THE ANNALS OF "DUNĂREA DE JOS" UNIVERSITY OF GALATI
FASCICLE III, 2013, VOL. 36, NO. 1, ISSN 1221-454X
ELECTROTECHNICS, ELECTRONICS, AUTOMATIC CONTROL, INFORMATICS

COUNTING MINIMUM COST BOUNDED DEGREE SUBTREES IN GRAPHS WITH SMALL 2-VERTEX-CONNECTED COMPONENTS

Mugurel Ionuţ Andreica

Computer Science Department, Politehnica University of Bucharest,
Splaiul Independenţei 313, 060042, sector 6, Bucharest, Romania,
email: mugurel.andreica@cs.pub.ro

Abstract: In this paper we present new algorithms for counting minimum cost bounded degree subtrees in connected graphs in which the 2-vertex-connected (biconnected) components have small sizes. The 2-vertex-connected components and the cut vertices can be organized into a block-cut vertex tree which is also a tree decomposition with small width of the graph. We present a dynamic programming algorithm which is very efficient on this particular tree decomposition and we also discuss methods of solving the problem given an arbitrary tree decomposition with small width. Among some of the most important results is an algorithm which can efficiently compute the number of subtrees of a (small) graph corresponding to each possible degree sequence.

Keywords: tree decomposition, block-cut vertex tree, 2-vertex-connected components, bounded-degree subtree.

1. INTRODUCTION

Bounded-degree subgraphs of a given base graph are important structures which arise in a wide variety of scenarios, like network design (Amini et al., 2009), security or parallel processing (Dekker et al., 2012). In this paper we consider a special class of bounded-degree subgraphs, namely *bounded-degree subtrees*. We present new algorithms for counting minimum cost bounded degree subtrees in connected graphs where the 2-vertex-connected (biconnected) components have small sizes. The 2-vertex-connected components and the cut vertices of any connected graph can be organized into a tree structure called a block-cut vertex tree (Andreica, 2006), (Pirzada, 2012). The block-cut vertex tree has two types of nodes, one type corresponding to cut vertices and another type corresponding to the 2-vertex-connected components. This tree is also a small-width tree decomposition (Kintali and Munteanu, 2012) of the graph (the width of the decomposition depends on the size of the largest 2-vertex-connected component). By employing a dynamic programming algorithm on this special tree decomposition we can efficiently find the minimum cost of a degree-bounded subtree of the graph, as well as the number of such minimum cost subtrees.

The time complexity of the obtained algorithm is of the form $O(f(K,D) \cdot N+M)$, where K is the maximum size of a 2-vertex-connected component, D is the maximum allowed degree, N is the number of vertices of the graph and M is the number of edges od the graph. Thus, when K and D are bounded by constants, the algorithm can be considered to have a linear time complexity. We should mention that, in theory, a solution with a time complexity of this form can be immediately derived from (Arnborg and Proskurowski, 1989) (because the problem can be expressed in extended monadic second-order logic). However, the $f(K,D)$ factor is difficult to estimate in this case and may be prohibitively high for a practical implementation. In this paper we will analyze the $f(K,D)$ factor carefully and will focus on obtaining solutions where this factor is as good as possible.

This paper was recommended for publication by Adrian Filipescu

The rest of this paper is structured as follows. We define the problem in more details in Section 2. Then, in Section 3, we present a dynamic programming algorithm which computes the minimum cost of a bounded-degree subtree and the number of such minimum cost subtrees given an arbitrary tree decomposition with small width of the graph. In Section 4 we show how we can use the special structure of the block-cut vertex tree in order to obtain an improved dynamic programming solution. This improved solution will also be based on counting the number of minimum cost bounded-degree subtrees of each biconnected component having every possible degree sequence. An initial solution which enumerates all the possible subtrees is given in Section 5 and much more efficient solutions are given in Sections 6 and 7. In Section 8 we present experimental results for the block-cut vertex tree-based dynamic programming algorithm which uses the algorithms from Sections 5, 6 and 7. In Section 9 we discuss some alternative algorithms to those presented in Sections 5, 6 and 7. In Section 10 we discuss related work and in Section 11 we conclude.

2. PROBLEM DEFINITION

We consider an undirected graph G with N vertices and M edges. Each edge (i,j) has an associated cost $c(i,j)$ (which may be positive, zero, or negative). A subtree of G consists of a subset of vertices V of G and a subset of edges E of G such that $|E|=|V|-1$, each endpoint of an edge from E is part of V and the edges from E form no cycles. Thus, the subtree is uniquely defined by the pair (V,E). The cost of a subtree is equal to the sum of the costs of the edges from the set E. The degree of a vertex v of the subtree is equal to the number of edges from E incident to it. In this paper we are interested in subtrees for which the maximum degree of any node in the subtree is at most equal to an upper limit D. We will call such subtrees *D-degree-bounded subtrees*. We want to compute the minimum cost of a D-degree-bounded subtree as well as the number of such minimum cost subtrees.

3. USING DYNAMIC PROGRAMMING ON AN ARBITRARY TREE DECOMPOSITION OF THE GRAPH

If the graph G has small treewidth (Kintali and Munteanu, 2012) and a tree decomposition of G can be computed (or is given), then we can use dynamic programming techniques in order to efficiently count the number of minimum cost D-degree-bounded subtrees of G. Each node X of the tree decomposition has an associated subset $v(X)$ of $nv(X)$ vertices of G: $v(X,1), v(X,2), \ldots, v(X,nv(X))$. The main properties of a tree decomposition are as follows:

- for every edge (i,j) of G we must have at least one node X of the tree decomposition such that both i and j belong to the subset $v(X)$

- if vertex i belongs to both $v(X)$ and $v(Y)$ then i belongs to the subsets of all the nodes Z located on the unique path from X to Y in the tree decomposition

Any tree decomposition can be easily transformed into a *nice tree decomposition* (Kintali and Munteanu, 2012) consisting of three types of nodes (besides the leaves):

- *Introduce node:* A node X is an *Introduce Node* if it has a single child Y, $nv(X)=nv(Y)+1$ and there exists an index j ($1 \le j \le nv(X)$) such that $v(X,i)=v(Y,i)$ for $1 \le i \le j-1$ and $v(X,i+1)=v(Y,i)$ for $j \le i \le nv(Y)$. Thus, node X *introduces* the vertex $v(X,j)$.

- *Forget node:* A node X is a *Forget Node* if it has a single child Y, $nv(X)=nv(Y)-1$ and there exists an index j ($1 \le j \le nv(Y)$) such that $v(X,i)=v(Y,i)$ for $1 \le i \le j-1$ and for $j+1 \le i \le nv(X)$. Thus, node X *forgets* the vertex $v(Y,j)$.

- *Join node:* A node X is a *Join node* if it has exactly two children Y and Z such that $nv(X)=nv(Y)=nv(Z)$ and the subsets $v(X)$, $v(Y)$ and $v(Z)$ are identical (although we may have the vertices in different orders in the three nodes).

We will use the nice tree decomposition for our dynamic programming algorithm instead of the original tree decomposition, because the description of the algorithm is simplified this way.

For each node X of the tree decomposition we will define by $GI[X]$ the subgraph induced by the vertices from $v(X)$ and the edges between them and by $GS[X]$ the subgraph induced by the vertices from $v(X)$ and all of its descendants in the tree decomposition, and the edges between these vertices.

For each node X of the tree decomposition we will compute a table $T(X)$ containing two values (*cmin* and *cnt*) for each possible state S. A state S is defined as a pair $(GP(S), DS(S))$, where:

- $GP(S)$ is a partition of $GI[X]$ into an arbitrary number (zero or more) of vertex-disjoint D-degree-bounded subforests: $GPS(S,1)$, $GPS(S,2)$, ..., $GPS(S,|GP(S)|)$ (with some vertices of $GI[X]$ possibly left out of any of the $|GP(S)|$ subforests), and

- $DS(S)$ is a degree sequence: $DS(S,1)$, $DS(S,2)$, ..., $DS(S,nv(X))$ such that $0 \le DS(S,i) \le D$.

Each subforest $GPS(S,i)$ consists of a subset of vertices $VGPS(S,i)$ of $v(X)$ and a subset of edges $EGPS(S,i)$, such that $|EGPS(S,i)| \le |VGPS(S,i)|-1$, all the endpoints of the edges from $EGPS(S,i)$ are part of

VGPS(S,i) and the edges from *EGPS(S,i)* form no cycles ($1 \le i \le |GP(S)|$). Moreover, as stated, the intersection of any two subsets *VGPS(S,i)* and *VGPS(S,j)* is void ($1 \le i,j \le |GP(S)|$) and some vertices of *v(X)* may be left out of any subset *VGPS(S,i)* ($1 \le i \le |GP(S)|$). If a vertex *v(X,j)* does not belong to any subset *VGPS(S,i)* ($1 \le i \le |GP(S)|$) then we must have *DS(S,j)=0*.

A state *S* for a node *X* defines a possible intersection of a D-degree-bounded subtree *ST* of *GS[X]* with *GI[X]*. The subforests *GPS(S,i)* contain the actual vertices and edges of the intersection. If a subforest contains multiple connected components (subtrees) this means that the vertices of the subforest were connected by vertices contained by descendants of the node *X*.

The degree sequence *DS(S)* contains the degrees of all the vertices of *v(X)* in *ST* (*DS(S,i)* is the degree of *v(X,i)*). Note that the degree of a vertex *v(X,i)* depends both on its subforest neighbors in *GI[X]* and on its *ST* subtree neighbors in *GS[X]\GI[X]* ($1 \le i \le nv(X)$). *T(X,S).cmin* will contain the minimum cost among all the D-degree-bounded subtrees *ST* of *GS[X]* corresponding to the state *S* (intersection and degree sequence) and *T(X,S).cnt* will contain the number of such minimum cost subtrees *ST*. If no subtree corresponds to a given state *S* then we will implicitly assume that $T(X,S).cmin = +\infty$ and that *T(X,S).cnt=0*. In terms of implementation we can use a hash table for *T(X)*, storing information only for those states *S* for which $T(X,S).cmin < +\infty$ and *T(X,S).cnt>0*. Then, if a state *S'* is not found in *T(X)* we will assume that $T(X,S').cmin = +\infty$ and *T(X,S').cnt=0*.

We will now show how to compute the tables *T(X)* for each node *X* of the tree decomposition. If *X* is a leaf node then we will simply generate all the possible partitions of *GI[X]* into any number of D-degree-bounded *subtrees* (including the possibility of leaving out some vertices of *v(X)*). For each such partition *GP(S)* we will have *DS(S,i)*=the degree of *v(X,i)* in its subtree (or *0*, if *v(X,i)* was left out of any subtree) ($1 \le i \le nv(X)$). Note that in this case we will have *T(X,S).cmin* = the sum of the edges costs in all the subtrees of the partition and *T(X,S).cnt=1*. A very simple method for generating all the partitions is to first select which vertices are left out and then decide which edges are kept among the edges connecting vertices which are not left out. Once the subset of vertices and edges is selected, we only need to verify that the connected components of the selected edges are D-degree-bounded subtrees.

If *X* is an *Introduce node* in the tree decomposition then we will proceed as follows. Let *Y* be the only child of *X* and let *v(X,j)* be the introduced vertex. Let *e(X,j)* be the set of edges having *v(X,j)* as an endpoint in *GI[X]*. We will consider every state *S* from *T(Y)* (such that *T(Y,S).cnt>0*) and every subset *se(X,j)* of at most *D* edges from *e(X,j)* (including the empty subset), such that if the edge *(v(X,i),v(Y,k))* is part of *se(X,j)* then we must have *DS(S,k)≤D-1* and *v(Y,k)* must belong to at least one subforest of *GP(S)* (i.e. it must not have been left out). Moreover, any two vertices *v(Y,a)* and *v(Y,b)* for which the edges *(v(X,j),v(Y,a))* and *(v(X,j),v(Y,b))* are part of *se(X,j)* must belong to different subforests in *GP(S)*.

All the subforests of *GP(S)* which are connected by edges from *se(X,j)* are merged into a single subforest (which will also include *v(X,j)*), thus obtaining a new partition *GP(S')* corresponding to a state *S'* for the node *X* (the subforests of *GP(S)* which are not connected by an edge to *v(X,j)* are *copied* as they are into *GP(S')*). We will obtain *DS(S')* from *DS(S)* by inserting on position *j* $DS(S',j)=|se(X,j)|$. Then, for every vertex *v(X,k)* such that the edge *(v(X,j),v(X,k))* is part of *se(X,j)* we will increment *DS(S',k)* by *1*. If $|se(X,j)|=0$ then *GP(S')* will be obtained from *GP(S)* by adding a new subforest containing the single node *v(X,j)*. Let *Cmin=T(Y,S)* + the sum of the costs of the edges from *se(X,j)* and *Cnt=T(Y,S).cnt*. If *Cmin<T(X,S').cmin* then we will set *T(X,S').cmin=Cmin* and *T(X,S').cnt=Cnt*; otherwise, if *Cmin=T(X,S').cmin* then we will increment *T(X,S').cnt* by *Cnt*.

The part presented so far corresponds to the case when the introduced vertex is selected to be part of the D-degree-bounded subtree. We must also consider the case when the introduced vertex is left out. In this case we will consider every state *S* from *T(Y)* and we will try to extend it to a valid state *S'* for *X*. We will have *GP(S')=GP(S)* and *DP(S')* is obtained from *DP(S)* by inserting on position *j* *DP(S',j)=0*. If *T(Y,S).cmin<T(X,S').cmin* then we will set *T(X,S').cmin=T(Y,S).cmin* and *T(X,S').cnt=T(Y,S).cnt*; otherwise, if *T(Y,S).cmin=T(X,S').cmin* then we will increment *T(X,S').cnt* by *T(Y,S).cnt*.

If *X* is a *Forget node* then let *Y* be its only child and let *v(Y,j)* be the *forgotten* vertex. We will consider all the states *S* of *T(Y)* (such that *T(Y,S).cnt>0*) in which:

- *v(Y,j)* belongs to no subforest of *GP(S)*

- *v(Y,j)* belongs to a subforest of *GP(S)* with at least two nodes

For each such state *S* we will obtain a new state *S'* as follows. If *v(Y,j)* belongs to no subforest of *GP(S)* then we will have *GP(S')=GP(S)*; otherwise we will obtain *GP(S')* from *GP(S)* by removing the vertex *v(Y,j)* and all the edges adjacent to it from the subforest to which it belongs. *DS(S')* will be obtained from *DS(S)* by simply removing the entry *DS(S,j)*. Then, if *T(Y,S).cmin<T(X,S').cmin* we will set *T(X,S').cmin=T(Y,S).cmin* and

T(X,S').cnt=T(Y,S).cnt; otherwise, if *T(Y,S).cmin=T(X,S').cmin* then we will increment *T(X,S')* by *T(Y,S)*.

If *X* is a *Join node* then let *Y* and *Z* be its two children. We will first reorder the vertices of *v(Y)* and *v(Z)* such that $v(X,i)=v(Y,i)=v(Z,i)$ ($1 \le i \le nv(X)$). The reordering will also modify the components *DS(SY)* and *DS(SZ)* of all the states *SY* of *T(Y)* and *SZ* of *T(Z)*. Then we will consider every pair of states *(SY,SZ)* such that *T(Y,SY).cnt>0, T(Z,SZ).cnt>0, GP(SX)=GP(SY)* and $DS(SY,i)+DS(SZ,i)-DegP(SY,v(Y,i)) \le D$ (for $1 \le i \le nv(X)$). We denoted by *DegP(SY,v(Y,i))* the degree of the vertex *v(Y,i)* in the subforest of *GP(SY)* to which it belongs (i.e. the number of edges of the subforest which are adjacent to *v(Y,i)*). If *v(Y,i)* does not belong to any subforest of *GP(SY)* then *DegP(SY,v(Y,i))=0*. We obtain a new state *SX* with *GP(SX)=GP(SY)* and $DS(SX,i)=DS(SY,i)+DS(SZ,i)-DegP(SY,v(Y,i))$ (for $1 \le i \le nv(X)$). Let *Cmin* be equal to *T(Y,SY)+T(Z,SZ)* minus the sum of the costs of the edges in all the subforests of *GP(SY)*, and let *Cnt* be equal to $T(Y,SY) \cdot T(Z,SZ)$. If *Cmin<T(X,SX).cmin* then we will set *T(X,SX).cmin=Cmin* and *T(X,SX).cnt=Cnt*; otherwise, if *Cmin=T(X,SX).cmin* then we will increment *T(X,SX).cnt* by *Cnt*.

In order to find the minimum cost of a D-degree-bounded subtree *MinCost* and the number of such minimum cost subtrees *NumSubtrees*, we will proceed as follows. We will initialize *MinCost=0* and *NumSubtrees=N+1* (corresponding to the empty subtree and the subtrees containing a single vertex). Then we will consider every node *X* whose parent is a *Forget* node. We will also consider the root node of the tree decomposition to belong to this category (because all of its vertices are *forgotten* after it). Let's denote the set of vertices forgotten by this node's parent by *F* (usually *F* contains only one vertex, except in the case of the root when it contains all of the root's vertices). We will consider all the states *S* for which *GP(S)* contains exactly one subforest, *T(X,S).cnt>0* and *DS(S,i)>0* for at least one position *i* ($1 \le i \le nv(X)$) for which *v(X,i)* belongs to *F* (i.e. we consider only subtrees containing at least one forgotten vertex). If *T(X,S).cmin<MinCost* we will set *MinCost=T(X,S).cmin* and *NumSubtrees=T(X,S).cnt*; otherwise, if *T(X,S).cmin=MinCost* then we will increment *NumSubtrees* by *T(X,S).cnt*.

Let's analyze the time complexity of the presented algorithm. Let *tw* be the width of the tree decomposition we used. The number of states computed for each node is of the order $NStates=(tw+1)^{2 \cdot tw} \cdot D^{tw+1}$. An *Introduce node* can be processed in time $O(2^{tw} \cdot NStates \cdot CopyState(tw+1))$, where *CopyStates(u)* denotes the time complexity of copying a state into another, where *u* is the maximum number of vertices (in its subforests and in its degree sequence). In this case *CopyState(u)* is of the order *O(u)*. A leaf node and a *Forget node* can be processed in time $O(Nstates \cdot CopyState(tw+1))$. A *Join node* can be processed in time $O((tw+1)^{2 \cdot tw} \cdot D^{2 \cdot (tw+1)} \cdot CopyState(u))$. It is obvious that handling a *Join node* is more complex than the other types of nodes and this dominates the time complexity of the algorithm, which becomes $O((tw+1)^{2 \cdot tw} \cdot D^{2 \cdot (tw+1)} \cdot CopyState(u) \cdot N)$, because the number of nodes of the tree decomposition is of the order *O(N)*.

Note that in our analysis we assumed that arithmetic operations involving the costs and the number of subtrees (the *cmin* and *cnt* fields of the tables) take constant time. Although this may be a valid assumption regarding the costs, note that the number of subtrees may be exponential in the number of nodes of the tree. Thus, the assumption that the fields *cmin* and *cnt* have constant size may be false. In this case we should add extra factors to the time complexity regarding the addition and multiplication of numbers with a non-constant number of bits. However, there are scenarios when even the *cnt* fields may be safely assumed to be constant – for instance, when we are not interested in the exact number of subtrees, but rather in the number of subtrees modulo a given number *P*. In these cases all the additions and multiplications involving the *cnt* fields can be performed modulo *P* – thus, these numbers will never exceed the number of bits *P* has. We will not discuss this issue in more details in the rest of the paper and in all the other time complexity analyses we will assume that arithmetic operations involving the *cmin* and *cnt* fields take constant time.

4. USING DYNAMIC PROGRAMMING ON THE BLOCK-CUT VERTEX TREE

The block-cut vertex tree of the graph is a tree decomposition of the graph obtained as follows. We have a node in the tree for each 2-vertex-connected component and a node for each cut vertex (critical node). Two nodes *X* and *Y* are adjacent in the tree if *X* corresponds to a 2-vertex-connected component *B*, *Y* corresponds to a cut vertex *C*, and *C* belongs to *B* (note that a cut vertex may belong to multiple 2-vertex-connected components). We will choose a node corresponding to a cut vertex as the root of the tree. If the graph is 2-vertex-connected (i.e. it has no cut vertices) we will artificially mark one of the graph vertices as a cut vertex (in this case the tree will only have two nodes).

Fig. 1 shows a graph with 13 vertices, where the cut vertices are marked with a darker background. Fig. 2 shows its block-cut vertex tree, where one of the cut vertices was chosen as the root.

For a node *X* corresponding to a cut vertex *C* we will have the subset of vertices *v(X)={C}*, i.e. *nv(X)=1*

and *v(X,1)=C*. For a node *Y* corresponding to a 2-vertex-connected component we will have the subset of vertices *v(Y)* as the subset of vertices belonging to the 2-vertex-connected component.

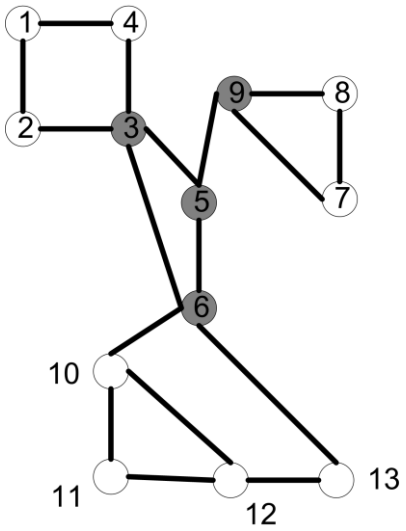

Fig.1. A graph with 13 vertices. Cut vertices (labeled 3, 5, 6 and 9) are marked with a darker background.

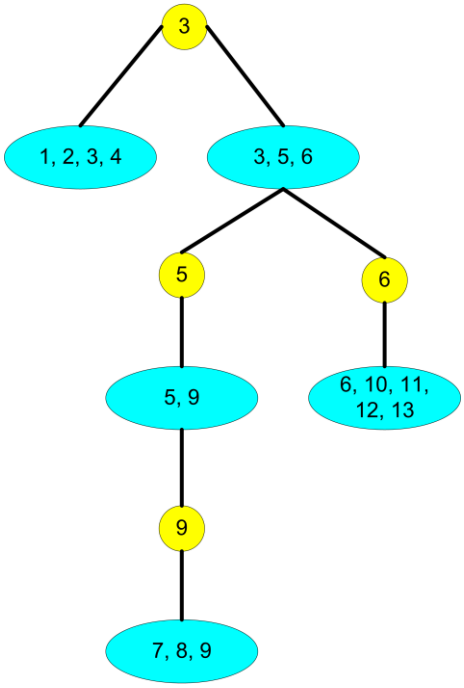

Fig.2. The block-cut vertex tree of the graph from Fig. 1.

We will denote by *nc(X)* the number of children in the block-cut vertex tree of the node *X*. The children will be denoted as *child(X,1), ..., child(X,nc(X))*. For a node *X* corresponding to a 2-vertex-connected component (i.e. *nv(X)≥2*) we will order the vertices in its subset as follows. *v(X,1)* will be the cut vertex *C* corresponding to the parent of node *X*. *v(X,i)* will be the cut vertex *C* corresponding to the child *child(X,i-1)* (for *2≤i≤nc(X)*+1). The remaining vertices of the subset (from *nc(X)+1* to *nv(X)*) may be ordered arbitrarily (they are not cut vertices and are only part of the subset *v(X)*).

With a block-cut vertex tree it will be enough to store some simplified states during the dynamic programming algorithm. In fact, a state *S* for a node *X* will consist only of a degree sequence *(DS(S,1), ..., DS(S,nv(X))* (with *0≤DS(S,i)≤D* for *1≤i≤nv(X)*). We do not need the partition into subforests because the intersection between a subtree of the graph and a 2-vertex-connected component is either void or consists of a single subtree. Moreover, in a block-cut vertex tree there are no *Join nodes*, for which the exact partition into subforests (and the edges of the subforests) was most important.

Since by using only the degree sequence we are unable to distinguish between subtrees containing a single vertex and subtrees containing zero vertices, our algorithm will compute only subtrees containing at least two vertices (thus, if *DS(S,i)=0* for a node *X* this will mean that the vertex *v(X,i)* is not part of the subtree). Subtrees with only one or with zero vertices will be considered separately. Apart from the simplified states we will compute some tables *T(X,i)* for each node *X* of the tree, having a similar meaning as before. The extra index *i* refers to the size of the degree sequence. For nodes *X* corresponding to 2-vertex-connected components this index will decrease below *nv(X)* because we will perform a state reduction after considering each of its children and also before considering any of its children.

If *X* is a node corresponding to a 2-vertex-connected component (i.e. *nv(X)≥2*) we will proceed as follows. For each possible sequence of degrees *DS(1), ..., DS(nv(X))* we will compute the minimum cost of a D-degree-bounded subtree fully contained in the 2-vertex-connected component and the number of such minimum cost subtrees in the table *T(X,nv(X))* (*T(X,nv(X),DS).cmin* and *T(X,nv(X),DS).cnt*). We can achieve this by enumerating all the D-degree-bounded subtrees (as in Section 5) or, more efficiently, by using the algorithms from Sections 6 and 7. If there are no subtrees corresponding to a degree sequence *DS* we will consider *T(X,nv(X),DS).cmin=+∞* and *T(X,nv(X),DS).cnt=0*.

Before proceeding further we will describe the reduction operation *Reduce(X, A, B)*. This operation will consider all the states from *T(X,A)* and *reduce* them into *T(X,B)* (*B<A*). The reduction operation is based on the premise that the degree information about the nodes *v(X,B+1), ..., v(X,A)* is no longer needed further. The *Reduce* operation can be implemented easily. We will consider all the states *DS* from *T(X,A)*. For each such state *DS* we will compute the state *DS'* containing only the first *B* entries of *DS* (i.e. *DS'(i)=DS(i)* for *1≤i≤B*). If *DS'*

contains at least a non-zero entry then we continue as follows. If $T(X,A,DS).cmin<T(X,B,DS').cmin$ then we set $T(X,B,DS').cmin = T(X,A,DS).cmin$ and $T(X,B,DS').cnt = T(X,A,DS).cnt$; otherwise, if $T(X,A,DS).cmin = T(X,B,DS').cmin$ then we will increment $T(X,B,DS').cnt$ by $T(X,A,DS).cnt$. We will also have, by definition, $T(X,B,DSZ_B).cmin=0$ and $T(X,B,DSZ_B).cnt=1$, where DSZ_B is a degree sequence containing B zeros.

After computing the table $T(X,nv(X))$ for a node X corresponding to a 2-vertex-connected component we will call $Reduce(X, nv(X), nc(X)+1)$ (only if $nv(X)>nc(X)$). Then we will consider, in order, the tables $T(X,j)$ (with j going down from $nc(X)+1$ to 2). We will consider each state DS of $T(X,j)$ and we will also consider each state DS' of $T(child(X,j-1),1)$ such that $DS(j)+DS'(1)\leq D$. We now have a new state DS'' which is identical to DS, except that we remove the last (j^{th}) entry from it. If DS'' contains at least one non-zero entry then we continue as follows. Let $Cmin$ be $T(X,j,DS).cmin+T(child(X,j-1),1,DS').cmin$ and let Cnt be $T(X,j,DS).cnt \cdot T(child(X,j-1),1,DS').cnt$. If $Cmin<T(X,j-1,DS'').cmin$ then we will set $T(X,j-1,DS'').cmin=Cmin$ and $T(X,j-1,DS'').cnt=Cnt$; otherwise, if $Cmin=T(X,j-1,DS'')$ we will increment $T(X,j-1,DS'').cnt$ by Cnt. We will also have, by definition, $T(X,j-1,DSZ_{j-1}).cmin=0$ and $T(X,j-1,DSZ_{j-1}).cnt=1$.

Let's assume now that X is a node corresponding to a cut vertex (i.e. $nv(X)=1$). In this case there are only $D+1$ possible states (one for each possible degree of the cut vertex in the subtree, from 0 to D). Note that since we only consider subtrees with at least two vertices (i.e. containing at least one edge), we will have $T(X,1,DSZ_1).cmin=0$ and $T(X,1,DSZ_1).cnt=1$ (corresponding to an empty subtree). We will use a knapsack-like dynamic programming algorithm for computing $T(X,1,DS)$ ($1\leq DS(1)\leq D$).

We will use a temporary table $Tmp(X)$ where $Tmp(X,j,DS)$ contains the computed values ($cmin$ and cnt) corresponding to the degree sequence DS after considering the first j children of X. We have $Tmp(X,0,DSZ_1).cmin=0$ and $Tmp(X,0,DSZ_1).cnt=1$ (no other states except for DSZ_1 are defined for $j=0$).

For $1\leq j\leq nc(X)$ we will consider all the states DS of $Tmp(X,j-1)$ (i.e. such that $Tmp(X,j-1,DS).cnt>0$) and all the states DS' of $T(child(X,j),1)$ (i.e. such that $T(child(X,j),1,DS').cnt>0$), such that $DS(1)+DS'(1)\leq D$. For each such combination we obtain a new state DS'' with a single entry $DS''(1)=DS(1)+DS'(1)$. If $DS''(1)>0$ then we continue as follows. Let $Cmin$ be $Tmp(X,j-1,DS).cmin+T(child(X,j),1,DS').cmin$ and let Cnt be $Tmp(X,j-1,DS).cnt \cdot T(child(X,j),1,DS').cnt$. If $Cmin< Tmp(X,j,DS'').cmin$ then we will set $Tmp(X,j,DS'').cmin=Cmin$ and $Tmp(X,j,DS'').cnt= Cnt$; otherwise, if $Cmin=Tmp(X,j,DS'')$ we will increment $Tmp(X,j,DS'').cnt$ by Cnt. We will have $T(X,1,DS)=Tmp(X,nc(X),DS)$ for every degree sequence DS such that $1\leq DS(1)\leq D$. At the end we will set $T(X,1,DSZ_1).cmin=0$ and $T(X,1,DSZ_1).cnt=1$.

In order to find the minimum cost of a D-degree-bounded subtree $MinCost$ and the number of such minimum cost subtrees $NumSubtrees$, we will proceed as follows. We will initialize $MinCost=+\infty$ and $NumSubtrees=0$. Then we will consider every node X corresponding to a cut vertex and every state DS of $T(X,1)$, such that $DS\neq DSZ_1$. If $T(X,1,DS).cmin<MinCost$ then we will set $MinCost=T(X,1,DS).cmin$ and $NumSubtrees= T(X,1,DS).cnt$; otherwise, if $MinCost= T(X,1,DS).cmin$ then we will increment $NumSubtrees$ by $T(X,1,DS).cnt$.

Afterwards we will consider nodes X corresponding to 2-vertex-connected components. For these nodes we will consider several situations. First of all, during the call $Reduce(X, nv(X), nc(X)+1)$, whenever we encounter a degree sequence DS' containing only zeros, we will compute $Cmin=T(X,nv(X),DS).cmin$ and $Cnt=T(X,nv(X),DS).cnt$. If $Cmin<MinCost$ then we will set $MinCost=Cmin$ and $NumSubtrees=Cnt$; otherwise, if $Cmin=MinCost$ then we will increment $NumSubtrees$ by Cnt. Second, while considering the states of the table $T(X,j)$ (with the purpose of filling in the table $T(X,j-1)$) ($2\leq j\leq nc(X)+1$), we may reach states $DS''=DSZ_{j-1}$. In these cases we will compute $Cmin$ and Cnt as if DS'' contained at least one non-zero entry and we update $MinCost$ and $NumSubtrees$ (if it is the case) according to the rules mentioned above.

The time complexity of the presented algorithm depends on the maximum size K of a 2-vertex-connected component. There are $O(min\{D+1,K\}^K)$ states generated initially for each 2-vertex-connected component (by using one of the algorithms from the next three sections). We will denote the time complexity for computing the values associated to these states by $TI(K,D)$. Then, for each cut vertex of the 2-vertex-connected component we traverse all the relevant states S and try to combine each such state with the $O(D+1)$ states computed for the cut vertex. Because of our state reduction procedure, if the 2-vertex-connected component contains $Q\geq 1$ cut vertices, handling all of them will take $O((D+1)\cdot((D+1)^Q+(D+1)^{Q-1}+\ldots+(D+1)^1)) = O((D+1)^{Q+1})$. The worst case situation occurs when we have large 2-vertex-connected components (with K vertices each, if possible) and as many as possible of these vertices are cut vertices. In order for a vertex of a 2-vertex-connected component to be a cut vertex in this scenario we need to attach it to another 2-vertex-connected component containing $K-1$ other vertices. Thus, in order to have P cut vertices we will need to have at least $N=K+P\cdot(K-1)$ vertices in our graph. Since we don't need exact numbers we can conclude

that we may have $O(N/K)$ 2-vertex-connected components and $O(N/K)$ cut vertices (this means that only $O(N/K^2)$ 2-vertex-connected-components can have all of their vertices as cut vertices). The overall time complexity is $O(N/K \cdot (TI(K,D)+K^2)+N/K^2 \cdot (D+1)^{K+1})$, where $TI(K,D)$ is, at best, $O(min\{D+1,K\}^K)$ (and, in the worst case, $TI(K,D)=O(K^K)$). The $O(N/K \cdot K^2)= O(N \cdot K)$ term stands for the number of edges of the graph. Constructing the block-cut vertex tree takes $O(N+M)$ time, which in this case is $O(N \cdot K)$.

5. ENUMERATING D-DEGREE-BOUNDED SUBTREES

In this section we present a recursive algorithm which generates all the D-degree-bounded subtrees of a graph. During the execution of the algorithm we will maintain three types of values:

- *CT*: the cost of the subtree generated so far
- a tuple $(deg_1, deg_2, \ldots, deg_N)$ representing the degrees of the vertices $1, 2, \ldots, N$, in the subtree generated so far ($0 \le deg_i \le D$)
- a tuple $(done_1, \ldots, done_N)$, where $done_i=0$ or 1; if $done_i=1$ it means that the degree deg_i of the vertex i is final (i.e. it cannot be increased any more by the algorithm); if $done_i=0$ the degree of vertex i is not final, yet.

The generated subtrees will be considered to be rooted at the smallest indexed vertex which is part of the subtree. Moreover, we will only generate subtrees containing at least one edge. The $N+1$ zero cost subtrees consisting of 1 vertex or of no vertex will be considered separately. The main loop of the algorithm will consider all the vertices r of the graph in increasing order ($1 \le r \le N$) and generate all the subtrees rooted at the vertex r. For each root r we initialize the three types of values as follows:

- $CT=0$
- $deg_k=0$ for $1 \le k \le N$
- $done_k=1$ for $1 \le k \le r-1$ and $done_k=0$ for $r \le k \le N$

The recursive function *GenerateSubtrees* described below will be called with the arguments r, 1, CT, deg_k ($1 \le k \le N$) and $done_k$ ($1 \le k \le N$). The *GenerateSubtrees* function adds new children to the node r in the current subtree which is being generated. Then the function chooses the next node to which children may be added (if any) and calls itself with a new set of arguments.

```
GenerateSubtrees(r, kmin, CT, deg_1, …, deg_N, done_1, …, done_N):
for k=kmin to D-deg_r do {
  let S(k)=the set of all subsets of k vertices v(1),…,v(k) such that v(i)≠r, done_v(i)=0 and the edge (r,v(i)) exists in the graph (for 1≤i≤k)
  for each subset (v(1),…,v(k)) in S(k) do {
    CT' = CT + sum of the values c(r,v(i)) (for 1≤i≤k)
    deg_i'=deg_i (for 1≤i≤N, i≠r and i≠v(j) for 1≤j≤k)
    deg_r'=deg_r+k
    deg_v(i)'=deg_v(i)+1 (1≤i≤k)
    done_i'=done_i (for 1≤i≤N, i≠r)
    done_r'=1
    let r'=the smallest index such that done_r'=0.
    if r does not exist then a new subtree was completely generated, having the sequence of degrees deg_1', …, deg_N' and cost CT'
    else call GenerateSubtrees(r', 0, deg_1', …, deg_N', done_1', …, done_N')
  }
}
```

In order to find the minimum cost of a D-degree-bounded subtree and the number of such subtrees we will initialize (in the beginning) two variables: $MinCost=0$ and $NumSubtrees=N+1$, corresponding to the $N+1$ zero cost subtrees containing at most 1 vertex. Then, in the *GenerateSubtrees* function, whenever a new subtree is completely generated, we need to compare CT' to $MinCost$. If $CT'<MinCost$ then we will set $MinCost=CT'$ and $NumSubtrees=1$; otherwise, if $CT'=MinCost$ then we will increment $NumSubtrees$ by 1.

6. IMPROVED COUNTING OF MINIMUM COST D-DEGREE-BOUNDED SUBTREES

In this section we will present an algorithm which, for each possible sequence of vertex degrees $(deg_1, \ldots, deg_N)$, computes the minimum cost D-degree-bounded subtree ($cmin(deg_1, \ldots, deg_N)$) as well as the number of such minimum cost subtrees ($cnt(deg_1, \ldots, deg_N)$).

The algorithm presented in the previous section can be used for this purpose, because for each generated subtree it also has its sequence of vertex degrees. However, we will present a faster algorithm in this section which does not need to generate all the subtrees for this. Note that the number of different degree sequences can be significantly smaller than the number of D-degree-bounded subtrees. A rough approximation of the number of degree sequences is $(D+1)^N$. However, the number of valid degree sequences is much smaller because the sum of the vertex degrees must be an even number at most equal to $2 \cdot N-2$. Moreover, if the sum of vertex degrees is $2 \cdot (G-1)$ (for $G \ge 2$) then we must have exactly G non-zero entries in the degree sequence. In Section 8 we present experimental results regarding the number of D-degree-bounded subtrees of a complete graph and the number of degree sequences.

The algorithm presented in this section will actually compute some auxiliary values *auxcmin(deg_1, …, deg_N, $done_1$, …, $done_N$)* and *auxcnt(deg_1, …, deg_N, $done_1$, …, $done_N$)*, where the $done_k$ ($1 \le k \le N$) values have the same meaning as in the previous section. We will assume that all the **cmin* values are initialized to $+\infty$ and all the **cnt* values are initialized to *0*.

We will use a data structure *DS* in which we will insert the tuples *(deg_1, …, deg_N, $done_1$, …, $done_N$)* for which we computed *auxcmin* and *auxcnt* values. As in the algorithm from the previous section we will consider every possible vertex *r* as a potential root for the counted subtrees. For each root *r* we will initialize the deg_k and $done_k$ ($1 \le k \le N$) values as before. We will set *auxcmin(deg_1, …, deg_N, $done_1$, …, $done_N$)=0* and *auxcnt(deg_1, …, deg_N, $done_1$, …, $done_N$)=1*. Then we will call the *CountSubtrees* function described below with the arguments *kmin=1* and the initialized deg_k and $done_k$ ($1 \le k \le N$) values.

```
CountSubtrees(kmin, deg1, …, degN, done1, …, doneN):
let r=the smallest index such that doner=0.
if (r does not exist) {
  if (auxcmin(deg1,…,degN,done1,…,doneN)
      < cmin(deg1,…,degN)) {
    cmin(deg1,…,degN) =
    auxcmin(deg1,…,degN,done1,…,doneN)
    cnt(deg1,…,degN) =
    auxcnt(deg1,…,degN,done1,…,doneN)
  } else if
  (auxcmin(deg1,…,degN,done1,…,doneN) =
   cmin(deg1,…,degN)) {
    cnt(deg1,…,degN) +=
    auxcnt(deg1,…,degN,done1,…,doneN)
  }
} else {
  for k=kmin to D-degr do {
    let S(k)=the set of all subsets of
k vertices v(1),…,v(k) such that
v(i)≠r, donev(i)=0 and the edge (r,v(i))
exists in the graph (for 1≤i≤k)
    for each subset (v(1),…,v(k)) in
S(k) do {
      cmin' =
      auxcmin(deg1,…,degN,done1,…,doneN)
      + sum of the values c(r,v(i))
      (for 1≤i≤k)
      cnt' =
      auxcnt(deg1,…,degN,done1,…,doneN)
      degi'=degi (for 1≤i≤N, i≠r and
i≠v(j) for 1≤j≤k)
      degr'=degr+k
      degv(i)'=degv(i)+1 (1≤i≤k)
      donei'=donei (for 1≤i≤N, i≠r)
      doner'=1
      if (cmin' <
          auxcmin(deg1',…,degN',
                  done1',…,doneN') {
        if (auxcmin(deg1',…,degN',
            done1',…,doneN')=+∞) then
          insert (deg1',…,degN',
             done1',…,doneN') into DS
          auxcmin(deg1',…,degN',
                  done1',…,doneN') = cmin'
          auxcnt(deg1',…,degN',
                 done1',…,doneN') = cnt'
      } else if (cmin' =
            auxcmin(deg1',…,degN',
                    done1',…,doneN') {
          auxcnt(deg1',…,degN',
                 done1',…,doneN') += cnt'
    }
  }
}
```

After considering every possible root for the counted subtrees the algorithm will proceed as follows. As long as the data structure *DS* is not empty we extract the *first* tuple *(deg_1,…,deg_N,$done_1$,…,$done_N$)* from *DS* and then call *CountSubtrees(0,deg_1,…,deg_N,$done_1$,…,$done_N$)*. The tuples from *DS* are sorted according to the number of values equal to *1* among the $done_1$, …, $done_N$ values. Thus, the first tuple from *DS* will be the one containing the smallest number of *1* values among the $done_1$, …, $done_N$ values. Note that *DS* may be implemented in multiple ways, from a standard priority queue to an array of queues, each position *P* of the array representing a queue storing all the tuples *(deg_1, …, deg_N, $done_1$, …, $done_N$)* having *P* values equal to *1* among the $done_1$, …, $done_N$ values.

It is easy to see that this algorithm does not need to generate each D-degree-bounded subtree independently. Instead, all the subtrees corresponding to the same degree sequence and with the same set of $done_1$, …, $done_N$ values are handled together (only the minimum cost of a subtree and the number of minimum cost subtrees are needed for each such class of subtrees – these numbers are stored in the *auxcmin* and *auxcnt* tables).

In order to find the minimum cost of a D-degree-bounded subtree and the number of such subtrees we will initialize, as before, *MinCost=0* and *NumSubtrees=1*. Then we will consider all the tuples *(deg_1,…,deg_N)* such that *cmin(deg_1,…,deg_N)*$<+\infty$. If *cmin(deg_1,…,deg_N)<MinCost* then we will set *MinCost=cmin(deg_1,…,deg_N)* and *NumSubtrees=cnt(deg_1,…,deg_N)*; otherwise, if *cmin(deg_1,…,deg_N)=MinCost* then we will add *cnt(deg_1,…,deg_N)* to *NumSubtrees*.

The total number of intermediate states (i.e. tuples *(deg_1, …, deg_N, $done_1$, …, $done_N$)*) generated by this algorithm is upper bounded by $O((2\cdot(D+1))^N)$. However, as we will see in Section 8, this number is significantly lower.

7. FURTHER IMPROVEMENTS FOR COUNTING MINIMUM COST D-DEGREE-BOUNDED SUBTREES

We can slightly improve the algorithm from the previous by using a different approach for computing

the entries of the tables *cmin* and *cnt*. We will also use an array of queues: *Q[S]* will contain states $(deg_1,\ldots,deg_N)$ for which $deg_1+\ldots+deg_N=S$. We will assume that all the *cmin* and *cnt* values for all the possible states are initially equal to $+\infty$ and 0, respectively. When a state $(deg_1,\ldots,deg_N)$ is extracted from a queue, we will have $cnt(deg_1,\ldots,deg_N)$ equal to the correct value multiplied by the number of degree values equal to *1* (i.e. the number of leaves in the trees corresponding to this degree sequence). Thus, we will have to update the $cnt(deg_1,\ldots,deg_N)$ value before using it, by dividing it to the number of leaves. Then, from each state $(deg_1,\ldots,deg_N)$ we will be able to generate new states $(deg_1',\ldots,deg_N')$ by adding a new leaf to the trees corresponding to that state. The algorithm consists of the function *CountSubtreesImproved*, presented below. The time complexity of the algorithm is $O(NumStates\cdot N^2)$, where *NumStates* is the number of different valid degree sequences. As discussed in the previous section, this number is upper bounded by $(D+1)^N$, but the experimental results from Section 8 will show that it is in fact much smaller.

CountSubtreesImproved():

```
for each edge (u,v) in the graph {
  deg_i = 0 (for 1≤i≤N, i≠u and i≠v)
  deg_u = deg_v = 1
  cmin(deg_1,…,deg_N) = c(u,v)
  cnt(deg_1,…,deg_N) = 2
  add (deg_1,…,deg_N) to Q[2]
}
for S=2 to 2·N-2 (S even) {
  while (Q[S] is not empty) {
    extract (deg_1,…,deg_N) from the front
of Q[S]
    let nl=the number of values i
(1≤i≤N) such that deg_i=1
    cmin(deg_1,…,deg_N) /= nl
    for each edge (u,v) of the graph
such that 1≤deg_u≤D-1 and deg_v=0 {
      deg_i' = deg_i (for 1≤i≤N, i≠u and
                      i≠v)
      deg_u' = deg_u+1
      deg_v' = 1
      S' = S+2
      cmin' = cmin(deg_1,…,deg_N)+c(u,v)
      cnt' = cnt(deg_1,…,deg_N)
      if (cmin'<cmin(deg_1',…,deg_N')) {
        if      (cmin(deg_1',…,deg_N')=+∞)
then add (deg_1',…,deg_N') to Q[S']
        cmin(deg_1',…,deg_N') = cmin'
        cnt(deg_1',…,deg_N') = cnt'
      } else if (cmin' =
                  cmin(deg_1',…,deg_N')) {
        cnt(deg_1',…,deg_N') += cnt'
    }
  }
}
```

A problem occurs when we want to compute the number of subtrees modulo a given number *P*. In this case the division by the number of leaves (when updating the *cnt* values) may pose some problems. If we want to compute the *cnt* values modulo *P* then the division by the number of leaves *nl* needs to be performed by multiplying the corresponding value by nl^{-1} (the multiplicative inverse of *nl* modulo *P*). However, depending on the number *P*, some numbers *nl* may not have a multiplicative inverse. In such cases we will have to compute the *cnt* values exactly and then compute their remainder when divided by *P* after having computed all the values.

8. EXPERIMENTAL RESULTS

We implemented the dynamic programming solution for graphs with small 2-vertex-connected components considering all the three algorithms from Sections 5, 6 and 7 for computing the minimum cost subtrees (and their numbers) corresponding to each degree sequence (within each biconnected components). All of our tests considered *D=3*.

We will first present (in Table 1) a comparison between the algorithms from Sections 5, 6 and 7 in terms of the number of generated subtrees (for the algorithm from Section 5), the number of degree sequences (for the algorithms from Sections 6 and 7) and the number of intermediate states for 2-vertex-connected components which are complete subgraphs (for the algorithm from Section 6).

Table 1. Variation of the number of subtrees with at least 2 vertices, number of distinct degree sequences and number of intermediate states with the number of vertices of a complete subgraph (K).

K	Number of subtrees with at least 2 vertices	Number of distinct degree sequences	$(D+1)^K$	Number of intermediate states	$2^K\cdot(D+1)^K$
2	1	1	16	2	64
3	6	6	64	13	512
4	34	28	256	68	4096
5	240	120	1024	331	32768
6	2205	495	4096	1577	262144
7	25466	2002	16384	7486	2097152
8	354956	8008	65536	35564	16777216
9	5793264	31824	262144	169128	134217728

In order to compare the algorithms from Sections 5, 6 and 7 in terms of running time we generated graphs with up to *100* vertices containing as many 2-vertex-connected components of a fixed size *K* as possible. Each 2-vertex-connected component was a complete subgraph with the cost of each edge generated randomly as an integer between *-100* and *100*. We ranged *K* from *5* to *9*. The tests were run on an Intel Atom N570 1.66 GHz CPU. The algorithms were

implemented in C++ and the code was compiled with the G++ compiler, version 3.3.1. Table 2 presents the running times of the three algorithms for each of the *5* values of *K*.

Table 2. Running time (in seconds) of the dynamic programming algorithm when using the algorithms from Sections 5, 6 or 7 for computing the number of minimum cost subtrees corresponding to each possible degree sequence for each 2-vertex-connected component.

K	Algorithm from Section 5	Algorithm from Section 6	Algorithm from Section 7
5	0.01	0.01	0.01
6	0.02	0.05	0.03
7	0.18	0.2	0.13
8	2.25	0.99	0.54
9	33.89	4.95	1.92

We can see that using the algorithms from Sections 6 and 7 for computing the values associated to the initial states of each 2-vertex-connected component becomes increasingly more efficient as *K* increases. The only part of the time complexity affected by the usage of the algorithms from Sections 5, 6 or 7 is the *TI(K,D)* factor.

We did not implement the dynamic programming algorithm which is capable of using an arbitrary tree decomposition, because it is obvious from the time complexity analysis that its running time would be significantly worse than the one based on the block-cut vertex tree (if the widths of the two tree decompositions have close values).

9. ALTERNATIVE SOLUTIONS FOR COUNTING MINIMUM COST D-DEGREE-BOUNDED SUBTREES

In this section we discuss two alternative solutions for the problem addressed by the algorithms from Sections 5, 6 and 7. For the first solution we will start with the degree sequence corresponding to an empty subtree (i.e. all degrees are equal to *0*). We will set $cmin(0, \ldots, 0)=0$ and $cnt(0, \ldots, 0)=1$. Then we will add at the back of a queue *Q* all the degree sequences corresponding to subtrees containing a single edge. For each edge *(u,v)* of the graph we will construct the degree sequence $(deg_1, \ldots, deg_N)$, where $deg_u=deg_v=1$ and $deg_i=0$ for $i\neq u$ and $i\neq v$. Then we will insert this sequence at the back of the queue *Q*. We will also maintain a hash table *HT* with the degree sequences already inserted in *Q*. Thus, whenever a degree sequence is inserted in *Q* it will also automatically be inserted in *HT*. As before, until *cmin(DS)* or *cnt(DS)* are explicitly initialized for a degree sequence *DS*, we will assume that $cmin(DS)=+\infty$ and $cnt(DS)=0$.

Then, as long as *Q* is not empty, we will extract from *Q* the degree sequence $(deg_1, \ldots, deg_N)$ located at the front of the queue. We will choose any vertex *u* such that $deg_u=1$. Then we will iterate through all the possible subtree neighbors *v* of *u*. If the degree sequence corresponds to a single edge (i.e. the sum of the degrees from the sequence is *2*) then *v* can be only one vertex: the other vertex besides *u* for which $deg_v=1$. Otherwise, *v* can be any of the vertices for which $deg_v\geq 2$. For each possible subtree neighbor *v* of *u* (with the extra condition that the edge *(u,v)* exists in the graph) we will construct the degree sequence $(deg_1', \ldots, deg_N')$ such that: $deg_i'=deg_i$ for $i\neq u$ and $i\neq v$, and $deg_i'=deg_i-1$ for $i=u$ or $i=v$. We will compute $cmin'=cmin(deg_1', \ldots, deg_N')+c(u,v)$ and $cnt'=cnt(deg_1', \ldots, deg_N')$. If $cmin'<cmin(deg_1, \ldots, deg_N)$ then we will set $cmin(deg_1, \ldots, deg_N)=cmin'$ and $cnt(deg_1, \ldots, deg_N)=cnt'$; otherwise, if $cmin'=cmin(deg_1, \ldots, deg_N)$ then we will add to $cnt(deg_1, \ldots, deg_N)$ the value *cnt'*.

After finalizing the computation of $cmin(deg_1, \ldots, deg_N)$ and $cnt(deg_1, \ldots, deg_N)$ we will generate new degree sequences by adding an extra tree edge to the current degree sequence. We will consider all the pairs of vertices *(u,v)*, such that $deg_u=0$, $1\leq deg_v\leq D-1$ and *(u,v)* is an edge in the graph. For each such pair *(u,v)* we will construct the degree sequence $(deg_1', \ldots, deg_N')$ such that: $deg_i'=deg_i$ for $i\neq u$ and $i\neq v$, and $deg_i'=deg_i+1$ for $i=u$ or $i=v$. If the degree sequence $(deg_1', \ldots, deg_N')$ was not yet inserted in *Q* (we look for it in the hash table *HT*) then we will insert the degree sequence $(deg_1', \ldots, deg_N')$ at the back of the queue *Q* (and also in *HT*, as we explained earlier).

This solution has a time complexity of $O(NumStates\cdot N^2)$, where *NumStates* is the number of different valid degree sequences (upper bounded by $(D+1)^N$). The time complexity could be improved if we had a more efficient method of generating all the valid degree sequences. Let's assume that we have a list of all the valid degree sequences $(deg_1, \ldots, deg_N)$, in order of increasing sum of degrees (i.e. in increasing order of $S=deg_1+\ldots+deg_N$), breaking ties arbitrarily. Then, by traversing the degree sequences in this order, we can compute $cmin(deg_1, \ldots, deg_N)$ and $cnt(deg_1, \ldots, deg_N)$ in $O(N)$ time for each degree sequence $(deg_1, \ldots, deg_N)$ (by using the method we just described).

Another solution consists of computing the following tables: $auxcmin(i,deg_1,\ldots,deg_i)$ and $auxcnt(i,deg_1, \ldots, deg_i)$ for all the valid degree sequences $(deg_1,\ldots,deg_i)$ for *i* vertices ($1\leq i\leq N$). For $i=1$ we have a single valid degree sequence: $deg_1=0$. We have $cmin(1,0)=0$ and $cnt(1,0)=1$. For $2\leq i\leq N$ we will proceed as follows. As before, we assume that any uninitialized entries in the *auxcmin(i)* table are equal to $+\infty$ and any

uninitialized entries in the *auxcnt(i)* table are equal to *0*. We will first consider all the valid degree sequences $(deg_1, ..., deg_i)$ with $deg_i=1$. We will iterate through all the graph neighbors $j<i$ of the vertex *i* such that: $deg_j \geq 2$ or ($deg_j=1$ and the only two non-zero entries of the degree sequence $(deg_1, ..., deg_i)$ are deg_j and deg_i). For each such neighbor *j* we will construct the degree sequence $(deg_1', ..., deg_{i-1}')$, where $deg_k'=deg_k$ for $k \neq j$ and $deg_j'=deg_j-1$. We set $cmin'=auxcmin(i-1, deg_1', ..., deg_{i-1}')+c(i,j)$ and $cnt'=auxcnt(i-1, deg_1', ..., deg_{i-1}')$. If $cmin'<auxcmin(i, deg_1, ..., deg_i)$ then we will set $auxcmin(i, deg_1, ..., deg_i)=cmin'$ and $auxcnt(i, deg_1, ..., deg_i)=cnt'$; otherwise, if $cmin'=auxcmin(i, deg_1, ..., deg_i)$ then we will add to $auxcnt(i, deg_1, ..., deg_i)$ the value *cnt'*.

Next we will consider all the valid degree sequences $(deg_1, ..., deg_i)$ with $2 \leq deg_i \leq D$, in increasing order of deg_i (breaking ties arbitrarily). For each such degree sequence we will consider all the valid degree sequences $(deg_1', ..., deg_i')$ such that $deg_j'=0$ or $deg_j'=deg_j$ (for $1 \leq j \leq i-1$), $deg_i'=1$ and the degree sequence $(deg_1'', ..., deg_i'')$ is also a valid degree sequence, where $deg_j''=deg_j-deg_j'$ (for $1 \leq j \leq i$). We will compute $cmin'=auxcmin(i, deg_1', ..., deg_i')+auxcmin(i, deg_1'', ..., deg_i'')$ and $cnt'=auxcnt(i, deg_1', ..., deg_i')\cdot auxcnt(i, deg_1'', ..., deg_i'')$. If $cmin'<auxcmin(i, deg_1, ..., deg_i)$ then we will set $auxcmin(i, deg_1, ..., deg_i)=cmin'$ and $auxcnt(i, deg_1, ..., deg_i)=cnt'$; otherwise, if $cmin'=auxcmin(i, deg_1, ..., deg_i)$ then we will add to $auxcnt(i, deg_1, ..., deg_i)$ the value *cnt'*. In order to avoid double-counting in this case, let's consider that *j* is the smallest index such that $deg_j \geq 1$. We will set $deg_j'=deg_j$ for this index *j* (i.e. we will not consider the case $deg_j'=0$, too); thus, we will have $deg_j''=0$.

We will also have $auxcmin(i, deg_j=0\ (1 \leq j \leq i))=0$ and $auxcnt(i, deg_j=0\ (1 \leq j \leq i))=1$.

In the end we have $cmin(deg_1, ..., deg_N)=auxcmin(N, deg_1, ..., deg_N)$ and $cnt(deg_1, ..., deg_N)=auxcnt(N, deg_1, ..., deg_N)$ for every valid degree sequence $(deg_1, ..., deg_N)$. A simple analysis shows us that the time complexity of this approach is upper bounded by $O((D+1)^N \cdot N+(D+1)^N \cdot 2^{N-2})$. Note that in this case we did not specify how to generate all the valid degree sequences for *i* vertices ($1 \leq i \leq N$). We could use the method from the previous solution or one of the methods presented in Sections 6 or 7. We also did not include the time complexity of the generation of valid degree sequences into the stated upper bound.

10. RELATED WORK

The problem of finding a minimum cost degree bounded subtree in an undirected graph has been studied from various perspectives in the scientific literature. Many papers considered the problem of computing an optimal *spanning* tree (or subgraph) under various degree constraints. Approximation algorithms for finding spanning trees which violate a maximum degree bound by a small constant while at the same time having a cost at most equal to that of the optimal degree bounded spanning tree were proposed in (Goemans, 2006) and (Singh and Lau, 2007). A more general approach regarding the constraints imposed on the spanning tree edges adjacent to each vertex was considered in (Zenklusen, 2012). Approximation algorithms for finding maximum bounded degree spanning subgraphs were proposed in (Feng et al., 2009). A branch-and-cut algorithm for finding a degree-constrained minimum spanning tree was presented in (Behle et al., 2007).

The problem of finding a minimum cost degree bounded subtree is similar to several other well-studied problems. For instance, when *D=2*, the problem is equivalent to finding the path of minimum total length (note that this is equivalent to the *longest path* problem if we negate all the cost values).

Algorithms based on dynamic programming on tree decompositions of graphs for finding optimal connected or degree-constrained subgraphs or vertex subsets were also presented in the literature. An algorithm for the *Connected Vertex Cover* problem was presented in (Moser, 2005). An algorithm for the *Connected Feedback Vertex Set* problem was described in (Misra et al., 2010). An algorithm for finding a minimum subgraph with minimum degree at least *D* (but not necessarily connected) was proposed in (Amini et al., 2009). A solution for the *Steiner Tree* problem was presented in (Chimani et al., 2012). A general method for developing dynamic programming algorithms on tree decompositions was presented in (Arnborg and Proskurowski, 1989) . However, the algorithms obtained by employing the proposed method are not the most efficient possible in terms of time complexity.

Instead of a tree decomposition some authors used a branch decomposition of the graph. For instance, in (Sau and Thilikos, 2010), the authors present a branch decomposition-based dynamic programming algorithm for finding a connected induced D-degree-bounded subgraph having a maximum number of edges (or vertices).

Counting certain types of subgraphs of a given base graph is a problem which has been considered many types in the scientific literature (e.g. counting spanning trees in dense graphs (Person, 2007)). However, counting subgraphs obeying some cost optimality criterion has received less attention.

11. CONCLUSIONS

In this paper we presented novel efficient algorithms for finding the minimum cost of a degree-bounded

subtree of a graph and the number of such subtrees, when the graph has 2-vertex-connected components with small sizes. The algorithm uses a special tree decomposition of the graph, called the block-cut vertex tree. The proposed solution was also evaluated experimentally.

We also presented a general solution which is capable of using any tree decomposition of the graph, but it is less efficient than the one based on the block-cut vertex tree.

Our solutions consider that the maximum degree of each vertex in the subtree can be at most *D*. The solutions can be modified in a straight-forward manner in order to have different upper bounds for different vertices. For instance, these bounds can be easily integrated in the algorithms from Sections 5, 6 and 7. In the dynamic programming algorithms, when the degrees of some vertices increase (either because of joining states from adjacent tree nodes or because new vertices are *introduced*), we need to replace the verification that the new degrees do not exceed *D* by the verification that the new degrees do not exceed the upper bounds of the corresponding vertices.

12. REFERENCES

Amini, O., D. Peleg, S. Perennes, I. Sau and S. Saurabh (2009). Degree-Constrained Subgraph Problems: Hardness and Approximation Results, *Lecture Notes in Computer Science*, **vol. 5426**, pp. 29-42.

Andreica, M. I. (2006). The Tree of Biconnected Components and Critical Nodes, *GInfo*, **vol. 16 (5)**, pp. 11-17.

Arnborg, S. and A. Proskurowski (1989). Linear Time Algorithms for NP-Hard Problems Restricted to Partial k-Trees, *Discrete Applied Mathematics*, **vol. 23**, pp. 11-24.

Behle, M., M. Junger and F. Liers (2007). A Primal Branch-and-Cut Algorithm for the Degree-Constrained Minimum Spanning Tree Problem, *Lecture Notes in Computer Science*, **vol. 4525**, pp 379-392.

Chimani, M., P. Mutzel and B. Zey (2012). Improved Steiner Tree Algorithms for Bounded Treewidth, *Journal of Discrete Algorithms*, **vol. 16**, pp. 67-78.

Dekker, A., H. Perez-Roses, G. Pineda-Villavicencio and P. Watters (2012). The Maximum Degree & Diameter-Bounded Subgraph and its Applications, *Journal of Mathematical Modelling and Algorithms*, **vol. 11 (3)**, pp. 249-268.

Feng, W., H. Ma, B. Zhang and H. Wang (2009). Approximating Bounded Degree Maximum Spanning Subgraphs. In: *Proceedings of the 8th International Symposium on Operations Research and Its Applications*, pp. 83-89.

Goemans, M. X. (2006). Minimum Bounded Degree Spanning Trees. In: *Proceedings of the 47th International Symposium on Foundations of Computer Science*, pp. 273-282.

Kintali, S. and S. Munteanu (2012). Computing Bounded Path Decompositions in Logspace, *Electronic Colloquim on Computational Complexity*, Report No. 126.

Misra, N., G. Philip, V. Raman, S. Saurabh and S. Sikdar (2010). FPT Algorithms for Connected Feedback Vertex Set. In: *Proceedings of the 4th international conference on Algorithms and Computation (WALCOM)*, pp. 269-280.

Moser, H. (2005). Exact Algorithms for Generalizations of Vertex Cover, Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena.

Y. Person (2007). Counting Spanning Trees in Dense Graphs. Diploma Thesis, Technical University Munchen.

S. Pirzada (2012). *An Introduction to Graph Theory*, Universities Press.

Sau, I. and D. M. Thilikos (2010). Subexponential Parameterized Algorithms for Degree-Constrained Subgraph Problems on Planar Graphs, *Journal of Discrete Algorithms*, **vol. 8 (3)**, pp. 330-338.

Singh, M. and L. C. Lau (2007). Approximating Minimum Bounded Degree Spanning Trees to within One of Optimal. In: *Proceedings of the 39th ACM Symposium on Theory of Computing*, pp. 661-670.

Zenklusen, R. (2012). Matroidal Degree-Bounded Minimum Spanning Trees. In: *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms*, pp. 1512-1521.