The generation of connected random graphs is non-trivial and important to many applications. In particular, it is not easy to sample a connected random graph from the space $\mathcal{G}$ of all connected graphs with uniformity; i.e. without a bias for graphs of a special kind. This entry contains two algorithms for sampling random connected graphs that do not overcome a special kind of bias. A third algorithm which overcomes all bias, though at great computational expense, is presented.

# 1 Bottom-up approach

Before proceeding, quick definitions.

- Let $\mathcal{T}_n$ the set of all trees of $n$ vertices, $\mathcal{G}_n$ the set of all graphs with $n$ vertices. We shall assume the vertices of these graphs are labeled $1, \ldots, n$.
- For any $T \in \mathcal{T}_n$, we define $\mathcal{U}_T := \{G \in \mathcal{G}_n : T \subseteq G\}$ and refer to it as *the universe* of $T$.
- Let $\Gamma(n) = \{\{x, y\} : x, y \in \{1, \ldots, n\}\}$.

In general, whenever I write of a tree $T$, I mean an arbitrary $T \in \mathcal{T}_n$; and whenever I write of a graph $G$, I mean an arbitrary $G \in \mathcal{G}_n$.

—

There are several properties of trees which make them attractive for graph generation. On one hand, any finite $G$ contains a finite number of spanning trees $T \subseteq G$; on the other, any tree spans a finite number of graphs. It is trivial to observe that $T \subseteq G$ if and only if there is some $S \in \Gamma(n)$ s.t. $E(G) = E(T) \cup S$. We can make use of this special relationship between $\Gamma(n)$ and $\mathcal{U}_T$ to produce random connected graphs.

Each tree is perfectly identified by its Prüfer sequence. Then, for a fixed $n$, the language $\{1, \ldots, n\}^{n-2}$ indexes a family of functions $\mathcal{F}$ defined as:

$$\begin{aligned} \mathcal{F}(w) &: \mathcal{U}_{T_w} \to \Gamma(n) \\ \mathcal{F}(w)(G) &= E(G) - E(T_w) \end{aligned}$$

where $T_w$ is the tree corresponding to the Prüfer sequence $w$. We shall use $\mathcal{F}_w$ to abbreviate $\mathcal{F}(w)$.

Since there is a perfect correspondence between any given $G \in \mathcal{U}_T$ and the set of edges which, if aggregated to $T$, produce $G$, $\mathcal{F}_w$ is a bijection. (This is easy to

prove.) This entails that any $G \in \mathcal{G}_n$ which may be spanned from $T_w$ corresponds to a unique set in $\Gamma(n)$—namely, that which extends $T$ into $G$.

Thus, we have established a series of generational relations. A Prüfer sequence maps to a unique tree $T$, the tree maps to a universe of connected graphs $\mathcal{U}_T$, and each graph in this universe is perfectly determined by a set in $\Gamma(n)$. This inspires the following effective procedure for generating random connected graphs of $n$ vertices.

*(1)* Generate randomly $p = p_1 \dots p_{n-2} \in \Sigma^{n-2}$.

*(2)* Span the tree $T = (V, E)$ of the Prüfer sequence $p$.

*(3)* Let $k \in_R \left\{0, \dots, \frac{n(n-1)}{2}\right\}$.

*(4)* Let $\ell_1, \dots, \ell_k \in_R \Gamma(n) - E(T)$, all distinct.

*(5)* Let $E = E \cup \{\ell_1, \dots, \ell_k\}$

Because all trees of $n$ vertices correspond to a sequence, all tres can be sampled. And all connected graphs can be derived from the set of all spanning trees. Then this procedure generates all graphes in $\mathcal{G}_n$.

The question is whether it is equally likely to generate any two graphs in $\mathcal{G}_n$. It is obvious that it is equally likely to generate any tree. And the probability that a given graph is generated depends entirely on the number of spanning trees it contains. Not all graphs have the same number of spanning trees. $\therefore$ It is more likely to generate a graph with many spanning trees than a graph with few spanning trees.

The second algorithm extends the input from only $n$, the number of vertices, to $m$, the number of edges. Thus, it specifies the problem further into the question of how to generate denser or sparser connected graphs of $n$ edges. The effective procedure to do this is:

*(1)* Generate randomly $p = p_1 \dots p_{n-2} \in \Sigma^{n-2}$.

*(2)* Span the tree $T = (V, E)$ of the Prüfer sequence $p$.

*(3)* If $V = \{v_1, \dots, v_n\}$, list all its non-neighbours; i.e. generate $\Gamma^c(v_1), \dots, \Gamma^c(v_n)$.

*(4)* Sample a random, non-saturated vertex $v \in V$, and sample a random vertex $w$ from $\Gamma^c(v)$.

*(5)* Add $\{v, w\}$ to $E$. Remove $v$ from the list of non-neighbours of $w$, and $w$ from the list of non-neighbours of $v$.

*(6)* If $|E| = m$, finish. Otherwise go to *4*.

It is clear that the procedure always finishes, and since the tree is connected the generated graph is connected. A pseudo-code algorithm may look as follows.

**Input:** $n, m$
$(V, E) =$ **gen_random_tree**$(n)$
$S = \left[ \Gamma^c(v_1), \ldots, \Gamma^c(v_n) \right]$ {Non-neighbours}
$C = \left[ |S[1]|, \ldots, |S|[n]| \right]$ {Cardinalities}
$V = [1, \ldots, n]$ {Non-saturated vertices }
$n' = n$
**while** $|E(T)| < m$ **do**
    $i :=$ **random**$(1, n')$
    $v := V[i]$
    **if** $(d(v) == n - 1)$ **do**
        **delete_at**$(V, i)$
        $n' := n' - 1$
    **else**
        $j =$ **random**$(1, C[v])$
        $w := S[v][j]$
        $E(T) := E(T) \cup \{v, w\}$
        $C[v] := C[v] - 1$
        **delete_at**$(S[v], j)$
        **delete_element**$(S[w], v)$
    **fi**
**od**
**return** $T$

Generating a tree from a random Prüfer sequence is $O(n^2)$. Listing all the non-neighbours is also $O(n^2)$. Within the while loop there are simply index manipulations, so the complexity of the loop is $\varphi \times O(1) = \varphi$, with $\varphi$ the complexity of the number of iterations.

All iterations add an edge except those where a saturated vertex is chosen. A saturated vertex may be chosen at most once. $\therefore$ There are $O(n)$ iterations where a saturated vertex is chosen. Since the rest of the iterations add an edge, their

number is fixed: $m - (n-1)$, where $(n-1)$ is the number of edges in the spanned tree. $\therefore$ There are exactly $m - n + 1$ edge-adding iterations.

$\therefore \varphi = O(n) + O(m - n + 1) = O(n) + O(m) = O(n^2)$

$\therefore$ The complexity of the algorithm is $O(n^2) + O(n^2) = O(n^2)$.