NEGATIVE CORRELATION OF EDGE EVENTS ON UNIFORM SPANNING FORESTS

PROF. GEOFFREY GRIMMETT, STEPHAN WINKLER

Contents

1. Introduction

Positive and negative association, or dependence, are fundamental properties enjoyed by broad families of events. While the former may be readily established by checking the so-called FKG lattice property, no such universal tool exists to prove negative association. Even ad-hoc arguments tailored to specific settings have proved elusive.

One well-known exception is negative association of certain increasing edge events on uniform spanning trees (UST). The ingenious proof exploits connections to the theory of electrical networks, invoking the Raleigh-principle. Unfortunately, it does not easily extend to more general settings.

We obtain two natural extensions of the sample space of UST by dismissing either one of the two defining properties of trees: acyclic or connected subgraphs give rise to the uniform spanning forest (USF) or uniform connected subgraph (UCG) measures, respectively.

This computational study addresses the conjecture that, for any fixed graph G, the presence of two distinct edges in a forest F is negatively associated, if F is drawn uniformly at random from the set of forests contained in G.

We present an algorithm, implementation and numerical results that establish the result for all graphs G on up to 8 vertices. Extrapolating certain distributional statistics suggests that counterexamples for G of higher order, should they exist, are probably sporadic.

2. Statement of the Conjecture

We begin by recapitulating some common graph-theoretic jargonthat will be used throughout this paper, and give a precise statement of the conjecture under investigation.

$G = (V, E)$ denotes an undirected *graph* with *vertex set* $V = V(G)$ and *edge set* $E = E(G) \subset \{e = \langle v, w\rangle : v \neq w \in V\}$. $|V|$ is called *order*, and $|G| := |E|$ is called *size* of G. If the vertex set V is understood, it is sometimes convenient to identify G with its edge set, writing $E = (V, E)$. Let $\mathcal{G}(V)$ be the set of undirected graphs with vertex set V. If the vertices are labelled in the natural way $V = [n] := \{1, \ldots, n\}$, we shall write $\mathcal{G} = \mathcal{G}([n])$. The particular graph with all edges present is called *complete*, and denoted K_n.

For convenience, we write set unions and differences as $X + Y := X \cup Y$ and $X - Y := X \setminus Y$, respectively. Similarly, we define $X \pm x := X \pm \{x\}$. Given graphs $F, G \in \mathcal{G}(V)$ with edge sets D, E, we write $G \pm F$ and $G \cap F$ for the graphs with edge sets $E \pm D$ and $E \cap D$, respectively. We say that F is a subgraph of G, or $F \subset G$, if $D \subset E$. $\mathcal{G}(V)$ is partially ordered by $\subset$, with the empty and complete graphs on V as smallest and largest element, respectively.

We may construct a bijection between graphs $G \in \mathcal{G}$ and N-tuples ω of binary digits, where $N = \binom{n}{2}$: each co-ordinate $\omega(e)$ of ω indicates the presence or absence of the corresponding edge e in G. $\mathcal{G}$ is thus identified with the boolean lattice $\Omega = 2^N$. We shall write $G(\omega)$ for the graph encoded by ω.

A probability measure on Ω is a non-negative function μ satisfying $\sum_{\omega\in\Omega} \mu(\omega) = 1$. We say that μ is *negatively associated* if $\int fg d\mu \leq \int f d\mu \int g d\mu$ for bounded increasing functions f, g depending on disjoint sets of edges.

In this study, we restrict ourselves to a specific choice of μ, f and g. Fix any graph $G \in \mathcal{G}$ and consider the set $\mathcal{F}(G) \subset \mathcal{G}$ of spanning forests contained in G. We denote its cardinality by $K(G) = |\mathcal{F}(G)|$ and define the Uniform Spanning Forest $USF(G)$ with probability measure $\mu = \mathbb{P}$,

$$\mathbb{P}(\omega) = \begin{cases} K(G)^{-1} & \text{if } F(\omega) \in \mathcal{F}(G) \\ 0 & \text{otherwise} \end{cases}$$

In other words, $F \sim USF(G)$ is a random spanning forest contained in G, chosen uniformly in that $\mathbb{P}(T = \tau) = K(G)^{-1}$. Finally, we consider $f(\omega) = 1_{\{e\in F(\omega)\}}$ and $g(\omega) = 1_{\{f\in F(\omega)\}}$ for distinct edges

$e, f \in E$. These functions are obviously bounded, increasing, and depend on disjoint sets of edges $\{e\}$, $\{f\}$. We can now state our conjecture that $\mathbb{P}$ is negatively associated for this particular choice of f, g.

Conjecture 2.1. *For every graph $G \in \mathcal{G}$, if $F \sim USF(G)$ and $e \neq f \in E$, then*

$$\mathbb{P}(e, f \in F) \leq \mathbb{P}(e \in F)\mathbb{P}(f \in F). \tag{2.1}$$

Since Ω is finite, computing these probabilities amounts to counting the combinatorial quantities involved. For this purpose, we introduce some notation. Given any graph $G \in \mathcal{G}$ and disjoint edge sets X, Y contained in E, we define $\mathcal{F}^X{}_Y(G)$ to denote the set of spanning forests $F \subset G$ containing X and disjoint from Y; formally $\mathcal{F}^X{}_Y(G) = \{F \in \mathcal{F}(G) : X \subset F, Y \cap F = \emptyset\}$. Let $K^X{}_Y(G) = |\mathcal{F}^X{}_Y(G)|$ denote its cardinality.

The edge sets X and Y will be referred to as *constraints*. For brevity, if either X or Y is empty, it is omitted from the index: $\mathcal{F}^\emptyset{}_Y = \mathcal{F}_Y$, $\mathcal{F}^X{}_\emptyset = \mathcal{F}^X$ and $\mathcal{F}^\emptyset{}_\emptyset = \mathcal{F}$. Singleton sets are denoted by their elements, without enclosing set brackets: $\mathcal{F}^{\{e\}}{}_{\{f\}} = \mathcal{F}^e{}_f$ etc. The same conventions apply to $K^X{}_Y(G)$. Inequality 2.1 now takes the form

$$K^{\{e,f\}}(G)K(G) \leq K^e(G)K_f(G)$$

Notice that $K = K^e + K_e$. Hence $K^e = K^{\{e,f\}} + K^e{}_f$ and, similarly, $K_e = K^f{}_e + K_{\{e,f\}}$. Substituting these expressions, we obtain

$$K^{\{e,f\}}(G)K_{\{e,f\}}(G) \leq K^e{}_f(G)K^f{}_e(G) \tag{2.2}$$

This is the form that we will take as starting point of our analysis. We define the *standard constraints* $X_k := \{e, f\}, \emptyset, e, f$ and $Y_k := \{e, f\} - X_k$ for $k = 1, 2, 3, 4$, respectively. Writing $K_k := K^{X_k}{}_{Y_k}(G)$, inequality 2.2 condenses to $K_1K_2 \leq K_3K_4$.

3. Algorithm

3.1. **Overview.** The difficulty of verifying inequality 2.2 numerically is due to the nesting of computations separately believed NP-hard. Naively, it is required

(1) to enumerate all possible graphs G of target order n,
(2) to consider, for each G, all pairs of distinct edges $e \neq f \in E$, and finally
(3) to compute, for each choice of G, e and f, the values of K_k appearing in equation 2.2.

Of course, this direct approach is not viable for reasons of complexity: first, there are $2^N = O(2^{n^2})$ subgraphs G of K_n to consider. Second, a graph of size s will allow $O(\binom{s}{2})$ distinct choices of edges $e \neq f$. Third, K_k is not easy to obtain: an elementary strategy involves examining all subgraphs F of G with $X_k \subset F$ and $Y_k \cap F = \emptyset$, a total of $2^{|G|-|Y_k|-|X_k|}$, or $O(2^{n^2})$, candidates. For each F, we must determine if it is indeed acyclic, another non-trivial task: examining one edge at a time until a cycle is discovered or F exhausted requires $O(|F|) = O(n^2)$ operations on average.

Nesting algorithms of (super-)exponential complexity results in unacceptable run-times: our rough analysis shows that a naive algorithm will require $O(n^2 2^{2n^2})$ operations, or $O(10^{10})$ for n as small as 4. In the following sections, we present methods that will extend the scope of computational verification up to $n = 8$, permitting non-trivial quantitative and qualitative deductions for even larger graphs.

Section 3.2 presents an ad-hoc counting algorithm to compute $K^X{}_Y(G)$ for a given set of parameters (G, X, Y): it starts from the smallest forest in G that satisfies constraints X, Y and enumerates its possible extensions.

Sections 3.3 - 3.5 address the crucial problem of reducing the number of explicit evaluations of ${K^X}_Y(G)$ required. We develop and prove a new method to index classes of parameters (G, X, Y) with common value ${K^X}_Y(G)$, exploiting invariance under certain isomorphisms.

This is heavily used in section 3.6, which describes an economical way of enumerating distinct instances (G, e, f) of conjecture 2.1: We fix the only two non-isomorphic choices of e, f (connected or not) ahead of time and examine the hierarchy of graphs G sandwiched between $\{e, f\}$ and K_n 'bottom-up', skipping instances identified as equivalent to earlier ones.

Finally, in section 3.7, we develop a formula expressing $K^X(D)$ as a sum over values $K^{X'}(D')$ for larger sets of constraints X' and smaller graphs D', which are cheaper to compute and may even be available from earlier computations.

3.2. **Explicit computation of ${K^X}_Y(G)$.** In order to compute ${K^X}_Y(G)$, we have to enumerate the set ${\mathcal{F}^X}_Y(G)$ of forests $F \subset G$ containing X and disjoined from Y. We shall refer to these forests as *valid.* We make two preliminary observations regarding the sets of constraints X and Y. (1) We may assume that X is itself acyclic, otherwise ${K^X}_Y(G)$ is trivially zero. (2) Y may be subsumed into G, since ${K^X}_Y(G) = K^X(D)$ with $D := G - Y$.

The partial order $\subset$ on $\mathcal{G}$ suggests two natural starting points: a minimal or (if any exists) a maximal valid forest F. The former is just the graph with edge set X, whereas the latter is more elusive and not necessarily unique: it would have to be constructed by dropping edges from D until no cycles remain, a rather unpleasant stopping condition algorithmically. We shall therefore proceed 'bottom-up', starting from X.

Our strategy is to enumerate all valid forests F of the same size before proceeding to the next layer in the hierarchy of graphs sandwiched between X and D. Let $\mathcal{F}_s := \{F \in \mathcal{F}^X(D) : |F| = s\}$ denote the layer of valid forests of size s for $s = |X|, \dots, |D|$. Clearly $\{\mathcal{F}_s\}$ forms a partition of $\mathcal{F}^X(D)$.

The construction of the layers $\mathcal{F}_s$ is best described inductively: We start from $\mathcal{F}_{|X|} = \{X\}$. Having found $\mathcal{F}_s$, we construct $\mathcal{F}_{s+1}$ by considering every $F \in \mathcal{F}_s(X)$ in turn. The forest F can be thought of as (vertex- and edge-)disjoint union of connected components C_i, noting that isolated vertices form singleton components. We seek to enlarge F by one edge without forming a cycle. Every such augmentation arises by merging two distinct components (whereas adding an edge to a component must form a cycle). Thus, from every pair of components $C_i \neq C_j$ of F, we obtain new elements F' of $\mathcal{F}_{s+1}(X)$ by adding an edge joining C_i to C_j. Formally,

$$\begin{aligned} \mathcal{F}_{|X|} &= \{X\} \\ \mathcal{F}_{s+1} &= \{F + e : F = \textstyle\bigsqcup_{i \in I} C_i \in \mathcal{F}_s \text{ and } e = \langle v_i, v_j \rangle \text{ with } v_i \in V(C_i), v_j \in V(C_j), i \neq j \in I\} \end{aligned}$$

Remark: This definition hides the fact that a given element of $\mathcal{F}_{s+1}$ may be arrived at in various ways from different elements on the layer below. Correct book-keeping, which involves storing forests as sets of components and detecting duplicates, is somewhat subtle to implement, as discussed in Chapter 3.

As soon as the highest layer $s = |D|$ is reached, we may read off

$${K^X}_Y(G) = K^X(D) = \sum_{s=|X|}^{|D|} |\mathcal{F}_s|.$$

3.3. **Indexing parameter classes** (G, X, Y)**.** Verifying inequality 2.2 for some graph G and choice of edges $e \neq f \in E$ requires the four values $K_k, k \in [4]$. Unfortunately, the above brute-force algorithm to compute ${K^X}_Y(G)$ becomes highly time-consuming for large graphs G and small sets of constraints X, Y, as the the layers $\mathcal{F}_s(X)$ grow fast in n and s. It is therefore crucial for good performance to avoid this explicit computation as often as possible.

Our strategy is to devise some function i that indexes equivalence classes of parameters (G, X, Y) with common value of $K^X{}_Y(G)$. Let this value be K_ι for all parameters with index $i(G, X, Y) = \iota$. Obviously, one explicit computation of K_ι suffices; the obtained value may be buffered in a database and retrieved to give $K^X{}_Y(G)$ for all parameters (G, X, Y) with same index ι.

To optimize performance, we seek to minimize the range of indices ι. Ideally, $i(G, X, Y) = i(G', X', Y')$ if and only if $K^X{}_Y(G) = K^{X'}{}_{Y'}(G')$. Such a function i being impossible to compute, however, we have to allow i to assume more than one value on parameters classes with common value of $K^X{}_Y(G)$. A good compromise is achieved by an easy-to-compute i with sufficiently small index space.

Our construction of such an index i relies on the following key claim (to be stated precisely and proved in section 3.5): Setting $D^{(\prime)} := G^{(\prime)} - Y^{(\prime)}$, a sufficient condition for $K^X(D)$ and $K^{X'}(D')$ to be equal is that

(K1) D and D' have similar structure,
(K2) X and X' as well as $D - X$ and $D' - X'$ have similar structure, respectively, and
(K3) X and $D - X$ are composed in a way similar to X' and $D' - X'$.

Conditions (K1) and (K2) are motivated by the fact that the number of spanning forests of D containing X depends on the structures of both D, X and $D - X$ (see figures **??** and **??**). However, the first two conditions are not sufficient, as illustrated by the graphs in figure **??**. It is intuitively clear that (K3) closes the gap, but non-trivial to decide precisely how it should to be enforced: for example, it is insufficient to ensure that X and D coincide in the same number of vertices as X' and D', even when all vertex degrees match (see counterexample **??**).

Section 3.4 develops the language required to precisely formulate the conditions (K1) - (K3). Section 3.5 gives the correct interpretation of (K3) and contains a proof of the above claim. As a corollary, we shall finally define $i(G, X, Y)$ to be the triple $(D', X', D' - X')$ for suitably defined canonical isomorphs of D, X and $D - X$.

3.4. **Canonical labels.**

The following definitions and conventions are based on McKay (**??**).

3.4.1. *Partitions.* A *partition* $\pi = (V_1, \ldots, V_k)$ of a set V is a sequence of disjoint non-empty subsets of V whose union is V The set of all partitions of V will be denoted by $\Pi(V)$. If V is the vertex set of a graph $G = (V, E)$ and $\pi \in \Pi(V)$, then the pair (G, π) is termed *partitioned graph.*

The components V_i of a partition π are called its *cells.* Note that the order of the cells is significant, but the order of the vertices within each cell is not. A *singleton*, or *trivial*, *cell* is a cell $V_i = \{v\}$ with a single element v, which is said to be *fixed* by π. If every cell of π is trivial, then π is a *discrete* partition. On the other hand, if π contains only one cell $V_1 = V$, then π is called *unit* parition.

Any partition π may be identified with a vertex-colouring $c : V \to [k]$ by assigning colour i to all vertices in cell V_i of π. In this alternative language, cells are also called colour classes.

3.4.2. *Permutation groups.* Let γ be a permutation of a set V. The image of $v \in V$ under γ will be denoted v^γ. Similarly, if $U \subset V$ then $U^\gamma = \{u^\gamma : u \in U\}$. If $\pi = (V_1, \ldots, V_k) \in \Pi(V)$ is a partition of V, we set $\pi^\gamma = \{V_1^\gamma, \ldots, V_k^\gamma\}$. If $G = (V, E)$ is a graph, then $G^\gamma \in \mathcal{G}(V)$ is the graph with vertices v^γ and w^γ adjacent if and only if v and w are adjacent in G, i.e. $E(G^\gamma) = \{\langle v^\gamma, w^\gamma \rangle : \langle v, w \rangle \in E\}$.

Two partitions π_1 and π_2 of V are called *compatible* if there exists a permutation γ of V such that $\pi_2 = \pi_1^\gamma$. In other words, π_1 and π_2 contain cells of the same size, with the same frequency, and in the same order.

Two graphs $G_1, G_2 \in \mathcal{G}(V)$ are called *isomorphic* if there exists a permutation γ of V (called *isomorphism*) such that $G_2 = G_1^\gamma$. We denote this by $G_1 \sim G_2$; it is easy to check that $\sim$ defines an equivalence relation on $\mathcal{G}(V)$. If the graphs G_i carry partitions π_i, we say that there is a *partition-preserving* isomorphism γ from G_1 to G_2 if $G_2 = G_1^\gamma$ and $\pi_2 = \pi_1^\gamma$.

The *automorphism group* $\mathrm{Aut}(G, \pi)$ of a partitioned graph (G, π) is the set of all permutations γ such that $G^\gamma = G$ and $\pi^\gamma = \pi$. Since the order of cells in partitions is significant, the last condition means that γ fixes the cells in π setwise. If π is the unit partition, we obtain the full automorphism group of G.

3.4.3. *Canonical labelling map $\mathcal{C}$.* A *canonical labelling map* is a function $\mathcal{C} : \mathcal{G}(V) \times \Pi(V) \to \mathcal{G}(V)$ such that, for any graph G, partition π of V, and permutation γ of V, we have

(C1) $\mathcal{C}(G, \pi) \sim G$,
(C2) $\mathcal{C}(G^\gamma, \pi^\gamma) = \mathcal{C}(G, \pi)$, and
(C3) If $\mathcal{C}(G, \pi^\gamma) = \mathcal{C}(G, \pi)$, then $\pi^\gamma = \pi^\delta$ for some $\delta \in \mathrm{Aut}(G)$.

Remarks: Properties (C1) - (C3) do not define $\mathcal{C}$ uniquely, but suffice for our purposes. We will later work with an arbitrary but fixed choice of $\mathcal{C}$. If π is the unit partition, $\pi = (V)$, we may abbreviate $\mathcal{C}(G) := \mathcal{C}(G, (V))$. The main use of a canonical label is to solve certain graph isomorphism problems, as shown by the following theorem:

Theorem 3.1. *(McKay 1981)*

Let $G_1, G_2 \in \mathcal{G}(V)$ be graphs, $\pi_1, \pi_2 \in \Pi(V)$ compatible partitions and γ a permutation of V. Then $\mathcal{C}(G_1, \pi_1) = \mathcal{C}(G_2, \pi_2)$ if and only if there is a partition-preserving isomorphism from G_1 to G_2.

Proof. Since π_1, π_2 are compatible, we may write $\pi_1 =: \pi$ and $\pi_2 = \pi^\gamma$ for some permutation γ of V.

$\Leftarrow$: Assume that there exists a permutation δ of V such that $G_2 = G_1^\delta$ and $\pi^\gamma = \pi^\delta$. Substituting, we have $\mathcal{C}(G_2, \pi^\gamma) = \mathcal{C}(G_1^\delta, \pi^\delta) = \mathcal{C}(G_1, \pi)$ by property (C2).

$\Rightarrow$: Suppose conversely that $\mathcal{C}(G_1, \pi) = \mathcal{C}(G_2, \pi^\gamma)$. By (C1), $G_2 = G_1^\beta$ for some permutation β of V. Therefore, $\mathcal{C}(G_2, \pi^\gamma) = \mathcal{C}(G_1^\beta, \pi^\gamma) = \mathcal{C}(G_1, \pi^{\gamma\beta^{-1}})$, by (C2). Hence $\mathcal{C}(G_1, \pi) = \mathcal{C}(G_1, \pi^{\gamma\beta^{-1}})$. By (3) there is some $\alpha \in \mathrm{Aut}(G_1)$ such that $\pi^{\gamma\beta^{-1}} = \pi^\alpha$, and so $\pi^\gamma = \pi^{\alpha\beta}$. But $\alpha \in \mathrm{Aut}(G_1)$, and so $G_2 = G_1^\beta = G_1^{\alpha\beta}$. This shows that $\delta = \alpha\beta$ exists as desired. □

Remark: In the language of vertex-colourings, Theorem 3.1 says that if two graphs G_1 and G_2 are coloured with the same number of vertices of each colour, then if $\mathcal{C}(G_1, \pi_1) = \mathcal{C}(G_2, \pi_2)$, there is a colour-preserving isomorphism from G_1 to G_2 (here, π_1 and π_2 are the colourings, with the colours in the same order in each).

3.5. **Sufficient condition for $K^X(D) = K^{X'}(D')$.** We are now in the position to precisely formulate and prove the result sketched in section 3.3 and, as a corollary, obtain the desired index function i. Resuming our earlier notation, we require a rather rigid criterion for the composition of $X, D - X$ and $X', D' - X'$ to guarantee that $K^X(D) = K^{X'}(D')$. This is obtained by constructing a certain vertex-colouring of D and D' and using the canonical labelling map $\mathcal{C}$ from section 3.4.

To begin with, let $\mathcal{C}(D)$ be the canonically-labelled isomorph of D (with unit partition), such that $\mathcal{C}(D) = D^\delta$ for some permutation δ of V.

Given two edge sets $A, B \subset E$, we define the *interface* $\mathcal{I}(A, B)$ to be the set of vertices incident with edges in both A and B, i.e. $\mathcal{I}(A, B) := \{v \in V : \langle v, a\rangle \in A, \langle v, b\rangle \in B\}$. Elements of $\mathcal{I}(A, B)$ are called *interface vertices.* Thus $\mathcal{I}(X, D - X)$ represents the interface of the complementary subgraphs X and $D - X$ of D.

We wish to 'fix' this interface whilst allowing portions supported by vertices 'inside' X and $D - X$ to vary up to isomorphism. For this purpose, we colour each vertex $v \in \mathcal{I}(X, D - X)$ with its label in the canonical isomorph of D, namely v^δ. This induces a coloring $c : V \to \{0, \dots, n\}$ of the vertex set of D: each vertex v takes colour v^δ if $v \in \mathcal{I}(X, D - X)$ and 0 (black) otherwise.

The colouring c translates into a partition of V by setting $V_i = \{v \in V : c(v) = i\}$. Observe that each V_i is either empty or a singleton cell, except perhaps the set V_0 of black vertices. Dropping all empty cells and relabelling the others in the same order, we obtain the partition $\pi = \pi(X, D - X, \delta)$ of the form $(V_0, \{c_1\}, \dots, \{c_k\})$, where $k = |\mathcal{I}(X, D - X)|$.

Theorem 3.2. *A sufficient condition for $K^X(D) = K^{X'}(D')$ is that*

(K1) *$\mathcal{C}(D) = \mathcal{C}(D')$,*
(K2) *$\mathcal{C}(X, \pi) = \mathcal{C}(X', \pi')$, and*
(K3) *$\mathcal{C}(D - X, \pi) = \mathcal{C}(D' - X', \pi')$.*

Before we embark on the proof of this result, we require the following

Lemma 3.3. If conditions (K1) - (K3) of Theorem 3.2 hold, then the partitions π and π' are compatible.

Proof. By construction, π and π' are compatible if the sets of black or, equivalently, interface vertices have the same cardinalities. Therefore, it is enough to show that $|\mathcal{I}(X, D-X)| = |\mathcal{I}(X', D'-X')|$, i.e. edges of X and $D-X$ coincide on the same number of vertices as those of X' and $D'-X'$.

We require some more graph-theoretic vocabulary: Let $\Gamma_G(v) := \{w \in V : \langle v, w\rangle \in E\}$ the set of *neighbours* of a vertex $v \in V$, and define $d_G(v) = |\Gamma_G(v)|$ the *degree* of v. Define the *degree sequence* $\mathbf{d}_G := (d_G(v) : v \in V)$ with degrees listed in increasing order.

Let $\mathbf{r}$, $\mathbf{s}$ and $\mathbf{t}$ denote the degree sequences $\mathbf{d}_D$, $\mathbf{d}_X$ and $\mathbf{d}_{D-X}$, respectively. At every vertex $v \in V$, we have $d_D(v) = d_X(v) + d_{D-X}(v)$. Equivalently, there exist permutations α, β of $[n]$ such that for all $i \in [n], r_i = s_j + t_k$ with $j = i^\alpha, k = i^\beta$. Thus, the i-th smallest degree of D corresponds to the unique pair (s_j, t_k).

A vertex v lies in the interface $\mathcal{I}(X, D-X)$ if and only if both summands $d_X(v)$ and $d_{D-X}(v)$ are nonzero, i.e. the corresponding pair (s,t) has $s, t \neq 0$. Let a, b and c denote the number of leading zeros in the degree sequences $\mathbf{r}$, $\mathbf{s}$ and $\mathbf{t}$, respectively. Notice that for $i \in [a]$, we have $r_i = 0 \Rightarrow s_j = t_k = 0$, so there are precisely a pairs $(0,0)$. This shows that $b + c - 2a$ zeros are paired with non-zero partners, leaving $n - a - (b + c - 2a)$ pairs (s,t) with $s, t \neq 0$. Hence $|\mathcal{I}(X, D-X)| = n + a - b - c$.

Finally, by assumption (K1) and using property (C1) of the canonical label $\mathcal{C}$, we have $D \sim \mathcal{C}(D) = \mathcal{C}(D') \sim D'$, hence $\mathbf{d}_D = \mathbf{d}_{D'}$. By (K2), (K3), analogous statements hold for $X^{(\prime)}$ and $D^{(\prime)} - X^{(\prime)}$, so that we obtain the same values a, b, c for both primed and unprimed edge sets. □

Proof of Theorem 3.2. Assume that conditions (K1) - (K3) hold. To show that $K^X(D) = K^{X'}(D')$, it is enough to construct a bijection $\phi : \mathcal{F}^X(D) \leftrightarrow \mathcal{F}^{X'}(D')$.

By Lemma 3.3, π and π' are compatible, so we may apply Theorem 3.1 to $(G_i, \pi_i) = (X^{(\prime)}, \pi^{(\prime)})$ and $(D^{(\prime)} - X^{(\prime)}, \pi^{(\prime)})$. This shows that there exist permutations ρ, σ of V such that

$$\begin{array}{lllllll} X' & = & X^\rho & \qquad & \pi' & = & \pi^\rho \\ D' - X' & = & (D-X)^\sigma & \qquad & \pi' & = & \pi^\sigma \end{array}$$

Let $\pi = (V_0, \{c_1\}, \dots, \{c_k\})$. Then $\pi^\rho = \pi^\sigma$ implies $c_i^\rho = c_i^\sigma$ for $i \in [k]$. This shows that ρ and σ agree on the interface $\mathcal{I}(X, D-X)$ $(\star)$.

We claim that a bijection $\phi : \mathcal{F}^X(D) \leftrightarrow \mathcal{F}^{X'}(D')$ is given by $F = X + A \mapsto F' = X^\rho + A^\sigma$. This claim is easily verified as follows: ϕ is well-defined by $(\star)$; F' contains $X' = X^\rho$; F' is acyclic (X', A' are separately acyclic as images of acyclic X, A, and jointly acyclic due to $(\star)$); and ϕ is bijective with inverse obtained by substituting ρ^{-1}, σ^{-1} for ρ, σ, respectively. □

As a direct consequence of Theorem 3.2, we obtain the following easy-to-compute and efficient index function $i(G, X, Y)$.

Corollary 3.4. *Let (G, X, Y) be an instance of conjecture 2.1, $D = G - Y$ and $\pi = \pi(X, D-X, \delta)$. Let i be given by the triple*

$$i(G, X, Y) = (\mathcal{C}(D), \mathcal{C}(X, \pi), \mathcal{C}(D - X, \pi)).$$

Then ${K^X}_Y(G)$ is equal to K_ι for all (G, X, Y) with common index $i(G, X, Y) = \iota$.

We make an important remark regarding efficiency: In the above construction of the partition $\pi = \pi(X, D-X, \delta)$, it is not relevant for the validity of Theorem 3.2 to colour each vertex $v \in \mathcal{I}(X, D-X)$ precisely with its label in the canonical isomorph of D, namely v^δ. In principle, we could have chosen arbitrary distinct colours, thus removing the dependence of π on δ.

However, these colours determine the order in which interface vertices appear as singleton cells in the induced partition π. This order is in turn significant for the canonical label $\mathcal{C}$. By consistently choosing the colour v^δ we maintain the same order across the whole class of isomorphs of D. This guarantees that i subdivides the space of parameters (G, X, Y) into possibly large classes with common index $i(G, X, Y) = \iota$ and, hence, the number of evaluations of ${K^X}_Y(G)$ required.

One drawback of the partition π depending on δ is that we have to recompute *all* components of i whenever a new graph D is considered and the corresponding δ affects π, even when X or $D - X$ have not changed themselves. Whether performance is improved overall will therefore depend essentially on the fast evaluation of $\mathcal{C}$ (see Chapter 3).

3.6. **Enumerating and verifying conjecture instances** (G, e, f)**.** In order to verify Conjecture 2.1 numerically for graphs of order n, we have to enumerate and verify inequality 2.2 for all possible choices of $G \subset K_n$ and $e \neq f \in E$. We call such a choice *conjecture instance* (G, e, f).

As seen in section 3.1, the number of conjecture instances grows extremely rapidly in n. Fortunately, there are large classes of equivalent conjecture instances. Here, (G, e, f) and (G', e', f') are called *equivalent* if inequality 2.1 is invariant under interchange of both sets of parameters, i.e. $K_1K_2 = K_1'K_2'$ and $K_3K_4 = K_3'K_4'$. We denote this by $(G, e, f) \sim (G', e', f')$.

We propose two measures to reduce complexity: (1) to choose an enumeration that *ab initio* avoids many equivalent instances; and (2) to skip instances identified as equivalent to earlier ones using index function i from section 3.3.

3.6.1. *Efficient enumeration.* Choosing G first confronts us with the hard problem of deciding which choices of $e \neq f \in E$ will produce distinct conjecture instances. To avoid this, we begin by fixing the only two non-isomorphic choices of e, f (connected or not); without loss of generality $E_1 = \{e_1, f_1\} := \{\langle 1,2\rangle, \langle 1,3\rangle\}$ and $E_2 = \{e_2, f_2\} = \{\langle 1,2\rangle, \langle 3,4\rangle\}$. We then examine the instances (G, e_i, f_i) for graphs G containing either (or both) E_i. This is sufficient: given any instance (G, e, f) with $e = \langle a, b\rangle, f = \langle c, d\rangle$ or $e = \langle a, b\rangle, f = \langle b, c\rangle$, there exists a permutation γ mapping a, b, c, d to $1, 2, 3, 4$, respectively. Hence $(G, e, f) \sim (G^\gamma, e_i, f_i)$ for $i = 1$ or 2, an instance of the form above.

Let $\mathcal{G}_s = \{G \in \mathcal{G} : |G| = s, E_i \subset G \text{ some } i = 1, 2\}$ be the layer of supergraphs G of E_i of size s, for $s = 2, \dots, N$. As in section 3.2, our algorithm to construct $\mathcal{G}_s$ is best described inductively: Begin with $\mathcal{G}_2 = \{E_1, E_2\}$. Having found $\mathcal{G}_s$, construct $\mathcal{G}_{s+1}$ by iterating through all $G \in \mathcal{G}_s$; for each G, enumerate all $N = \binom{n}{2}$ possible edges $e_i = \langle v_i, w_i\rangle \in E(K_n)$ in lex order (i.e. $e_i <_e e_j$ if $v_i < v_j$ or $v_i = v_j, w_i < w_j$). If e_i is not already present in G, add $G + e_i$ to $\mathcal{G}_{s+1}$.

We incorporate a simple optimization to avoid some apparently equivalent instances from the start; it is most effective on low layers $\mathcal{G}_s$ with $s \ll N$, when the order n is large. Let $v^\star$ be the smallest isolated vertex of G. Then, by construction, all higher vertices $w \in \{v^\star, \dots, n\}$ are also isolated. It is easy to show that a conjecture instance $(G + \langle u, w\rangle, e_i, f_i)$ will be equivalent to $(G + \langle u, v^\star\rangle, e_i, f_i)$ for all $u < v^\star < w$, and $(G + \langle w, w'\rangle, e_i, f_i) \sim (G + \langle v^\star, v^\star + 1\rangle, e_i, f_i)$ for all $v^\star \leq w < w'$. We may therefore restrict our attention to edges e_i with $v_i, w_i \leq v^\star$ (with the exception of the last edge to be added, namely $\langle v^\star, v^\star + 1\rangle$).

3.6.2. *Skipping equivalent instances.* Having constructed $\mathcal{G}_s$, we must verify all instances (G, e_i, f_i) for $G \in \mathcal{G}_s$ and $E_i \subset G$. However, the cardinality of $\mathcal{G}_s$ grows prohibitively fast, like $O(\binom{N}{s})$, which in any computer implementation is bound to quickly exhaust the available resources. To address this critical issue, we require a criterion to identify equivalent instances. This will enable us to skip many redundant checks of inequality 2.2, and to reduce $\mathcal{G}_s$ by deleting graphs G for which both instances (G, e_1, f_1) and (G, e_2, f_2) are equivalent to earlier ones. We use the following proposition:

Proposition 3.5. *(Equivalence of conjecture instances)*

Let $G, G' \in \mathcal{G}$ and $e \neq f \in E \cap E'$. If conditions (K1) - (K3) of Theorem 3.2 hold for $X^{(\prime)} = \{e, f\}$ and $D^{(\prime)} = G^{(\prime)}$, then $(G, e, f) \sim (G', e, f)$.

Proof. Recall that by definition, $(G, e, f) \sim (G', e, f)$ if $K_1K_2 = K'_1K'_2$ and $K_3K_4 = K'_3K'_4$. By Theorem 3.2, the assumptions immediately imply $K_1 = K'_1$, so all we have to show is (1) $K_2 = K'_2$ and (2) $K_3K_4 = K'_3K'_4$.

Let ρ, σ as in the proof of Theorem 3.2. In this context, $X' = X^\rho$ reads $\{e, f\} = \{e, f\}^\rho$, so that ρ either fixes or swaps the edges e and f. To show (1), we construct a bijection $\phi : \mathcal{F}_{\{e,f\}}(G) \leftrightarrow \mathcal{F}_{\{e,f\}}(G')$ by mapping $F \mapsto F' = F^\sigma$.

Statement (2) follows from the observation that mapping $F = e + A \in \mathcal{F}^e{}_f(G)$ to $e^\rho + A^\sigma$ constitutes a bijection between $\mathcal{F}^e{}_f(G)$ and *either* $\mathcal{F}^e{}_f(G')$ *or* $\mathcal{F}^f{}_e(G')$, depending if ρ fixes or swaps e and f, and we have $K_3 = K'_3$ or $K_3 = K'_4$ accordingly. By an analogous argument, K_4 equals K'_4 or K'_3, respectively. This establishes (2). □

The main use of this result is the following: Assume that for some instance (G, e_i, f_i), K_1 is successfully retrieved from a database of values $K^X(D)$ using index $i(G, E_i, \emptyset) = \iota$, say. Then this value springs from an earlier computation for parameters (G', X', Y') with same index $i(G', X', Y') = \iota$. We deduce that $X' \sim E_i$, and so $X' = E_i$, since this is the only such choice that we have ever used. Similarly, $Y' \sim \emptyset \Rightarrow Y' = \emptyset$. Applying Proposition 3.5 with $e = e_i, f = f_i$, we conclude that (G, e_i, f_i) is equivalent to (G', e_i, f_i). Since inequality 2.2 has already been checked for the latter instance, we are safe to skip the former now.

More importantly, we can be sure that for any supergraph H of G, there is a supergraph H' of G' with $(H, e_i, f_i) \sim (H', e_i, f_i)$. Hence, if both instances (G, e_1, f_1) and (G, e_2, f_2) were identified as equivalent to earlier ones, we no longer need to consider descendants of G when constructing $\mathcal{G}_t$ for $t > s$; we may safely delete G from $\mathcal{G}_s$ altogether. (If only one of these two instances, (G, e_j, f_j) say, was found equivalent to an earlier one, we must leave G in $\mathcal{G}_s$; at least we know in advance that we may skip later instances (H, e_j, f_j) for all descendants H of G.)

3.7. **Recursive formula for $K^X(D)$.** Despite our efforts, a large number of explicit computations of $K^X(D)$ remain inevitable. As remarked above, the cost of computing $K^X(D)$ explicitly grows exponentially in the size of X and D. It is therefore preferable to expand this quantity into a sum over $K^{X'}(D')$ for larger sets of constraints X' and smaller graphs D', which are cheaper to compute, or may even be available from earlier computations. We propose a method based on the following

Proposition 3.6. *For all $d \in D - X$,*

$$K^X(D) = K^X(D - d) + K^{X+d}(D). \tag{3.1}$$

Proof. This identity is immediate from the definition of $K^X{}_Y(G)$: Partition $\mathcal{F}^X(D) = \mathcal{F}_1 \bigsqcup \mathcal{F}_2$ by putting F into $\mathcal{F}_1$ if it does not contain d, and into $\mathcal{F}_2$ otherwise. Observe that we may bijectively identify $\mathcal{F}_1$ with $\mathcal{F}^X(D - d)$ and $\mathcal{F}_2$ with $\mathcal{F}^{X+d}(D)$. The result follows by taking cardinalities. □

Repeatedly applying proposition 3.1 to the last summand in equation 3.1 yields a recursive formula for $K^X(D)$, as follows. By assumption, D is of the form $D = X + \{d_1, \dots, d_k\}$ with $k = |D| - |X|$. We define the increasing sequence of constraints $X_j = X + \{d_1, \dots, d_{j-1}\}$ (assuming $X_1 = X$) and the *reduced graphs* $D_j = D - d_j$ for $j \in [k]$. The corresponding values $K^{X_j}(D_j)$ are called *minors* of $K^X(D)$. Then

Corollary 3.7.

$$K^X(D) = \sum_{j=1}^{k} K^{X_j}(D_j) + K^D(D) \tag{3.2}$$

We make several observations:

(1) Equation 2.2 shows how to obtain one value of K by aggregating several others. Despite appearance, this saves runtime for the following reasons: (a.) If X is a standard constraint, then the first summand on the rhs, $K^{X_1}(D_1) = K^X(D - d_1)$, has already been computed on the layer below D. (b.) $K^D(D)$ is simply 0 or 1, depending if D is itself acyclic. (c.) If we buffer not only the result $K^X(D)$, but all the minors $K^{X_j}(D_j)$ obtained on the way, we may expect that over time, some minors can be retrieved from earlier calculations using index function i.

(2) To increase the probability of minors being available from earlier computations, we buffer the intermediate results $K^{X_j}(D)$: identity 3.1 yields the recursive expression

$$\begin{aligned} K^{X_j}(D) &= K^{X_j}(D_j + d_j) \\ &= K^{X_j}(D_j) + K^{X_{j+1}}(D) \end{aligned}$$

Observe that $X_{k+1} = D$. Hence, for $j = k$, the rhs is just the sum of the last two terms in 3.2. We may then compute $K^{X_j}(D)$ for $j = k-1, k-2, \ldots, 1$ in turn, at each stage adding the j-th term of 3.2 on to a running total. When $j = 1$ is reached, we can read off the overall result $K^X(D) = K^{X_1}(D)$.

(3) To further accelerate the evaluation of 3.2, observe that $K^X(D) = 0$ whenever X contains a cycle. Since the sequence X_j is increasing, if some X_j contains a cycle, then so does $X_{j'}$ for all $j' \geq j$. Let $j^\star$ be the largest j with X_j acyclic. If $j^\star < k$, then 3.2 simplifies to

$$K^X(D) = \sum_{j=1}^{j^\star} K^{X_j}(D_j)$$

To determine $j^\star$ algorithmically when X is a standard constraint, we start from the edge set $X_1 = X$ (which is acyclic as $|X| \leq 2$) and add edges whilst keeping track of the components of X_j. So long as addition of an edge merges disjoint components, X_j remains acyclic. Once this fails to hold, we have gone past, and hence identified, $j^\star$.

4. Implementation

4.1. **Overview.** In this Chapter, we present our implementation of the above algorithm in C++, available at `http://www.statslab.cam.ac.uk/~grg/rsf.zip`. The source code is fully commented and compiles with both gcc version 3.2 and Visual C++ 6.0. We have chosen C++ for a variety of reasons: First, for speed and for the benefits of object-oriented design. Second, to profit from the powerful Standard Template Library (STL) when handling advanced data structures such as vectors, sets and maps. Third, to be able to incorporate the excellent C-program *nauty* by Brendan D. McKay, needed to obtain canonically-labelled isomorphs of (partitioned) graphs.

In section 4.2, we document our representations (classes) of the various mathematical objects under discussion, including permutations, edges, components, forests, and general graphs. We further describe the data structures used to manage sets of forests and graphs such as $\mathcal{F}_s$ and $\mathcal{G}_s$ (see 3.2 and 3.6), and how we realized the database of values $K^X(D)$, indexed by i (see 3.3 - 3.5).

Section 4.3 is devoted to our implementation of the algorithm itself. We discuss programming issues relevant for the three main components: explicit computation of ${K^X}_Y(G)$ (see 3.2), enumeration and verification of conjecture instances (G, e, f) (see 3.6), and the recursive formula for $K^X(D)$ from section 3.7.

The findings of our investigation are summarized in section 4.4. Notably, we confirm Conjecture 2.1 for all graphs G of order $n \leq 8$, present distributional statistics and evaluate the efficacy (and limitations) of the various optimizations used.

4.2. **Classes and data structures.**

4.2.1. *Class architecture.* The class CPermutation stores the image of some permutation γ of $[n]$ as vector and possesses methods permute and inverse to compute x^γ and γ^{-1}. The class CEdge represents undirected edges $e = \langle v, w \rangle$. For reasons that will emerge later, it supports operators ==, < to test for equality and precedence, where the latter is defined by $e <_e e' :\Leftrightarrow v < v'$ if $v \neq v'$, and $w < w'$ otherwise.

The classes CVertices and CEdges wrap vectors _V, _E of types int, CEdge, listing the elements of a vertex set V and an edge set E, respectively. These classes also carry operators for lex ordering. Individual or sets of elements can be added and removed, alternatively at the end of the list (using push and pop) or preserving lex order (using add and remove). Both classes possess a method combine that merges two ordered lists of elements into one. Finally, CEdges can evaluate $v^\star$, the smallest isolated vertex of the graph $G = (\mathbb{N}, E)$ (see subsection 3.6.1).

Components C are connected subgraphs (V, E) of a graph $G = (V', E')$, with $V \subset V'$ and $E \subset E'$. In our implementation, they are represented by the class CComponent, which contains objects _V, _E of type CVertices, CEdges, respectively. There are three constructors: one to instantiate the singleton $(\{v\}, \emptyset)$; another to construct a component $(\{v, w\}, \{e\})$ from a single edge $e = \langle v, w \rangle$; a third to merge disjoint sub-components C_1 and C_2 by adding an edge e assumed incident with both. A method add_edge is provided to augment C by some incident edge $f \notin C$. A total order is imposed on components by giving precedence to components with fewer edges. Formally,

$$(V,E) <_C (V',E') \quad :\Leftrightarrow \quad \begin{array}{ll} |E| < |E'| & \text{if } |E| \neq |E'| \\ v < v' & \text{if both are isolated vertices,} \quad V^{(\prime)} = \{v^{(\prime)}\}, E^{(\prime)} = \emptyset \\ E <_e E' & \text{under lex order on edge lists, otherwise} \end{array}$$

Forests $F = \bigsqcup_{i=1}^{k} C_i = (C_1, \ldots, C_k)$ are modelled by the class CForest. Its member vector<CComponent*> _C stores pointers to the objects representing C_i, so they may be centrally administered and shared among multiple forests. Forests may be constructed from either a set of components (the notion of a set will be made precise below) or another forest, replacing two old components with a new one. Given forests F, F', we define their order by giving precedence to forests with fewer components:

$$F <_F F' \quad :\Leftrightarrow \quad \begin{array}{ll} k < k' & \text{if } k \neq k' \\ (C_1, \ldots, C_k) <_C (C'_1, \ldots, C'_k) & \text{under lex order on component lists, otherwise} \end{array}$$

The class CGraph plays a pivotal role for the present implementation. When instantiated, it represents a graph $G = ([n], E)$ of order n. Rather than storing E, we record Matrix _A[1], the adjacency matrix of G. This allows simple and fast book-keeping (e.g. add_edge(s), remove_edge(s) and has_edge), but renders the class inappropriate for large sets of graphs such as $\mathcal{G}_s$. Instances may be constructed from a specified edge set E, or randomly sampled fixing the desired size s or probability p for edges to appear independently at random. When not instantiated, the class CGraph acts as interface, whose static methods implement components of our algorithm not associated with an individual graph object (see section 4.3 below).

To store graphs more economically, we introduce a dedicated class CGraphcode. Employing McKay's highly optimised technique, the adjacency matrix A_{ij} is encoded in a vector r_j of unsigned longs. This appears as member Graphcode _code, where Graphcode is a placeholder for vector<unsigned long>. For comparatively small graphs like those considered in this study, one r_j suffices to store the j-th row of A. We impose a (somewhat artificial) ordering on instances G of class CGraphcode by setting $G <_G G' :\Leftrightarrow (r_1, \ldots, r_n) < (r'_1, \ldots, r'_n)$ under lex order. A subclass CGraphdata is derived from CGraphcode to store further information pertinent to the encoded graph, as will be described below.

Remark: Objects of all classes are able to produce an easy-to-read textual representation of their current data. For CEdge, CEdges and CVertices, there exists an operator << to write the object to a given ostream&. All other classes implement a method report.

[1] The type Matrix is short for vector< vector<int> >.

4.2.2. *Sets and maps.* Apart from `vectors`, we make extensive use of the template classes `set<`T`,`$<_T$`>` and `map<`S`,`T`>` provided by STL. A `set<`T`,`$<_T$`>` is a sorted associative container that can store and retrieve unique objects of some type T, ordered by $<_T$. Internally, `sets` use balanced binary trees to ensure that the times for insertion, random access and deletion grow logarithmically in the size of the set. Elements are stored in maps as pairs in which each unique key (of type S) has an associated value (of type T).

It is easily seen that the class `set<`T`,`$<_T$`>` meets all requirements to represent sets of forests and graphs such as $\mathcal{F}_s$ and $\mathcal{G}_s$ from sections 3.2 and 3.6. Hence we define types `Forests` and `Graphs` by `set<CForest, less_F>` and `set<CGraphdata, less_G>`, respectively. `less_F`, `less_G` are function objects implementing the total orders $<_F, <_G$. We note in passing one further situation where this approach is fruitful. Recall that `CForest`-objects do not keep local copies of, but merely pointers to, their constituent components. These are administrated in a central repository, for which we use the type `Components` := `set<CComponent, less_C>`.

The subtle question remains how to buffer $K^X{}_Y(G) = K^X(D)$ in a database indexed by the triple $i = (\mathcal{C}(D), \mathcal{C}(X,\pi), \mathcal{C}(D-X,\pi))$. It is crucial to retain these values precisely as long as they might be queried. It turns out that a simple and effective criterion for this is given by the size of D. Assume $G \in \mathcal{G}_s$ and note that for standard constraints $Y \subset \{e,f\}$, we have $s-2 \leq |D| \leq s$. Therefore, when we examine the layer of graphs $\mathcal{G}_s$, D and hence $\mathcal{C}(D)$ are confined to the last three layers. When we use the recursive formula from 3.7, we have to make it four, since all reduced graphs $D_j = D - d_j$ are of size $|D|-1$.

In consequence, we need to retain $K^X(D)$ only while D lies on $\mathcal{G}_t$ for $t = s, \dots, s-3$. In our implementation, these are represented by the array `Graphs graph_DB[4]`, aka graph database, such that `graph_DB[t%4]` $= \mathcal{G}_t$. We separate the first component $\mathcal{C}(D)$ from the triple i to serve as 'main category'. In order to retrieve a value $K^X(D)$, we first dial the `CGraphdata` object representing $\mathcal{C}(D)$ from the appropriate layer of the graph database.

At this stage, the template class `map<`S`, `T`>` is used to map the latter two components of i, namely the pair $(\mathcal{C}(X,\pi), \mathcal{C}(D-X,\pi))$, to the desired result $K^X(D)$. The former is of type S = `Key` := `pair<Graphcode, Graphcode>`. For the latter it would be sufficient to choose type `unsigned long`. Nonetheless, we define a dedicated class T = `CKdata` to be able to compile certain usage statistics. In conclusion, every `CGraphdata`-object has a member `map<Key, CKdata> _K_DB` to perform the look-up of values $K^X(D)$.

Two final remarks: (1) This architecture makes it trivial, upon completion of a layer $\mathcal{G}_s$, to discard all obsolete values $K^X(D)$: simply clear the set `graph_DB[(s-3)%4]` corresponding to $\mathcal{G}_{s-3}$. It can then be reused for $\mathcal{G}_{s+1}$. (2) Observe that among all graphs on a given layer $\mathcal{G}_t$, only canonically labelled ones need to be possess a map `_K_DB`. For all others, indeed the vast majority, a plain object of class `CGraphcode` would be enough, saving a lot of memory. This is incorporated by imposing that the graph database `graph_DB` only stores canonical labelled graphs $\mathcal{C}(G)$. In return, the `CGraphdata`-object representing $\mathcal{C}(G)$ contains a `vector<CGraphcode> _G` which lists all other $G \in \mathcal{G}_s$ with $G \sim \mathcal{C}(G)$.

4.3. **Implementing our algorithm.** In this section, we discuss the methods implementing our algorithm as described in Chapter 3. To provide a tidy interface, they are all bundled in the class `CGraph`, although some are static and do not operate on a particular graph object.

4.3.1. *Explicit computation of $K^X{}_Y(G)$.* The ad-hoc approach from section 3.2 is implemented in the method `count_forests` in a straightforward way. First, the auxiliary function `construct_forest` converts the edge set X into a set of components $\{C_i\}$, from which we instantiate the smallest valid forest. An array `Forests forest_DB[2]` serves to store the current layer $\mathcal{F}_s$ and the next layer $\mathcal{F}_{s+1}$ being constructed, while the set `Components component_DB` handles the component repository.

Iterating through all forests $F \in \mathcal{F}_s$, all pairs of components $C_i \neq C_j$ of F, and all edges e between them, we keep constructing (potentially) new components $C' = C_i + C_j + e$ and (replacing C_i, C_j by C' in F) new forests $F' \in \mathcal{F}_{s+1}$. Whether C' and F' are actually new is not a concern: this is automatically taken care of by the sets `component_DB` and `forest_DB[(s+1)%2]`. We keep a running total of distinct forests encountered to give the final result.

4.3.2. *Enumerating and verifying conjecture instances* (G, e, f). The most complex task of our program is distributed across the following three main functions:

(1) `check_conjecture_up_to_K_n`. This public method is called by the user specifying n. We begin by defining the edge sets E_i (using the notation of section 3.6.1) and standard constraints X_{ik}, Y_{ik} with $i \in [2], k \in [4]$, and place E_1, E_2 into `graph_DB[2]`. Iterating through all layers from $s = 2$ to $N - 1$, we enumerate all canonically-labelled graphs $\mathcal{C}(G)$ stored in `graph_DB[s%4]` and all $G \sim \mathcal{C}(G)$ listed in the `CGraphdata`-object representing $\mathcal{C}(G)$. For each G, we call...

(2) `analyse_supergraphs`. Iterating through all edges $e \notin E$ in lex order up to $\langle v^\star, v^\star + 1\rangle$, we form new graphs $G' := G + e$ and retrieve/insert their canonically-labelled isomorphs $\mathcal{C}(G')$ from/into `graph_DB`. We thus proceed to checking any new conjecture instances brought about by G'. Two conditions determine for which i the instance (G', e_i, f_i) may be skipped at once: (1) if $E_i \not\subset G$ or (2) if, for some ancestor $H \subset G$, (H, e_i, f_i) was found equivalent to another instance. If these do not apply, we pass (G', e_i, f_i) on to the method...

(3) `check_instance`. From this method, the retrieval/computation of the four values K_k (using `obtain_K`) is coordinated. If K_1 can be retrieved from an earlier computation, an instance equivalent to (G', e_i, f_i) has been checked before (as shown in section 3.6), and we may return without delay. Otherwise, we obtain the remaining K_k and check $K_1K_2 \leq K_3K_4$ (using `check_inequality`). In the former case, upon return to `analyse_supergraphs`, we record that (H, e_i, f_i) may be skipped for all decendants $H \supset G'$. Depending if we arrive at this conclusion for both $i = 1, 2$, we will or will not include G' in $\mathcal{G}_{s+1}$.

We briefly mention three auxiliary methods used throughout this process:

- `compute_isomorph` provides an interface to the *nauty*-library, and serves a double purpose: (1) Given a partitioned graph (G, π) with π based on an interface $\mathcal{I}$ and a permutation δ, it computes $\mathcal{C}(G, \pi)$. Here, $\mathcal{I}$ is represented by `vector<bool> I`, with `I[v-1]` true iff $v \in \mathcal{I}$. (2) Given an (unpartitioned) graph G, it provides $\mathcal{C}(G)$ as well as $\delta : G \to \mathcal{C}(G)$.
- `obtain_CG`. Given a `CGraph` G, this routine first uses `compute_isomorph` to create a `CGraphdata`-object representing $\mathcal{C}(G)$. Taking this as key, it attempts to retrieve an equivalent object (equal under the operator `==`) from `graph_DB`. If successful, the key is discarded, otherwise it automatically becomes a new entry. Finally, $\mathcal{C}(G)$ is returned.
- `obtain_K`. Given G, X, Y, this method co-ordinates the retrieval or, if neccesary, evaluation of ${K^X}_Y(G)$. It prepares the edge sets $D, D - X$ and obtains $\mathcal{C}(D), \delta$ (using `compute_isomorph`) as well as $\mathcal{I}(X, D - X)$ (using `mark_I`). It queries `graph_DB` for $\mathcal{C}(D)$ and then tries to read ${K^X}_Y(G)$ from the corresponding `_K_DB`-object. Failing that, it computes the value afresh using `compute_K` (see below).

4.3.3. *Recursive formula for* $K^X(D)$. Finally we present the method `compute_K` implementing formula 3.2 for $K^X(D)$, which is straightforward using the machinery developed above. First, `compute_j_max` is called to determine the upper bound $j^\star$ of the summation (for this purpose, X_j is iteratively enlarged until a cycle forms, almost as in `construct_forest`). Then `obtain_K_minor` provides the minors $K^{X_j}(D_j)$, while `store_K_tot` buffers the intermediate results $K^{X_j}(D)$.

4.4. **Results and statistics.**

5. Conclusion