Algorithmica (2021) 83:2427–2468
https://doi.org/10.1007/s00453-021-00836-5

The Maximum Binary Tree Problem

Karthekeyan Chandrasekaran[1] · **Elena Grigorescu**[2] · **Gabriel Istrate**[3] · **Shubhang Kulkarni**[1] · **Young-San Lin**[2] · **Minshen Zhu**[2]

Received: 10 October 2020 / Accepted: 15 May 2021 / Published online: 31 May 2021

Abstract

We introduce and investigate the approximability of the *maximum binary tree problem* (MBT) in directed and undirected graphs. The goal in MBT is to find a maximum-sized binary tree in a given graph. MBT is a natural variant of the well-studied longest path problem, since both can be viewed as finding a maximum-sized tree of bounded degree in a given graph. The connection to longest path motivates the study of MBT in directed acyclic graphs (DAGs), since the longest path problem is solvable efficiently in DAGs. In contrast, we show that MBT in DAGs is hard: it has no efficient $\exp(-O(\log n/\log\log n))$-approximation under the exponential time hypothesis, where n is the number of vertices in the input graph. In undirected graphs, we show that MBT has no efficient $\exp(-O(\log^{0.63} n))$-approximation under the exponential time hypothesis. Our inapproximability results rely on self-improving reductions and structural properties of binary trees. We also show constant-factor inapproximability assuming $\mathbf{P} \neq \mathbf{NP}$. In addition to inapproximability results, we present algorithmic results along two different flavors: (1) We design a randomized algorithm to verify if a given directed graph on n vertices contains a binary tree of size k in $2^k\mathsf{poly}(n)$ time. (2) Motivated by the longest heapable subsequence problem, introduced by Byers, Heeringa, Mitzenmacher, and Zervas, *ANALCO 2011*, which is equivalent to MBT in *permutation DAGs*, we design efficient algorithms for MBT in bipartite permutation graphs.

Keywords Maximum binary tree · Inapproximability · Fixed-parameter tractability

✉ Karthekeyan Chandrasekaran
karthe@illinois.edu

Extended author information available on the last page of the article

1 Introduction

A general degree-constrained subgraph problem asks for an optimal subgraph of a given graph with specified properties while also satisfying degree constraints on all vertices. Degree-constrained subgraph problems have numerous applications in the field of network design and consequently, have been studied extensively in the algorithms and approximation literature [1, 15–17, 30, 34, 35]. In this work, we introduce and study the *maximum binary tree problem* in directed and undirected graphs. In the maximum binary tree problem (MBT), we are given an input graph G and the goal is to find a *binary tree* in G with maximum number of vertices.

Our first motivation for studying MBT arises from the viewpoint that it is a variant of the longest path problem: In the longest path problem, the goal is to find a maximum-sized tree in which every vertex has degree at most 2. In MBT, the goal is to find a maximum-sized tree in which every vertex has degree at most 3. Certainly, one may generalize both these problems to finding a *maximum-sized degree-constrained tree* in a given graph. In this work we focus on binary trees; however, all our results extend to the maximum-sized degree-constrained tree problem for *constant* degree bound.

Our second motivation for studying MBT is its connection to the *longest heapable subsequence problem* introduced by Byers et al. [11]. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be a permutation on n elements. Byers et al. define a subsequence (not necessarily contiguous) of σ to be *heapable* if the elements of the subsequence can be sequentially inserted to form a binary *min-heap* data structure. Namely, insertions subsequent to the first element, which takes the root position, happen below previously placed elements. The longest heapable subsequence problem asks for a maximum-length heapable subsequence of a given sequence. This generalizes the well-known longest increasing subsequence problem. Porfilio [33] showed that the longest heapable subsequence problem is equivalent to MBT in permutation directed acyclic graphs (abbreviated *permutation DAGs*): a permutation DAG associated with the sequence σ is obtained by introducing a vertex u_i for every sequence element σ_i, and arcs (u_i, u_j) for every pair (i, j) such that $i > j$ and $\sigma_i \geq \sigma_j$. On the other hand, for sequences of intervals the maximum binary problem is easily solvable by a greedy algorithm [6] (see also [22] for further results and open problems on the heapability of partial orders). These results motivate the study of MBT in restricted graph families.

We now formally define MBT in undirected graphs, which we denote as UndirMaxBinaryTree. A *binary tree* of an *undirected* graph G is a subgraph T of G that is connected and acyclic with the degree of u in T being at most 3 for every vertex u in T. In UndirMaxBinaryTree, the input is an undirected graph G and the goal is to find a binary tree in G with maximum number of vertices. In the rooted variant of this problem, the input is an undirected graph G along with a specified root vertex r and the goal is to find a binary tree containing r in G with maximum number of vertices such that the degree of r in the tree is at most 2. We focus on the unrooted variant of the problem and mention that it reduces to the rooted variant. We emphasize that a binary tree T of G is not necessarily spanning (i.e., may not contain all

vertices of the given graph). The problem of verifying whether a given undirected graph has a *spanning* binary tree is **NP**-complete. This follows by a reduction from the Hamiltonian path problem: Given an undirected graph $G = (V, E)$, create a pendant vertex v' adjacent to v for every vertex $v \in V$. The resulting graph has a spanning binary tree if and only if G has a Hamiltonian path.

Next, we formally define MBT in directed graphs. A *tree of a directed graph G* is a subgraph T of G such that T is acyclic and has a unique vertex, termed as the root, with the property that every vertex v in T has a unique directed path *to the root* in T. A *binary tree of a directed graph G* is a tree T such that the incoming-degree of every vertex u in T is at most 2 while the outgoing-degree of every vertex u in T is at most 1. In the rooted variant of the maximum binary tree problem for directed graphs, the input is a directed graph G along with a specified root r and the goal is to find an r-rooted binary tree T in G with maximum number of vertices. The problem of verifying whether a given directed graph has a *spanning* binary tree is **NP**-complete (by a similar reduction as that for undirected graphs).

The connection to the longest path problem as well as the longest heapable subsequence problem motivates the study of the maximum binary tree problem in directed acyclic graphs (DAGs). In contrast to directed graphs, the longest path problem in DAGs can be solved in polynomial-time (e.g., using dynamic programming or LP-based techniques). Moreover, verifying whether a given DAG contains a spanning binary tree is solvable in polynomial-time using the following characterization: a given DAG on vertex set V contains a spanning binary tree if and only if the partition matroid corresponding to the in-degree of every vertex being at most two and the partition matroid corresponding to the out-degree of every vertex being at most one have a common independent set of size $|V| - 1$. These observations raise the intriguing possibility of solving the maximum binary tree problem in DAGs in polynomial-time. For this reason, we focus on DAGs within the family of directed graphs in this work. We denote the maximum binary tree problem in DAGs as DAGMaxBinaryTree.

The rooted and the unrooted variants of the maximum binary tree problem in DAGs are polynomial-time equivalent by simple transformations. Indeed, the unrooted variant can be solved by solving the rooted variant for every choice of the root. To see the other direction, suppose we would like to find a maximum r-rooted binary tree in a given DAG $G = (V, E)$. Then, we discard from G all outgoing arcs from r and all vertices that cannot reach r (i.e., we consider the sub-DAG induced by the descendents of r) and find an unrooted maximum binary tree in the resulting DAG. If this binary tree is rooted at a vertex $r' \neq r$, then it can be extended to an r-rooted binary tree by including an arbitrary $r' \to r$ path P—since the graph is a DAG, any such path P will not visit a vertex that is already in the tree (apart from r'). The equivalence is also approximation preserving. For this reason, we only study the rooted variant of the problem in DAGs.

We present inapproximability results for MBT in DAGs and undirected graphs. On the algorithmic side, we show that MBT in directed graphs is fixed-parameter tractable when parameterized by the solution size. We observe that the equivalence of the longest heapable subsequence to MBT in permutation DAGs motivates the study of MBT even in restricted graph families. As a first step towards

understanding MBT in permutation DAGs, we design an algorithm for bipartite permutation graphs. We use a variety of tools including self-improving and gadget reductions for our inapproximability results, and algebraic and structural techniques for our algorithmic results.

1.1 Related Work

Degree-constrained subgraph problems appeared as early as 1978 in the textbook of Garey and Johnson [18] and have garnered plenty of attention in the approximation community [1, 15–17, 24, 30, 34, 35]. A rich line of works have addressed the minimum degree spanning tree problem as well as the minimum cost degree-constrained spanning tree problem leading to powerful rounding techniques and a deep understanding of the spanning tree polytope [12, 13, 16, 19, 26, 29, 35]. Approximation and bicriteria approximation algorithms for the counterparts of these problems in directed graphs, namely degree-constrained arborescence and min-cost degree-constrained arborescence, have also been studied in the literature [7].

In the *maximum-edge* degree-constrained connected subgraph problem, the goal is to find a connected degree-constrained subgraph of a given graph with maximum number of edges. This problem does not admit a PTAS [3] and has been studied from the perspective of fixed-parameter tractability [4]. MBT could be viewed as a *maximum-vertex* degree-constrained connected subgraph problem, where the goal is to maximize the number of *vertices* as opposed to the number of *edges*—the degree-constrained connected subgraph maximizing the number of vertices may be assumed to be acyclic and hence, a tree. It is believed that the *connectivity constraint* makes the maximum-edge degree-constrained connected subgraph problem to become extremely difficult to approximate. Our results formalize this belief when the objective is to maximize the number of vertices.

Switching the objective with the constraint in the maximum-vertex degree-constrained connected subgraph problem leads to the minimum-degree k-tree problem: here the goal is to find a minimum degree subgraph that is a tree with at least k vertices. Minimum degree k-tree admits a $O(\sqrt{(k/\Delta^*)\log k})$-approximation, where Δ^* is the optimal degree and does not admit a $o(\log n)$-approximation [24]. We note that the hardness reduction here (from set cover) crucially requires the optimal solution value Δ^* to grow with the number n of vertices in the input instance, and hence, does not imply any hardness result for input instances in which Δ^* is a constant. Moreover, the approximation result implies that a tree of degree $O(\sqrt{k \log k})$ containing k vertices can be found in polynomial time if the input graph contains a constant-degree tree with k vertices.

We consider the maximum binary tree problem to be a generalization of the longest path problem as both can be viewed as asking for a maximum-sized degree-constrained connected acyclic subgraph. The longest path problem in undirected graphs admits an $\Omega\big((\log n/\log\log n)^2/n\big)$-approximation [9], but it is APX-hard and does not admit a $2^{-O(\log^{1-\epsilon} n)}$-approximation for any constant $\epsilon > 0$ unless $\mathbf{NP} \subseteq \mathbf{DTIME}\left(2^{\log^{O(1/\epsilon)} n}\right)$ [23]. Our hardness results for the max binary

tree problem in undirected graphs bolsters this connection. The longest path problem in directed graphs is much harder: For every $\varepsilon > 0$ it cannot be approximated to within a factor of $1/n^{1-\varepsilon}$ unless $\mathbf{P} = \mathbf{NP}$, and it cannot be approximated to within a factor of $(\log^{2+\epsilon} n)/n$ under the Exponential Time Hypothesis [9]. However, the longest path problem in DAGs is solvable in polynomial time. Our hardness results for the max binary tree problem in DAGs are in stark contrast to the polynomial-time solvability of the longest path problem in DAGs.

On the algorithmic side, the color-coding technique introduced by Alon et al. [2] can be used to decide whether an undirected graph $G = (V, E)$ contains a copy of a bounded treewidth pattern graph $H = (V_H, E_H)$ where $|V_H| = O(\log |V|)$, and if so, then find one in polynomial time. The idea here is to randomly color the vertices of G by $O(\log |V|)$ colors and to find a maximum colorful copy of H using dynamic programming. We note that the same dynamic programming approach can be modified to find a maximum colorful binary tree. This algorithm can be derandomized, thus leading to a deterministic $\Omega((1/n) \log n)$-approximation to UndirMaxBinaryTree.

In parameterized complexity, designing algorithms with running time $\beta^k \mathsf{poly}(n)$ ($\beta > 1$ is a constant) for problems like k-Path and k-Tree is a central topic. For k-Path, the color-coding technique mentioned above already implies a $(2e)^k \mathsf{poly}(n)$-time algorithm. Koutis [27] noticed that k-Path can be reduced to detecting whether a given polynomial contains a multilinear term. Using algebraic methods for the latter problem, Koutis obtained a $2^{1.5k} \mathsf{poly}(n)$ time algorithm for k-Path. This was later improved by Williams [39] to $2^k \mathsf{poly}(n)$. The current state-of-art algorithm is due to Björklund et al. [8], which is also an algebraic algorithm with running time $1.66^k \mathsf{poly}(n)$. All of these algorithms are randomized. Our study of the k-BinaryTree problem, which is the problem of deciding whether a given graph G contains a binary tree of size at least k, is inspired by this line of results. Our construction of the polynomial employs the idea of branching walk first introduced by Nederlof [31]. This idea has also been used in other graph connectivity problems [10, 20].

Several **NP**-hard problems are known to be solvable in specific families of graphs. Bipartite permutation graphs is one such family which is known to exhibit this behaviour [25, 36–38]. Our polynomial-time solvability result for these families of graphs crucially identifies the existence of structured optimal solutions to reduce the search space and solves the problem over this reduced search space.

1.2 Our Contributions

1.2.1 Inapproximability Results

Directed Graphs We first focus on directed graphs and in particular, on directed acyclic graphs. It is well-known that the longest path problem in DAGs is solvable in polynomial-time. In contrast, we show that DAGMaxBinaryTree does not even admit a constant-factor approximation. Furthermore, if DAGMaxBinaryTree

admitted a polynomial-time $\exp(-O(\log n/\log\log n))$-approximation algorithm then the Exponential Time Hypothesis would be violated.

Theorem 1 *We have the following inapproximability results for* DAGMaxBinaryTree *on n-vertex input graphs*:

1. DAGMaxBinaryTree *does not admit a polynomial-time constant-factor approximation assuming* $\mathbf{P} \neq \mathbf{NP}$.
2. *If* DAGMaxBinaryTree *admits a polynomial-time* $\exp(-O(\log n/\log\log n))$*-approximation, then* $\mathbf{NP} \subseteq \mathbf{DTIME}\left(\exp\left(O\left(\sqrt{n}\right)\right)\right)$, *refuting the Exponential Time Hypothesis*.
3. *For any* $\varepsilon > 0$, *if* DAGMaxBinaryTree *admits a quasi-polynomial time* $\exp\left(-O\left(\log^{1-\varepsilon} n\right)\right)$*-approximation, then* $\mathbf{NP} \subseteq \mathbf{DTIME}\left(\exp\left(\log^{O(1/\varepsilon)} n\right)\right)$, *thus refuting the Exponential Time Hypothesis*.

LP-Based Approach The longest path problem in DAGs can be solved using a linear program (LP) based on cut constraints. Based on this connection, an integer program (IP) based on cut constraints can be formulated for DAGMaxBinaryTree. In Sect. 5.3, we show that the LP-relaxation of this cut-constraints-based-IP has an integrality gap of $\Omega(n^{1/3})$ in n-vertex DAGs.

Undirected Graphs Next, we turn to undirected graphs. We show that UndirMaxBinaryTree does not have a constant-factor approximation and does not admit a quasi-polynomial-time $\exp(-O(\log^{0.63} n))$-approximation under the Exponential Time Hypothesis.

Theorem 2 *We have the following inapproximability results for* UndirMaxBinaryTree *on n-vertex input graphs*:

1. UndirMaxBinaryTree *does not admit a polynomial-time constant-factor approximation assuming* $\mathbf{P} \neq \mathbf{NP}$.
2. *For* $c = \log_3 2$ *and any* $\varepsilon > 0$, *if* UndirMaxBinaryTree *admits a quasi-polynomial time* $\exp(-O(\log^{c-\varepsilon} n))$*-approximation, then* $\mathbf{NP} \subseteq \mathbf{DTIME}\left(\exp\left(\log^{O(1/\varepsilon)} n\right)\right)$, *thus refuting the Exponential Time Hypothesis*.

We summarize our hardness results for MBT on various graph families in Table 1 and contrast them with the corresponding known hardness results for the longest path problem on those families.

1.2.2 Algorithmic Results

Fixed-Parameter Tractability We denote the decision variant of MBT as k-BinaryTree—here the goal is to verify if a given graph contains a binary tree with at least k vertices. Since k-BinaryTree is **NP**-hard when k is part of the input, it is

Table 1 Summary of inapproximability results

Family	Assumption	Max binary tree	Longest path
DAGs	$\mathbf{P} \neq \mathbf{NP}$	No poly-time $\Omega(1)$-apx (Theorem 1)	Poly-time solvable
	ETH	No poly-time $\exp(-O(\frac{\log n}{\log\log n}))$-apx No quasi-poly-time $\exp(-O(\log^{1-\epsilon} n))$-apx (Theorem 1)	Poly-time solvable
Directed	$\mathbf{P} \neq \mathbf{NP}$	Same as DAGs (Theorem 1)	No poly-time $\frac{1}{n^{1-\epsilon}}$-apx [9]
	ETH	Same as DAGs (Theorem 1)	Same as $\mathbf{P} \neq \mathbf{NP}$
Undirected	$\mathbf{P} \neq \mathbf{NP}$	No poly-time $\Omega(1)$-apx (Theorem 2)	No poly-time $\Omega(1)$-apx [23]
	ETH	No quasi-poly-time $\exp(-O(\log^{0.63-\epsilon} n))$-apx (Theorem 2)	No quasi-poly-time $\exp(-O(\log^{1-\epsilon} n))$-apx [23]

Here, n refers to the number of vertices in the input graph and ϵ is any positive constant. We include the known results for longest path for comparison

desirable to have an algorithm that runs in time $f(k)\mathsf{poly}(n)$ (i.e., a fixed parameter algorithm parameterized by the solution size). Our first algorithmic result achieves precisely this goal. Our algorithm is based on algebraic techniques.

Theorem 3 *k-binary-tree There exists a randomized algorithm that takes a directed graph $G = (V, E)$, a positive integer k, and a real value $\delta \in (0, 1)$ as input, runs in time $2^k\mathsf{poly}(|V|)\log(1/\delta)$ and*

1. *outputs 'no' if G does not contain a binary tree of size k;*
2. *outputs a binary tree of size k with probability $1 - \delta$ if G contains one.*

Bipartite Permutation Graphs Next, motivated by its connection to the max heapable subsequence problem, we study MBT in *bipartite permutation graphs*. A bipartite permutation graph is a permutation graph (undirected) which is also bipartite. We show that bipartite permutation graphs admit an efficient algorithm for MBT. Our algorithm exploits structural properties of bipartite permutation graphs. We believe that these structural properties could be helpful in solving MBT in permutation graphs which, in turn, could provide key insights towards solving MBT in permutation DAGs.

Theorem 4 *There exists an algorithm to solve* UndirMaxBinaryTree *in n-vertex bipartite permutation graphs that runs in time $O(n^3)$.*

We summarize our algorithmic results for MBT in Table 2 and contrast them with the corresponding best known bounds for the longest path problem.

Table 2 Summary of algorithmic results

Problem	Max binary tree	Longest path
FPT parameterized by solution size (Dir.)	2^kpoly(n)-time (Theorem 3)	1.66^kpoly(n)-time [8]
Bipartite permutation graphs (Undir.)	$O(n^3)$-time (Theorem 4)	$O(n)$-time [38]

Here, n refers to the number of vertices in the input graph. We include the known results for longest path for comparison

We remark again that our inapproximability as well as algorithmic results are also applicable to the maximum degree-constrained tree problem for larger, but constant degree constraint. We focus on the degree constraint corresponding to binary trees for the sake of simplicity in exposition.

1.3 Proof Techniques

In this section, we outline the techniques underlying our results.

1.3.1 Inapproximability Results

At a very high level, our inapproximability results for MBT rely on the proof strategy for hardness of longest path due to Karger et al. [23], which has two main steps: (1) a *self-improving* reduction whose amplification implies that a constant-factor approximation immediately leads to a PTAS, and (2) a proof that there is no PTAS. However, we achieve both these steps in a completely different manner compared to the approach of Karger, Motwani, and Ramkumar. Both their steps are tailored for the longest path problem, but fail for the maximum degree-constrained tree problem. Our results for MBT require several novel ideas, as described next.

Karger, Motwani and Ramkumar's self-improving reduction for the longest path proceeds as follows: given an undirected graph G, they obtain a squared graph G^2 by replacing each edge $\{u, v\}$ of G with a copy of G by adding edges from u and v to all vertices in that edge copy. Let $OPT(G)$ be the length of the longest path in G. They make the following two observations: Obs (i) $OPT(G^2) \geq OPT(G)^2$ and Obs (ii) a path in G^2 of length at least $\alpha OPT(G^2)$ can be used to recover a path in G of length at least $\sqrt{\alpha} OPT(G)$. The first observation is because we can extend any path P in G into a path of length $|E(P)|^2$ by traversing each edge copy also along P. The second observation is because for any path P_2 in G^2 either P_2 restricted to some edge copy of G (i.e., subgraph of the edge copy formed by the vertices and edges it has in common with P_2) is a path of length at least $\sqrt{|E(P_2)|}$ or projecting P_2 to G (i.e., replacing each sub-path of P_2 in each edge copy by a single edge) gives a path of length at

least $\sqrt{|E(P_2)|}$. We note that a similar construction of the squared graph for directed graphs also has the above mentioned observations: replace each directed arc (u, v) of G with a copy of G by adding arcs from u to all vertices in that edge copy and from all vertices in that edge copy to v.

In order to obtain inapproximability results for the maximum binary tree problem, we first introduce different constructions for the squared graph in the self-improving reduction compared to the ones by Karger et al. Moreover, our constructions of the squared graph differ substantially between undirected and directed graphs. Interestingly, our constructions also generalize naturally to the max degree-constrained tree problem. Secondly, although our reduction for showing the lack of PTAS in undirected graphs for MBT is also from TSP(1, 2), it is completely different from that of Karger et al. and, once again, generalizes to the max degree-constrained tree problem. Thirdly, we show the lack of PTAS in DAGs for MBT by reducing from the max 3-coloring problem. This reduction is altogether new—the reader might recall that the longest path problem in DAGs is solvable in polynomial-time, so there cannot be a counterpart of this step (i.e., lack of PTAS in DAGs) for longest path. We next present further details underlying our proofs.

Self-improving Reduction for Directed Graphs We focus on the rooted variant of MBT in directed graphs. We first assume that the given graph G contains a source (if not, adding such a source vertex with arcs to all the vertices changes the optimum only by one). In contrast to the squared graph described above (i.e., instead of adding edge copies), we replace every vertex in G by a copy of G (that we call as a vertex copy) and for every arc (u, v) in G, we add an arc from the root node (we call the vertices within vertex copies as *nodes*) of the vertex copy corresponding to u to the source node of the vertex copy corresponding to v. Finally, we declare the root node of the root vertex copy to be the root node of G^2. Let $\alpha \in (0, 1]$ and $OPT(G)$ be the number of vertices in the maximum binary tree in G. With this construction of the squared graph, we show that (1) $OPT(G^2) \geq OPT(G)^2$ and (2) an α-approximate rooted binary tree T_2 in G^2 can be used to recover a rooted binary tree T_1 in G which is a $\sqrt{\alpha}$-approximation. We emphasize that if G is a DAG, then the graph G^2 obtained by this construction is also a DAG.

Inapproximability for DAGs In order to show the constant-factor inapproximability result for DAGs, it suffices to show that there is no PTAS (due to the self-improving reduction for directed graphs described above). We show the lack of a PTAS in DAGs by reducing from the max 3-coloring problem in 3-colorable graphs. It is known that this problem is APX-hard—in particular, there is no polynomial-time algorithm to find a coloring that colors at least 32/33-fraction of the edges properly [21]. Our reduction encodes the coloring problem into a DAGMaxBinaryTree instance in a way that recovers a consistent coloring for the vertices while also being proper for a large fraction of the edges. Our ETH-based inapproximability result is also a consequence of this reduction in conjunction with the self-improving reduction. We again emphasize that there is no counterpart of APX-hardness in DAGs for max binary tree in the longest path literature.

Self-improving Reduction for Undirected Graphs For UndirMaxBinaryTree, the self-improving reduction is more involved. Our above-mentioned reduction for

DirMaxBinaryTree heavily exploits the directed nature of the graph (e.g., uses source vertices) and hence, is not applicable for undirected graphs. Moreover, the same choice of squared graph G^2 as Karger et al. [23] fails since Obs (ii) does not hold any more: the tree T_2 restricted to each edge copy may not be a tree (but it will be a forest). However, we observe that T_2 restricted to each edge copy may result in a forest with up to four binary trees in it. This observation and a more careful projection can be used to recover a tree of size at least $\sqrt{|V(T_2)|}/4$ (let us call this weakened Obs (ii)). Yet, weakened Obs (ii) is insufficient for a self-improving reduction. One approach to fix this would be to construct a *different squared graph* $G^{\boxtimes 2}$ that strengthens Obs (i) to guarantee that $OPT(G^{\boxtimes 2}) \geq 16OPT(G)^2$ while still allowing us to recover a binary tree of size $\sqrt{|V(T_2)|}/4$ in G from a binary tree T_2 in $G^{\boxtimes 2}$. Such a strengthened Obs (i) coupled with weakened Obs (ii) would complete the self-improving reduction. Our reduction is a variant of this approach: we introduce a construction of the squared-graph that strengthens Obs (i) by a factor of 2 while also weakening Obs (ii) only by a factor of 2. We prove these two properties of the construction by relying on a handshake-like property of binary trees which is a relationship between the number of vertices of each degree and the total number of vertices in the binary tree.

Inapproximability for Undirected Graphs In order to show the constant-factor inapproximability result, it suffices to show that there is no PTAS (due to the self-improving reduction). We show the lack of a PTAS by reducing from TSP(1, 2). We mention that Karger et al. [23] also show the lack of a PTAS for the longest path problem by reducing from TSP(1, 2). However, our reduction is much different from their reduction. Our reduction mainly relies on the fact that if we add a pendant node to each vertex of a graph G and obtain a binary tree T that has a large number of such pendants, then the binary tree restricted to G cannot have too many nodes of degree three. Our ETH-based inapproximability result is also a consequence of this reduction in conjunction with the self-improving reduction.

1.3.2 Algorithmic Results

A $2^k\mathsf{poly}(n)$ *Time Algorithm for* k-BinaryTree. The proof of this result is inspired by the algebrization technique introduced in [27, 28, 39] for designing randomized algorithms for k-Path and k-Tree—in k-Path, the goal is to recover a path of length k in the given graph while k-Tree asks to recover a *given tree* on k vertices in the given graph. Their idea is to encode a path (or the given tree) as a multilinear monomial term in a carefully constructed polynomial, which is efficiently computable using an arithmetic circuit. Then, a result due to Williams [39] is used to verify if the constructed polynomial contains a multilinear term—Williams' result gives an efficient randomized algorithm, which on input a small circuit that computes the polynomial, outputs 'yes' if a multilinear term exists in the sum of products representation of the input polynomial, and 'no' otherwise. The subgraph that is sought may then be extracted using an additional pass over the graph. Our main technical contribution is the construction of a polynomial P_G whose multilinear terms correspond to binary trees of size k in G and which is efficiently computable by an arithmetic circuit. We remark that the polynomial constructions in previous results do

not readily generalize for our problem. Our key contribution is the construction of a suitable polynomial, based on a carefully designed recursion.

Efficient Algorithm for Bipartite Permutation Graphs Our main structural insight for bipartite permutation graphs is that there exists a maximum binary tree which is *crossing-free* with respect to the so-called *strong ordering* of the vertices. With this insight, MBT in bipartite permutation graphs reduces to finding a maximum crossing-free binary tree. We solve this latter problem by dynamic programming.

1.4 Organization

We present the $2^k\mathsf{poly}(n)$ time algorithm for k-BinaryTree in Sect. 4. We present our hardness results for DAGs in Sect. 5. We formulate an IP for DAGs and discuss its integrality gap in Sect. 5.3. We show our hardness results for undirected graphs in Sect. 6. We design an efficient algorithm for bipartite permutation graphs in Sect. 7. We conclude with a few open problems in Sect. 8.

1.5 Preliminaries

PTAS and APX-Hardness We say that a maximization problem has a *polynomial-time approximation scheme* (PTAS) if it admits an algorithm that for each fixed $\varepsilon > 0$, and for each instance, outputs a solution of size at least $(1-\varepsilon)$ times the optimal solution (we refer to such solutions as $(1-\varepsilon)$-approximate solutions), in time polynomial in the size of the input instance. A problem is said to be in the class APX if it has a polynomial-time constant-factor approximation algorithm. A problem is APX-hard if there is a PTAS reduction from every problem in APX to that problem.

MBT in Directed Graphs Given a directed graph $G = (V, E)$ and a vertex $r \in V$, we say that a subgraph T where $V(T) \subseteq V$ and $E(T) \subseteq E$, is an *r-rooted tree* in G if T is acyclic and every vertex v in T has a *unique* directed path (in T) to r. If the in-degree of each vertex in T is at most 2, then T is an *r-rooted binary tree*.

The problem of interest in directed graphs is the following:

---

rooted-DirMaxBinaryTree

Given: A directed graph $G = (V, E)$ and a root $r \in V$.

Goal: An r-rooted binary tree T in G with maximum number of vertices.

The problem DAGMaxBinaryTree is a special case of rooted-DirMaxBinaryTree in which the input directed graph is a DAG. We recall that the rooted and unrooted variants of the maximum binary tree problem in DAGs are equivalent.

MBT in Undirected Graphs Given an undirected graph $G = (V, E)$, we say that a subgraph T, where $V(T) \subseteq V$ and $E(T) \subseteq E$, is a *binary tree* in G if T is connected, acyclic, and $\deg_T(v) \leq 3$ for every vertex $v \in V(T)$. We will focus on the unrooted variant, i.e., UndirMaxBinaryTree, since the inapproximability results for the rooted variant are implied by inapproximability results for the unrooted variant.

UndirMaxBinaryTree
Given: An undirected graph G.
Goal: A binary tree in G with maximum number of vertices.

2 A 2^kpoly(n) Time Algorithm for k-BinaryTree

In this section, we present a randomized algorithm that solves k-BinaryTree exactly and runs in time 2^kpoly(n) where n is the number of vertices in the input graph. We recall that k-BinaryTree is the problem of deciding whether a given directed graph contains a binary tree of size k. Our algorithm is inspired by an algebraic approach for solving the k-Path problem—the algebraic approach relies on efficient detection of multilinear terms in a given polynomial.

k-Path, polynomials and Multilinear Terms We begin with a recap of the algebraic approach to solve k-Path—here, the goal is to verify if a given (directed or undirected) graph G contains a path of length at least k. There has been a rich line of research dedicated to designing algorithms for k-Path with running time β^kpoly(n) where $\beta > 1$ is a constant and n is the number of vertices in G (cf. [2, 8, 27, 39]). In particular, the algorithms in [27, 39] are based on detecting multilinear terms in a polynomial.

We now recall the problem of detecting multilinear terms in a polynomial. Here, we are given a polynomial with coefficients in a finite field $\mathbb{F}_q$ and the goal is to verify if it has a multilinear term. We emphasize that the input polynomial is given *implicitly* by an arithmetic circuit consisting of addition and multiplication gates. In other words, the algorithm is allowed to evaluate the polynomial at any point but does not have direct access to the sum-of-product expansion of the polynomial. We recall that a multilinear term in a polynomial $p \in \mathbb{F}_q[x_1, x_2, \cdots, x_m]$ is a monomial in the sum-of-products expansion of p consisting of only degree-1 variables. For example, in the following polynomial

$$p(x_1, x_2, x_3) = x_1^2 x_2 + x_3 + x_1 x_2 x_3,$$

the monomials x_3 and $x_1x_2x_3$ are multilinear terms, whereas $x_1^2x_2$ is not a multilinear term since x_1 has degree 2. We will use the algorithm mentioned in the following theorem as a black box for detecting multilinear terms in a given polynomial.

Theorem 5 (Theorem 3.1 in [39]) *Let $P(x_1, \cdots, x_n)$ be a polynomial of degree at most k, represented by an arithmetic circuit of size $s(n)$ with addition gates (of unbounded fan-in), multiplication gates (of fan-in two), and no scalar multiplications. There is a randomized algorithm that on input P runs in $2^k s(n) \cdot \mathsf{poly}(n) \log(1/\delta)$ time, outputs 'yes' with probability $1 - \delta$ if there is a multilinear term in the sum-product expansion of P, and outputs 'no' if there is no multilinear term.*

The idea behind solving k-Path with the help of this theorem is to construct a polynomial p_G based on the input graph G so that p_G contains a multilinear term

if and only if G contains a simple path of length k. At the same time, p_G should be computable by an arithmetic circuit of size $\mathsf{poly}(n)$. Koutis and Williams achieved these properties using the following polynomial:

$$p_G(x_1, \cdots, x_n) := \sum_{\left(v_{i_1}, v_{i_2}, \ldots, v_{i_k}\right) : a walk in G} x_{i_1} x_{i_2} \ldots x_{i_k}.$$

We recall that a walk in G is a sequence of vertices in which neighbouring vertices are adjacent in G. From the definition, it is easy to observe that there is a one-to-one correspondence between simple k-paths in G and multilinear terms in p_G. Moreover, it can be shown that there is an arithmetic circuit of size $O\left(k^2(m+n)\right)$ that computes p_G, where m is the number of edges and n is the number of vertices in G. See Chapter 10.4 of [14] for alternative constructions of this polynomial.

*The Polynomial Construction for k-*BinaryTree Following the above-mentioned approach, we construct a polynomial P_G with the property that P_G contains a multilinear term if and only if G contains a binary tree of size k. Unfortunately, there is no immediate generalization of walks of length k that characterize binary trees on k vertices. So, instead of defining the polynomial conceptually, we will define the polynomial recursively by building the arithmetic circuit that computes P_G, and will prove the correspondence between multilinear terms in P_G and binary trees of size k in G. In the definition of our polynomial, we also need to introduce an auxiliary variable to eliminate low-degree multilinear terms in P_G (which is not an issue in the construction of the polynomial for k-Path).

Let $G = (V, E)$ be the given directed graph. For $v \in V$, let $\Delta_v^{in} := \{u \in V : (u, v) \in E\}$. We begin by defining a polynomial $P_v^{(k)}$ for every $v \in V$ and every positive integer k, in $(n+1)$ variables $\left\{x_v\right\}_{v \in V} \cup \{y\}$:

$$P_v^{(k)} := \begin{cases} x_v & \text{if } k = 1 \\ x_v \cdot y^{k-1} & \text{if } k > 1 \text{ and} \Delta_v^{in} = \emptyset \\ x_v \left(\sum_{u \in \Delta_v^{in}} P_u^{(k-1)} + \sum_{\ell=1}^{k-2} \left(\sum_{u_1 \in \Delta_v^{in}} P_{u_1}^{(\ell)} \right) \left(\sum_{u_2 \in \Delta_v^{in}} P_{u_2}^{(k-1-\ell)} \right) \right) & \text{if } k > 1 \text{ and} \Delta_v^{in} \neq \emptyset \end{cases}$$

Next, we define $P_G^{(k)} := \sum_{v \in V} P_v^{(k)}$. We recall that a polynomial is homogenous if every monomial has the same degree. By induction on k, the polynomial $P_v^{(k)}$ is a degree-k homogeneous polynomial and so is $P_G^{(k)}$. Moreover, by the recursive definition, we see that $P_v^{(k)}$ can be represented as an arithmetic circuit of size $O(k^2 n)$ since there are kn polynomials in total, and computing each requires $O(1)$ addition gates (with unbounded fan-in) and $O(k)$ multiplication gates (with fan-in two). We show the following connection between multilinear terms in $P_G^{(k)}$ and binary trees in G.

Lemma 1 *The graph G has a binary tree of size k rooted at r if and only if there is a multilinear term of the form $\prod_{v \in S} x_v$ in $P_r^{(k)}$ where $|S| = k$.*

Proof We first show the forward direction, i.e., if G has a binary tree T of size k rooted at r, then there is a multilinear term of the form $\prod_{v \in T} x_v$ in $P_r^{(k)}$. We prove this

by induction on k. The base case $k = 1$ follows since $P_r^{(1)} = x_r$. Suppose that the forward direction holds when $|T| \leq k-1$. For $|T| = k$, we consider two cases.

1. The root r has only one child c. The subtree T_c of T rooted at c has size $k-1$. By induction hypothesis there is a multilinear term $\prod_{v \in T_c} x_v$ in $P_c^{(k-1)}$. Since $c \in \Delta_r^{in}$, for some polynomial Q we can write
$$P_r^{(k)} = x_r\big(P_c^{(k-1)} + Q\big).$$
Therefore $x_r \cdot \prod_{v \in T_c} x_v$ is a term in $P_r^{(k)}$. This term is multilinear and equals to $\prod_{v \in T} x_v$ since $r \notin T_c$.
2. The root r has two children c_1, c_2. Suppose that the subtree T_{c_1} rooted at c_1 has size ℓ, thus the subtree T_{c_2} rooted at c_2 has size $k-1-\ell$. The induction hypothesis implies that $P_{c_1}^{(\ell)}$ has a multilinear term $\prod_{v \in T_{c_1}} x_v$, and $P_{c_2}^{(k-1-\ell)}$ has a multilinear term $\prod_{v \in T_{c_2}} x_v$. Since $c_1, c_2 \in \Delta_r^{in}$, for some polynomial Q we can write
$$P_r^{(k)} = x_r\Big(P_{c_1}^{(\ell)} P_{c_2}^{(k-1-\ell)} + Q\Big).$$
Therefore $x_r\Big(\prod_{v \in T_{c_1}} x_v\Big)\Big(\prod_{v \in T_{c_2}} x_v\Big)$ is a term in $P_r^{(k)}$. This term is multilinear and equals to $\prod_{v \in T} x_v$ because T is the disjoint union of r, T_{c_1} and T_{c_2}.

In both cases, the polynomial $P_r^{(k)}$ has a multilinear term $\prod_{v \in T} x_v$. This completes the inductive step.

Next, we show that if $P_r^{(k)}$ has a multilinear term of the form $\prod_{v \in S} x_v$ where $|S| = k$, then there is a binary tree T rooted at r in G with vertex set S. We prove this also by induction on k. The base case $k = 1$ is trivial since $P_r^{(1)} = x_r$ and there is a binary tree of size 1 rooted at r. Suppose that the statement holds for $k-1$ or less ($k > 1$).

Let $\prod_{v \in S} x_v$ be a multilinear term in $P_r^{(k)}$. We note that $r \in S$ since every term in $P_r^{(k)}$ contains x_r. Moreover, we may assume that $\Delta_r^{in} \neq \emptyset$ since otherwise $P_r^{(k)} = x_r \cdot y^{k-1}$ which does not contain any term of the form $\prod_{v \in S} x_v$. According to the definition of $P_r^{(k)}$, we could have two cases.

1. The term $\prod_{v \in S \setminus \{r\}} x_v$ is a multilinear term in $P_c^{(k-1)}$ for some $c \in \Delta_r^{in}$. The induction hypothesis implies that there is a binary tree T_c rooted at c with vertex set $S \setminus \{r\}$. Let T be the binary tree obtained by adding the edge (c, r) to T_c. Then T is a binary tree rooted at r with vertex set S.
2. The term $\prod_{v \in S \setminus \{r\}} x_v$ is a multilinear term in $P_{c_1}^{(\ell)} P_{c_2}^{(k-1-\ell)}$ for some $c_1, c_2 \in \Delta_r^{in}$ and some integer $1 \leq \ell \leq k-2$. In this case, since $P_{c_1}^{(\ell)}$ and $P_{c_2}^{(k-1-\ell)}$ are homogeneous polynomials of degree ℓ and $k-1-\ell$, we can partition $S \setminus \{r\}$ into two sets S_1 and S_2 with $|S_1| = \ell$ and $|S_2| = k-1-\ell$ such that $\prod_{v \in S_1} x_v$ is a multilinear term in $P_{c_1}^{(\ell)}$, and $\prod_{v \in S_2} x_v$ is a multilinear term in $P_{c_2}^{(\ell)}$. Applying the induction hypothesis, we obtain a binary tree T_{c_1} (rooted at c_1) with vertex set S_1 and a binary tree T_{c_2} (rooted at c_2) with vertex set S_2. Let T be the binary tree obtained by adding edges (c_1, r) and (c_2, r) to $T_{c_1} \cup T_{c_2}$. Then T is a binary tree rooted at r with vertex set $S_1 \cup S_2 \cup \{r\} = S$.

In both cases, we can find a binary tree T rooted at r with vertex set S. This completes the inductive step. □

With this choice of $P_G^{(k)}$, we call the algorithm appearing in Theorem 5 on input polynomial $\tilde{P}_G^{(k)} := y \cdot P_G^{(k)}$, and output the result. We note that every multilinear term of the form $\prod_{v \in S} x_v$ in $P_G^{(k)}$ becomes a multilinear term of the form $y \cdot \prod_{v \in S} x_v$ in $\tilde{P}_G^{(k)}$, and every multilinear term of the form $y \cdot \prod_{v \in S} x_v$ in $P_G^{(k)}$ becomes $y^2 \cdot \prod_{v \in S} x_v$ in $\tilde{P}_G^{(k)}$, which is no longer a multilinear term. In light of Lemma 1, the graph G contains a binary tree of size k if and only if the degree-$(k+1)$ homogeneous polynomial $\tilde{P}_G^{(k)}$ has a multilinear term. The running time is $2^{k+1} \cdot O(k^2 n) \cdot \mathsf{poly}(n+1) \log(1/\delta) = 2^k \cdot \mathsf{poly}(n) \log(1/\delta)$.

We remark that this algorithm does not immediately tell us the tree T (namely the edges in T). However, we can find the edges in T with high probability via a reduction from the search variant to the decision variant. This is formalized in the next lemma.

Lemma 2 *Suppose that there is an algorithm $\mathcal{A}$ which takes as input a directed graph $G = (V, E)$, an integer k and $\delta' \in (0,1)$ runs in time $2^k \mathsf{poly}(|V|) \log\left(1/\delta'\right)$ and*

- outputs 'yes' with probability at least $1 - \delta'$ if G contains a binary tree of size k,
- outputs 'no' with probability 1 if G does not contain a binary tree of size k.

Then there also exists an algorithm $\mathcal{A}'$ which for every $\delta \in (0,1)$ outputs a binary tree T of size k with probability at least $1 - \delta$ when the answer is 'yes', and runs in time $2^k \mathsf{poly}(|V|) \log(1/\delta)$.

Proof The algorithm $\mathcal{A}'$ iterates through all arcs $e \in E$ and calls $\mathcal{A}$ on $(G - e, k)$ with $\delta' = \delta/m$ where $G - e = (V, E \setminus \{e\})$ and $m = |E|$. If for some $e \in E$ the call to $\mathcal{A}$ outputs 'yes', we remove e from G (i.e., set $G \leftarrow G - e$) and continue the process. We will show that when the algorithm terminates, the arcs in G constitute a binary tree of size k (if there exists one) with probability at least $1 - \delta$.

Suppose the order in which $\mathcal{A}'$ processes the arcs is $e_1, e_2, \cdots, e_m$, and the graph at iteration t is denoted by $G^{(t)}$. Let B_t denote the event "$G^{(t-1)} - e_t$ contains a binary tree of size k, but the call to $\mathcal{A}\left(G^{(t-1)} - e_t, k\right)$ returns no". Due to the assumption we made for $\mathcal{A}$, event B_t happens with probability at most δ'. Since the algorithm $\mathcal{A}$ has perfect soundness, whenever $\mathcal{A}'$ removes an edge we are certain that the remaining graph still contains a binary tree of size k (otherwise the call to $\mathcal{A}$ would never return 'yes'). That means if $G^{(0)} = G$ contains a binary tree of size k then $G^{(t)}$ contains a binary tree of size k for all $0 \le t \le m$. Therefore if none of the events B_t happens, the final graph $G^{(m)}$ is a binary tree of size k. The probability of failure is upper bounded by

$$\Pr\left[\bigcup_{t=1}^{m} B_t\right] \le m \cdot \delta' = m \cdot \frac{\delta}{m} = \delta.$$

Since algorithm $\mathcal{A}'$ makes m calls to algorithm $\mathcal{A}$, the running time of $\mathcal{A}'$ is $m \cdot 2^k \mathsf{poly}(|V|) \log\left(1/\delta'\right) = 2^k \mathsf{poly}(|V|) \log(1/\delta)$. □

Theorem 5 in conjunction with Lemmas 1 and 2 complete the proof of Theorem 3.

3 Hardness Results for DAGs

In this section, we show the inapproximability of finding a maximum binary tree in DAGs. The *size* of a binary tree denotes the number of vertices in the tree.

3.1 Self-improvability for Directed Graphs

We show that if there exists a constant-factor approximation algorithm for RootedDirMaxBinaryTree, then such an algorithm can be turned into a PTAS as in Theorem 6. We emphasize that this result holds for arbitrary directed graphs and not just DAGs. The idea is to define a *squared graph* and gradually boost up the approximation ratio by running the constant-factor approximation algorithm on squared graphs and extracting better solutions for the original graph. We note that our notion of squared graph is similar to that of Karger, Motwani and Ramkumar [23], and differs from standard graph theoretic definitions of the same. We define our squared graph next.

Definition 1 Given a directed graph $G = (V, E)$ with root r, the *squared graph* G^2 is the directed graph obtained by performing the following operations on G:

1. Construct $G' = (V', E')$ by introducing a source vertex s, i.e., $V' := V \cup \{s\}$. We add arcs from s to every vertex in G, i.e., $E' := E \cup \{(s, v) : v \in V\}$.
2. For each $u \in V$ (we note that V does not include the source vertex), we create a copy of G' that we denote as a *vertex copy* G'_u. We call the vertices within vertex copies as nodes, and denote the root node of G'_u by r_u, and the source node of G'_u by s_u.
3. For each $(u, v) \in E$, we create an arc $\left(r_u, s_v\right)$.
4. We declare the root of G^2 to be r_r, i.e. the root node of the vertex copy G'_r.

We define $G^{2^{k+1}}$ recursively as $G^{2^{k+1}} := \left(G^{2^k}\right)^2$ with the base case $G^1 := G$.

Given a directed graph G with $n-1$ vertices, let $n_k := \left|V\left(G^{2^k}\right)\right|$ so that $n_0 = n - 1$. Then, n_k satisfies the following recurrence relation.

$$n_k = n_{k-1}\left(n_{k-1} + 1\right) = n_{k-1}^2 + n_{k-1}.$$

Hence, we have

$$n_k + 1 \leq \left(n_{k-1} + 1\right)^2 \leq \left(n_{k-2} + 1\right)^{2^2} \leq \ldots \leq \left(n_0 + 1\right)^{2^k} = n^{2^k}.$$

We use $OPT(G)$ to denote the size (number of vertices) of a maximum binary tree in G. The following lemma shows that $OPT(G)$ is super-multiplicative under the squaring operation.

Lemma 3 *For any fixed root r, $OPT(G^2) \geq OPT(G)^2$.*

Proof Suppose we have an optimal r-rooted binary tree T_1 of G, i.e. $|V(T_1)| = OPT(G)$. We construct an r_r-rooted binary tree T_2 of G^2 as follows (Fig. 1):

1. For $v \in V(G)$, define $T'_v = T_v \cup \{s_v\}$ to be the optimal r_v-rooted binary tree in the vertex copy G'_v where T_v is identical to T_1 and the source node s_v is connected to an arbitrary leaf node in T_v.
2. For every vertex $v \in T_1$, add T'_v to T_2. This step generates $|V(T_1)| \cdot (|V(T_1)| + 1)$ nodes in T_2.
3. Connect the copies selected in step 2 by adding the arc (r_u, s_v) to T_2 for every arc $(u, v) \in T_1$.

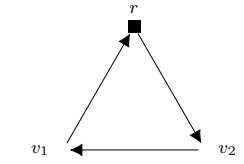
Since T_1 is an r-rooted binary tree (in G), it follows that T_2 is an r_r-rooted binary tree (in G^2). Moreover, the size of T_2 is

$$|V(T_2)| = |V(T_1)| \cdot (|V(T_1)| + 1) \geq OPT(G)^2,$$

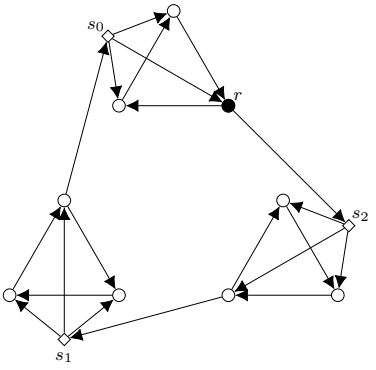
which cannot exceed $OPT(G^2)$. □

The following lemma shows that a large binary tree in G^2 can be used to obtain a large binary tree in G.

Lemma 4 *For every $\alpha \in (0, 1]$, given an r_r-rooted binary tree T_2 in G^2 with size*



(a) G rooted at the black node



(b) G^2 rooted at the black node. Source nodes are represented by diamonds.

Fig. 1 Directed squared graph

$$|V(T_2)| \geq \alpha OPT(G^2) - 1,$$

there is a linear-time (*in the size of* G^2) *algorithm that finds an r-rooted binary tree* T_1 *of G with size*

$$|V(T_1)| \geq \sqrt{\alpha} OPT(G) - 1.$$

Proof Let $U := \{v : v \in V(G) \text{ such that } r_v \in V(T_2)\}$ and $A := \{(v,w) : v,w \in V(G), (r_v, s_w) \in E(T_2)\}$. We note that $T_1' := (U,A)$ is an r-rooted binary tree in G. This is because the path from every $v \in U$ to the root r is preserved, and the in-degree of every vertex $w \in U$ is bounded by the in-degree of s_w (in T_2), which is thus at most 2, and similarly the out-degree of every vertex is at most 1. We also remark that T_1' can be found in linear time. If $|U| \geq \sqrt{\alpha} OPT(G) > \sqrt{\alpha} OPT(G) - 1$, then the lemma is already proved. So, we may assume that $|U| < \sqrt{\alpha} OPT(G)$.

We now consider $T_v' := (V(T_2) \cap V(G_v'), E(T_2) \cap E(G_v'))$ for $v \in U$. We can view T_v' as the restriction of T_2 to G_v', hence every node of T_v' has out-degree at most 2. Since T_2 is an r_r-rooted binary tree in G^2, every node in $V(T_2) \cap V(G_v')$ has a unique directed path (in T_2) to r_r, which must go through r_v, thus every node in $V(T_2) \cap V(G_v')$ has a unique directed path to r_v. It follows that T_v' is an r_v-rooted binary tree in the vertex copy G_v'.

We now show that there exists $v \in U$ such that $|V(T_v')| \geq \sqrt{\alpha} OPT(G)$. Suppose not, which means for every $v \in U$ we have $|V(T_v')| < \sqrt{\alpha} OPT(G)$. Then

$$\begin{aligned} |V(T_2)| &= \sum_{v \in U} |V(T_v')| < \sum_{v \in U} \left(\sqrt{\alpha} OPT(G) \right) < \sqrt{\alpha} OPT(G) \cdot \sqrt{\alpha} OPT(G) \\ &= \alpha OPT(G)^2 \leq \alpha \cdot OPT(G^2), \end{aligned}$$

a contradiction. The last inequality is due to Lemma 3.

In linear time we can find a binary tree T_v' with the desired size $|V(T_v')| \geq \sqrt{\alpha} OPT(G)$. To complete the proof of the lemma, we let $T_1 := T_v' \setminus \{s_v\}$ which is (isomorphic to) an r-rooted binary tree in G with size at least $\sqrt{\alpha} OPT(G) - 1$. □

Theorem 6 *If* ROOTED-DIRMAXBINARYTREE *has a polynomial-time algorithm that achieves a constant-factor approximation*, *then it has a PTAS*.

Proof Suppose that we have a polynomial-time algorithm $\mathcal{A}$ that achieves an α -approximation for ROOTED-DIRMAXBINARYTREE. Given a directed graph G, root r and $\varepsilon > 0$, let

$$k := 1 + \left\lceil \log_2 \frac{\log_2 \alpha}{\log_2 (1 - \varepsilon)} \right\rceil$$

be an integer constant that depends on α and ε. We construct G^{2^k} and run algorithm $\mathcal{A}$ on G^{2^k}. Then, we get a binary tree in G^{2^k} of size at least $\alpha OPT\left(G^{2^k}\right) - 1$. By Lemma 4, we can obtain an r-rooted binary tree in G of size at least

$$\alpha^{2^{-k}} OPT(G) - 1 \geq \alpha^{2^{-k+1}} OPT(G) \geq (1-\varepsilon) OPT(G).$$

The first inequality holds as long as

$$OPT(G) \geq \frac{1}{\sqrt{1-\varepsilon} - (1-\varepsilon)} \geq \frac{1}{\alpha^{2^{-k}} - \alpha^{2^{-k+1}}}.$$

We note that if $OPT(G)$ is smaller than $1/\left(\alpha^{2^{-k}} - \alpha^{2^{-k+1}}\right)$ which is a constant, then we can solve the problem exactly by brute force in polynomial time. Finally, we also observe that for fixed ε, the running time of this algorithm is polynomial since there are at most $n^{2^k} = n^{O(1)}$ vertices in the graph G^{2^k}. □

3.2 APX-Hardness for DAGs

Next, we show the inapproximability results for DAGs. We begin by recalling DAGMaxBinaryTree:

DAGMaxBinaryTree

Given: A directed acyclic graph $G = (V, E)$ and a root $r \in V$.

Goal: An r-rooted binary tree in G with the largest number of vertices.

We may assume that the root is the only vertex that has no outgoing arcs as we may discard all vertices that cannot reach the root. We show that DAGMaxBinaryTree is APX-hard by reducing from the following problem.

Max-3-Colorable-Subgraph

Given: An undirected graph G that is 3-colorable.

Goal: A 3-coloring of G that maximizes the fraction of properly colored edges.

It is known that finding a 3-coloring that properly colors at least 32/33-fraction of edges in a given 3-colorable graph is NP-hard [5, 21]. In particular, Max-3-Colorable-Subgraph is APX-hard. We reduce Max-3-Colorable-Subgraph to DAGMaxBinaryTree. Let $G = (V, E)$ be the input 3-colorable undirected graph with $n := |V|$ and $m := |E|$. For $\varepsilon > 0$ to be fixed later, we construct a DAG, denoted $D(G, \varepsilon)$, as follows (see Fig. 2 for an illustration):

1. Create a directed binary tree B rooted at vertex c with $n := |V|$ leaf vertices. We will identify each leaf by a unique vertex $v \in V$. Create a super root a and arc $c \to a$. This tree and the super root would have $2n$ vertices, including the super root vertex a, n leaf vertices, and $n - 1$ internal vertices.

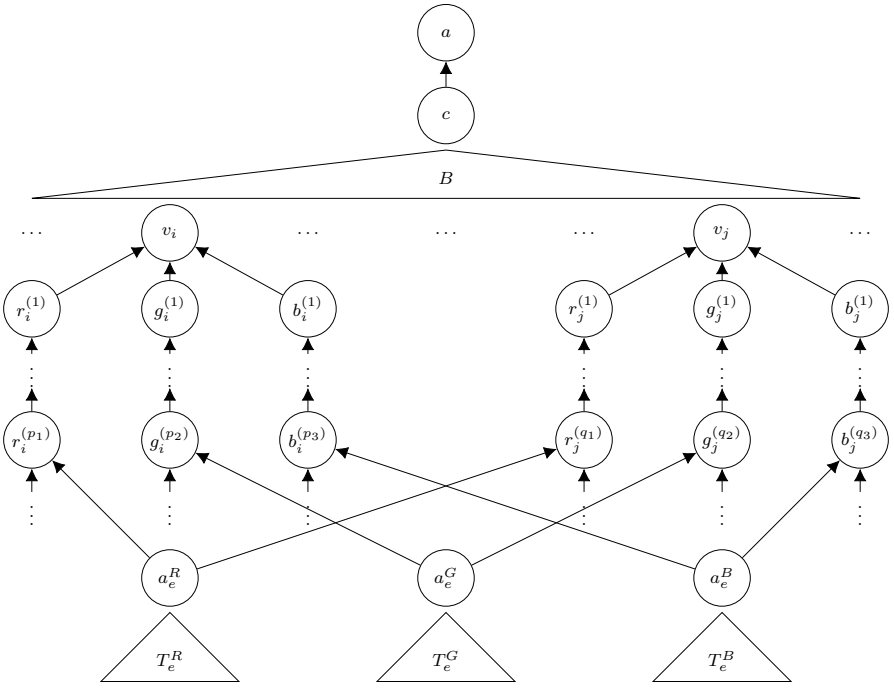


Fig. 2 DAG $D(G,\varepsilon)$ constructed in the reduction from Max-3-Colorable-Subgraph to DAGMaxBinaryTree

2. For every $i \in V$, we introduce three directed paths of length n that will be referred to as R_i, G_i and B_i. Let R_i be structured as $r_i^{(1)} \leftarrow r_i^{(2)} \leftarrow \cdots \leftarrow r_i^{(n)}$, and similarly introduce $g_i^{(k)}$ and $b_i^{(k)}$ with the same structure. Also add arcs $r_i^{(1)} \rightarrow v_i$, $g_i^{(1)} \rightarrow v_i$ and $b_i^{(1)} \rightarrow v_i$.
3. For every edge $e = \{i,j\} \in E$, introduce three directed binary trees that will be referred to as T_e^R, T_e^G, and T_e^B, each with $t = \left\lceil \frac{2\varepsilon n(n+1)+4n^2}{\varepsilon m} \right\rceil$ vertices. Let the roots of the binary trees T_e^R, T_e^G, and T_e^B be a_e^R, a_e^G, and a_e^B respectively. Add arcs $a_e^R \rightarrow r_i^{(p_1)}$ and $a_e^R \rightarrow r_j^{(q_1)}$ where $r_i^{(p_1)}$ and $r_j^{(q_1)}$ are two vertices in R_i and R_j with in-degree strictly smaller than 2. We note that R_i is a path with n vertices so such a vertex always exists. Similarly connect a_e^G to $g_i^{(p_2)}$ and $g_j^{(q_2)}$, and a_e^B to $b_i^{(p_3)}$ and $b_j^{(q_3)}$ in the directed paths G_i and B_i, respectively.

The constructed graph $D(G,\varepsilon)$ is a DAG. We fix a to be the root. The number of vertices N in $D(G,\varepsilon)$ is $N = 3mt + 3n \cdot n + 2n = 3mt + 3n^2 + 2n$. We note that every vertex $v_i \in V$ has in-degree exactly 2 in every a-rooted maximal binary tree in $D(G,\varepsilon)$. The idea of this reduction is to encode the color of v_i as the unique path among R_i, G_i, B_i that is *not* in the subtree under v_i. The following two lemmas summarize the main properties of the DAG constructed above.

Lemma 5 *Let T be a maximal a-rooted binary tree of $D(G,\varepsilon)$. If $|V(T)| \geq (1-\varepsilon/4)(N-n^2)$, then at most εm vertices among $\cup_{e\in E}\{a_e^R, a_e^G, a_e^B\}$ are not in T.*

Proof Suppose more than εm such vertices are missing from T. For each vertex a_e^R that is not in T, the corresponding subtree T_e^R is also not in T (same for a_e^G and a_e^B). Therefore

$$|V(T)| < N - \varepsilon mt = 3mt + 3n^2 + 2n - \varepsilon mt = \left(1-\frac{\varepsilon}{4}\right)\cdot 3mt + 3n^2 + 2n - \frac{\varepsilon}{4}mt.$$

The choice of t implies that $\varepsilon mt/4 > \varepsilon n(n+1)/2 + n^2$. Therefore

$$\begin{aligned} |V(T)| &< \left(1-\frac{\varepsilon}{4}\right)\cdot 3mt + 3n^2 + 2n - \frac{\varepsilon n(n+1)}{2} - n^2 \\ &< \left(1-\frac{\varepsilon}{4}\right)\cdot 3mt + \left(1-\frac{\varepsilon}{4}\right)\left(2n^2+2n\right) \\ &= \left(1-\frac{\varepsilon}{4}\right)(N-n^2), \end{aligned}$$

a contradiction. □

Lemma 6 *If G is 3-colorable, then every a-rooted maximum binary tree in $D(G,\varepsilon)$ has size exactly $N-n^2$.*

Proof We first note that every binary subtree of $D(G,\varepsilon)$ has size at most $N-n^2$. This is because there are n vertices with in-degree 3 (namely $v_1, v_2, \cdots, v_n$). For each such vertex v_i, there are 3 vertices $r_i^{(1)}, g_i^{(1)}$ and $b_i^{(1)}$ whose only outgoing arc is to v_i. Moreover, each vertex $r_i^{(1)}$ (and similarly $g_i^{(1)}$ and $b_i^{(1)}$) is the end-vertex of an induced path of length n.

Suppose G is 3-colorable. We now construct an a-rooted binary tree T of size $N-n^2$ in $D(G,\varepsilon)$. We focus on the vertices to be discarded so that we may construct a binary spanning tree with the remaining vertices. Let $\sigma : V \to \{Red, Green, Blue\}$ be a proper 3-coloring of G. If $\sigma(v_i) = Red$, we discard the path R_i. The cases where $\sigma(v_i) \in \{Green, Blue\}$ are similar. Since there are no monochromatic edges, there do not exist $e = \{v_i, v_j\} \in E$ and $C \in \{R,G,B\}$ such that both parents of a_e^C are not in T. Therefore every binary tree T_e^C is contained as a subtree in T. □

Theorem 7 *Suppose there is a PTAS for* DAGMaxBinaryTree *on DAGs, then for every $\varepsilon > 0$ there is a polynomial-time algorithm which takes as input an undirected 3-colorable graph G, and outputs a 3-coloring of G that properly colors at least $(1-\varepsilon)m$ edges.*

Proof Let $G = (V,E)$ be the given undirected 3-colorable graph. We construct $D(G,\varepsilon)$ in polynomial time. We note that the constructed graph $D(G,\varepsilon)$ is a directed acyclic graph. We now run the PTAS for DAGMaxBinaryTree on $D(G,\varepsilon)$ and root a to obtain a $(1-\varepsilon/4)$-approximate maximum binary tree in $D(G,\varepsilon)$. By Lemma 6

and the fact that G is 3-colorable, the PTAS will output an a-rooted binary tree T of size at least

$$\left(1-\frac{\varepsilon}{4}\right)(N-n^2).$$

We may assume that T is a maximal binary tree in $D(G,\varepsilon)$ (if not, then add more vertices to T until we cannot add any further). Maximality ensures that the vertices v_i are in the tree T and moreover, the in-degree of v_i in T is exactly 2. For each $v_i \in V$, let c_i be the unique vertex among $\left\{r_i^{(1)}, g_i^{(1)}, b_i^{(1)}\right\}$ that is not in T. We define a coloring $\sigma : V \to \{Red, Green, Blue\}$ of G as

$$\forall v_i \in V, \quad \sigma(v_i) = \begin{cases} Red & ifc_i = r_i^{(1)} \\ Green & ifc_i = g_i^{(1)} \\ Blue & ifc_i = b_i^{(1)}. \end{cases}$$

We now argue that the coloring is proper for at least $(1-\varepsilon)$-fraction of the edges of G. Suppose we have an edge $e = \{v_i, v_j\}$ which is monochromatic under σ, and suppose w.l.o.g. $\sigma(v_i) = \sigma(v_j) = Red$. This means that neither $r_i^{(1)}$ nor $r_j^{(1)}$ is included in T. Therefore $a_e^R \notin T$ since neither of the two vertices with incoming arcs from a_e^R are in T. By Lemma 5, we know that at most εm vertices among $\cup_{e\in E}\{a_e^R, a_e^G, a_e^B\}$ can be excluded from T. Hence, the coloring σ that we obtained can violate at most εm edges in G. □

Finally, we prove Theorem 1 using the self-improving argument (Theorem 6) and the APX-hardness of DAGMaxBinaryTree (Theorem 7).

Proof of Theorem 1

1. We observe that the graph G^2 constructed in Sect. 5 for the self-improving reduction is a DAG if G is a DAG. Therefore, by Theorem 6, a polynomial-time constant-factor approximation for DAGMaxBinaryTree would imply a PTAS for DAGMaxBinaryTree, a contradiction to APX-hardness shown in Theorem 7.
2. Next we show hardness under the Exponential Time Hypothesis. Suppose there is a polynomial-time algorithm $\mathcal{A}$ for DAGMaxBinaryTree that achieves an $\exp\left(-C \cdot \log_2 n / \log_2 \log_2 n\right)$-approximation for some constant $C > 0$. Given the input graph G with $n-1$ vertices, let k be an integer that satisfies

$$2^{\sqrt{n}} \le n^{2^k} \le 2^{2\sqrt{n}},$$

and run $\mathcal{A}$ on G^{2^k} to obtain a binary tree with size at least

$$\exp\left(-C \cdot \log_2 N / \log_2 \log_2 N\right) OPT\left(G^{2^k}\right),$$

where $N = n^{2^k}$ upper bounds the size of G^{2^k}. Recursively running the algorithm suggested in Theorem 6 k times gives us a binary tree in G with size at least

$$\exp\left(-C\cdot\frac{\log_2 N}{\log_2\log_2 N\cdot 2^k}\right)OPT(G)-1$$
$$\geq \exp\left(-C\cdot\frac{2\sqrt{n}}{\log_2\sqrt{n}}\cdot\frac{\log_2 n}{\sqrt{n}}\right)OPT(G)-1$$
$$\geq \exp(-4C)OPT(G)-1\geq\frac{1}{2}\cdot\exp(-4C)OPT(G).$$

The last inequality holds as long as

$$OPT(G)\geq 2\cdot e^{4C}.$$

We note that if $OPT(G)$ is smaller than $2e^{4C}$ which is a constant, we can solve the problem exactly by brute force in polynomial time. Otherwise the above procedure can be regarded as a constant-factor approximation for DAGMaxBinaryTree. The running time is polynomial in

$$N=n^{2^k}=\exp\left(O\left(\sqrt{n}\right)\right),$$

which is sub-exponential. Moreover, from item 1 we know that it is **NP**-hard to approximate DAGMaxBinaryTree within a constant factor, thus $\mathbf{NP}\subseteq\mathbf{DTIME}\left(\exp\left(O\left(\sqrt{n}\right)\right)\right)$.

3. The proof of this item is almost identical to the previous one except that we choose a different integer k. Suppose there is an algorithm $\mathcal{A}'$ for DAGMaxBinaryTree that achieves a $\exp\left(-C\cdot\log^{1-\epsilon}n\right)$-approximation for some constant $C>0$, and runs in time $\exp\left(O\left(\log^d n\right)\right)$ for some constant $d>0$. We show that there is an algorithm that achieves a constant-factor approximation for DAGMaxBinaryTree, and runs in time $\exp\left(O\left(\log^{d/\epsilon}n\right)\right)$. Given a DAG G on $n-1$ vertices as input for DAGMaxBinaryTree, let $k=\left\lceil\left(\frac{1}{\epsilon}-1\right)\log_2\log n\right\rceil$ be an integer that satisfies

$$\left(2^k\log n\right)^{1-\epsilon}\leq 2^k\leq 2(\log n)^{\frac{1}{\epsilon}-1}.$$

Running $\mathcal{A}'$ on G^{2^k} gives us a binary tree with size at least

$$\exp\left(-C\cdot\log^{1-\epsilon}N\right)OPT\left(G^{2^k}\right),$$

where $N=n^{2^k}$ upper bounds the size of G^{2^k}. Recursively running the algorithm suggested in Theorem 6 k times gives us a binary tree in G with size at least

$$\exp\left(-C\cdot\frac{\log^{1-\epsilon}N}{2^k}\right)OPT(G)-1$$
$$\geq \exp\left(-C\cdot\frac{\left(2^k\log n\right)^{1-\epsilon}}{2^k}\right)OPT(G)-1$$
$$\geq \exp(-C)OPT(G)-1\geq\frac{1}{2}\cdot\exp(-C)OPT(G).$$

The last inequality holds as long as

$$OPT(G)\geq 2\cdot e^C.$$

We note that if $OPT(G)$ is smaller than $2e^C$ which is a constant, we can solve the problem exactly by brute force in polynomial time. Otherwise the above procedure can be regarded as a constant-factor approximation for DAGMaxBinaryTree. The running time is quasi-polynomial in N, i.e. for some constant $C'>0$, the running time is upper-bounded by

$$\exp\left(C'\left(\log^d N\right)\right)=\exp\left(C'\left(\left(2^k\log n\right)^d\right)\right)\leq\exp\left(C'\left(\log^{d/\epsilon}n\right)\right).$$

□

3.3 An IP and Its Integrality Gap for DAGs

Let $G=(V,E)$ with root $r\in V$ be the input graph. We use indicator variables Y_u for the vertices $u\in V$ and X_e for the arcs $e\in E$ to determine the set of vertexs and arcs chosen in the solution. With these variables, the objective is to maximize the number of chosen vertices. Let $\delta^{out}(u)$ and $\delta^{in}(u)$ be the set of incoming and outgoing edges of u respectively. Constraints (2) ensure that each chosen vertex has at most two incoming arcs. Constraints (3) ensure that each chosen non-root vertex has an outgoing arc. Constraints (4) are cut constraints that ensure that every subset S of vertices containing a chosen vertex u but not the root has at least one outgoing arc.

$$\text{maximize}\quad\sum_{v\in V}Y_v \tag{1}$$

$$\text{subject to}\quad\sum_{e\in\delta^{in}(u)}X_e\leq 2Y_u\qquad\forall\, u\in V, \tag{2}$$

$$\sum_{e\in\delta^{out}(u)}X_e=Y_u\qquad\forall\, u\in V\setminus\{r\}, \tag{3}$$

$$\sum_{e\in\delta^{out}(S)}X_e\geq Y_u\forall\, u\in S\subset V\setminus\{r\}, \tag{4}$$

$$0 \leq Y_u \quad \leq 1 \forall\, u \in V, \tag{5}$$

$$0 \leq X_e \quad \leq 1 \forall\, e \in E, \tag{6}$$

$$Y \in \mathbb{Z}^{|V|}, X \quad \in \mathbb{Z}^{|E|}. \tag{7}$$

We note that a similar IP formulation can also be written for the longest $s \to t$ path problem by replacing the factor 2 in the RHS of (2) with a factor of 1. It can be shown that extreme point solutions for the LP-relaxation of such an IP are in fact integral. Owing to the similarity between the longest $s \to t$ path problem in DAGs and DAGMaxBinaryTree (as degree bounded maximum subtree problems), it might be tempting to conjecture that LP-based techniques might be helpful for DAGMaxBinaryTree. However, in contrast to the LP-relaxation for longest $s \to t$ path problem in DAGs (which is integral), the LP-relaxation of the above IP (by removing Constraints 7) for DAGMaxBinaryTree has very large integrality gap.

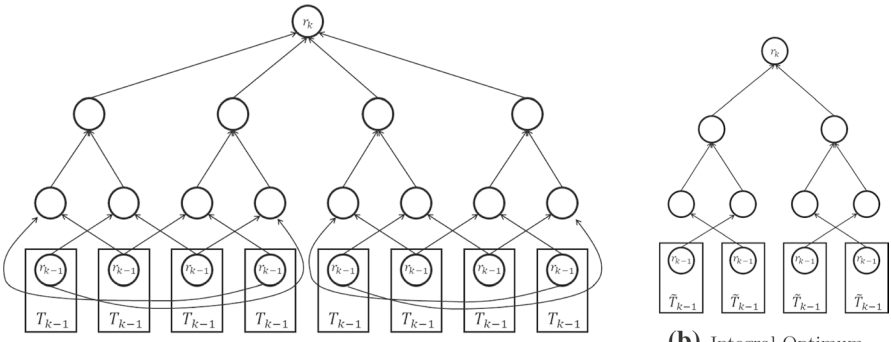
Theorem 8 *The integrality gap of the LP-relaxation of the above IP, even in DAGs, is* $\Omega(n^{1/3})$, *where n is the number of vertices in the input DAG.*

Proof We construct an integrality gap graph T_k recursively as shown in Fig. 3a with the base graph T_1 being a single vertex labeled r_1. We will denote the root vertices of $T_1, T_2, \ldots, T_{k-1}, T_k$ to be *special* vertices. The layered construction and the direction of the arcs illustrate that the graph T_n is a DAG. The number V_k of vertices in the graph T_k satisfies the recursion

$$V_k = 8V_{k-1} + 13$$

with $V_1 = 1$. Thus, $V_k = (13/7)(8^{k-1} - 1)$.

Due to the degree constraints, the optimal integral solution $\tilde{T}_k$ is obtained using a recursive construction as shown in Fig. 3b with the base graph $\tilde{T}_1 = T_1$. The number of arcs in the optimal integral solution satisfies the recursion



(a) Integrality Gap Graph T_k

(b) Integral Optimum Solution: $\tilde{T}_k$

Fig. 3 DAG integrality gap

$$\text{IP-OPT}(k) = 4\text{IP-OPT}(k-1) + 7$$

with IP-OPT(1) = 0. Thus, IP-OPT(k) = (7/3)($4^{k-1} - 1$).

In order to show large integrality gap, we give an LP-feasible solution with large objective value. The LP-feasible solution that we consider is $X_e := 1/2$ for every arc e in the graph with

$$Y_u := \begin{cases} 1 \text{ if u is a special vertex,} \\ \frac{1}{2} \text{ otherwise.} \end{cases}$$

We now argue that this solution satisfies all constraints of the LP-relaxation. The in-degree and out-degree constraints hold by definition. We now show that the cut constraints, i.e., constraints (4), are satisfied. We have two cases:

Case 1. Suppose u is a non-special vertex. For every non-special vertex u, we have a path from u to the root r. So, every cut S containing u but not r has an arc leaving it and hence $\sum_{e \in \delta^{out}(S)} X_e \geq 1/2 = Y_u$.

Case 2. Suppose u is a special verteex. For every special vertex u, we have two arc-disjoint paths from u to the root r. So, every cut S containing u but not r has at least 2 arcs leaving it and hence $\sum_{e \in \delta^{out}(S)} X_e \geq 1 = Y_u$.

The objective value of this LP-feasible solution satisfies the recursion

$$\text{LP-obj}(k) = 8\text{LP-obj}(k-1) + 14$$

with LP-obj(1) = 0. Thus, LP-obj(k) = 2($8^{k-1} - 1$). Consequently, the integrality gap of the LP for instance T_k is $\Omega(2^{k-1}) = \Omega(V_k^{1/3})$. □

4 Hardness Results for Undirected Graphs

We show the inapproximability of finding a maximum binary tree in undirected graphs. We use $OPT(G)$ to denote the size (number of vertices) of a maximum binary tree in G.

4.1 Self-improvability

This section is devoted to proving the following theorem.

Theorem 9 *If* UndirMaxBinaryTree *has a polynomial-time algorithm that achieves a constant-factor approximation, then it has a PTAS.*

In the previous section we showed an analogous self-improvability for rooted-DirMaxBinaryTree (Theorem 6) by defining a squared graph and gradually boosting up the approximation ratio by running the constant-factor approximation algorithm on squared graphs. We use the same high-level strategy to prove Theorem 9. We need a different notion of a squared graph, which we formally introduce next.

Definition 2 For an undirected graph G, its squared graph $G^{\boxtimes 2}$ is defined as the graph obtained by performing the following operations in G.

1. Replace each edge $\{u, v\} \in E(G)$ with a copy $G_{u,v}$ of G. Connect u and v to all vertices in $G_{u,v}$. We will refer to these copies as *edge* copies.
2. For each vertex $v \in V(G)$, introduce two copies of G denoted by $G_v^{(1)}$ and $G_v^{(2)}$, and connect v to all vertices in $G_v^{(1)}$ and $G_v^{(2)}$. We will refer to these copies as *pendant* copies.

We will use $V(G)$ to denote the original vertices of G and the same vertices in the graph $G^{\boxtimes 2}$ (see Fig. 4 for an example). We will refer to vertices from vertex copies, edge copies, and pendant copies as nodes. We define $G^{\boxtimes 2^{k+1}}$ recursively as $G^{\boxtimes 2^{k+1}} := \left(G^{\boxtimes 2^k}\right)^{\boxtimes 2}$ with the base case $G^{\boxtimes 1} := G$.

Given an undirected graph G with n vertices, the number of vertices in $G^{\boxtimes 2^k}$ is upper bounded by n^{3^k} since at most $|E(G)| + 2|V(G)| \leq n^2$ copies of G are introduced in $G^{\boxtimes 2}$ when $n \geq 3$.

For a binary tree T and $d \in \{0, 1, 2, 3\}$, let $I_d(T) \subseteq V(T)$ be the set of vertices with degree exactly d. The following lemma is our main tool in the reduction.

Lemma 7 *For every non-empty binary tree T, we have*

$$3|I_0(T)| + 2|I_1(T)| + |I_2(T)| = |V(T)| + 2,$$

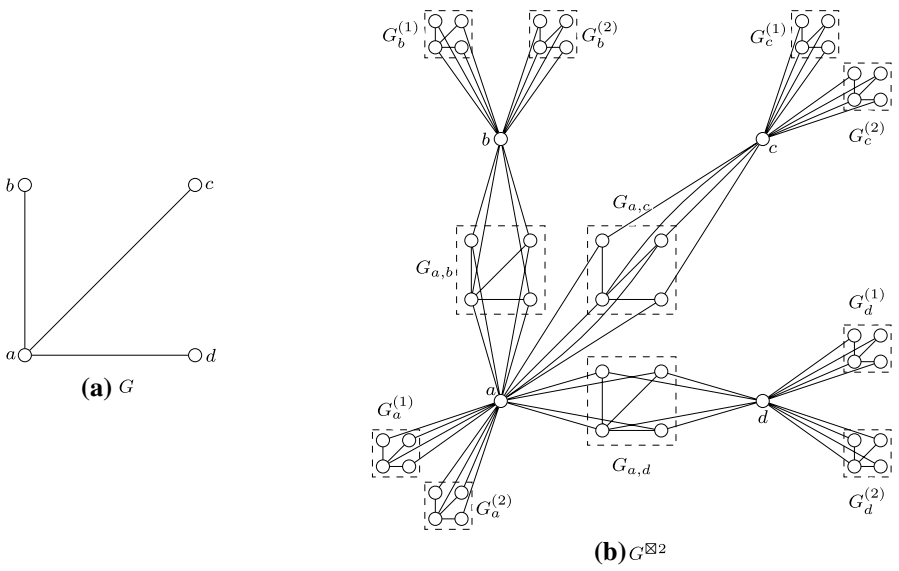


Fig. 4 A graph G and its squared graph $G^{\boxtimes 2}$

Proof We prove by induction on $|V(T)|$. When $|V(T)| = 1$, the tree T has one degree-0 vertex. Therefore $|I_0(T)| = 1$ and $|I_1(T)| = |I_2(T)| = 0$, thus $3|I_0(T)| + 2|I_0(T)| + |I_1(T)| = 3 = |V(T)| + 2$ holds.

Suppose that the statement holds for all binary trees with $t - 1$ vertices for $t \geq 2$. Let T be a binary tree with t vertices. When $t \geq 2$ there are no degree-0 vertices, so we only need to verify that $2|I_1(T)| + |I_2(T)| = |V(T)| + 2$. Let ℓ be an arbitrary leaf node in T and p be its unique neighbor. Then $\deg_T(\ell) = 1$. Removing ℓ results in a binary tree T' with $t - 1$ vertices. We have the following cases:

1. If $p \in I_3(T)$, then $I_1(T') = I_1(T) \setminus \{\ell\}$ and $I_2(T') = I_2(T) \cup \{p\}$.
2. If $p \in I_2(T)$, then $I_1(T') = \big(I_1(T) \setminus \{\ell\}\big) \cup \{p\}$ and $I_2(T') = I_2(T) \setminus \{p\}$.
3. If $p \in I_1(T)$, then $I_0(T') = \{p\}$, $I_1(T') = I_1(T) \setminus \{p, \ell\}$ and $I_2(T') = I_2(T)$.

In all cases, we have

$$2|I_1(T)| + |I_2(T)| = 3|I_0(T')| + 2|I_1(T')| + |I_2(T')| + 1 = |V(T')| + 3 = t + 2$$

where the second equality is due to the induction hypothesis. This completes the inductive step. □

Remark 1 Lemma 7 can be generalized to any non-empty tree T with maximum degree k to conclude that $\sum_{i=0}^{k-1}(k-i)|I_i(T)| = (k-2)|V(T)| + 2$, where $I_d(T) \subseteq V(T)$ is the set of vertices with degree exactly d. This can be proved by induction on $|V(T)|$ in a very similar fashion to the proof of Lemma 7.

The next lemma shows that $OPT(G)$ is super-multiplicative under the squaring operation.

Lemma 8 $OPT\big(G^{\boxtimes 2}\big) \geq 2OPT(G)^2 + 2OPT(G)$.

Proof Let T_1 be an optimal binary tree in G, i.e. $|V(T_1)| = OPT(G)$. We construct a binary tree T_2 in $G^{\boxtimes 2}$ as follows:

1. For $\{u, v\} \in E(G)$, let $T_{u,v}$ be the optimal binary tree (identical to T_1) in the edge copy $G_{u,v}$. For $v \in V(G)$ and $i \in \{1, 2\}$, let $T_v^{(i)}$ be the optimal binary tree in the pendant copy $G_v^{(i)}$.
2. For every edge $\{u, v\} \in T_1$, add $T_{u,v}$ into T_2 along with two edges $\{u, \ell\}$ and $\{v, \ell\}$ where ℓ is an arbitrary leaf node in $T_{u,v}$. This step generates $|V(T_1)| + |V(T_1)| \cdot (|V(T_1)| - 1) = |V(T_1)|^2$ vertices in T_2 of which $|V(T_1)|(|V(T_1)| - 1)$ are nodes, since the number of edges in $E(T_1)$ is $|V(T_1)| - 1$.
3. For $v \in I_1(T_1)$, add both $T_v^{(1)}$ and $T_v^{(2)}$ to T_2 by connecting v to $\ell^{(1)}$ and $\ell^{(2)}$, where $\ell^{(1)}$ and $\ell^{(2)}$ are arbitrary leaf nodes in $T_v^{(1)}$ and $T_v^{(2)}$, respectively. For $u \in I_2(T_1)$, add only $T_u^{(1)}$ to T_2 by connecting u to a leaf node in $T_u^{(1)}$. By Lemma 7, this step generates $|V(T_1)| \cdot (|V(T_1)| + 2)$ nodes in T_2.

Since T_1 is a binary tree, it follows that T_2 is a binary tree. Moreover, the size of $V(T_2)$ is

$$|V(T_2)| = |V(T_1)|^2 + |V(T_1)| \cdot (|V(T_1)| + 2) = 2OPT(G)^2 + 2OPT(G),$$

which cannot exceed $OPT\left(G^{\boxtimes 2}\right)$. □

The next two lemmas show that a large binary tree in $G^{\boxtimes 2}$ can be used to obtain a large binary tree in G.

Lemma 9 *Given T_2 as a binary tree in $G^{\boxtimes 2}$, there is a linear-time (in the size of $G^{\boxtimes 2}$) algorithm that finds a binary tree T_1' in G with vertex set $V(T_2) \cap V(G)$.*

Proof Given a binary tree T_2 in the squared graph $G^{\boxtimes 2}$, the algorithm finds a binary tree T_1' in G by going through every edge $\{u, v\} \in E(G)$ and adding it to T_1' whenever the unique path from u to v in T_2 goes through the edge copy $G_{u,v}$. We discard the edge $\{u, v\}$ if there does not exist a path through $G_{u,v}$ connecting u and v in T_2.

By construction, the subgraph returned by the algorithm has maximum degree 3 and is acyclic. Moreover, it is connected since the path between any two vertices $u, v \in T_2$ is preserved in T_1'. Therefore, T_1' is a binary tree in G with vertex set $V(T_2) \cap V(G)$. □

Lemma 10 *For every $\alpha \in (0, 1]$, given a binary tree T_2 in $G^{\boxtimes 2}$ with size*

$$|V(T_2)| \geq \alpha OPT\left(G^{\boxtimes 2}\right) - 1,$$

there is a linear-time (in the size of $G^{\boxtimes 2}$) algorithm that finds a binary tree T_1 in G with size

$$|V(T_1)| \geq \sqrt{\alpha} OPT(G) - 1.$$

Proof Running the algorithm suggested in Lemma 9 gives us a binary tree T_1' in G with vertex set $V(T_1') = V(T_2) \cap V(G)$ in linear time. Therefore if $|V(T_2) \cap V(G)| \geq \sqrt{\alpha} OPT(G) > \sqrt{\alpha} OPT(G) - 1$ then the lemma is already proved. So, we may assume that $|V(T_2) \cap V(G)| < \sqrt{\alpha} OPT(G)$.

Let us first deal with the case when $V(T_2) \cap V(G) = \emptyset$. In this case T_2 completely resides within some edge copy or pendant copy of $G^{\boxtimes 2}$. That means T_2 is already a binary tree in G with size

$$\left|V(T_2)\right| \geq \alpha OPT\left(G^{\boxtimes 2}\right) - 1 \geq \left(\sqrt{\alpha} OPT(G)\right)^2 - 1 \geq \sqrt{\alpha} OPT(G) - 1,$$

where the second inequality uses Lemma 8 and the third inequality assumes $\sqrt{\alpha} OPT(G) \geq 1$.

Suppose T_1' is not the empty tree. Removing vertices in $V(G)$ from T_2 results in a forest with each connected component of the forest residing completely within one of the copies of G (either edge or pendent copy). Let $\mathcal{F}$ be the set of trees in this forest.

Claim *We have*

$$|\mathcal{F}| < 2\sqrt{\alpha}OPT(G) + 1.$$

Proof Each tree $T^{(j)} \in \mathcal{F}$ is connected to one or two vertices in $V(T_2) \cap V(G)$. Let $\mathcal{F}_j$ be the set of trees connected to exactly j vertices in $V(G) \cap V(T_2)$. Then $\mathcal{F} = \mathcal{F}_2 \cup \mathcal{F}_1$. We now bound the size of $\mathcal{F}_2$ and $\mathcal{F}_1$. If $T^{(j)}$ is connected to two vertices $u, v \in V(G) \cap V(T_2)$, then $\{u, v\}$ must be an edge in T'_1. Moreover, if there are two distinct trees $T^{(j_1)}$, $T^{(j_2)}$ both connected to $u, v \in V(G) \cap V(T_2)$ then there will be a cycle. Therefore $|\mathcal{F}_2| \le |E(T'_1)| = |V(T'_1)| - 1$. As to $\mathcal{F}_1$, for $j \in \{0, 1, 2, 3\}$, every vertex $v \in I_j(T'_1)$ is connected to at most $3 - j$ trees in $\mathcal{F}_1$. Every vertex $v \in V(G) \setminus V(T'_1)$ is not connected to any tree in $\mathcal{F}_1$. Therefore $|\mathcal{F}_1| \le 3|I_0(T'_1)| + 2|I_1(T'_1)| + |I_2(T'_1)|$. This gives an upper bound on $|\mathcal{F}|$.

$$\begin{aligned}|\mathcal{F}| = |\mathcal{F}_1| + |\mathcal{F}_2| &\le |V(T'_1)| - 1 + 3|I_0(T'_1)| + 2|I_1(T'_1)| + |I_2(T'_1)| \\ &= 2|V(T'_1)| + 1 = 2|V(T_2) \cap V(G)| + 1 < 2\sqrt{\alpha}OPT(G) + 1,\end{aligned}$$

where the second equality is due to Lemma 7. □

Claim *There exists a tree $T^* \in \mathcal{F}$ with size at least*

$$|V(T^*)| \ge \sqrt{\alpha}OPT(G) - 1.$$

Proof Suppose not. Then every tree $T \in \mathcal{F}$ has $|V(T)| < \sqrt{\alpha}OPT(G) - 1$. Then,

$$\begin{aligned}|V(T_2)| &= |V(T_2) \cap V(G)| + \sum_{T^{(j)} \in \mathcal{F}} \left|V(T^{(j)})\right| \\ &< \sqrt{\alpha}OPT(G) + \left(\sqrt{\alpha}OPT(G) - 1\right)\left(2\sqrt{\alpha}OPT(G) + 1\right) \qquad \text{(By Claim 4.1)} \\ &= 2\alpha OPT(G)^2 - 1 \\ &< \alpha\left(2OPT(G)^2 + 2OPT(G)\right) - 1 \\ &\le \alpha OPT\left(G^{\boxtimes 2}\right) - 1\end{aligned}$$

which is a contradiction. The last inequality here is due to Lemma 8. □

Since T^* resides in one of the copies of G, it is a binary tree in G. A DFS would find T^* in linear time. □

Now we are ready prove the main theorem of this subsection. Our proof strategy is similar to that of Theorem 6.

Proof of Theorem 9 Suppose that we have a polynomial-time algorithm $\mathcal{A}$ that achieves an α-approximation for UndirMaxBinaryTree. Given an undirected graph G and $\varepsilon > 0$, let

$$k := 1 + \left\lceil \log_2 \frac{\log_2 \alpha}{\log_2(1 - \varepsilon)} \right\rceil$$

be an integer constant that depends on α and ε. We construct $G^{\boxtimes 2^k}$ and run algorithm $\mathcal{A}$ on it. We get a binary tree in $G^{\boxtimes 2^k}$ of size at least $\alpha OPT\left(G^{\boxtimes 2^k}\right) - 1$. By Lemma 10, we can obtain a binary tree in G of size at least

$$\alpha^{2^{-k}} OPT(G) - 1 \geq \alpha^{2^{-k+1}} OPT(G) \geq (1-\varepsilon) OPT(G).$$

The first inequality holds as long as

$$OPT(G) \geq \frac{1}{\sqrt{1-\varepsilon} - (1-\varepsilon)} \geq \frac{1}{\alpha^{2^{-k}} - \alpha^{2^{-k+1}}}.$$

We note that if $OPT(G)$ is smaller than $1/\left(\sqrt{1-\varepsilon} - (1-\varepsilon)\right)$ which is a constant, then we can solve the problem exactly by brute force in polynomial time. Finally, we also observe that for fixed constant ε, the running time of this algorithm is polynomial since there are at most $n^{3^k} = n^{O(1)}$ vertices in the graph $G^{\boxtimes 2^k}$. □

4.2 APX-Hardness

In this section, we show that UndirMaxBinaryTree is APX-hard. The reduction is from the following problem, denoted as TSP(1,2).

TSP(1,2)

Given: A complete undirected graph K_n with edge weights w_{ij} where $w_{ij} \in \{1,2\}\ \forall i,j \in [n]$.

Goal: A tour with minimum weight which starts and finishes at the same vertex and visits every other vertex exactly once.

For an instance $K_n = ([n], E_1 \cup E_2)$ of TSP(1,2) where E_1 is the set of edges with weight 1, and E_2 is the set of edges with weight 2, define two subgraphs $S_1 = ([n], E_1)$ and $S_2 = ([n], E_2)$. It is convenient to think of S_1 and S_2 as unweighted graphs.

Theorem 10 ([32], Theorem 9 of [23]) TSP(1,2) *is APX-hard even on instances with optimal value n, i.e. instances whose associated subgraph S_1 has a Hamiltonian cycle.*

The following lemma shows that a binary tree in K_n with a small number of degree-3 vertices can be transformed into a path without too much increase in total weight.

Lemma 11 *Let K_n be a weighted complete graph with edge weights in $\{1,2\}$. Let T be a binary tree in K_n with ℓ vertices and let w be the sum of the edge weights of T. If at most d vertices in T have degree 3, then in linear time we can find a path in K_n with length ℓ and total weight at most $w + d$.*

Proof Pick an arbitrary root vertex r with $\deg_T(r) \leq 2$ and consider the r-rooted tree T. Perform the following operation on T bottom-up to convert every vertex of degree-3 in T into a vertex of degree-2.

1. Find a vertex $v \in T$ with degree 3 such that the two subtrees of v are paths. Suppose that the path on the left is $(a_1, a_2, \cdots, a_p)$ and the path on the right is $(b_1, b_2, \cdots, b_q)$ with a_1 and b_1 being adjacent to v in T.
2. Merge the two paths into a longer path $(a_1, \cdots, a_p, b_1, b_2, \cdots, b_q)$ by replacing edge (v, b_1) with (a_p, b_1).

This operation converts v into a degree-2 vertex and increases the total weight of T by at most 1 (in case of $w(v, b_1) = 1$ and $w(a_p, b_1) = 2$). Since no new degree-3 vertices are introduced during the operation, the total number of degree-3 vertices decreases by one. The path claimed by the lemma is thus obtained by recursively performing this operation d times. We note that the final path is effectively a post-order traversal of T. □

We also need the following structural result.

Lemma 12 *Let $H = (V, E)$ be a graph with n vertices and let $\tilde{H} = (V \cup V', E \cup E')$ where $V' := \{v' : v \in V\}$ and $E' := \{\{v, v'\} : v \in V\}$. Suppose we have a binary tree $\tilde{T}$ in $\tilde{H}$ of size at least $(2 - \varepsilon)n$. Then, the graph T obtained by restricting $\tilde{T}$ to H is a binary tree with at most εn vertices of degree* 3.

Proof Let us denote the set V' of added vertices as pendants. Since the pendants have degree 1 in $\tilde{H}$, the restricted graph T is a binary tree. We now show that the number of vertices in T with degree 3 is small. We note that the number of vertices in $\tilde{H}$ is $2n$ and hence $\tilde{T}$ contains all but at most εn vertices of $\tilde{H}$. For every vertex v with $\deg_T(v) = 3$, its pendant v' is not in $\tilde{T}$. Thus, in order for $\tilde{T}$ to contain all but εn vertices of $\tilde{H}$, the number of vertices of degree 3 in T cannot exceed εn. □

The following lemma implies that if there is a PTAS for UndirMaxBinaryTree then there is a PTAS for TSP(1,2).

Lemma 13 *Suppose there is a PTAS for* UndirMaxBinaryTree (*even restricted to graphs that contain binary spanning trees*), *then for every $\varepsilon > 0$ there is a polynomial-time algorithm which takes as input an undirected complete graph K_n with edge weights in $\{1, 2\}$ whose associated subgraph S_1 has a Hamiltonian cycle to output a tour with weight at most $(1 + \varepsilon)n$.*

Proof Let K_n be the input instance of TSP(1,2) with S_1 and S_2 defined as above. Let $\tilde{S}_1$ be the graph constructed from S_1 as follows: For every $i \in [n]$, introduce a new vertex v_i adjacent to vertex i in S_1. We will refer to v_i as the pendant of vertex i.

We note that $\tilde{S}_1$ has a spanning binary tree (i.e. $OPT(\tilde{S}_1) = 2n$) because S_1 has a Hamiltonian cycle. Therefore we can run the PTAS for UndirMaxBinaryTree on

graph $\tilde{S}_1$ with error parameter $\varepsilon/4$ (which still takes polynomial time). The PTAS would output a binary tree $\tilde{T}$ of size at least

$$\left(1-\frac{\varepsilon}{4}\right)\cdot OPT(\tilde{S}_1)=\left(1-\frac{\varepsilon}{4}\right)\cdot 2n=2n-\frac{\varepsilon}{2}n,$$

By Lemma 12, we obtain a binary tree T in S_1 with the following properties:

1. T contains at least $2n-\varepsilon n/2-n=(1-\varepsilon/2)n$ vertices and
2. T has at most $\varepsilon n/2$ vertices with degree 3.

By Lemma 11 we can transform T into a path in K_n that has total weight at most $(1-\varepsilon/2)n+\varepsilon n/2=n$ and contains at least $(1-\varepsilon/2)n$ vertices. This path can be extended to a valid tour by including the missing vertices using edges with weight 2. Such a tour will have total weight at most $n+2(\varepsilon n/2)=(1+\varepsilon)n$. □

Now we are ready to prove Theorem 2.

Theorem 2 *We have the following inapproximability results for* DAGMaxBinaryTree *on n-vertex input graphs*:

1. DAGMaxBinaryTree *does not admit a polynomial-time constant-factor approximation assuming* $\mathbf{P}\neq\mathbf{NP}$.
2. *If* DAGMaxBinaryTree *admits a polynomial-time* $\exp\left(-O(\log n/\log\log n)\right)$*-approximation, then* $\mathbf{NP}\subseteq\mathbf{DTIME}\left(\exp\left(O\left(\sqrt{n}\right)\right)\right)$*, refuting the Exponential Time Hypothesis*.
3. *For any* $\varepsilon>0$*, if* DAGMaxBinaryTree *admits a quasi-polynomial time* $\exp\left(-O\left(\log^{1-\epsilon} n\right)\right)$*-approximation, then* $\mathbf{NP}\subseteq\mathbf{DTIME}\left(\exp\left(\log^{O(1/\epsilon)} n\right)\right)$*, thus refuting the Exponential Time Hypothesis*.

Proof

1. Suppose there is a polynomial-time algorithm for UndirMaxBinaryTree that achieves a constant-factor approximation. By Theorem 9, the problem also has a PTAS. By Lemma 13, TSP(1,2) would also have a PTAS, thus contradicting its APX-hardness. Therefore UndirMaxBinaryTree does not admit a polynomial-time constant-factor approximation assuming $\mathbf{P}\neq\mathbf{NP}$.
2. Next we show hardness under the exponential time hypothesis. Suppose there is a polynomial-time algorithm $\mathcal{A}$ for UndirMaxBinaryTree that achieves an $\exp\left(-C\cdot(\log n)^{\log_3 2-\varepsilon}\right)$-approximation for constants $C,\varepsilon>0$. We show that there is an algorithm that achieves a constant-factor approximation for UndirMaxBinaryTree, and runs in time $\exp\left(O\left((\log n)^{\epsilon^{-1}\log_3 2}\right)\right)$. Given an undirected graph G on n vertices as input for UndirMaxBinaryTree, let $k=\left\lceil\left(\varepsilon^{-1}\log_3 2-1\right)\log_3\log n\right\rceil$, which is an integer that satisfies

$$\left(3^k \log n\right)^{\log_3 2-\varepsilon} \leq 2^k \text{ and } 3^k \leq 3(\log n)^{\varepsilon^{-1}\log_3 2-1}.$$

Running $\mathcal{A}$ on $G^{\boxtimes 2^k}$ gives us a binary tree with size at least

$$\exp\left(-C \cdot (\log N)^{\log_3 2-\varepsilon}\right) OPT\left(G^{\boxtimes 2^k}\right)$$

where $N \leq n^{3^k}$ is the size of $G^{\boxtimes 2^k}$. Recursively applying Lemma 10 gives us a binary tree in G with number of vertices being at least

$$\begin{aligned}
&\exp\left(-C \cdot \frac{(\log N)^{\log_3 2-\varepsilon}}{2^k}\right) OPT(G) - 1 \\
&\geq \exp\left(-C \cdot \frac{(3^k \log n)^{\log_3 2-\varepsilon}}{2^k}\right) OPT(G) - 1 \\
&\geq \exp(-C) OPT(G) - 1 \geq \frac{1}{2} \cdot \exp(-C) OPT(G).
\end{aligned}$$

The last inequality holds as long as

$$OPT(G) \geq 2 \cdot e^C.$$

We note that when $OPT(G)$ is smaller than $2e^C$ which is a constant, we can solve the problem exactly by brute force in polynomial time. Otherwise the above procedure can be regarded as a constant-factor approximation for UndirMaxBinaryTree. The running time is quasi-polynomial in N, i.e. for some constant $C' > 0$, the running time is upper-bounded by

$$\exp\left(C'\left(\log^d N\right)\right) \leq \exp\left(C'\left(\left(3^k \cdot \log n\right)^d\right)\right) \leq \exp\left(C'\left((\log n)^{\varepsilon^{-1} d \log_3 2}\right)\right).$$

Moreover, from item 1 we know that it is **NP**-hard to approximate UndirMaxBinaryTree within a constant factor, therefore $\mathbf{NP} \subseteq \mathbf{DTIME}\left(\exp\left(O\left((\log n)^{\varepsilon^{-1} d \log_3 2}\right)\right)\right)$.

□

We remark that APX-hardness for UndirMaxBinaryTree on graphs with spanning binary trees does not rule out constant-factor approximation algorithms on such instances. This is because our squaring operation might lose spanning binary trees ($G^{\boxtimes 2}$ does not necessarily contain a spanning binary tree even if G does).

5 An Efficient Algorithm for Bipartite Permutation Graphs

In this section we prove Theorem 4. We begin with some structural properties of bipartite permutation graphs that will be helpful in designing the algorithm.

5.1 Structural Properties of Bipartite Permutation Graphs

Definition 3 A *strong ordering* σ of a bipartite graph $G = (S, T, E)$ is an ordering of S and an ordering of T such that

$$\forall s <_\sigma s' \in S, t <_\sigma t' \in T, \quad (s,t') \in E \text{and} (s',t) \in E \implies (s,t) \in E \text{and} (s',t') \in E.$$

Informally, strong ordering essentially states that the existence of *cross edges* implies the existence of *parallel edges*. The following theorem from [37] shows that strong ordering exactly characterizes bipartite permutation graphs.

Theorem 11 (Theorem 1 of [37]) *A bipartite graph $G = (S, T, E)$ is also a permutation graph if and only if G has a strong ordering.*

Corollary 1 *Let G be a bipartite permutation graph and $H = G[V_H]$ be an induced subgraph. Then H is also a bipartite permutation graph.*

Proof Let σ be a strong ordering of G. The corollary follows by applying Theorem 11 and observing that the projection of σ onto V_H is a strong ordering of H. □

In the following, when we are given a bipartite permutation graph $G = (S, T, E)$ along with a strong ordering σ (or simply a strongly ordered bipartite permutation graph), we always assume that the elements in S and T are sorted in ascending order according to σ:

$$s_1 <_\sigma s_2 <_\sigma \cdots <_\sigma s_p, \quad t_1 <_\sigma t_2 <_\sigma \cdots <_\sigma t_q.$$

Here $p := |S|$ and $q := |T|$.

The following lemma shows that in a bipartite permutation graph the neighborhood of a vertex $v \in G$ has a nice consecutive structure.

Lemma 14 *Let $G = (S, T, E)$ be a connected bipartite permutation graph and σ be a strong ordering of G. For every $s_i \in S$, there exist $a_i \le b_i \in [q]$ such that*

$$N(s_i) = [t_{a_i}, t_{b_i}] := \{t_{a_i}, t_{a_i+1}, \cdots, t_{b_i-1}, t_{b_i}\}.$$

Moreover, for any $s_i, s_j \in S$ such that $s_i <_\sigma s_j$ we have

$$a_i \le a_j, \quad b_i \le b_j.$$

Proof For the first part, let $s \in S$ be an arbitrary vertex and let t_1, t_2 be the smallest and largest elements in $N(s)$ (with respect to σ), respectively. Consider any $t \in T$ satisfying $t_1 <_\sigma t <_\sigma t_2$. We want to show that $t \in N(s)$. Since G is connected, there must be some $s' \in S$ adjacent to t. Suppose $s' <_\sigma s$. Since σ is a strong ordering, we have

$$(s', t_2) \in E \text{and} (s,t) \in E \implies (s,t) \in E.$$

A symmetric argument holds for the case $s <_\sigma s'$. Therefore $t \in N(s)$.

For the second part, suppose for the sake of contradiction that $a_j < a_i$. Recall that $s_i <_\sigma s_j$, we have

$$(s_i, t_{a_i}) \in E and (s_j, t_{a_j}) \in E \implies (s_i, t_{a_j}) \in E.$$

That means $t_{a_j} \in N(s_i) = [t_{a_i}, t_{b_i}]$, contradicting with $a_j < a_i$. A symmetric argument can be used to prove $b_i \leq b_j$. □

Another important property of (connected) bipartite permutation graphs is that they contain crossing-free spanning trees.

Definition 4 Given a bipartite permutation graph $G = (S, T, E)$ and a strong ordering σ of G, we say a subgraph H has an *edge crossing* (w.r.t. the strong ordering σ) if H contains two edges (s_1, t_1) and (s_2, t_2) such that $s_1 <_\sigma s_2$ and $t_2 <_\sigma t_1$. Otherwise we say H is *crossing-free*.

We need the following theorem from [36].

Theorem 12 (Corollary 4.19 of [36]) *Let G be a strongly ordered connected bipartite permutation graph. There exists a minimum degree spanning tree (MDST) of G which is crossing-free.*

Lemma 15 *Let G be a strongly ordered bipartite permutation graph. There exists a maximum binary tree in G which is crossing-free.*

Proof Consider any maximum binary tree $H = (V_H, E_H)$ in G and the induced subgraph $G[V_H]$. By Corollary 1 we have that $G[V_H]$ is a bipartite permutation graph. Moreover, $G[V_H]$ contains a spanning binary tree and is thus connected. By Theorem 12 there is a crossing-free MDST of $G[V_H]$, which we will denote by H'. We note that H' is a binary tree, and that H' has the same size as H since they are both spanning trees of $G[V_H]$. Therefore H' is a maximum binary tree in G which is crossing-free. □

The next lemma inspires the definition of subproblems which lead us to the Dynamic Programming based algorithm.

Lemma 16 *Let $G = (S, T, E)$ be a strongly ordered bipartite permutation graph, and let $H = (V_H, E_H)$ be a connected crossing-free subgraph of G. Let s_1 and t_1 be the two minimum vertices (w.r.t. the strong ordering) in $S \cap V_H$ and $T \cap V_H$, respectively. Then we have $\{s_1, t_1\} \in E_H$, and that one of s_1 and t_1 has degree 1.*

Proof Suppose for the sake of contradiction that $\{s_1, t_1\} \notin E_H$. Since H is connected, there exists a path $(s_1, t_2, \cdots, s_2, t_1)$ where $t_2 >_\sigma t_1$ and $s_2 >_\sigma s_1$. However, the two edges $\{s_1, t_2\}$ and $\{s_2, t_1\}$ constitute an edge crossing which is a contradiction to the assumption that H is crossing-free.

We proved that $\{s_1, t_1\} \in E_H$. Suppose both s_1 and t_1 have at least one more neighbors, say t_2 and s_2, respectively, then once more $\{s_1, t_2\}$ and $\{s_2, t_1\}$ constitute an edge crossing. Therefore one of s_1 and t_1 has degree 1. □

5.2 The Algorithm

In this section, we give a dynamic programming approach for solving UndirMax-BinaryTree on bipartite permutation graphs. We first focus on connected, strongly ordered bipartite permutation graphs. Theorem 4 will follow from the fact that a strong ordering can be found in linear time.

Let $G = (S, T, E)$ be a strongly ordered bipartite permutation graph with $|S| = p$ and $|T| = q$. For $i \in [p], j \in [q]$, define

$$S_i := \{s_i, s_{i+1}, \cdots, s_p\}, \quad T_j := \{t_j, t_{j+1}, \cdots, t_q\}.$$

We also use the convention $S_{p+1} = T_{q+1} = \emptyset$. Define $[i, j] := G[S_i \cup T_j]$, i.e. the subgraph of G induced by $S_i \cup T_j$.

For $i \in [p]$ and $j \in [q]$, let $\mathsf{MBT}_S(i, j)$ be the maximum cardinality (number of edges) of a binary tree H in $[i, j]$ such that

1. H is crossing-free,
2. $\{s_i, t_j\} \in E_H$,
3. s_i is a leaf vertex in H.

$\mathsf{MBT}_T(i, j)$ is similarly defined except that in the last constraint we require t_j (instead of s_i) to be a leaf vertex in H. Finally let

$$\mathsf{MBT}(G) := \max_{i \in [p], j \in [q]} \{\max \{\mathsf{MBT}_S(i, j), \mathsf{MBT}_T(i, j)\}\}.$$

Lemma 17 *Let $G = (S, T, E)$ be a strongly ordered bipartite permutation graph. Let $\mathsf{OPT}(G)$ be the cardinality of the maximum binary tree in G. Then $\mathsf{MBT}(G) = \mathsf{OPT}(G)$.*

Proof Since it is trivial that $\mathsf{OPT}(G) \geq \mathsf{MBT}(G)$, we focus on the other direction $\mathsf{OPT}(G) \leq \mathsf{MBT}(G)$.

Let $H = (V_H, E_H)$ be a maximum binary tree in G, i.e. $|E_H| = \mathsf{OPT}(G)$. By Lemma 15, we can further assume that H is a crossing-free. Let s_i be the minimum vertex in $S \cap V_H$ and let t_j be the minimum vertex in $T \cap V_H$. Since H is a connected crossing-free subgraph, by Lemma 16 we have that $\{s_i, t_j\} \in E_H$, and that one of s_i and t_j is a leaf vertex in H. Observing that H is also a maximum binary tree in the subgraph $[i, j]$, we have

$$\mathsf{OPT}(G) = |E_H| = \max \{\mathsf{MBT}_S(i, j), \mathsf{MBT}_T(i, j)\} \leq \mathsf{MBT}(G).$$

□

Now in order to compute $\mathsf{OPT}(G)$, it suffices to compute $\mathsf{MBT}(G)$ which amounts to solving the subproblems $\mathsf{MBT}_S(i,j)$ and $\mathsf{MBT}_T(i,j)$. The following recurrence relations immediately give a dynamic programming algorithm:

$$\mathsf{MBT}_S(i,j) = \begin{cases} 0 & \text{if } s_i \notin N_{[i,j]}(t_j), \\ 1 & \text{if } N_{[i,j]}(t_j) = \{s_i\}, \\ \max\left\{ \mathsf{MBT}_T(i+1,j)+1, \max\limits_{2 \le k \le \left|N_{[i,j]}(t_j)\right|-1} \{\mathsf{MBT}_T(i+k,j)+2\} \right\} & \text{if } \left|N_{[i,j]}(t_j)\right| \ge 2. \end{cases} \tag{8}$$

$$\mathsf{MBT}_T(i,j) = \begin{cases} 0 & \text{if } t_j \notin N_{[i,j]}(s_i), \\ 1 & \text{if } N_{[i,j]}(s_i) = \{t_j\}, \\ \max\left\{ \mathsf{MBT}_S(i,j+1)+1, \max\limits_{2 \le k \le \left|N_{[i,j]}(s_i)\right|-1} \{\mathsf{MBT}_S(i,j+k)+2\} \right\} & \text{if } \left|N_{[i,j]}(s_i)\right| \ge 2. \end{cases} \tag{9}$$

The boundary conditions are given by

$$\mathsf{MBT}_S(p+1,q+1) = \mathsf{MBT}_T(p+1,q+1) = 0.$$

Lemma 18 *$\mathsf{MBT}_S(i,j)$ and $\mathsf{MBT}_T(i,j)$ satisfy the recurrence relations (8) and (9).*

Proof Since S and T are symmetric, we will only prove that $\mathsf{MBT}_S(i,j)$ satisfies relation (8).

Case 1: $s_i \notin N_{[i,j]}(t_j)$. Clearly $\mathsf{MBT}_S(i,j) = 0$ since $\{s_i, t_j\} \notin E$ implies that constraint 2 cannot be satisfied by any binary tree.

Case 2: $N_{[i,j]}(t_j) = \{s_i\}$, i.e. s_i is the unique neighbor of t_j in the graph $[i,j]$. Since by constraint 3 vertex s_i has to be a leaf vertex in H, we know that $\{s_i, t_j\}$ is the only binary tree which satisfies all 3 constraints. In this case $\mathsf{MBT}_S(i,j) = 1$.

Case 3: Case 1 and Case 2 do not occur, which implies $d := \left|N_{[i,j]}(t_j)\right| \ge 2$. Consider the optimal binary tree $H = (V_H, E_H)$ satisfying all 3 constraints. Let s_{i+k} $(1 \le k \le d-1)$ be the "furthest" neighbor of t_j, i.e. the maximal element in $N_H(t_j) \setminus \{s_i\}$. We further consider 2 possibilities.

- $k = 1$. We note that t_j is a degree-2 vertex in this case. Consider the binary tree $H' = (V_{H'}, E_{H'})$ where $V_{H'} = V_H \setminus \{s_i\}$ and $E_{H'} = E_H \setminus \{\{s_i, t_j\}\}$. H' is a feasible solution to the subproblem $\mathsf{MBT}_T(i+1,j)$ since H' is a crossing-free binary tree which contains t_j as a leaf vertex and the edge $\{s_{i+1}, t_j\}$. We deduce that $|E_H| = |E_{H'}| + 1 \le \mathsf{MBT}_T(i+1,j) + 1$.
- $k \ge 2$. Since H is maximum, t_j must have another neighbor other than s_i and s_{i+k}. By Lemma 14, $s_{i+\ell}$ is a neighbor of t_j for any $0 \le \ell \le k$. Since H is crossing-free, that third neighbor of t_j is a leaf vertex in H. Therefore without loss of generality we can assume that it is s_{i+1}. Consider the binary tree $H' = (V_{H'}, E_{H'})$ where $V_{H'} = V_H \setminus \{s_i, s_{i+1}\}$ and $E_{H'} = E_H \setminus \{\{s_i, t_j\}, \{s_{i+1}, t_j\}\}$. H' is a feasible solution to the subproblem $\mathsf{MBT}_T(i+k,j)$ since H' is a crossing-free binary

tree which contains t_j as a leaf vertex and the edge $\{s_{i+k}, t_j\}$. We deduce that $|E_H| = |E_{H'}| + 2 \leq \mathsf{MBT}_T(i+k, j) + 2$.

Thus, we conclude that

$$\mathsf{MBT}_S(i,j) \leq \max\left\{\mathsf{MBT}_T(i+1,j)+1, \max_{2\leq k\leq d-1}\left\{\mathsf{MBT}_T(i+k,j)+2\right\}\right\}.$$

To see the other direction of the inequality, we note that a feasible solution to $\mathsf{MBT}_T(i+1,j)$ induces a feasible solution to $\mathsf{MBT}_S(i,j)$ by including the edge $\{s_i, t_j\}$, and a feasible solution to $\mathsf{MBT}_T(i+k,j)$ induces a feasible solution to $\mathsf{MBT}_S(i,j)$ by including the edges $\{s_i, t_j\}$ and $\{s_{i+1}, t_j\}$. □

We now give a formal proof of Theorem 4.

Theorem 4 *There exists an algorithm to solve* UndirMaxBinaryTree *in n-vertex bipartite permutation graphs that runs in time* $O(n^3)$.

Proof Given a bipartite permutation graph G with n vertices and m edges, there is an $O(m+n)$ time algorithm for finding a strong ordering of G (cf. [37]). Suppose G has connected components $G_1, G_2, \cdots, G_\ell$ and G_i has n_i vertices, hence $\sum_{i=1}^{\ell} n_i = n$. Every G_i is a (strongly ordered) connected bipartite permutation graph. Since any binary tree in G completely resides in one connected component of G, it suffices to solve $\mathsf{MBT}(G_i)$ for every G_i and return $\max_{1\leq i\leq \ell}\{\mathsf{MBT}(G_i)\}$. Solving $\mathsf{MBT}(G_i)$ requires $O(n_i^3)$ time since there are $O(n_i^2)$ subproblems ($\mathsf{MBT}_S(i,j)$ and $\mathsf{MBT}_T(i,j)$ for $i,j \in [n_i]$) solving each of which requires $O(n_i)$ time. The overall running time is $O\left(\sum_{i=1}^{\ell} n_i^3\right) = O(n^3)$. □

6 Conclusion and Open Problems

In this work, we introduced the maximum binary tree problem (MBT) and presented hardness of approximation results for undirected, directed, and directed acyclic graphs, a fixed-parameter algorithm with the solution as the parameter, and efficient algorithms for bipartite permutation graphs. Our work raises several open questions that we state below.

6.1 Inapproximability of DirMaxBinaryTree

The view that MBT is a variant of the longest path problem leads to the natural question of whether the inapproximability results for MBT match that of longest path: Is MBT in directed graphs (or even in DAGs) hard to approximate within a factor of $1/n^{1-\varepsilon}$ (we recall that longest path is hard to approximate within a factor of $1/n^{1-\varepsilon}$ [9])? We remark that the self-improving technique is weak to handle $1/n^{1-\varepsilon}$

-approximations since the squaring operation yields no improvement. The reduction in [9] showing $1/n^{1-\epsilon}$-inapproximability of longest paths is from a restricted version of the vertex-disjoint paths problem and is very specific to paths. Furthermore, directed cycles play a crucial role in their reduction for a fundamental reason: longest path is polynomial-time solvable in DAGs. However, it is unclear if directed cycles are the source of hardness for MBT in digraphs (since MBT is already hard in DAGs).

6.2 Bicriteria Approximations

Given our inapproximability results, one natural algorithmic possibility is that of bicriteria approximations: can we find a tree with at least $\alpha \cdot OPT$ vertices while violating the degree bound by a factor of at most β? In particular, this motivates an intriguing direction concerning the longest path problem: Given an undirected/directed graph G with a path of length k, can we find a c_1-degree tree in G with at least k/c_2 vertices for some constants c_1 and c_2 efficiently?

6.3 Maximum Binary Tree in Permutation DAGs

Finally, it would be interesting to resolve the complexity of MBT in permutation DAGs (and permutation graphs). This would also resolve the open problem posed by Byers, Heeringa, Mitzenmacher, and Zervas of whether the maximum heapable subsequence problem is solvable in polynomial time [11].

Acknowledgements Karthekeyan Chandrasekaran is supported by NSF CCF-1814613 and NSF CCF-1907937. Elena Grigorescu, Young-San Lin, and Minshen Zhu are supported by NSF CCF-1910659 and NSF CCF-1910411. Gabriel Istrate was supported by a grant of Ministry of Research and Innovation, CNCS - UEFISCDI project number PN-III-P4-ID-PCE-2016-0842, within PNCDI III. We thank our anonymous reviewers for many constructive comments that helped improve the presentation of the paper. A preliminary version of this work appeared in the proceedings of the 28th Annual European Symposium on Algorithms (ESA, 2020). This version contains complete proofs, an integer program, and an integrality gap instance.

References

1. Addario-Berry, L., Dalal, K., Reed, A.: Degree constrained subgraphs. Electron. Notes Discrete Math. **19**, 257–263 (2005)
2. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4), 844–856 (1995)
3. Amini, O., Peleg, D., Pérennes, S., Sau, I., Saurabh, S.: Degree-constrained subgraph problems: hardness and approximation results. In: Approximation and Online Algorithms, pp. 29–42 (2009)
4. Amini, O., Sau, I., Saurabh, S.: Parameterized complexity of the smallest degree-constrained subgraph problem. In: Parameterized and Exact Computation, pp. 13–29 (2008)
5. Austrin, P., O'Donnell, R., Wright, J.: A new point of NP-hardness for 2-to-1 label-cover. In: Proceedings of the 15th Annual International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX '12, pp. 1–12 (2012)

6. Balogh, J., Bonchiş, C., Diniş, D., Istrate, G., Todinca, I.: On the heapability of finite partial orders. In: Discrete Mathematics and Theoretical Computer Science, vol. 22 (2020)
7. Bansal, N., Khandekar, R., Nagarajan, V.: Additive guarantees for degree-bounded directed network design. SIAM J. Comput. **39**(4), 1413–1431 (2009)
8. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. J. Comput. Syst. Sci. **87**, 119–139 (2017)
9. Björklund, A., Husfeldt, T., Khanna, S.: Approximating longest directed paths and cycles. In: Languages and Programming, Automata, pp. 222–233 (2004)
10. Björklund, A., Kaski, P., Kowalik, Ł: Constrained multilinear detection and generalized graph motifs. Algorithmica **74**(2), 947–967 (2016)
11. Byers, J., Heeringa, B., Mitzenmacher, M., Zervas, G.: Heapable sequences and subseqeuences. In: Proceedings of the Meeting on Analytic Algorithmics and Combinatorics, ANALCO '11, pp. 33–44 (2011)
12. Chaudhuri, K., Rao, S., Riesenfeld, S., Talwar, K.: A push-relabel approximation algorithm for approximating the minimum-degree mst problem and its generalization to matroids. Theoret. Comput. Sci. **410**(44), 4489–4503 (2009)
13. Chaudhuri, K., Rao, S., Riesenfeld, S., Talwar, K.: What would edmonds do? Augmenting paths and witnesses for degree-bounded MSTs. Algorithmica **55**(1), 157–189 (2009)
14. Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M.: Springer, Parameterized Algorithms (2015)
15. Erdös, P., Faudree, R.J., Rousseau, C.C.: Subgraphs of minimal degree k. Discrete Math. **85**, 53–58 (1990)
16. Fürer, M., Raghavachari, B.: Approximating the minimum-degree Steiner tree to within one of optimal. J. Algorithms **17**(3), 409–423 (1994)
17. Gabow, H.N.: An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83, pp. 448–456 (1983)
18. Garey, M., Johnson, D.: Computers and Intractability. W. H Freeman and Company, New York (1979)
19. Goemans, M.X.: Minimum bounded degree spanning trees. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS '06, pp. 273–282 (2006)
20. Guillemot, S., Sikora, F.: Finding and counting vertex-colored subtrees. Algorithmica **65**(4), 828–844 (2013)
21. Guruswami, V., Sinop, A.K.: Improved inapproximability results for maximum k-colorable subgraph. Theory Comput. **9**, 413–435 (2013)
22. Istrate, G., Bonchiş, C.: Heapability, interactive particle systems, partial orders: results and open problems. In: Proceedings of DCFS'2016, 18th International Conference on Descriptional Complexity of Formal Systems, Springer, pp. 18–28 (2016)
23. Karger, D.R., Motwani, R., Ramkumar, G.D.S.: On approximating the longest path in a graph. Algorithmica **18**, 82–98 (1997)
24. Khandekar, R., Kortsarz, G., Nutov, Z.: On some network design problems with degree constraints. J. Comput. Syst. Sci. **79**(5), 725–736 (2013)
25. Kloks, T., Kratsch, D., Müller, H.: Bandwidth of chain graphs. Inf. Process. Lett. **68**, 313–315 (1998)
26. Könemann, J., Ravi, R.: A matter of degree: improved approximation algorithms for degree-bounded minimum spanning trees. SIAM J. Comput. **31**, 1783–1793 (2002)
27. Koutis, I..: Faster algebraic algorithms for path and packing problems. In: International Colloquium on Automata, Languages, and Programming, Springer, pp. 575–586 (2008)
28. Koutis, I., Williams, R..: Limits and applications of group algebras for parameterized problems. In: International Colloquium on Automata, Languages, and Programming, Springer, pp. 653–664 (2009)
29. Könemann, J., Ravi, R.: Primal-dual meets local search: approximating msts with nonuniform degree bounds. SIAM J. Comput. **34**, 763–773 (2005)
30. Lap, C.L., Joseph, N., Mohammad, S., Mohit, S.: Survivable network design with degree or order constraints. SIAM J. Comput. **39**(3), 1062–1087 (2009)
31. Nederlof, J.: Fast polynomial-space algorithms using möbius inversion: improving on steiner tree and related problems. In: International Colloquium on Automata, Languages, and Programming, Springer, pp. 713–725 (2009)

32. Papadimitriou, C.H., Yannakakis, M.: The traveling salesman problem with distances one and two. Math. Oper. Res. **18**(1), 1–11 (1993)
33. Porfilio, J.: A combinatorial characterization of heapability. Master's Thesis, Williams College (2015)
34. Ravi, R., Marathe, M., Ravi, S.S., Rosenkrantz, D.: Approximation algorithms for degree-constrained minimum-cost network-design problems. Algorithmica **31**, 58–78 (2001)
35. Singh, M., Lau, L.C.: Approximating minimum bounded degree spanning trees to within one of optimal. J. ACM **62**(1), 1–19 (2015)
36. Smith, J.: Minimum degree spanning trees on bipartite permutation graphs. Master's Thesis, University of Alberta (2011)
37. Spinrad, J., Brandstädt, A., Stewart, L.: Bipartite permutation graphs. Discrete Appl. Math. **18**(3), 279–292 (1987)
38. Uehara, R., Uno, Y.: Efficient algorithms for the longest path problem. In: Proceedings of the 15th International Conference on Algorithms and Computation, ISAAC '04, pp. 871–883 (2004)
39. Williams, R.: Finding paths of length k in $O^*(2^k)$ time. Inf. Process. Lett. **109**(6), 315–318 (2009)

Authors and Affiliations

Karthekeyan Chandrasekaran[1] · Elena Grigorescu[2] · Gabriel Istrate[3] · Shubhang Kulkarni[1] · Young-San Lin[2] · Minshen Zhu[2]

Elena Grigorescu
elena-g@purdue.edu

Gabriel Istrate
gabrielistrate@acm.org

Shubhang Kulkarni
smkulka2@illinois.edu

Young-San Lin
lin532@purdue.edu

Minshen Zhu
zhu628@purdue.edu

[1] University of Illinois, Urbana-Champaign, Urbana, USA

[2] Purdue University, West Lafayette, USA

[3] West University of Timişoara, Romania and the e-Austria Research Institute, Timisoara, Romania