.3.1) as an operating system. Experiments were conducted using the Java Hotspot 64-Bit Server Virtual Machine (build 11.2-b01) installed on our servers. Note that our algorithm was implemented as a single-thread.

##### A. Biological Networks

As a first example we present the results of our algorithm for a real-world graph. We run Bonsai on the protein-protein interaction network used by Dittrich et al. [9] for discovering functional modules. The node scores provided in this dataset are real numbers. Therefore, in order to execute our algorithm, we map the scores to non-negative values by adding to each score the minimum score in the network. The graph contains 2034 proteins (nodes) and 8399 interactions (edges).

Table 1 contains the experimental evaluation of this network for different cardinalities  $k$  and error bounds  $\epsilon$ . Note that the implementation of our algorithm returns a first candidate solution after the first execution of the UnrootedGrowth procedure, followed by a call to the TreeDP routine. In the table,  $t_{\text{final}}$  denotes the total running time of the algorithm,  $t_{\text{first}}$  the time to return the first

candidate solution and  $w_{\text{first}}, w_{\text{final}}$  the weight of the first candidate tree and the weight of the final tree respectively.

$k$	$\epsilon$	$t_{\text{first}}$ [s]	$t_{\text{final}}$ [s]	$w_{\text{first}}$	$w_{\text{final}}$	$w_{\text{first}}/w_{\text{final}}$
5	0.1	0.004	10.100	40.9	52.5	0.78
	0.5	0.003	6.503	40.9	52.5	0.78
	1.0	0.004	6.275	40.9	52.5	0.78
20	0.1	0.005	8.389	163.2	185.2	0.88
	0.5	0.005	6.427	163.2	185.2	0.88
	1.0	0.005	7.802	163.2	185.2	0.88
100	0.1	0.009	15.588	642.6	726.0	0.89
	0.5	0.010	11.369	642.6	726.0	0.89
	1.0	0.009	8.218	642.6	726.0	0.89

Table 1: Experimental results for the biological network

##### B. Synthetic Graphs

In the following, we demonstrate the running time and quality of our algorithm for synthetically created graphs. We execute the Bonsai( $G, w, k, \epsilon$ ) algorithm over a wide variety of settings:

- as input graphs we generate power-law random graphs using the R-MAT graph generator [6] (with parameters  $a = 0.45, b = 0.15, c = 0.15, d = 0.25$ ) with  $n \in \{i \cdot 10^3 \mid i = 2, 5, 10, 20\}$  nodes and  $m \in \{4n, 10n, 50n\}$  edges,
- a weight function  $w : V \rightarrow \mathbb{R}_{\geq 0}$ , that assigns power-law distributed values from the interval  $[0, 1]$  to the nodes,
- cardinality constraints  $k \in \{5, 10, 20, 100\}$ , and
- error bound  $\epsilon = 0.5$ .

Figure 3 provides an overview over the resulting running times with an error bound of  $\epsilon = 0.5$  and different graph sizes (nodes, edges) and cardinality values using a logarithmic scale. The lower part of each bar represents the required time for computing the first candidate solution (UnrootedGrowth followed by TreeDP). The full bar represents the total running time for the complete Bonsai algorithm.

The impact of the cardinality  $k$  is negligible in all the problem instances. This is due to the fact that we use the weight of the greedy solution as the lower bound for the value OPT, which is a much tighter bound than the maximal node weight.

In Figure 4 we compare the weight of the first returned candidate with the weight of final solution over a varying number of vertices and edges for different cardinalities  $k$ . It is striking that the difference between the weight of the first candidate tree (lower part of each bar) and the weight of the final output (full bar) is in all the cases very small, although the time to obtain it is almost an order of magnitude lower than the total running time of the algorithm. Note also that on average in all experiments we obtain a solution that is much better than the worst-case approximation guarantee, as – by the design of the experiments – the value OPT is upper-bounded by  $k$ .

#### V. IMPLEMENTATION DETAILS

In this section, we briefly present the details of our implementation of Bonsai and its constituent subroutines. Our algorithms were implemented using Java 1.6. To the best of our knowledge, our algorithm is the *first ever practical implementation* of a constant-factor approximation algorithm for the KCT problem.

Figure 3: Running times (wallclock) for various number of vertices, edges, and cardinality constraints

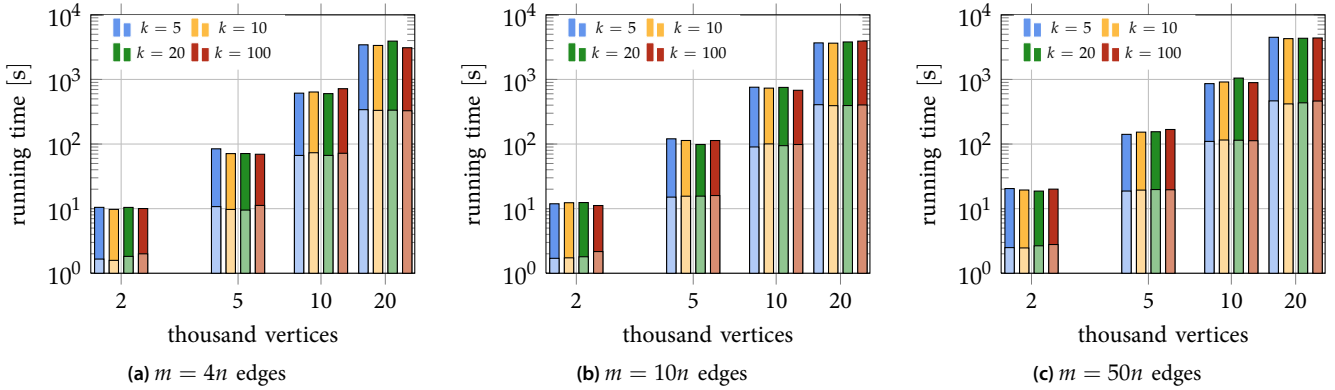
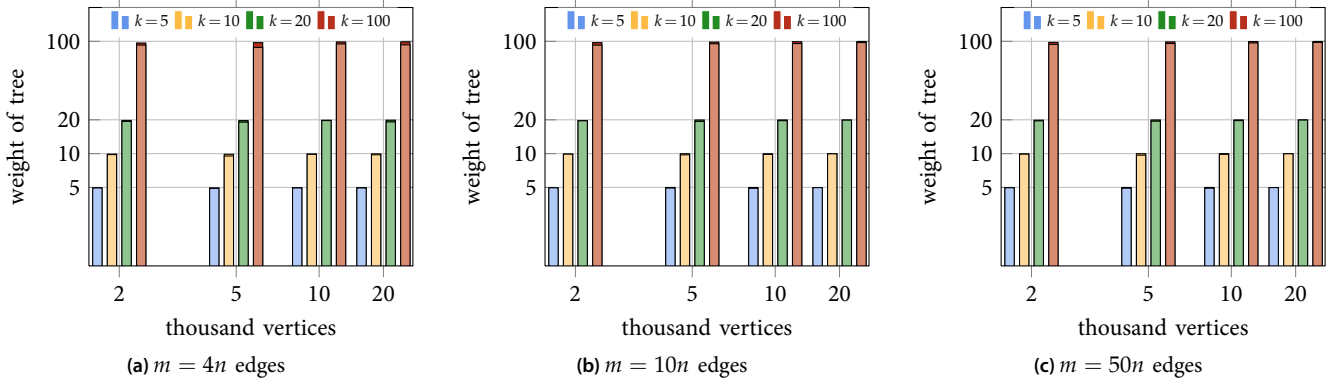


Figure 4: Solution quality for various number of vertices, edges, and cardinality constraints



Our implementation of `UnrootedGrowth` essentially follows the ideas outlined in [20] and [10], and we omit its detailed discussion due to the lack of space. Briefly, the key data structure used during running of `UnrootedGrowth` is a collection of *min-heaps*, each of which corresponds to a component, i. e. a connected set of nodes – either *active* (the cluster of nodes that continues to grow), or *passive* (the cluster of nodes which have stopped growing). We implemented min-heaps as `FibonacciHeaps`, which are known to be superior in performance [8].

Despite a highly optimized implementation of `UnrootedGrowth`, a naive implementation of `Bonsai` exhibits super-linear growth in execution time as we increase the size of the graph – which is not surprising in itself since `UnrootedGrowth` which we call polylogarithmically many times, has complexity in  $\mathcal{O}(n^2 \log(n))$ . As a result, we explored the potential of optimizing the number of probes

The key to optimizing the number of probes made in the inner binary-search lies in keeping track of the best results found in earlier iterations of outer binary-search, and using them to cut down the range of values to be considered as multiplication parameters  $\lambda$ . In other words, after the first complete execution of the `HeavySubtree` routine, we can improve the range of values  $[\lambda_1, \lambda_2]$  that are explored in its subsequent executions. Consider an instance of the `HeavySubtree` routine for one of the choices, say  $\text{OPT}_i$ , of the outer binary search. Also, let the weights of the *final trees*

obtained for already probed choices of `OPT` be  $W_0, W_1, \dots, W_{i-1}$ , and corresponding values of  $\lambda$  be  $l_0, l_1, \dots, l_{i-1}$ . If we keep track of the  $W_i$  and  $l_i$  values, we can tighten the range of values explored from both sides:

- (i) **Improving  $\lambda_1$ .** If the weight of the final tree found in an earlier iteration is *smaller* than the value of  $\text{OPT}_i$  being used in the current execution of `HeavySubtree`, then we know that the best  $\lambda$  we can hope to find presently cannot be *smaller*. That is, if  $\text{OPT}_i \leq W_j$  for some  $0 < j < i$ , then  $l_i$  (i. e., the final value found by the current `HeavySubtree` instance) is strictly lower-bounded by  $l_j$ .
- (ii) **Improving  $\lambda_2$ .** In every iteration of `HeavySubtree`, we can continue to upper-bound the value of  $\lambda_2$  by the best value  $l_j$  found in earlier iteration,  $j$ , where  $\text{OPT}_i \geq W_j$ .

It should be noted that the inner search optimization presented above does not lead to any loss in the quality of results found.

## VI. CONCLUSIONS

In this paper we have provided a practical constant-factor approximation algorithm for the KCT problem, named `Bonsai`. Our algorithm works by reducing KCT to certain instances of the related PCST problem. We have exploited an existing approximation algorithm for this related setting and derived an algorithm with approximation guarantee of  $\frac{1}{5(1+\epsilon)}$  for the KCT problem. Furthermore, we proposed various optimizations to our algorithm that lead

to an implementation that is very flexible and runs reasonably fast on the problem instances considered. Using a mixture of synthetic and real-world graphs we were able to demonstrate the practical viability of our approach. The Bonsai algorithm can return a first candidate solution after the first execution of the PCST subroutine. We have shown empirically that the quality of this first solution is in all considered cases close to the optimum.

As we do not make any assumption on the distribution of the node weights, our algorithm is suitable for a variety of application scenarios. Possibilities include practical applications such as identifying the most-deviant parts of protein-protein interaction networks for designing biomedical trials, as well as others like using weights based on the structural properties of the graph (like node degrees or PageRank values) or interestingness-scores (like activity measures for articles in the Wikipedia graph to extract an active topical core). Interesting future work encompasses running our algorithm for these choices of weights and on graphs from various data sources. Other possible future directions include solving the KCT problem in the presence of both edge and node weights.

#### ACKNOWLEDGEMENTS

This work is supported by DFG (German Research Foundation) within the priority research program 1355 “Scalable Visual Analytics”.

#### REFERENCES

- [1] E. Althaus, G. W. Klau, O. Kohlbacher, H.-P. Lenhof, and K. Reinert. Integer Linear Programming in Computational Biology. In *Efficient Algorithms*, volume 5760/2009 of *Lecture Notes in Computer Science*, pages 199–218. Springer, 2009.
- [2] C. Blum. Revisiting Dynamic Programming for Finding Optimal Subtrees in Trees. *European Journal of Operational Research*, 177(1):102–114, 2007.
- [3] C. Blum and M. J. Blesa. *Optimization Techniques for Solving Complex Problems*, chapter 23: Solving the KCT Problem: Large-Scale Neighborhood Search and Solution Merging, pages 407–421. Wiley Series on Parallel and Distributed Computing. Wiley, 2009.
- [4] C. Blum and M. Ehrgott. Local Search Algorithms for the  $k$ -Cardinality Tree Problem. *Discrete Applied Mathematics*, 128(2-3):511–540, 2003.
- [5] J. Brimberg, D. Urošević, and N. Mladenović. Variable Neighborhood Search for the Vertex Weighted  $k$ -Cardinality Tree Problem. *European Journal of Operational Research*, 171(1):74–84, 2006.
- [6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM’04: Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.
- [7] F. A. Chudak, T. Roughgarden, and D. P. Williamson. Approximate  $k$ -MSTs and  $k$ -Steiner Trees via the Primal-Dual Method and Lagrangean Relaxation. *Mathematical Programming*, 100(2):411–421, 2004.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.
- [9] M. T. Dittrich, G. W. Klau, A. Rosenwald, T. Dandekar, and T. Müller. Identifying Functional Modules in Protein-Protein Interaction Networks: An Integrated Exact Approach. *Bioinformatics*, 24(13):i223–i231, 2008.
- [10] P. Feofiloff, C. G. Fernandes, C. E. Ferreira, and J. C. de Pina.  $\mathcal{O}(n^2 \log(n))$  Implementation of an Approximation for the Prize-Collecting Steiner Tree Problem. 2002.
- [11] P. Feofiloff, C. G. Fernandes, C. E. Ferreira, and J. C. de Pina. Primal-Dual Approximation Algorithms for the Prize-Collecting Steiner Tree Problem. *Information Processing Letters*, 103(5):195–202, 2007.
- [12] M. Fischetti, H. W. Hamacher, K. Jørnsten, and F. Maffioli. Weighted  $k$ -Cardinality Trees: Complexity and Polyhedral Structure. *Networks*, 24:11–21, 1994.
- [13] D. Gibson, R. Kumar, and A. Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. In *VLDB’05: Proceedings of the 31st Intl. Conference on Very Large Data Bases*, pages 721–732. VLDB Endowment, 2005.
- [14] M. X. Goemans and D. P. Williamson. A General Approximation Technique for Constrained Forest Problems. In *SODA ’92: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 307–316, 1992.
- [15] H. He and A. K. Singh. Efficient Algorithms for Mining Significant Substructures in Graphs with Quality Guarantees. In *ICDM ’07: Proceedings of the 7th IEEE Conference on Data Mining*, pages 163–172, Washington, DC, USA, 2007. IEEE Computer Society.
- [16] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *SIGMOD’07: Proceedings of the 2007 ACM SIGMOD Intl. Conference on Management of Data*, pages 305–316, New York, NY, USA, 2007. ACM.
- [17] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining Coherent Dense Subgraphs Across Massive Biological Networks for Functional Discovery. *Bioinformatics*, 21(S1):213–221, 2005.
- [18] J. Huan, W. Wang, and J. Prins. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In *ICDM ’03: Proceedings of the 3rd IEEE Conference on Data Mining*. IEEE Computer Society, 2003.
- [19] K. Jain and V. V. Vazirani. Approximation Algorithms for Metric Facility Location and  $k$ -Median Problems using the Primal-Dual Schema and Lagrangian Relaxation. *Journal of the ACM*, 48(2):274–296, 2001.
- [20] D. S. Johnson, M. Minkoff, and S. Phillips. The Prize Collecting Steiner Tree Problem: Theory and Practice. In *SODA ’00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 760–769, 2000.
- [21] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph Summarization with Bounded Error. In *SIGMOD’08: Proceedings of the 2008 ACM SIGMOD Intl. Conference on Management of Data*, pages 419–432. ACM, 2008.
- [22] A. Segev. The Node-Weighted Steiner Tree Problem. *Networks*, 17:1–17, 1987.
- [23] S. Seufert, S. Bedathur, J. Mestre, and G. Weikum. Bonsai: Growing Interesting Small Trees. Technical Report MPI-I-2010-5-005, Max-Planck-Institute for Informatics, October 2010.
- [24] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top- $k$  Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. In *ICDE’09: Proceedings of the 2009 IEEE Intl. Conference on Data Engineering*, pages 405–416, Washington, DC, USA, 2009. IEEE Computer Society.
- [25] N. Wang, S. Parthasarathy, K.-L. Tan, and A. K. H. Tung. CSV: Visualizing and Mining Cohesive Subgraphs. In *SIGMOD’08: Proceedings of the 2008 ACM SIGMOD Intl. Conference on Management of Data*, pages 445–458, New York, NY, USA, 2008.