A Moderately Exponential Time Algorithm for FULL DEGREE SPANNING TREE*

Serge Gaspers Saket Saurabh
Alexey A. Stepanov
Department of Informatics, University of Bergen,
N-5020 Bergen, Norway.
{serge|saket}@ii.uib.no, ljosha@ljosha.org

Abstract

We consider the well studied FULL DEGREE SPANNING TREE problem, a NP-complete variant of the SPANNING TREE problem, in the realm of moderately exponential time exact algorithms. In this problem, given a graph G, the objective is to find a spanning tree T of G which maximizes the number of vertices that have the same degree in T as in G. This problem is motivated by its application in fluid networks and is basically a graph-theoretic abstraction of the problem of placing flow meters in fluid networks. We give an exact algorithm for FULL DEGREE SPANNING TREE running in time $\mathcal{O}(1.9465^n)$. This adds FULL DEGREE SPANNING TREE to a very small list of "non-local problems", like FEEDBACK VERTEX SET and CONNECTED DOMINATING SET, for which non-trivial (non brute force enumeration) exact algorithms are known.

1 Introduction

The problem of finding a spanning tree of a connected graph arises at various places in practice and theory, like the analysis of communication or distribu-

*Additional support by the Norwegian Research Council. A preliminary version of this paper appeared in the Proceedings of the 5th Annual Conference on Theory and Applications of Models of Computation (TAMC08) [10]

tion networks, or modeling problems, and can be solved efficiently in polynomial time. On the other hand, if we want to find a spanning tree with some additional properties like maximizing the number of leaves or minimizing the maximum degree of the tree, the problem becomes NP-complete. This paper deals with one of the NP hard variants of SPANNING TREE, namely FULL DEGREE SPANNING TREE from the view point of moderately exponential time algorithms.

> FULL DEGREE SPANNING TREE (FDST): Given an undirected connected graph $G = (V, E)$, find a spanning tree T of G which maximizes the number of vertices of *full degree*, that is the vertices having the same degree in T as in G.

The FDST problem is motivated by its applications in water distribution and electrical networks [16, 17, 18, 19]. Pothof and Schut [19] studied this problem in the context of water distribution networks where the goal is to determine or control the flows in the network by installing and using a small number of flow meters. It turns out that to measure flows in all pipes, it is sufficient to find a full degree spanning tree T of the network and install flow meters (or pressure gauges) at each vertex of T that does not have full degree. We refer to [1, 4, 12] for a more detailed description of various applications of FDST.

The FDST problem has attracted a lot of attention recently and has been studied extensively from different algorithmic paradigms, developed for coping with NP-completeness. Pothof and Schut [19] studied this problem first and gave a simple heuristic based algorithm. Bhatia et al. [1] studied it from the view point of approximation algorithms and gave an algorithm of factor $\mathcal{O}(\sqrt{n})$. On the negative side, they show that FDST is hard to approximate within a factor of $\mathcal{O}(n^{\frac{1}{2}-\epsilon})$, for any $\epsilon > 0$, unless $coR = NP$, a well known complexity-theoretic hypothesis. Guo et al. [11] studied the problem in the realm of parameterized complexity and observed that the problem is W[1]-complete. The problem which is dual to FDST is also studied in the literature, that is the problem of finding a spanning tree that minimizes the number of vertices not having full degree. For this dual version of the problem, Khuller et al [12] gave an approximation algorithm of factor $2 + \epsilon$ for any fixed $\epsilon > 0$, and Guo et al. [11] gave a fixed parameter tractable algorithm running in time $4^k n^{\mathcal{O}(1)}$. FDST has also been studied on special graph classes like planar graphs, bounded degree graphs and graphs of bounded treewidth [4]. The goal of this paper is to study FULL DEGREE SPANNING TREE in the context of moderately exponential time algorithms, another coping strategy to deal with NP-completeness. We give a $\mathcal{O}(1.9465^n)$ time algorithm breaking the trivial $2^n n^{\mathcal{O}(1)}$ barrier.

Exact exponential time algorithms have an old history [5, 15] but the last few years have seen a renewed interest in the field. This has led to the advancement of the state of the art on exact algorithms and many new techniques based on Inclusion-Exclusion, Measure & Conquer and various other combinatorial tools have been developed to design and analyze exact algorithms [2, 3, 7, 8, 13]. Branch & Reduce has always been one of the most important tools in the area but its applicability was mostly limited to 'local problems' (where the decision on one element of the input has direct consequences for its neighboring elements) like MAXIMUM INDEPENDENT SET, SAT and various other problems, until recently. In 2006, Fomin et al.[9] devised an algorithm for CONNECTED DOMINATING SET (or MAXIMUM LEAF SPANNING TREE) and Razgon [20] for FEEDBACK VERTEX SET combining sophisticated branching and a clever use of measure. Our algorithm adheres to this machinery and adds an important real life problem to this small list. We also need to use an involved measure, which is a function of the number of vertices and the number of edges to be added to the spanning tree, to get the desired running time.

2 Preliminaries

Let G be a graph. We use $V(G)$ and $E(G)$ to denote the vertices and the edges of G respectively. We simply write V and E if the graph is clear from the context. For $V' \subseteq V$ we define an *induced subgraph* $G[V'] = (V', E')$, where $E' = \{uv \in E : u, v \in V'\}$.

Let $v \in V$, we denote by $N(v)$ the *neighborhood* of v, namely $N(v) = \{u \in V : uv \in E\}$. The *closed neighborhood* $N[v]$ of v is $N(v) \cup \{v\}$. In the same way we define $N[S]$ for $S \subseteq V$ as $N[S] = \cup_{v \in S} N[v]$ and $N(S) = N[S] \setminus S$. We define the *degree* of vertex v in G as the number of vertices adjacent to v in G. Namely, $d_G(v) = |\{u \in V(G) : uv \in E(G)\}|$.

Let G be a graph and T be a spanning tree of G. A vertex $v \in V(G)$ is a *full degree* vertex in T, if $d_G(v) = d_T(v)$. We define a *full degree spanning tree* to be a spanning tree with the maximum number of full degree vertices. One can similarly define *full degree spanning forest* by replacing tree with forest in the earlier definition.

A set $I \subseteq V$ is called an *independent set* for G if no vertex v in I has a neighbor in I.

3 Algorithm for Full Degree Spanning Tree

In this Section we give an exact algorithm for the FDST problem.

Given an input graph $G = (V, E)$, the basic idea is that if we know a subset S of V for which there exists a spanning tree T where all the vertices in S have full degree then, given this set S, we can construct a spanning tree T where all the vertices in S have full degree in polynomial time. Our first observation towards this is that all the edges incident to the vertices in S, that is

$$E_S = \{uv \in E \text{ such that } u \in S \text{ or } v \in S\ \} \tag{1}$$

induce a forest. For our polynomial time algorithm we start with the forest (V, E_S) and then complete this forest into a spanning tree by adding edges to connect the components of the forest. The last step can be done by using a slightly modified version of the Spanning Tree algorithm of Kruskal [14] that we denote by `poly_fdst`(G, S). Moreover, it has been shown in [11] that a full degree spanning tree can be computed in polynomial time if the decision on degree 2 vertices is left open. That is, for a maximum set of full degree vertices S, `poly_fdst`$(G, S \setminus S_2)$ returns a spanning tree with the same number of full degree vertices as `poly_fdst`(G, S) where S_2 is the set of degree 2 vertices in S. Therefore, our algorithm may postpone the decision on degree 2 vertices to the polynomial time procedure.

The rest of the section is devoted to finding a subset of vertices S for which `poly_fdst`(G, S) returns a spanning tree with the largest number of full degree vertices.

Our algorithm follows a branching strategy and as a partial solution keeps a set of vertices S for which there exists a spanning tree where the vertices in S have full degree. The algorithm grows one component of the forest $(N[S], E_S)$ at a time. The vertices of this component are denoted by S_a, the active set. We denote $S \setminus S_a$ by S_b, the inactive set. The standard branching step chooses a vertex v that could be included in S_a and then recursively tries to find a solution by including v in S_a and not including v in S. But when v is not included in S, it cannot be removed from further consideration as cycles involving v might be created later on in (V, E_S) by adding neighbors of v to S. Hence we resort to a coloring scheme for the vertices, which can also be thought of as a partition of the vertex set of the input graph. At any point of the execution of the algorithm, the vertices are partitioned as below:

1. *Selected* $S = S_a \cup S_b$: The set of vertices which are decided to be of full degree. The set S_a corresponds to the *active* set of vertices which we use in the current stage of the algorithm. The set S_b is the *inactive* set of vertices which were decided to be of full degree in an earlier stage of the algorithm.

2. *Discarded* D: The set of vertices which are not required to be of full degree.

3. *Postponed* P: The subset of vertices of degree 2 which are to be decided later by `poly_fdst`(G, S).

4. *Undecided* U: The set of vertices which are not in S, D or P, that is those vertices which are yet to be decided. So, $U = V \setminus (S \cup D \cup P)$.

Next we define a generalized form of the FDST problem based on the above partition of the vertex set. But before that we need the following definition.

Definition 1 *Given a vertex set $S \subseteq V$, we define the* partial spanning forest *of G induced by S as $T(S) = (N[S], E_S)$ where E_S is defined as in Equation (1).*

For our generalized problem, we denote by $G = (S_a, S_b, D, P, U, E)$ the graph (V, E) with vertex set $V = S_a \cup S_b \cup D \cup P \cup U$ partitioned as above.

> Generalized Full Degree Spanning Tree (GFDST): Given an instance $G = (S_a, S_b, D, P, U, E)$ such that $T(S_a)$ is connected and acyclic, and no vertex of $T(S_b)$ is adjacent to a vertex in U, the objective is to find a spanning forest which maximizes the number of vertices of $U \cup P$ of full degree under the constraint that all the vertices in $S = S_a \cup S_b$ have full degree.

If we start with a graph G, an instance of FDST, with the vertex partition $S = D = P = \emptyset$ and $U = V$ then the problem we will have at every intermediate step of the recursive algorithm is GFDST. Also, note that a full degree spanning forest of a connected graph can easily be extended to a full degree spanning tree and that a full degree spanning tree is a full degree spanning forest.

As suggested earlier our algorithm is based on branching and will have some reduction rules that can be applied in polynomial time, leading to a refined partitioning of the vertices. Before we come to the detailed description of the algorithm, we introduce a few more important definitions. For given sets S, D, P and U, we say that an edge is

(a) *unexplored* if one of its endpoints is in U and the other one in $U \cup D \cup P$,

(b) *forced* if at least one of its endpoints is in S, and

(c) *superfluous* if both its endpoints are in D.

The basic step of our algorithm chooses an undecided vertex $u \in U$ and considers two subcases that it solves recursively: either u is *selected*, that is u is moved from U to S_a, or u is *discarded*, that is moved from U to D. But the main idea is to choose a vertex in a way that the connectivity of $T(S_a)$ is maintained in both recursive calls. To do so we choose u from $U \cap N[N[S_a]]$. This brings us to the following definition.

Definition 2 *The vertices in $U \cap N[N[S_a]]$ are called* candidate *vertices.*

On the other hand, if S_a is not empty and there are no candidate vertices, then $S_b := S_b \cup S_a$ and $S_a := \emptyset$. If U is not empty, then the algorithm starts to grow a new component.

Now we are ready to describe the algorithm in details. We start with a procedure for reduction rules in the next subsection and prove that these rules are correct.

3.1 Reduction Rules

Given an instance $G = (S_a, S_b, D, P, U, E)$ of GFDST, a reduced instance of G is computed by the following procedure.

`Reduce(`$G = (S_a, S_b, D, P, U, E)$`)`

R1 If there is a superfluous edge e, then return `Reduce(`$(S_a, S_b, D, P, U, E \setminus \{e\})$`)`

R2 If there is a vertex $u \in D \cup U$ such that $d(u) = 1$, then remove the unique edge e incident on it and return `Reduce(`$(S_a, S_b, D, P, U, E \setminus \{e\})$`)`.

R3 If there is an undecided vertex $u \in U$ such that $T(S_a \cup \{u\})$ contains a cycle, then discard u, that is return `Reduce(`$(S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E)$`)`.

R4 If there is a candidate vertex u that is incident to at most one vertex in $U \cup D \cup P$, then select u, and return `Reduce(`$(S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E)$`)`.

R5 If $S = \emptyset$ and there exists a vertex $u \in U$ of degree 2, then select u and return `Reduce(`$(S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E)$`)`.

R6 If there is a candidate vertex u of degree 2 that has a neighbor v in D, then put u into P, remove the edge uv and return `Reduce(`$(S_a, S_b, D, P \cup \{u\}, U \setminus \{u\}, E \setminus \{\{u, v\}\})$`)`.

R7 If there is no candidate vertex and $U \neq \emptyset$ then return `Reduce(`$(\emptyset, S_a \cup S_b, D, P, U, E)$`)`.

Else return G

Now we argue about the correctness of the reduction rules. More precisely, we prove that there exists a spanning forest of G such that a maximum number of vertices preserve their degree and the partitioning of the vertices into the sets S, D, P and U of the graph resulting from a call to `Reduce(`$G = (S_a, S_b, D, P, U, E)$`)` is respected. Note that the reduction rules are applied in the order of their appearance.

The correctness of **R1** follows from the fact that discarded vertices are not required to have full degree.

For the correctness of reduction rule **R2**, consider a vertex $u \in D \cup U$ of degree 1 with unique neighbor w. Let $G' = (S_a, S_b, D, P, U, E \setminus \{uw\})$ be the graph resulting from the application of the reduction rule. Note that the edge uw is not part of any cycle and that a full degree spanning forest of G can be obtained from a full degree spanning forest of G' by adding the edge uw.

As Algorithm `poly_fdst`(G, S) adds edges to make the obtained spanning forest into a spanning tree, the edge uw is added to the final solution.

For the correctness of reduction rule **R3**, it is enough to observe that if for a subset $S \subseteq V$, there exists a spanning tree T such that all the vertices of S have full degree then $T(S)$ is a forest.

We prove the correctness of **R4** and **R5** by the following lemmata.

Lemma 3 *Let $G = (V, E)$ be a graph and T be a full degree spanning forest for G. If $v \in V$ is a vertex of degree $d_G(v) - 1$ in T, then there exists a full degree spanning forest T' such that v has degree $d_G(v)$ in T'.*

Proof: Let $u \in V$ be the neighbor of v such that uv is not an edge of T. Note that both u and v do not have full degree in T, are not adjacent and belong to the same tree in T. The last assertion follows from the fact that if u and v belong to two different trees of T then one can safely add uv to T and obtain a forest T' that has a larger number of full degree vertices, contradicting that T is a full degree spanning forest. Now, adding the edge uv to T creates a unique cycle passing through u and v. We obtain the new forest T' by removing the other edge incident to u on the cycle, say uw, $w \neq v$. So, $T' = T \setminus \{uw\} + \{uv\}$. The number of full degree vertices in T' is at least as high as in T as v becomes a full degree vertex and at most one vertex, w, could become non full degree. ■

We also need a generalized version of Lemma 3.

Lemma 4 *Let $G = (S_a, S_b, D, P, U, E)$ be a graph and T be a full degree spanning forest for G such that the vertices in $S = S_a \cup S_b$ have full degree. Let $v \in U$ be a candidate vertex such that its neighbors in $D \cup P \cup U$ are not incident to a forced edge. If v has degree $d_G(v) - 1$ in T, then there exists a full degree spanning forest T' such that v has degree $d_G(v)$ in T' and the vertices in S have full degree.*

Proof: The proof is similar to the one of Lemma 3. The only difference is that we need to show that the vertices of S remain of full degree and for that we need to show that all the edges of $T(S)$ remain in T'. To this observe that all the edges incident to the neighbors of v in $D \cup P \cup U$ in T do not belong to edges of $T(S)$, that is they are not forced edges. So if uv is the unique edge incident to v missing in T then we can add uv to T and remove the other non-forced edge on u from the unique cycle in $T + \{uv\}$ and get the desired T'. ■

First, observe that every vertex of P has has degree 0. Therefore, all vertices in U are in different connected components than the vertices in S_b.

Now consider reduction rule **R4**. If u is a candidate vertex with unique neighbor w in $D \cup P \cup U$ then (a) $u \in N(S_a)$ and (b) all the edges incident to w are not forced, otherwise reduction rule **R2** or **R3** would have applied. Now the correctness of the reduction rule follows from Lemma 4.

To prove the correctness of reduction rule **R5**, we need to show that there exists a spanning forest where u has full degree. Suppose not and let T be any full degree spanning forest of G. Without loss of generality, suppose that u has degree 1 in T (if u is an isolated vertex in T, then add one edge incident to u to T; this does not create any cycle in T and does not decrease the number of vertices of full degree in T). Let v be the unique neighbor of u in T. But since $S_a = \emptyset$, there are no forced edges incident to neighbors of u and we can apply Lemma 4 again and conclude.

Reduction rule **R6** postpones the decision for certain degree 2 vertices. The edge uv may be deleted as the decision on this edge is made by the polynomial time procedure. Note that u becomes of degree 1, which triggers reduction rule **R2**, removing the other edge incident on u as there exists a full degree spanning tree respecting our partitioning of vertices containing this edge.

Reduction rule **R7** only makes the active set inactive in order to start a new component of the spanning forest as the current component cannot be grown any further and is thus correct.

This finishes the correctness proof of the reduction rules. Before we go into the details of the algorithm we would like to point out that all our reduction rules preserve the connectivity of $T(S_a)$.

3.2 Algorithm

In this section we describe our algorithm in details. Given an instance $G = (S_a, S_b, D, P, U, E)$ of GDPST, our algorithm recursively solves the problem by choosing a candidate vertex $u \in U$ and including u in S_a or in D and then returning as solution the one with a maximum number of full degree vertices. The algorithm has various cases based on the number of unexplored edges incident to u.

Algorithm `fdst`(G), described below, returns a super-set S^* of $S = S_a \cup S_b$ corresponding to the full degree vertices (except some degree 2 vertices) in a full degree spanning forest respecting the initial choices for S and D. After this, poly_fdst(G, S^*) returns a full degree spanning tree of G.

We describe the procedure poly_fdst(G, S) now in more details. Initially, all edges are unweighted. To each edge incident to a vertex in S, assign weight 1. To each unweighted edge incident to a vertex in P, assign weight 2. To each remaining unweighted edge, assign weight 3. Then compute a minimum spanning tree T using the algorithm of Kruskal [14] of the resulting weighted graph. Thus, all edges of $T(S)$ are in T ($T(S)$ is acyclic), and connecting two different connected components of $T(S)$ is done preferably by making vertices of P of full degree rather than adding edges incident to discarded vertices.

The description of the algorithm consists of the application of the reduction rules and a sequence of cases. A case consists of a condition (first sentence) and a procedure to be executed if the condition holds. The first case which applies is used in the algorithm. Thus, inside a given case, the conditions of all previous cases are assumed to be false.

fdst$(G = (S_a, S_b, D, P, U, E))$

Replace G by Reduce(G).

Case 1: U is a set of isolated vertices. Return $S \cup U$.

Case 2: $S_a = \emptyset$. Choose a vertex $u \in U$ of degree at least 3. Return the best solution among fdst$((S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E))$ and fdst$((S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E))$.

Case 3: There is a candidate vertex u with at least 3 unexplored incident edges. Make two recursive calls: fdst$((S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E))$ and fdst$((S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E))$, and return the best solution.

Case 4: There is a candidate vertex $u \in N(S_a)$ with at least one neighbor v in U and exactly two unexplored incident edges. Make two recursive calls: fdst$((S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E))$ and fdst$((S_a, S_b, D \cup \{u, v\}, P, U \setminus \{u, v\}, E))$, and return best solution .

From now on let v_1 and v_2 denote the discarded neighbors of a candidate vertex u (see Figure 1).

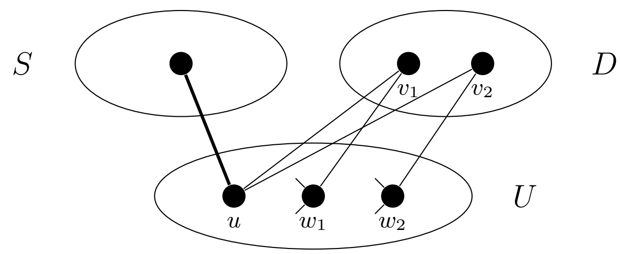

Figure 1: Illustration of Case 6. Cases 5, 7 and 8 are similar.

Case 5: Either v_1 and v_2 have a common neighbor $x \neq u$; or
v_1 (or v_2) has a neighbor $x \neq u$ that is a candidate vertex; or
v_1 (or v_2) has a neighbor x of degree 2.
Make two recursive calls: $\texttt{fdst}((S_a \cup \{u\}, S_b, D, P, U \setminus \{u\}, E))$ and $\texttt{fdst}((S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E))$, and return the best solution.

Case 6: Both v_1 and v_2 have degree 2. Let w_1 and w_2 ($w_1 \neq w_2$) be the other (different from u) neighbors of v_1 and v_2 in U respectively. Make recursive calls as usual, but also explore all the possibilities for w_1 and w_2 if $u \in S$. When u is in S, recurse on all possible ways one can add a subset of $A = \{w_1, w_2\}$ to S. That is make recursive calls $\texttt{fdst}((S, D \cup \{u\}, U \setminus \{u\}, E))$ and $\texttt{fdst}((S \cup \{u\} \cup X, D \cup (A - X), U \setminus (\{u\} \cup A), E))$ for each independent set $X \subseteq A$, and return the best solution.

Case 7: One of $\{v_1, v_2\}$ has degree ≥ 3. Let $\{u, w_1, w_2, w_3\} \subseteq N(\{v_1, v_2\})$ and let $A = \{w_1, w_2, w_3\}$. Make recursive calls $\texttt{fdst}((S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E))$ and $\texttt{fdst}((S_a \cup \{u\} \cup X, S_b, D \cup (A - X), P, U \setminus (\{u\} \cup A), E))$ for each independent set $X \subseteq A$, and return the best solution.

Case 8: Both v_1 and v_2 have degree ≥ 3. Let $\{u, w_1, w_2, w_3, w_4\} \subseteq N(\{v_1, v_2\})$ and let $A = \{w_1, w_2, w_3, w_4\}$. Make recursive calls $\texttt{fdst}((S_a, S_b, D \cup \{u\}, P, U \setminus \{u\}, E))$ and $\texttt{fdst}((S_a \cup \{u\} \cup X, S_b, D \cup (A - X), P, U \setminus (\{u\} \cup A), E))$ for each independent set $X \subseteq A$, and return the best solution.

---

4 Correctness and Time Complexity of the Algorithm

We prove the correctness and the time complexity of Algorithm `fdst` in the following theorem.

Theorem 5 *Given an input graph $G = (S_a, S_b, D, P, U, E)$ on n vertices such that $T(S = S_a \cup S_b)$ is acyclic and $T(S_a)$ is connected, Algorithm* `fdst` *returns a set S^*, $S \subseteq S^* \subseteq S \cup U$, such that* `poly_fdst`$(G, S^*)$ *returns a spanning tree with a maximum number of full degree vertices, in time $\mathcal{O}(1.9465^n)$.*

Proof: The correctness of the reduction rules is described in Section 3.1.

To see that to every instance the algorithm applies some reduction or branching rule, it is sufficient to note that if no reduction rule, nor Cases 1–4 apply, then every candidate vertex is in $N(S_a)$ and has exactly two neighbors in D: candidate vertices with at least 3 unexplored incident edges are handled by Case 3, candidate vertices with at most one unexplored incident edge are taken care of by reduction rule **R4**, and candidate vertices with exactly two unexplored incident edges satisfy the condition of reduction rule **R6** or Case 4, or belong to $N(S_a)$ and have two neighbors in D.

The correctness of **Case 1** follows, as any isolated vertex belonging to U has full degree in any spanning forest. The remaining cases, except **Case 4**, of Algorithm `fdst` are branching steps where the algorithm chooses a vertex $u \in U$ and tries both possibilities: $u \in S_a$ or $u \in D$. Sometimes the algorithm branches further by looking at the local neighborhood of u and trying all possible ways these vertices can be added to either S_a or D. Since all possibilities are tried to add vertices of U to D or S_a in **Cases 3** and **5** to **8**, these cases are correct and do not need any further justifications. The correctness of **Case 4** requires special attention. Here we use the fact that there exists a full degree spanning forest with all the vertices in S having full degree, such that either $u \in S$ or u and its neighbor $v \in U$ are in D. We prove the correctness of this assertion by contradiction. Suppose all the full degree spanning forests such that all the vertices in S are of full degree have u of non full degree and v of full degree. But since $u \in N(S_a)$ and all the neighbors of u in $D \cup U$ do not have any incident forced edges, we can use Lemma 4 to get a spanning forest which contains u and is a full degree spanning forest with all the vertices in S having full degree.

Now we move on to the time complexity of the algorithm. The measure of subproblems is generally chosen as a function of structure, like vertices, edges or other graph parameters, which change during the recursive steps of the algorithm. In our algorithm, this change is reflected when vertices are moved to either S, D or P from U. The second observation is that any spanning tree on n vertices has at most $n-1$ edges and hence when we select a vertex in S we increase the number of edges in $T(S)$ and decrease the number of edges we can add to $T(S)$. Finally we also gain when the degree of a vertex becomes two because reduction rules apply as soon as the degree 2 vertex is in $N(S_a)$ (**R4**) or in $N(N(S_a))$ and has a neighbor in D. (Candidate vertices in $N(N(S_a))$ of degree 2 that are adjacent only to vertices in U fall into one of these two categories once their neighbor in $N(S_a)$ is put in either S_a or D.) Our measure is precisely a function of these three parameters and is defined as follows:

$$\mu(G) = \eta|U_2| + \beta|U_{\geq 3}| + \alpha m', \tag{2}$$

where U_2 is the subset of undecided vertices of degree 2, $U_{\geq 3}$ is the subset of undecided vertices of degree at least 3, $m' = n - 1 - |E(T(S))|$ is the number of edges that can be added to the spanning tree and $\eta = 0.5, \beta = 0.722$ and $\alpha = 0.23887$ are numerically obtained constants to optimize the running time. We write μ instead of $\mu(G)$ if G is clear from the context. We prove that the problem can be solved for an instance of size μ in time $\mathcal{O}(2^{\mu})$. As $\mu \leq 0.96087n$, the final running time of the algorithm will be $\mathcal{O}(2^{0.96087n}) = \mathcal{O}(1.9465^n)$. Denote by $P[\mu]$ the maximum number of times the algorithm is called recursively on a problem of size μ (i. e. the number of leaves in the search tree). Then the running time $T(\mu)$ of the algorithm is bounded by $P[\mu] \cdot n^{\mathcal{O}(1)}$ because in any node of the search tree, the algorithm executes only a polynomial number of steps. We use induction on μ to prove that $P[\mu] \leq 2^{\mu}$. Then $T(\mu) = 2^{\mu} \cdot n^{\mathcal{O}(1)} = \mathcal{O}(1.9465^n)$. Clearly, $P[0] = 1$. Suppose that $P[k] \leq 2^k$ for every $k < \mu$ and consider a problem of size μ.

Case 2: In this case, the number of vertices in $U_{\geq 3}$ decreases by one in both recursive calls and the number of edges in $T(S)$ increases by at least 3 in the first recursive call. Thus,

$$P[\mu] \leq P[\mu - \beta - 3\alpha] + P[\mu - \beta].$$

Case 3: This case has the same recurrence as **Case 2** as the number of vertices in $U_{\geq 3}$ decreases by one in both recursive calls and the number of edges in $T(S)$ increases by at least 3 in the first recursive call.

Case 4: When the algorithm adds u to S, the number of vertices in $U_{\geq 3}$ decreases by one and the number of edges in $T(S)$ increases by 2 while in the other case, $|U_{\geq 3}|$ decreases by two or $|U_2|$ and $|U_{\geq 3}|$ both decrease by one. So we get:

$$\begin{aligned} P[\mu] &\leq P[\mu-\beta-2\alpha]+P[\mu-2\beta], \text{ and} \\ P[\mu] &\leq P[\mu-\beta-2\alpha]+P[\mu-\eta-\beta]. \end{aligned}$$

Case 5: When the algorithm adds u to S, reduction rule **R3** or **R6** applies to x. We obtain the following recurrences, based on the degree of x:

$$\begin{aligned} P[\mu] &\leq P[\mu-\eta-\beta-2\alpha]+P[\mu-\beta], \text{ and} \\ P[\mu] &\leq P[\mu-2\beta-2\alpha]+P[\mu-\beta]. \end{aligned}$$

Case 6: In this case we distinguish two subcases based on the degrees of w_1 and w_2. Our first subcase is when either w_1 or w_2 has degree 3 and the other subcase is when both w_1 and w_2 have degree at least 4. (Note that because of **Case 4**, v_1 and v_2 do not have a common neighbor and do not have a neighbor of degree 2). Suppose w_1 has degree 3. When the algorithm adds u to D, the edges uv_1 and uv_2 are removed (**R1**), the degree of v_1 is reduced to 1 and then reduction rule **R2** is applied and makes w_1 of degree 2. So, in this subcase, μ decreases by $2\beta-\eta$. The analysis of the remaining branches is standard and we get the following recurrence:

$$P[\mu] \leq P[\mu-3\beta-2\alpha]+2P[\mu-3\beta-5\alpha]+P[\mu-3\beta-8\alpha]+P[\mu-2\beta+\eta].$$

For the other subcase we get the following recurrence:

$$P[\mu] \leq P[\mu-3\beta-2\alpha]+2P[\mu-3\beta-6\alpha]+P[\mu-3\beta-10\alpha]+P[\mu-\beta].$$

Case 7: This case is similar to **Case 6**. Without loss of generality, suppose v_1 has degree 2 and has w_1 as neighbor. If w_1 has degree 3, then it becomes of degree 2 when u is discarded. Thus, we obtain the recurrence

$$\begin{aligned} P[\mu] \leq\ & P[\mu-4\beta-2\alpha]+3P[\mu-4\beta-5\alpha]+3P[\mu-4\beta-8\alpha]+ \\ & P[\mu-4\beta-11\alpha]+P[\mu-2\beta+\eta]. \end{aligned}$$

On the other hand, if w_1 has degree at least 4, at least one more edge is added to the spanning forest each time w_1 is put into S_a and we obtain the recurrence

$$\begin{aligned} P[\mu] \leq\ & P[\mu - 4\beta - 2\alpha] + 2P[\mu - 4\beta - 5\alpha] + P[\mu - 4\beta - 6\alpha] + \\ & P[\mu - 4\beta - 8\alpha] + 2P[\mu - 4\beta - 9\alpha] + P[\mu - 4\beta - 12\alpha] + \\ & P[\mu - \beta]. \end{aligned}$$

Case 8: This case easily gives the recurrence

$$\begin{aligned} P[\mu] \leq\ & P[\mu - 5\beta - 2\alpha] + 4P[\mu - 5\beta - 5\alpha] + 6P[\mu - 5\beta - 8\alpha] + \\ & 4P[\mu - 5\beta - 11\alpha] + P[\mu - 5\beta - 14\alpha] + P[\mu - \beta]. \end{aligned}$$

In each of these recurrences, $P[\mu] \leq 2^{\mu}$ which completes the proof of the theorem. ■

The bottleneck of the analysis is the recurrence in **Case 8**. Therefore, an improvement of this case would lead to a faster algorithm.

We provide a Python program in the appendix for checking that the values for η, β and α satisfy all the recurrences.

5 Conclusion

In this paper we have given an exact algorithm for the Full Degree Spanning Tree problem. The most important feature of our algorithm is the way we exploit connectivity arguments to reduce the size of the graph in the recursive steps of the algorithm. We think that this idea of combining connectivity while developing Branch & Reduce algorithms could be useful for various other non-local problems and in particular for other NP-complete variants of the Spanning Tree problem. Although the theoretical bound we obtained for our algorithm seems to be only slightly better than a brute-force enumeration algorithm, practice shows that Branch & Reduce algorithms perform usually better than the running time proved by a worst case analysis of the algorithm. Therefore we believe that this algorithm, combined with good heuristics, could be useful in practical applications.

One problem which we would like to mention here is Minimum Maximum Degree Spanning Tree, where, given an input graph G, the objective is to find a spanning tree T of G such that the maximum degree of T is minimized.

This problem is a generalization of the famous Hamiltonian Path problem for which no algorithm faster than $2^n n^{\mathcal{O}(1)}$ is known. It remains open to find even a $2^n n^{\mathcal{O}(1)}$ time algorithm for the Minimum Maximum Degree Spanning Tree problem.

Acknowledgment:

We would like to thank Henning Fernau, Fedor V. Fomin and Daniel Raible for useful discussions on a preliminary version of the presented algorithm.

References

[1] R. Bhatia, S. Khuller, R. Pless, Y. J. Sussmann, The full degree spanning tree problem, Networks 36(4): 203–209, (2000).

[2] A. Björklund, T. Husfeldt, Inclusion–Exclusion Algorithms for Counting Set Partitions, in the proceedings of FOCS'06, 575–582, (2006).

[3] A. Björklund, T. Husfeldt, P. Kaski and M. Koivisto, Fourier meets Möbius: Fast Subset Convolution, in the proceedings of STOC'07, 67–74, (2007).

[4] H. Broersma, O. R. Koppius, H. Tuinstra, A. Huck, T. Kloks, D. Kratsch, H. Müller, Degree-preserving trees, Networks 35(1): 26–39 (2000).

[5] N. Christofides, An Algorithm for the Chromatic Number of a Graph, Computer Journal, 14(1): 38–39, (1971).

[6] F. V. Fomin, S. Gaspers, A. V. Pyatkin, Finding a Minimum Feedback Vertex Set in Time $O(1.7548^n)$, in the proceedings of IWPEC'06, LNCS 4169, 184–191 (2006).

[7] F. V. Fomin, F. Grandoni, D. Kratsch, Measure and Conquer: Domination - A Case Study, in the proceedings of ICALP'05, LNCS 3580, 191–203, (2005).

[8] F. V. Fomin, F. Grandoni, D. Kratsch, Measure and Conquer: A simple $O(2^{0.288n})$ Independent Set Algorithm, in the proceedings of SODA'06, 18–25, (2006).

[9] F. V. Fomin, F. Grandoni, D. Kratsch, Solving Connected Dominating Set Faster Than 2^n, in the proceedings of FSTTCS'06, LNCS 4337, 152–163, (2006).

[10] S. Gaspers, S. Saurabh, A. A. Stepanov, A Moderately Exponential Time Algorithm for Full Degree Spanning Tree, in the proceedings of TAMC08, LNCS 4978, 479–489, (2008).

[11] J. Guo, R. Niedermeier, S. Wernicke, Fixed-Parameter Tractability Results for Full-Degree Spanning Tree and Its Dual, in the proceedings of IWPEC'06, LNCS 4169, 203–214 (2006).

[12] S. Khuller, R. Bhatia, R. Pless: On Local Search and Placement of Meters in Networks. SIAM Journal of Computing 32(2): 470–487, (2003).

[13] M. Koivisto, An $O(2^n)$ Algorithm for Graph Colouring and Other Partitioning Problems via Inclusion-Exclusion, in the proceedings of FOCS'06, 583–590, (2006).

[14] J. B. Kruskal, On the Shortest Spanning Subtree and the Traveling Salesman Problem, in the proceedings of the American Mathematical Society 7: 48–50, (1956).

[15] E. L. Lawler, A Note on the Complexity of the Chromatic Number Problem, Information Processing Letters 5 (3): 66–67, (1976).

[16] M. Lewinter, Interpolation Theorem for the Number of Degree-Preserving Vertices of Spanning Trees, IEEE Transaction Circ. Syst. 34: 205, (1987).

[17] L. E. Ormsbee, Implicit Network Calibration, Journal of Water Resources, Planning and Management, 115(2): 243–257, (1989).

[18] L. E. Ormsbee and D. J. Wood, Explicit Pipe Network Calibration, Journal of Water Resources, Planning and Management, 112(2): 166–182, (1986).

[19] I. W. M. Pothof and J. Schut, Graph-theoretic approach to identifiability in a water distribution network, Memorandum 1283, Universiteit Twente, (1995).

[20] I. Razgon, Exact Computation of Maximum Induced Forest, in the proceedings of SWAT'06, LNCS 4059, 160–171 (2006).

Appendix: Program to Check the Analysis

```
# weights
alpha = 0.23887
beta = 0.722
eta = 0.5

# case 2
print 2**(-beta-3*alpha) + 2**(-beta) < 1
# case 3
# case 4
print 2**(-beta-2*alpha) + 2**(-2*beta) < 1
print 2**(-beta-2*alpha) + 2**(-eta-beta) < 1
# case 5
print 2**(-eta-beta-2*alpha) + 2**(-beta) < 1
print 2**(-2*beta-2*alpha) + 2**(-beta) < 1
# case 6
print 2**(-3*beta-2*alpha) + 2*2**(-3*beta-5*alpha)
    + 2**(-3*beta-8*alpha) + 2**(-2*beta+eta) < 1
print 2**(-3*beta-2*alpha) + 2*2**(-3*beta-6*alpha)
    + 2**(-3*beta-10*alpha) + 2**(-beta) < 1
# case 7
print 2**(-4*beta-2*alpha) + 3*2**(-4*beta-5*alpha)
    + 3*2**(-4*beta-8*alpha) + 2**(-4*beta-11*alpha)
    + 2**(-2*beta+eta) < 1
print 2**(-4*beta-2*alpha) + 2*2**(-4*beta-5*alpha)
    + 2**(-4*beta-6*alpha) + 2**(-4*beta-8*alpha)
    + 2*2**(-4**beta-9*alpha) + 2**(-4*beta-12*alpha)
    + 2**(-beta) < 1
# case 8
print 2**(-5*beta-2*alpha) + 4*2**(-5*beta-5*alpha)
    + 6*2**(-5*beta-8*alpha) + 4*2**(-5*beta-11*alpha)
    + 2**(-5*beta-14*alpha) + 2**(-beta) < 1
```