Computing the Tutte Polynomial on Graphs of Bounded Clique-Width ⋆

Omer Giménez[1⋆⋆], Petr Hliněný[2⋆⋆⋆], and Marc Noy[1†]

[1] Department of Applied Mathematics
Technical University of Catalonia
Jordi Girona 1–3, 08034 Barcelona, Spain
e-mail: [omer.gimenez, marc.noy]@upc.edu

[2] Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
e-mail: hlineny@member.ams.org

March 28, 2006

Abstract. The Tutte polynomial is a notoriously hard graph invariant, and efficient algorithms for it are known only for a few special graph classes, like for those of bounded tree-width. The notion of clique-width extends the definition of cographs (graphs without induced P_4), and it is a more general notion than that of tree-width. We show a subexponential algorithm (running in time $\exp O(n^{1-\varepsilon})$) for computing the Tutte polynomial on graphs of bounded clique-width. In fact, our algorithm computes the more general U-polynomial.

Keywords: Tutte polynomial, cographs, clique-width, subexponential algorithm, U polynomial.
2000 Math Subjects Classification: **05C85, 68R10**

1 Introduction

The Tutte polynomial $T(G; x, y)$ of a graph G is a powerful invariant with many applications, not only in graph theory but also in other fields such as knot theory and statistical physics. One important feature of the Tutte polynomial is that by evaluating $T(G; x, y)$ at special points in the plane one obtains several parameters of G. For example, $T(G; 1, 1)$ is the number of spanning trees of G and $T(G; 2, 1)$ is the number of forests (that is, spanning acyclic subgraphs) of G.

⋆ An extended abstract published in WG 2005, Proceedings, Lecture Notes in Computer Science 3787, Springer Verlag (2005).
⋆⋆ Supported by Beca Fundació Crèdit Andorrà and Project MTM2005-08618-C02-01.
⋆⋆⋆ Supported partly by Czech research grant GAČR 201/05/0050, and by the Institute of Theoretical Computer Science, project 1M0545.
† Supported by Project MTM2005-08618-C02-01.

A question that has received much attention is whether the evaluation of $T(G;x,y)$ at a particular point of the (x,y) plane can be done in polynomial time. Jaeger, Vertigan and Welsh [9] showed that evaluating the Tutte polynomial of a graph is #P-hard at every point except those lying on the hyperbola $(x-1)(y-1)=1$ and eight special points, including at $(1,1)$ which gives the number of spanning trees. In each of the exceptional cases the evaluation can be done in polynomial time. On the other hand, the Tutte polynomial can be computed in polynomial time for graphs of bounded tree-width. This was obtained independently by Andrzejak [2] and Noble [13]. Recently Hliněný [8] has obtained the same result for matroids of bounded branch-width representable over a fixed finite field, which is a substantial generalization of the previous results. See [6] for additional references on this subject.

In this paper we study the problem of computing the Tutte polynomial for cographs and, more generally, for graphs of bounded clique-width. A graph has clique-width $\leq k$ if it can be constructed using k labels and the following four operations: 1) create a new vertex with label i; 2) take the disjoint union of several labeled graphs; 3) add all edges between vertices of label i and label j; and 4) relabel all vertices with label i to have label j. An expression defining a graph G built from the above four operations using k labels is a *k-expression* for G. When we say that a graph G has clique-width $\leq k$, we always assume that a k-expression for G is given.

A *cograph* is a graph of clique-width at most two; equivalently, it is a graph containing no induced path P_4 on four vertices (see Section 4).

Although a class of graphs with bounded tree-width has also bounded clique-width, the converse is not true. For instance, complete graphs have clique-width two. It is well-known that all problems expressible in monadic second order logic of incidence graphs become polynomial time solvable when restricted to graphs of bounded tree-width. For bounded clique-width less is true: all problems become polynomial time solvable if they are expressible in monadic second-order logic using quantifiers on vertices but not on edges (adjacency graphs) [3].

Our main results are as follows:

Theorem 1.1. *The Tutte polynomial of a cograph with n vertices can be computed in time* $\exp\left(O(n^{2/3})\right)$.

Theorem 1.2. *Let G be a graph with n vertices of clique-width k along with a k-expression for G as an input. Then the Tutte polynomial of G can be computed in time* $\exp\left(O(n^{1-1/(k+2)})\right)$.

Theorem 1.2 is not likely to hold for the class of all graphs, since it would imply the existence of a subexponential algorithm for 3-coloring, hence also for 3-SAT; that is considered highly unlike in the Computer Science community. Of course, the main open question is whether there exists a *polynomial time* algorithm for computing the Tutte polynomial of graphs of bounded clique-width. We discuss this issue in the last section.

In fact, our algorithms compute not only the Tutte polynomial, but the so-called U polynomial (see [14]), which is a stronger polynomial invariant. More-

over, we may skip the requirement of having a k-expression for G as an input in Theorem 1.2, if we do not care about an asymptotic behaviour in the exponent: Just to prove a subexponential upper bound we may use the approximation algorithm for clique-width by Oum and Seymour [16, 15] (see Section 4).

Since our algorithms are quite complicated, for an illustration, we first present in Section 2 a simplified algorithm computing the number of forests in a cograph, that is, evaluating $T(G; 2, 1)$ for graphs of clique-width ≤ 2. In Section 3 we extend the algorithm to the computation of the full Tutte polynomial on cographs. Section 4 then discusses more closely the notion of graph clique-width. Finally, in Section 5 we prove our main result, Theorem 1.2.

2 Forests in Cographs

The problem of computing the number of spanning forests in an arbitrary graph is $\#P$-hard [9]. In this section we show the existence of a subexponential algorithm for the class of cographs.

2.1 Definition and Signatures

The class of *cographs* is defined recursively as follows:

1. A single vertex is a cograph.
2. A disjoint union of two cographs is a cograph.
3. A complete union of two cographs is a cograph.

Here a *complete union* of two graphs $G \boxtimes H$ means the operation of taking a disjoint union $G \dot\cup H$, and adding all edges between $V(G)$ and $V(H)$. (We better avoid using the notation $\oplus$ at all since in the context of clique-width it is used to denote a disjoint union while in some other areas it denotes a complete union.) A cograph G can be represented by a tree, whose internal nodes correspond to operations (2) and (3) above, and whose leaves correspond to single vertices. We call such a tree an *expression* for G.

For example, all cliques are cographs, and the complement of a cograph is a cograph again. Cographs have a long history of theoretical and algorithmic research. In particular, they are known to be exactly the graphs without induced paths on four vertices (P_4-free).

Let us call a *signature* a multiset of positive integers. The *size* $\|\boldsymbol{\alpha}\|$ of a signature $\boldsymbol{\alpha}$ is the sum of all elements in $\boldsymbol{\alpha}$, respecting repetition in the multiset. A signature $\boldsymbol{\alpha}$ of size n is represented by the *characteristic vector* $\boldsymbol{\alpha} = (a_1, a_2, \ldots, a_n)$, where there are $a_i \geq 0$ elements i in $\boldsymbol{\alpha}$, and $\sum_{i=1}^{n} i \cdot a_i = n$. (On the other hand, the *cardinality* of $\boldsymbol{\alpha}$ is $|\boldsymbol{\alpha}| = \sum_{i=1}^{n} a_i$, as usual.) An important fact we need is the following. Recall that $\Theta(f)$ is a usual shortcut for all functions having the same asymptotic growth rate as f.

Lemma 2.1. *There are $2^{\Theta(\sqrt{n})}$ distinct signatures of size n.*

Proof. Each signature actually corresponds to a partition of n into an unordered sum of positive integers. It is well-known [11, Chapter 15] that there are $2^{\Theta(\sqrt{n})}$ of those. ∎

We call a *double-signature* a multiset of ordered pairs of non-negative integers, excluding the pair $(0,0)$. The *size* $\|\boldsymbol{\beta}\|$ of a double-signature $\boldsymbol{\beta}$ is the sum of all $(x+y)$ for $(x,y) \in \boldsymbol{\beta}$, respecting repetition in the multiset. We, moreover, need to prove:

Lemma 2.2. *There are* $\exp\left(\Theta(n^{2/3})\right)$ *distinct double-signatures of size* n.

Lemma 2.2 is a particular case of Lemma 5.1, which is proved in Section 5.

Lemma 2.3. *A double-signature* $\boldsymbol{\beta}$ *of size* n *has at most* $\exp\left(O(n^{2/3})\right)$ *different submultisets (i.e. of different characteristic vectors).*

Proof. Just count all double-signatures of size $\leq n$. ∎

2.2 Forest Signature Table

Let us now consider a graph G and a forest $U \subset G$. The signature $\boldsymbol{\alpha}$ of U is the multiset of sizes of the connected components of U. (Obviously, $\boldsymbol{\alpha}$ has size $|V(G)|$ if U spans all the vertices.) We call a *(spanning) forest signature table* of the graph G a vector $\boldsymbol{T}$ (realized as an array $\boldsymbol{T}[\ldots]$); such that $\boldsymbol{T}$ records, for each signature $\boldsymbol{\alpha}$ of size $|V(G)|$, the number of spanning forests $U \subset G$ having signature $\boldsymbol{\alpha}$ (as $\boldsymbol{T}[\boldsymbol{\alpha}]$). For simplicity we usually skip the word "spanning" if it is clear from the context. We are going to compute the forest signature table of a cograph G recursively along the way G has been constructed. For that we describe two algorithms.

Let us denote by Σ_G the set of all signatures of size $|V(G)|$. It is important to keep in mind that signatures are considered as multisets, which concerns also set operations. For instance, a *multiset union* $\boldsymbol{\gamma} \uplus \boldsymbol{\delta}$ is obtained as the sum of the characteristic vectors of $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$, and a *multiset difference* $\boldsymbol{\gamma} \setminus \boldsymbol{\delta}$ is defined by the non-negative difference of those.

Algorithm 2.4. *Combining the spanning forest signature tables of graphs* F *and* G *into the one of the disjoint union* $H = F \dot\cup G$.

`Input:` Graphs F, G, and their forest signature tables $\boldsymbol{T}_F, \boldsymbol{T}_G$.
`Output:` The forest signature table $\boldsymbol{T}_H$ of $H = F \dot\cup G$.

`create` *empty table* $\boldsymbol{T}_H$ *of forest signatures of size* $|V(H)|$`;`
`for` *all signatures* $\boldsymbol{\alpha}_F \in \Sigma_F$, $\boldsymbol{\alpha}_G \in \Sigma_G$ `do`
  `set` $\boldsymbol{\alpha} = \boldsymbol{\alpha}_F \uplus \boldsymbol{\alpha}_G$ *(a multiset union)*`;`
  `add` $\boldsymbol{T}_H[\boldsymbol{\alpha}] \mathrel{+}= \boldsymbol{T}_F[\boldsymbol{\alpha}_F] \cdot \boldsymbol{T}_G[\boldsymbol{\alpha}_G]$`;`
`done.`

The running time of this algorithm is proportional to the number of pairs of signatures (α_F, α_G), which is $\exp\big(O(n^{2/3})\big)$, where $n = |V(H)|$; this is due to Lemma 2.2 and the fact that we have the $O(\ )$ expression in the exponent.

The second algorithm is, on the other hand, more complicated. It involves double-signatures in the following meaning: Consider a graph H with vertices partitioned into two parts $V(H) = V_1 \cup V_2$, and a forest $U \subset H$. The double-signature of U (wrt. V_1, V_2) is the multiset of pairs $\big(|V(C) \cap V_1|, |V(C) \cap V_2|\big)$ over all connected components C of U. See an illustration in Fig. 1.

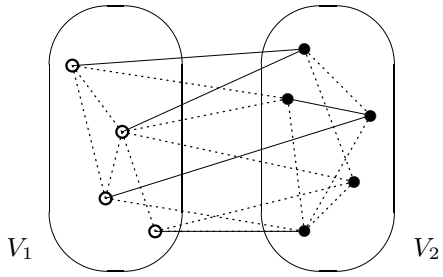

Fig. 1. An illustration of a spanning forest (solid edges) in a graph partitioned into V_1 (white) and V_2 (black); this forest has double signature $\{(2,1),(1,2),(0,1),(1,1)\}$.

The idea behind the algorithm is to obtain the double-signatures (for $V_1 = V(F)$ and $V_2 = V(G)$) of the spanning forests in $H = F \boxtimes G$ from the signatures of the spanning forests in F and G. For every pair of forests $U_F \subset F$ and $U_G \subset G$, the algorithm iteratively counts the different ways in which each component of U_G can be joined to components of U_F. During the process, double signatures are needed to distinguish between former vertices of F and of G in already joined components. In fact, the algorithm works with pairs of signatures $\boldsymbol{\alpha}_F$ and $\boldsymbol{\alpha}_G$, that is, with whole classes of forests instead of particular forests. We also remark that a submultiset is considered among all possible selections of repeated elements, as if they were pairwise distinct.

Algorithm 2.5. *Combining the spanning forest signature tables of graphs F and G into the one of the complete union $H = F \boxtimes G$.*

`Input:` Graphs F, G, and their forest signature tables $\boldsymbol{T}_F, \boldsymbol{T}_G$.
`Output:` The forest signature table $\boldsymbol{T}_H$ of $H = F \boxtimes G$.

`create` *empty table* $\boldsymbol{T}_H$ *of forest signatures of size* $|V(H)|$`;`
`for` *all signatures* $\boldsymbol{\alpha}_F \in \Sigma_F$, $\boldsymbol{\alpha}_G \in \Sigma_G$ `do`
  `set` $z = |V(F)|$`;`
  `create` *empty table* $\boldsymbol{X}$ *of forest double-signatures of size* z`;`
    `//` *Imagine particular forests* $U_F \subset F$, $U_G \subset G$ *of signature* $\boldsymbol{\alpha}_F, \boldsymbol{\alpha}_G$,
    `//` *and a selected component* $C \subset U_G$ *of size* c.

```
   set X[double-signature {(a,0) : a ∈ α_F}] = 1;
   for each c ∈ α_G (with repetition) do
      create empty table X' of forest double-signatures of size z + c;
      for all double signatures β of size z s.t. X[β] > 0 do
(†)      for all submultisets γ ⊆ β (with repetition) do
            set d_1 = Σ_{(x,y)∈γ} x, d_2 = Σ_{(x,y)∈γ} y;
            set double-signature β' = (β \ γ) ⊎ {(d_1, d_2 + c)};
(*)         add X'[β'] += X[β] · Π_{(x,y)∈γ} cx;
         done
      done
      set X = X', z = z + c;  dispose X';
   done
   for all double-signatures β of size |V(H)| do
      set signature α_0 = {x + y : (x, y) ∈ β};
      add T_H[α_0] += X[β] · T_F[α_F] · T_G[α_G];
   done
done.
```

Proof of Algorithm 2.5. We now explain the algorithm, and show its correctness. It is more easily understood if one imagines particular forests (representatives) $U_F \subset F$ and $U_G \subset G$ in the place of the signatures $\boldsymbol{\alpha}_F$ and $\boldsymbol{\alpha}_G$ chosen in the first `for` cycle. Then one may routinely verify that all subsequent computations depend only on the forest signatures $\boldsymbol{\alpha}_F$, $\boldsymbol{\alpha}_G$ (not on the particular forests), and hence it is correct to finally multiply the computed values in $\boldsymbol{X}$ by the numbers $\boldsymbol{T}_F[\boldsymbol{\alpha}_F] \cdot \boldsymbol{T}_G[\boldsymbol{\alpha}_G]$.

In the tables $\boldsymbol{X}, \boldsymbol{X}'$ we iteratively compute the numbers of all spanning forests in H that result by adding some edges between the forests U_F and U_G. For a particular iteration, imagine a new forest component of size c in G, that is to be joined with another forest component in $F \boxtimes G$ which has x vertices in $V(F)$. There are $c \cdot x$ edges to consider between the components, and we have to select exactly one connecting edge to maintain acyclicity, so there are cx choices there. These numbers are naturally multiplied (*) when joining more components together. See the steps in Fig. 2

So the core of the algorithm in the second cycle '`for` each $c \in \boldsymbol{\alpha}_G \ldots$' reads: We consider an arbitrary order $C_1, C_2, \ldots, C_k$ on the connected components of U_G. For $i = 1, 2, \ldots, k$, we take the component C_i, and count all possible ways how to connect C_i by selected edges to a subset (†) of components of each of the previously constructed forests on $V(F \cup C_1 \cup \ldots \cup C_{i-1})$ which are recorded in the table $\boldsymbol{X}$. The other ends of those selected edges are considered only among vertices in $V(F)$. (Recall that the complete union $H = F \boxtimes G$ has added *all* edges between $V(F)$ and $V(C_i)$.) We then record (*) numbers of all the new forests on $V(F \cup C_1 \cup \ldots \cup C_i)$ in a new table $\boldsymbol{X}'$ that will play the role of $\boldsymbol{X}$ in the next iteration.

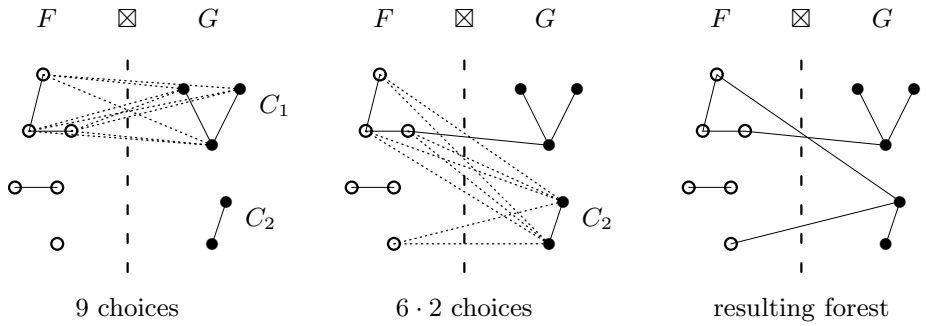

Fig. 2. An illustration of inner iterations of Algorithm 2.5: Particular spanning forests U_F, U_G are chosen in F and G (other edges are not shown here), and the components C_1, C_2 of U_G are then joined to some of the four components of U_F. Possible choices of edges for these joins are shown in dotted lines.

More precisely, after finishing iteration $i = 1, 2, \ldots, k$ described in the previous paragraph, each entry $\boldsymbol{X}'[\boldsymbol{\beta}]$ equals the number of all forests U' of signature $\boldsymbol{\beta}$ spanning $V(F \cup C_1 \cup \ldots \cup C_i)$ such that $U' \restriction V(F) = U_F$ and $U' \restriction V(G) = U_G \restriction C_1 \cup \ldots \cup C_i$. That follows easily by an induction from the previous arguments. At the end we count each spanning forest $U \subseteq H$ such that $U \restriction V(F) = U_F$ and $U \restriction V(G) = U_G$ exactly once. Finally, the double-signatures in the table $\boldsymbol{X}$ partition the vertices into $V(F)$ and $V(G)$, but that is no longer needed. So we "simplify" them – we record the resulting numbers only by the (single) forest signatures in the resulting table $\boldsymbol{T}_H$. ∎

2.3 Time Analysis

To get a fine time-complexity analysis of Algorithm 2.5, we have to insert a slight modification. (A problem may occur in the original Algorithm 2.5 in the fourth nested cycle '`for` all submultisets $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}$' if $\boldsymbol{\beta}$ consists, say, of $n/2$ copies of the element 2. Then there are up to $\exp\big(\Theta(n)\big)$ submultisets γ to consider.)

Algorithm 2.6. *Same as Algorithm 2.5, except the program line (†) now reads*

$$\texttt{for } \textit{all different submultisets } \boldsymbol{\gamma} \subseteq \boldsymbol{\beta} \texttt{ do}\,,$$

and the line (*) reads

$$\texttt{add } \boldsymbol{X}'[\boldsymbol{\beta}'] \mathrel{+}= \boldsymbol{X}[\boldsymbol{\beta}] \cdot \prod_{(x,y)\in\boldsymbol{\gamma}} cx \cdot \prod_{(x,y)\in\langle\boldsymbol{\beta}\rangle} \binom{\mu_{\boldsymbol{\beta}}(x,y)}{\mu_{\boldsymbol{\gamma}}(x,y)}\,,$$

where $\langle\boldsymbol{\alpha}\rangle$ denotes the ordinary set formed by elements of a multiset $\boldsymbol{\alpha}$, and $\mu_{\boldsymbol{\alpha}} z$ is the repetition of an element z in $\boldsymbol{\alpha}$.

Proof of Algorithm 2.6. We prove that this algorithm computes the same results as Algorithm 2.5. Notice that the outcome of the computation between the lines

(†) and (*) depends only on the characteristic vector of $\boldsymbol{\gamma}$. Hence instead of all $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}$, it is enough to consider (much less of) pairwise different submultisets $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}$, and then multiply the resulting number by all possible choices (combinations) of repeated elements of $\boldsymbol{\gamma}$ from $\boldsymbol{\beta}$, as we do here in Algorithm 2.6. ∎

Now, since we use $O(\,)$ in the exponent, it is enough to argue that each of the `for` cycles in Algorithm 2.6 (2.5) is iterated at most $\exp\big(O(n^{2/3})\big)$ times. That follows easily from Lemmas 2.1, 2.2, and 2.3. We conclude:

Lemma 2.7. *Algorithm 2.6 runs in time* $\exp\big(O(n^{2/3})\big)$ *where* $n = |V(H)|$. ∎

We remark that the improvement presented in Algorithm 2.6 has been fully incorporated in the subsequent algorithms, without further notices.

Theorem 2.8. *The number of spanning forests in an* n*-vertex cograph can be computed in time* $\exp\big(O(n^{2/3})\big)$.

Proof. Consider a cograph G and a tree expression defining it. The forest signature table of a single vertex is trivial, and by Algorithms 2.4 and 2.6, the forest signature tables of a union or a complete union of two cographs can be computed in time claimed. Finally, knowing the forest signature table $\boldsymbol{T}$ of G, the number of all spanning forests of G is computed by adding up the entries of $\boldsymbol{T}$. ∎

Here we should note that the expression defining a cograph can be found in linear time [5], and hence we do not require it on the input.

3 The Tutte Polynomial of a Cograph

The Tutte polynomial can be defined in a number of equivalent ways. For our purposes, given a graph $G = (V, E)$ we define the Tutte polynomial as

$$T(G; x, y) = \sum_{F \subseteq E} (x-1)^{r(E)-r(F)} (y-1)^{|F|-r(F)},$$

where $r(F) = |V| - k(F)$ and $k(F)$ is the number of connected components of the spanning subgraph induced by the edge-subset F. It is clear that knowing $T(G; x, y)$ is the same as knowing, for every i and j, how many spanning subgraphs with the edge set F in G are there with $|F| = i$ and $k(F) = j$.

Consider a spanning subgraph $W \subset G$ determined on $V(W) = V(G)$ by an arbitrary subset $F \subset E(G)$, $F = E(W)$. The sizes of the connected components of W define a signature of size $|V(G)|$. In the *(spanning) subgraph signature table* $\boldsymbol{S}$ of G, for each signature $\boldsymbol{\alpha}$ of size $|V(G)|$ and each number of edges $f \in \{0, 1, 2, \ldots, |E(G)|\}$, we record the number $\boldsymbol{S}[\boldsymbol{\alpha}, f]$ of all spanning subgraphs of G having f edges and having component sizes according to the signature $\boldsymbol{\alpha}$. We abbreviate by $\boldsymbol{\gamma}\restriction_i$ the multiset formed by all the i-th coordinates (repetitions accounted for) of the elements of a double-signature $\boldsymbol{\gamma}$.

In order to prove Theorem 1.1 we need analogues of Algorithms 2.4 and 2.5 for computing subgraph signature tables. The algorithm for disjoint unions is again straightforward and we omit it; the one for complete unions comes next.

Besides adding edge number as the second index to the signature tables, the only other major difference of this algorithm from Algorithm 2.5 is that the single line (*) is replaced with another `for` cycle calling a procedure `CellSel` of further Algorithm 3.2.

Algorithm 3.1. *A modification of Algorithm 2.6 (2.5) for computing the (spanning) subgraph signature table of the complete union $H = F \boxtimes G$.*

`Input:` *Graphs F, G, and their subgraph signature tables $\boldsymbol{S}_F, \boldsymbol{S}_G$.*
`Output:` *The subgraph signature table $\boldsymbol{S}_H$ of $H = F \boxtimes G$.*

`create` *empty table $\boldsymbol{S}_H$ of subgraph signatures of size $|V(H)|$*;
`for` *all $\boldsymbol{\alpha}_F \in \Sigma_F$, and $e_F = 0, 1, \ldots, |E(F)|$ s.t. $\boldsymbol{S}_F[\boldsymbol{\alpha}_F, e_F] > 0$* `do`
  `for` *all $\boldsymbol{\alpha}_G \in \Sigma_G$, and $e_G = 0, \ldots, |E(G)|$ s.t. $\boldsymbol{S}_G[\boldsymbol{\alpha}_G, e_G] > 0$* `do`
    `set` $z = |V(F)|$;
    `create` *empty table $\boldsymbol{Y}$ of subgraph double-signatures of size z*;
    `set` $\boldsymbol{Y}\big[$*double-signature* $\{(a,0) : a \in \boldsymbol{\alpha}_F\}$, $e_F\big] = 1$;
    `for` *each $c \in \boldsymbol{\alpha}_G$ (with repetition)* `do`
      `create` *empty table $\boldsymbol{Y}'$ of subgraph double-sign. of size $z + c$*;
      `for` *all $\boldsymbol{\beta}$ of size z, and e s.t. $\boldsymbol{Y}[\boldsymbol{\beta}, e] > 0$* `do`
        `for` *all different submultisets $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}$* `do`
          `set` $r = \displaystyle\prod_{(x,y)\in\langle\boldsymbol{\beta}\rangle} \binom{\mu_{\boldsymbol{\beta}}(x,y)}{\mu_{\boldsymbol{\gamma}}(x,y)}$;
          `set` $d_1 = \|\boldsymbol{\gamma}\restriction_1\| = \sum_{(x,y)\in\boldsymbol{\gamma}} x$, $d_2 = \|\boldsymbol{\gamma}\restriction_2\| = \sum_{(x,y)\in\boldsymbol{\gamma}} y$;
          `set` *double-signature* $\boldsymbol{\beta}' = (\boldsymbol{\beta} \setminus \boldsymbol{\gamma}) \uplus \{(d_1, d_2 + c)\}$;
          `for` $f = |\boldsymbol{\gamma}|, |\boldsymbol{\gamma}| + 1, \ldots, c \cdot d_1$ `do`
            `set` *multiset* $D = c \cdot (\boldsymbol{\gamma}\restriction_1) = \{cx : (x,y) \in \boldsymbol{\gamma}\}$;
            `call` *Algorithm 3.2:* $p =$ `CellSel`(D, f);
            `add` $\boldsymbol{Y}'[\boldsymbol{\beta}', e + f] \mathrel{+}= \boldsymbol{Y}[\boldsymbol{\beta}, e] \cdot r \cdot p$;
          `done`
        `done`
      `done`
      `set` $\boldsymbol{Y} = \boldsymbol{Y}'$, $z = z + c$; `dispose` $\boldsymbol{Y}'$;
    `done`
    `for` *all double-sign. $\boldsymbol{\beta}$ of size $|V(H)|$, and f, s.t. $\boldsymbol{Y}[\boldsymbol{\beta}, f] > 0$* `do`
      `set` *signature* $\boldsymbol{\alpha}_0 = \{x + y : (x,y) \in \boldsymbol{\beta}\}$;
      `add` $\boldsymbol{S}_H[\boldsymbol{\alpha}_0, f + e_G] \mathrel{+}= \boldsymbol{Y}[\boldsymbol{\beta}, f] \cdot \boldsymbol{S}_F[\boldsymbol{\alpha}_F, e_F] \cdot \boldsymbol{S}_G[\boldsymbol{\alpha}_G, e_G]$;
    `done`
  `done`
`done.`

Proof of Algorithm 3.1. This algorithm is similar to the improved version of Algorithm 2.6, and so we only sketch the proof here. The main new difficulty lies in counting the different ways in which a connected component of c vertices in $\boldsymbol{\alpha}_G$ can be connected with f edges to the selected components of signatures $(x, y) \in \boldsymbol{\gamma}$. Recall that when counting forests we had no such difficulty, since we joined the component of $\boldsymbol{\alpha}_G$ to each component of $\boldsymbol{\gamma}$ with exactly one edge; thus we used exactly $f = |\boldsymbol{\gamma}|$ edges chosen in $\prod_{(x,y)\in\boldsymbol{\gamma}} cx$ different ways. The procedure '`CellSel`$(D,\ f)$' counts this for spanning subgraphs, and we defer the explanation to Algorithm 3.2. See also Fig. 3.

Finally, notice that the edge numbers in tables $\boldsymbol{Y}$, $\boldsymbol{Y}'$ do not account for the edges from $E(G)$, since we do not know how many edges has each one of the components of $\boldsymbol{\alpha}_G$. Those edges are summed up at the end, when obtaining the signatures for H from the double-signatures stored in $\boldsymbol{Y}$. ∎

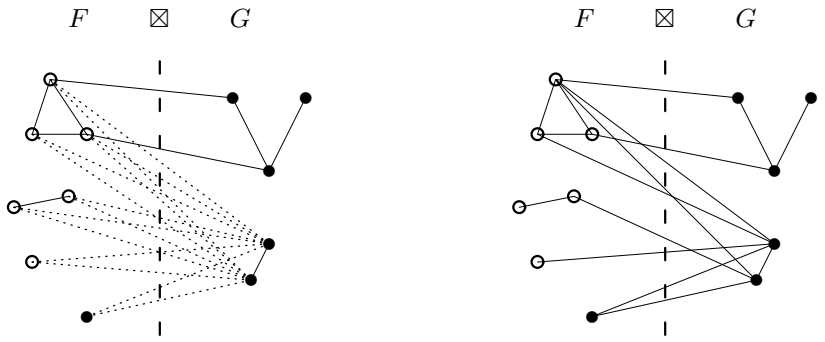

Fig. 3. How cellular selections arise in Algorithm 3.1 when adding edges to a spanning subgraph; here we are selecting $f = 7$ edges out of cell sizes $\{6, 4, 2, 2\}$ (possible edge choices shown in dotted lines).

Algorithm 3.2. *Computing the number of cellular selections: We are selecting ℓ elements from the union $C_1 \cup C_2 \cup \ldots \cup C_k$, where C_i for $i = 1, 2, \ldots, k$ are pairwise disjoint cells of sizes $d_i = |C_i|$, and we require that some element is selected from every cell.*

`Input:` A multiset $D = \{d_1, d_2, \ldots, d_k\}$ of cell sizes, and a number ℓ.

`Output:` The number `CellSel`(D, ℓ) of all such possible selections.

`create` *table* $\boldsymbol{u}[1..k][1..\ell]$, *filled with* 0`;`
`for` $j = 1, 2, \ldots, d_1$ `do` `set` $\boldsymbol{u}[1][j] = \binom{d_1}{j}$`;`
`set` $z = d_1$`;`

```
for i = 2, 3, ..., k do
   add z += d_i;
   for j = i, i+1, ..., min(ℓ, z) do
      for s = 1, 2, ..., min(j − (i − 1), d_i) do
         add u[i][j] += u[i − 1][j − s] · (d_i choose s);
      done
   done
done
return u[k][ℓ].
```

Proof of Algorithm 3.2. Let $u_{i,j} = \boldsymbol{u}[i][j]$ be the number of cellular selections of j elements chosen among the first i cells. These numbers satisfy the recurrence relation

$$u_{i,j} = \sum_{s=1}^{r} u_{i-1,j-s} \cdot \binom{d_i}{s}$$

where r is the maximum number of elements than can be selected from the i-th cell to obtain a total of j elements. Since the i-th cell has d_i elements available, and the $i-1$ previous cells contributed at least one element each to the resulting j elements, it follows that $r = \min\{j-(i-1), d_i\}$.

Algorithm 3.2 just applies the previous recurrence in a correct order, and avoids useless computations like with values of j too small or too large. It runs in $O(k\ell^2)$ steps. ∎

Proof of Theorem 1.1. As in Theorem 2.8, the subgraph signature table $\boldsymbol{S}$ of a cograph can be computed in time proportional to the number of all possible double-signatures of size n, i.e. in $\exp\big(O(n^{2/3})\big)$. Then, summing the entries of $\boldsymbol{S}$, we compute the numbers of spanning subgraphs with a given number of edges and a number of components. As we have remarked previously, these numbers give (efficiently) the Tutte polynomial. ∎

The U polynomial of an n-vertex graph G is defined in [14] as

$$U(G; \mathbf{x}, y) = \sum_{F \subseteq E} x_{n_1} \cdots x_{n_k} (y-1)^{|F|-r(F)},$$

where $n_1, \ldots, n_k$ are the vertex sizes of the components of the spanning subgraph (V, F). If we let $x_1 = \cdots = x_n = x - 1$ in the expression above, we recover the Tutte polynomial $T(G; x, y)$ up to a power of $x-1$. It is clear that the subgraph signature table of a graph is precisely equivalent to the U polynomial, hence in the statement of Theorem 1.1 we can replace "U polynomial" for "Tutte polynomial".

4 Graph Clique-Width

In this section we give a more precise definition of clique-width and some of its properties.

Graphs from now on are labelled on the vertices; $V_i(G)$ denotes the set of vertices in G that have label i. A graph has clique-width $\leq k$ if it can be constructed using k labels and the following four operations:

1. $v(i)$: creates a new vertex with label i.
2. $\dot\cup$: produces the union of several disjoint graphs, without modifying the labels.
3. $\eta_{i,j}$, $i \neq j$: joins all the vertices labelled i to all the vertices labelled j. This operation does not create multiple edges.
4. $\rho_{i,j}$: all vertices labelled i are relabelled to have label j. It allows one to merge two label classes into one, thus freeing a label for later use.

An expression defining a graph G built from the above four operations using k labels is a *k-expression* for G.

Now we explain why cographs have clique-width at most two. Operations 1 and 2 are analogous to rules 1 and 2 for cographs introduced in Section 2.1. In order to perform the complete union $G \boxtimes H$ of two cographs (rule 3), one relabels all vertices of G to have label 1 and all vertices of H to have label 2 (using operations $\rho_{i,j}$), and then applies the operation $\eta_{1,2}$. The interested reader may check why the path P_4 has clique-width greater than two. Furthermore, we show in Fig. 4 examples of the graphs C_5 and C_7 having clique-width 3 and 4, respectively. (Actually, all cycles have clique-width at most 4.)

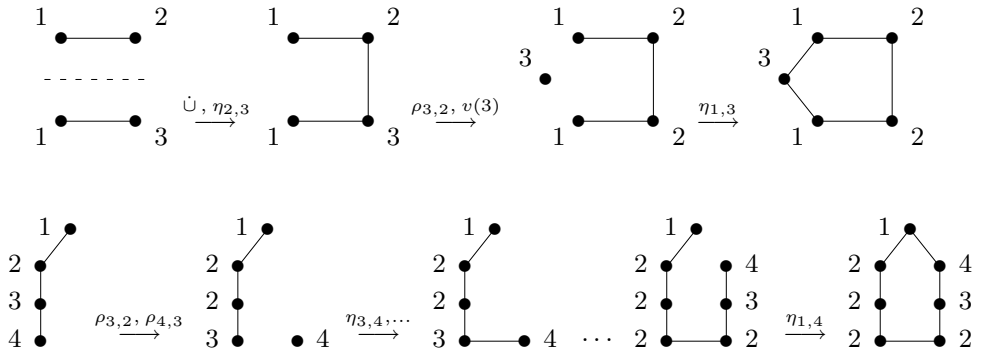

Fig. 4. An example – optimal expressions defining the cycles C_5 and C_7 (starting from trivially constructed subgraphs on the left).

A k-expression for G is *irredundant* if, whenever operation $\eta_{i,j}$ is applied, no vertex with label i has been previously joined to any vertex of label j. It can be shown [4] that for every k-expression, one can construct an irredundant k-expression defining the same graph. Hence in the next section we assume that graphs with bounded clique-width are defined by means of irredundant expressions.

Another important question concerns computing the clique-width of a graph, and more importantly, finding a defining k-expression. Until recently [16], algo-

rithms running on graphs of bounded clique-width *needed* a corresponding k-expression on the input. The first (and currently the only known) efficient way of approximating [16, 15] the expression for a graph of bounded clique-width, uses a new notion of rank-width [16]. It is remarkable how close is computation of rank-width on graphs [15] to computation of branch-width on binary matroids [7]: It is that rank-width of a bipartite graph equals branch-width of the matroid formed by the associated binary adjacency matrix minus one, and a simple translation can be used for general graphs. However, this interesting topic is far beyond the scope of our paper, and so we refer interested readers to the cited papers.

5 The Tutte Polynomial for Bounded Clique-Width

In this section we prove Theorem 1.2. Analogously to Section 2, the most involved part is Algorithm 5.4, which corresponds to adding edges in the operation $\eta_{i,j}$.

Instead of double-signatures, we need k-signatures to mark the different labels of the vertices belonging to the components of a subgraph. A k-signature is a multiset of k-tuples of non-negative integers, excluding the k-tuple $(0,\ldots,0)$. The *size* $\|\boldsymbol{\beta}\|$ of a k-signature $\boldsymbol{\beta}$ is the sum of all $(x_1+\cdots+x_k)$ for $(x_1,\ldots,x_k)\in\boldsymbol{\beta}$, respecting repetition in the multiset. As in the case of double-signatures we have:

Lemma 5.1. *There are* $\exp\left(\Theta(n^{k/(k+1)})\right)$ *distinct k-signatures of size n, for each fixed k.*

Since the subsequent proof is quite involved, it is worth mentioning that an easy encoding argument gives an upper bound of $\exp\left(O(n^{k/(k+1)}\log n)\right)$, which is almost as good: We limit the number of nonzero entries of the characteristic vector of a k-signature. In the worst case, at most all those $\Theta(t^k)$ entries corresponding to $(c_1,\ldots,c_k)$ are nonzero where $0\le c_i\le t$, and t is such that $\Theta(t^{k+1})=n$ (the size of the signature). Hence the characteristic vector of a k-signature has at most $\Theta\left(n^{k/(k+1)}\right)$ nonzero entries, and we may encode all k-signatures of size n by choosing those nonzero entries in all possible ways, and then trying all values between 1 and n for each;

$$\binom{n^k}{\Theta(n^{k/(k+1)})}\cdot n^{\Theta(n^{k/(k+1)})}=n^{\Theta(k\cdot n^{k/(k+1)})}=\exp\left(O(n^{k/(k+1)}\log n)\right).$$

Proof of Lemma 5.1. The proof is based on generating functions and complex analysis. Let p_n be the number of distinct k-signatures of size n. The associated generating function is equal to

$$P(z)=\sum_{n\ge 0}p_n z^n=\prod_{n\ge 1}\frac{1}{(1-z^n)^{\binom{n+k-1}{k-1}}}.$$

The reason is that the number of non-negative (ordered) solutions of $x_1+\cdots+x_k=n$ is equal to $\binom{n+k-1}{n}=\binom{n+k-1}{k-1}$. The infinite product encodes the fact that we are taking multisets of those k-tuples.

According to a result of Meinardus (see Theorem 6.2 in [1]), the asymptotic behaviour of p_n is determined by the associated Dirichlet series

$$D(s) = \sum_{n\geq 1} \frac{\binom{n+k-1}{k-1}}{n^s}.$$

Provided some analytical conditions on $D(s)$ hold, that in our case are easy to check, we have

$$p_n \sim Cn^\gamma \exp\left(Kn^{\rho/(\rho+1)}\right),$$

where C, γ, K are constants and ρ is the unique (simple) pole of $D(s)$ in a suitable region $\mathrm{Re}(s) > -C_0$, where $0 < C_0 < 1$.

Now it is clear that $D(s)$ can be expressed as a linear combination of $\zeta(s-k+1), \zeta(s-k+2), \ldots, \zeta(s)$, where $\zeta(s) = \sum_{n\geq 1} n^{-s}$ is the Riemann zeta function. Since $\zeta(s)$ has a unique simple pole at $s = 1$ for $\mathrm{Re}(s) > 0$, the pole of $D(s)$ we are looking for is at $\rho = k$, and this proves the result. ∎

The next two algorithms, which correspond to the operations $\dot\cup$ and $\rho_{i,j}$, need no special analysis.

Algorithm 5.2. *Combining the spanning subgraph k-signature tables of k-labeled graphs F and G into the one of the disjoint union $H = F \dot\cup G$.*

`Input:` Graphs F, G, and their subgraph k-signature tables $\boldsymbol{S}_F, \boldsymbol{S}_G$.
`Output:` The subgraph k-signature table $\boldsymbol{S}_H$ of $H = F \dot\cup G$.

`create` *empty table* $\boldsymbol{S}_H$ *of subgraph k-signatures of size* $|V(H)|$`;`
`for` *all k-signatures* $\boldsymbol{\alpha}_F \in \Sigma_F^k$, *and* $e_F = 0, 1, \ldots, |E(F)|$ `do`
  `for` *all k-signatures* $\boldsymbol{\alpha}_G \in \Sigma_G^k$, *and* $e_G = 0, 1, \ldots, |E(G)|$ `do`
    `set` $\boldsymbol{\alpha} = \boldsymbol{\alpha}_F \uplus \boldsymbol{\alpha}_G$ *(a multiset union)*`;`
    `add` $\boldsymbol{S}_H[\boldsymbol{\alpha}, e_F + e_G] \mathrel{+}= \boldsymbol{S}_F[\boldsymbol{\alpha}_F, e_F] \cdot \boldsymbol{S}_G[\boldsymbol{\alpha}_G, e_G]$`;`
`done.`

Algorithm 5.3. *Modifying the spanning subgraph k-signature table of a k-labeled graph G into the one of $\rho_{i,j}$, the relabeling $1 \to 2$ of G.*

`Input:` A k-labeled graph G, and its subgraph k-signature table $\boldsymbol{S}_G$.
`Output:` The subgraph k-signature table $\boldsymbol{S}_H$ of $H = \rho_{1,2}(G)$.

`create` *empty table* $\boldsymbol{S}_H$ *of subgraph k-signatures of size* $|V(G)|$`;`
`for` *all k-signatures* $\boldsymbol{\alpha}_G \in \Sigma_G^k$, *and* $e_G = 0, 1, \ldots, |E(G)|$ `do`
  `set` *k-signature* $\boldsymbol{\alpha}'_G = \{(0, a_1 + a_2, \ldots, a_k) : (a_1, a_2, \ldots, a_k) \in \boldsymbol{\alpha}_G\}$`;`
  `add` $\boldsymbol{S}_H[\boldsymbol{\alpha}'_G, e_G] \mathrel{+}= \boldsymbol{S}_G[\boldsymbol{\alpha}_G, e_G]$`;`
`done.`

The next algorithm computes the k-signature table of the graph $H = \eta_{1,2}(G)$ from that of G, assuming that there were no edges in G between vertices with labels 1 and 2 (like in an irredundant expression). To understand the main idea,

let us consider the following method for obtaining all the spanning subgraphs in H such that their restriction to G gives a certain subgraph G_α. Start by relabeling every vertex with label 1 in G to 0, meaning that these vertices have not been processed yet. Then choose a vertex v with label 0, relabel to 1, and consider all the different subgraphs of H we can generate from the original one by adding new edges from v to some (selected) vertices with label 2. Obviously, the restriction of any of these subgraphs to G still gives G_α. Iterate the process for every generated subgraph and for every vertex with label 0 until none remains.

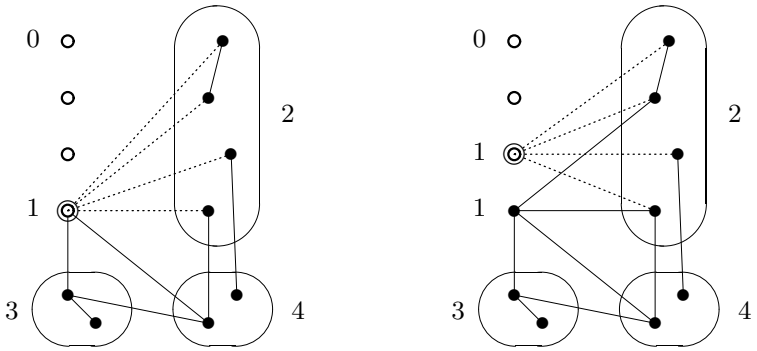

Fig. 5. Two steps illustrating the method of counting all spanning subgraphs of $H = \eta_{1,2}(G)$; the hollow vertices are not processed yet (label 0), and the dotted edges show possible choices of new edges from the marked vertex (v).

See an illustration of the method in Fig. 5. Algorithm 5.4 essentially follows this process, but it improves the running time by working with the signatures instead of subgraphs, and with connected components instead of single vertices, as it has been done in Algorithms 2.5 and 3.1.

The algorithm counts all spanning subgraphs of H such that their restriction to G is of a signature $\boldsymbol{\alpha}$, for every possible $\boldsymbol{\alpha}$. A $(k+1)$-signature table $\boldsymbol{Y}$ is used to store all intermediate results of the computation (together). Instead of processing a vertex v by one at a time, we choose a component B with some labels 0 from a signature $\boldsymbol{\beta}$ in Y, and then we choose a submultiset $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}$ signing the components that B will be joined to, using exactly f of the new edges of H. Procedure `CellSel` (Algorithm 3.2) computes efficiently the number of ways this can be done. We add the resulting signatures and numbers to the table $\boldsymbol{Y}$, and we repeat until no signature with vertices labeled 0 remains in $\boldsymbol{Y}$. We then update the table $\boldsymbol{S}_H$ with the signatures computed in the table $\boldsymbol{Y}$, and we start again from a new signature $\boldsymbol{\alpha}$ of G.

Algorithm 5.4. *Updating the subgraph k-signature table of a k-labeled graph G, such that there is no edge between the labels* 1 *and* 2 *in G, into the one of the graph H obtained from G by adding all edges between the labels* 1 *and* 2.

`Input:` A k-labeled graph G, and its subgraph k-signature table $\boldsymbol{S}_G$.

`Output:` The subgraph k-signature table $\boldsymbol{S}_H$ of $H = \eta_{1,2}(G)$.

`create` *empty table* $\boldsymbol{S}_H$ *of subgraph* k*-signatures of size* $|V(H)|$`;`
`for` *all* $\boldsymbol{\alpha} \in \Sigma_G^k$, *and* $e = 0, 1, \ldots, |E(G)|$ *s.t.* $\boldsymbol{S}_G[\boldsymbol{\alpha}, e] > 0$ `do`
  `//` *Imagine a part. span. subgraph in* G *of a* k*-signature* $\boldsymbol{\alpha}$ *with* e *edges.*
  `create` *empty table* $\boldsymbol{Y}$ *of subgraph* $(k+1)$*-signatures of size* $|V(H)|$`;`
  `set` $(k+1)$*-signature* $\boldsymbol{\alpha}_0 = \big\{(a_1, 0, a_2, \ldots, a_k) : (a_1, a_2, \ldots, a_k) \in \boldsymbol{\alpha}\big\}$`;`
  `set` $\boldsymbol{Y}[\boldsymbol{\alpha}_0,\, e] \,=\, 1$`;`
  `while` *exists* $\boldsymbol{\beta} \in \Sigma_H^{k+1}$, $b = (b_0, b_1, \ldots, b_k) \in \boldsymbol{\beta}$, *and* e', *such that* $\boldsymbol{Y}[\boldsymbol{\beta},\, e'] > 0$ *and* $b_0 > 0$ `do`
    `select` *such* $\boldsymbol{\beta}$, e', *and* b, *with maximal* $\|\boldsymbol{\beta}\restriction_0\|$`;`
      `//` *Imagine a part. spanning subgraph in* H *with* e' *edges and*
      `//` *a* $(k+1)$*-signature* $\boldsymbol{\beta}$, *and its component* B *corresp. to* b*:*
      `//` b_0 *of the vert. of* B *are going to be "joined to" labels* 2 *in* H.
    `set` $y = Y[\boldsymbol{\beta},\, e']$`,` $Y[\boldsymbol{\beta},\, e'] = 0$`,` $\boldsymbol{\beta}' = \boldsymbol{\beta} \setminus \{b\}$`;`
    `for` *all different submultisets* $\boldsymbol{\gamma} \subseteq \boldsymbol{\beta}'$ *such that* $c_2 > 0$ *for each* $(c_0, c_1, c_2, \ldots, c_k) \in \boldsymbol{\gamma}$ `do`
      `//` *We connect* B *to all components in* $\boldsymbol{\gamma}$.
      `set` $r = \displaystyle\prod_{d \in \langle \boldsymbol{\beta}' \rangle} \binom{\#_{\boldsymbol{\beta}'}\, d}{\#_{\boldsymbol{\gamma}}\, d}$`;`
      `for` $i = 0, 1, \ldots, k$ `do` $d_i = \|\boldsymbol{\gamma}\restriction_i\| = \sum_{(c_0, \ldots, c_k) \in \boldsymbol{\gamma}} c_i$`;`
      `set` $d = (d_0, d_1 + b_0 + b_1, d_2 + b_2, \ldots, d_k + b_k)$`;`
      `set` $(k+1)$*-signature* $\boldsymbol{\delta} = \big(\boldsymbol{\beta}' \setminus \boldsymbol{\gamma}\big) \uplus \{d\}$`;`
      `set` *multiset* $D = b_0 \cdot (\boldsymbol{\gamma}\restriction_2) = \{b_0 c_2 : (c_0, c_1, c_2, \ldots, c_k) \in \boldsymbol{\gamma}\}$`;`
      `for` $f = |\boldsymbol{\gamma}|, |\boldsymbol{\gamma}| + 1, \ldots, b_0 \cdot (d_2 + b_2)$ `do`
        `//` *We count all the cellular selections from* $D \uplus \{b_2\}$ *here,*
        `//` *but we allow to select nothing from* $\{b_2\}$ *as well.*
        `call` *Algorithm 3.2:*
(‡) $p \,=$ `CellSel`$(D \uplus \{b_2\},\, f) +$ `CellSel`$(D,\, f)$`;`
        `add` $\boldsymbol{Y}[\boldsymbol{\delta},\, e' + f] \mathrel{+}= y \cdot r \cdot p$`;`
      `done`
    `done`
  `done`
  `for` *all* $\boldsymbol{\beta} \in \Sigma_H^{k+1}$, *and* f, *such that* $\boldsymbol{Y}[\boldsymbol{\beta}, f] > 0$ `do`
    `set` *signature* $\boldsymbol{\alpha}_0 \,=\, \{(b_1, \ldots, b_k) : (b_0, b_1, \ldots, b_k) \in \boldsymbol{\beta}\}$`;`
    `add` $\boldsymbol{S}_H[\boldsymbol{\alpha}_0, f] \mathrel{+}= \boldsymbol{Y}[\boldsymbol{\beta},\, f] \cdot \boldsymbol{S}_G[\boldsymbol{\alpha},\, e]$`;`
  `done`
`done.`

Proof of Algorithm 5.4. The idea is analogous to the proofs of Algorithms 2.5 and 3.1. The only noticeable differences are the following two:

– Since we are now adding edges inside the same graph (instead of composing two previously disjoint graphs), we need an artificial new label 0 for marking those vertices that still have to be processed, among those having original label 1. One little advantage of the extra label is that now we can store all intermediate results of our computation in the same $(k+1)$-signature table $\boldsymbol{Y}$, and to control the computation we just need one large `while` cycle.
– Unlike for cographs, we now have to consider the possibility that our selected component B has vertices of both labels 0 and 2, and hence we may want to add some edges induced on $V(B)$ as well. This is taken care of on the line (‡) of the algorithm.

Again, it is easier to imagine a particular spanning subgraph G_α with e edges in the place of the signature $\boldsymbol{\alpha}$, since subsequent computations do not depend on a particular choice of G_α. Under this assumption table $\boldsymbol{Y}$ counts $(k+1)$-labelled subgraphs of H whose restriction to G is G_α. It is convenient to see table $\boldsymbol{Y}$ as a `set` of subgraphs, stored according to signature for the sake of efficiency.

This set $\boldsymbol{Y}$ contains at the beginning the subgraph G_α, where vertices of label 1 receive label 0 to mark that they are unprocessed. At every iteration of the `while` cycle we choose some subgraphs of $\boldsymbol{Y}$ and we process some of its vertices of label 0, marking them with label 1 and considering all possible ways of joining them to vertices of label 2. The resulting subgraphs are stored in the table $\boldsymbol{Y}$ in place of the former ones. The `while` loop ends when no subgraph in $\boldsymbol{Y}$ has vertices of label 0.

To be more precise, at every iteration of the `while` loop we choose subgraphs of $(k+1)$-signature $\boldsymbol{\beta}$ and e' edges. (Imagine again a particular subgraph G_β.) We select a component B with $b_0 > 0$ vertices of label 0. We process component B by joining its vertices of label 0 to some of the $\|\boldsymbol{\beta}\restriction_2\|$ vertices of label 2 in G_β. Hence the components containing the vertices of label 2 may be joined to B. The `for` loop iterates over all suitable subsets $\boldsymbol{\gamma}$ of such components to account for all possible resulting signatures. Components not in $\boldsymbol{\gamma}$ receive no edge from B, while components in $\boldsymbol{\gamma}$ receive at least one edge, so we call procedure `CellSel` to count efficiently the number of ways $\boldsymbol{\gamma}$ may be joined to B. In fact, `CellSel` is called twice, since the vertices of label 0 and 2 in B itself may be joined by either none or some edges.

Observe that a pair of descendants of, say, G_β differ in at least one edge joining vertices of labels 1 and 2, and further descendants of them will still differ at the same edge, since new edges are only added between vertices of labels 0 and 2. This implies that $\boldsymbol{Y}$ is free of duplicates, because all subgraphs have a common ancestor G_α. So, at the end, we count each spanning subgraph $W \subseteq H$ such that $W \restriction G = G_\alpha$ exactly once in the table $\boldsymbol{Y}$. After multiplying by $\boldsymbol{S}_G[\boldsymbol{\alpha}, e]$, we record in $\boldsymbol{S}_H$ the number of all spanning subgraphs of H having their restriction to G of signature $\boldsymbol{\alpha}$, for each $\boldsymbol{\alpha}$. ∎

Proof of Theorem 1.2. The idea is the same as in the proof of Theorem 1.1 at the end of Section 3. In the `while` cycle we always choose signature $\boldsymbol{\beta}$ among those with the maximal number of unprocessed vertices ($\|\boldsymbol{\beta} \restriction_0\|$), and then

we strictly decrease the number of those. Hence we never process the same pair $(\boldsymbol{\beta}, e)$ twice. So time complexity is again dominated by the lengths of the tables involved. Since we need a table of $(k+1)$-signatures, the complexity $\exp\left(O(n^{(k+1)/(k+1+1)})\right) = \exp\left(O(n^{1-\frac{1}{k+2}})\right)$ follows from Lemma 5.1. ∎

Finally we remark that, exactly as in the case of cographs, the algorithm computes the full U polynomial on graphs of bounded clique-width.

6 Concluding remarks

We have shown that the Tutte and U polynomials can be computed in subexponential time for cographs, and more generally for graphs with bounded clique-width. Such a result is very unlikely to hold for all graphs. Of course, the important question of whether the Tutte polynomial can be computed in polynomial time, or the problem is #P–hard even for graphs of bounded clique-width, remains open. (The U polynomial is obviously not computable in polynomial time due to its size.)

On the other hand, the *chromatic* polynomial for graphs of bounded clique-width can be computed in polynomial time (although not FPT). This follows by adapting the algorithm in [10] for computing the chromatic number, keeping track also of the number of r-colorings for $r = 1, \ldots, n$, where n is the number of vertices; see in [12]. To our knowledge, that is possibly the only currently known natural example of graph classes other than chordal graphs, where the chromatic polynomial can be computed in polynomial time, but the complexity of computing the Tutte polynomial is undecided.

References

1. G.E. Andrews, *The theory of partitions*, Cambridge U. Press, Cambridge, 1984.
2. A. Andrzejak, *An Algorithm for the Tutte Polynomials of Graphs of Bounded Treewidth*, Discrete Math. 190 (1998), 39–54.
3. B. Courcelle, J.A. Makowsky, U. Rotics, *Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width*, Theory Comput. Systems 33 (2000), 125–150.
4. B. Courcelle, S. Olariu, *Upper bounds to the clique width of graphs*, Discrete Appl. Math. 101 (2000), 77–114.
5. D.G. Corneil, Y. Perl, and L.K. Stewart, *A linear recognition algorithm for cographs*, SIAM J. Comput. 14 (1985), 926–934.
6. O. Giménez, M. Noy, *On the complexity of computing the Tutte polynomial of bicircular matroids*, Combin. Probab. Computing, to appear.
7. P. Hliněný, *A Parametrized Algorithm for Matroid Branch-Width*, SIAM J. Computing 35 (2005), 259–277 (electronic).
8. P. Hliněný, *The Tutte Polynomial for Matroids of Bounded Branch-Width*, Combin. Probab. Computing, to appear (2006).
9. F. Jaeger, D.L. Vertigan, D.J.A. Welsh, *On the Computational Complexity of the Jones and Tutte Polynomials*, Math. Proc. Camb. Phil. Soc. 108 (1990), 35–53.

10. D. Kobler, U. Rotics, *Edge dominating set and colorings on graphs with fixed clique-width*, Discrete Applied Math. 126 (2003), 197–221.
11. J.H. van Lint, R.M. Wilson, *A Course in Combinatorics*, Cambridge University Press, Cambridge, 1992.
12. J.A. Makowsky, U. Rotics *Computing the chromatic polynomial on graphs of bounded clique-width*, preprint, September 2005.
13. S.D. Noble, *Evaluating the Tutte Polynomial for Graphs of Bounded Tree-Width*, Combin. Probab. Computing 7 (1998), 307–321.
14. S.D. Noble, D.J.A. Welsh, *A weighted graph polynomial from chromatic invariants of knots*, Ann. Inst. Fourier (Grenoble) 49 (1999), 1057–1087.
15. Sang-Il Oum, *Approximating Rank-width and Clique-width Quickly*, In: WG 2005, Proccedings, Lecture Notes in Computer Science 3787 (2005), 49–58.
16. Sang-Il Oum, P.D. Seymour, *Approximating Clique-width and Rank-width*, J. Combin. Theory Ser. B, to appear (2006).