Chapter 1

RNC Algorithms for the Uniform Generation of Combinatorial Structures

Michele Zito, Ida Pu, Martyn Amos and Alan Gibbons*
University of Warwick, Coventry CV4 7AL, UK

Abstract

We describe several RNC algorithms for generating graphs and subgraphs uniformly at random. For example, unlabelled undirected graphs are generated in $O(\lg^3 n)$ time using $O\left(\frac{\varepsilon n^2}{\lg^3 n}\right)$ processors if their number is known in advance and in $O(\lg n)$ time using $O\left(\frac{\varepsilon n^2}{\lg n}\right)$ processors otherwise. In both cases the error probability is the inverse of a polynomial in ε. Thus ε may be chosen to trade-off processors for error probability. Also, for an arbitrary graph, we describe RNC algorithms for the uniform generation of its subgraphs that are either non-simple paths or spanning trees. The work measure for the subgraph algorithms is essentially determined by the transitive closure bottleneck. As for sequential algorithms, the general notion of constructing generators from counters also applies to parallel algorithms although this approach is not employed by all the algorithms of this paper.

1 Introduction.

This paper is concerned with problems of generating combinatorial structures uniformly at random (u.a.r.) by means of efficient parallel algorithms. Our model of computation is the well-known (see [11], for example) *parallel random access machine* (PRAM) augmented with a facility to generate random numbers in $[0, 1]$. By efficient algorithms we mean algorithms that run in polylogarithmic time in the problem size using a polynomial number of processors. Such problems define the class NC, or specifically RNC if the facility for generating random numbers is employed. Our algorithms generally run on the exclusive-read exclusive-write variant of the PRAM (the EREW PRAM).

Our domain of discourse is graph theory. Very little seems to be known about the parallel complexity of generation problems. With the exception of [5] the authors are not aware of any other graph theoretic parallel uniform generation algorithm. In the spirit of work in sequential algorithms, the general notion of constructing generators from counters may apply to parallel algorithms as well but this approach is not employed by all the algorithms of this paper.

*Partially supported by the ESPRIT BRA programme contract No.20244, ALCOM IT. E-address: amg@dcs.warwick.ac.uk

A few graph theoretic problems have very easy solutions. For example, labelled graphs or, more generally, subgraphs of a given labelled graph can be generated u.a.r. by simply selecting edges with probability $1/2$, using n^2 processors this only requires constant time. We also observe that it is easy to generate u.a.r. the Eulerian subgraphs of a given graph. Given a subset of the fundamental circuits of G (each such circuit is chosen to be in the subset with probability $1/2$), an Eulerian subgraph is generated u.a.r. by taking the ring sum of the subset. A spanning tree of G and thence a set of fundamental circuits can be found by standard NC algorithms.

An interesting aspect of some of the algorithmic solutions involves the conversion of sequential expected time to parallel worst case time with reasonably small error probability (i.e. a small probability that the parallel algorithm fails to produce any result). In these cases, algorithmic work can be traded-off for higher success probability. For problem size n, we will abbreviate phrases of the form *with probability at least* $1 - n^{-c}$ *for any* $c > 0$ by the acronym whp.

Those complexity parameters of our algorithms which employ counters are generally dominated by the cost of counting which, for the problems we consider, is determined by repeated matrix multiplication. The well-known costliness of this in terms of numbers of processors is frequently referred to as the *transitive closure bottleneck*. On the other hand, the work required by these algorithms is within logarithmic factors of the sequential algorithms known to us.

In Section 2 we describe two algorithms for selecting unlabelled graphs uniformly at random. The first is based on a sequential method described in [9] and shows that the uniform generation problem for unlabelled graphs can be reduced to the problem of counting the number of such graphs. As a by-product we describe an NC algorithm for enumerating a polynomial number of integer partitions in *weak* ascending lexicographic order. The second algorithm originates from a sequential strategy described in [27]. This approach requires no information about the number of graphs and uses a simpler sampling method.

In Section 3, given an arbitrary graph G, we describe RNC algorithms for the uniform generation of non-simple paths and spanning trees of G. As a subprocedure, the uniform generation of spanning trees algorithm requires a parallel solution to the problem of generating random

walks which we also describe.

The final section is devoted to conclusions and open problems.

2 Unlabelled Graphs.

Sequential methods for generating graphs u.a.r. can be grouped into two broad classes. The first class includes methods based, either explicitly or implicitly (via recurrence relations or asymptotic enumeration), on counting formulas (see [26] for a survey). Graph theoretic problems are reduced to number theoretic ones, then concepts from linear algebra or combinatorics are used to obtain the desired algorithm. Very often this approach leads to algorithms running in time linear in the size of the object to be generated. Another option is given by Markov chain based methods (see [22] for a complete description and many exemplar applications). The combinatorial structure of interest is generated by simulating a suitably defined Markov chain until the process converges to its stationary distribution. The Markov chain approach is particularly useful when no exact uniform generation algorithm is known. In many cases it is possible to prove that an object can be generated in polynomial time *almost* u.a.r. (see for instance [14, 15]). The rest of this section describes parallel algorithms based on the first method described above.

2.1 Conjugacy Classes and Exact Counting.

In this section we describe a parallel algorithm for generating unlabelled graphs u.a.r. Some preliminaries are needed. Let $\mathcal{G}$ be a given finite group acting on a set $\Omega \neq \emptyset$. The action of $\mathcal{G}$ induces an equivalence relation $\sim$ on Ω ($\alpha \sim \beta$ iff $\alpha = g\beta$ for some $g \in \mathcal{G}$). The equivalence classes are called *orbits*. For each $g \in \mathcal{G}$, define $Fix(g) = \{\alpha : g\alpha = \alpha\}$. Basic facts:

(a) (Frobenius-Burnside Lemma) The number of orbits is $m = \frac{1}{|\mathcal{G}|}\sum_{g\in\mathcal{G}} |Fix(g)|$.

(b) For each orbit ω, $|\{(g,\alpha) : \alpha \in \omega \cap Fix(g)\}| = |\mathcal{G}|$.

(c) Given a conjugacy class (cf. [20]) $C \subseteq \mathcal{G}$, for all $f, g \in C$ and for each $\omega \in \Omega$, $|Fix(f) \cap \omega| = |Fix(g) \cap \omega|$ and $|Fix(f)| = |Fix(g)|$.

Dixon and Wilf [9] describe a sequential algorithm for generating orbits $\omega \subseteq \Omega$ u.a.r. as follows:

(1) select a conjugacy class $C \subseteq \mathcal{G}$ with probability $\frac{|C||Fix(g)|}{m|\mathcal{G}|}$ (where g is any member of the class);
(2) select u.a.r. $\alpha \in Fix(g)$ and return its orbit.

Using (a)-(c) above it is easy to show that it is equally likely for α to be in any of the orbits.

The algorithm above can be adapted to generate unlabelled graphs of given order. Ω is the set of all labelled graphs with n vertices while $\mathcal{G} = S_n$, where S_n is the permutation group. The action of $\mathcal{G}$ on Ω is a mapping which, given a graph and a permutation, relabels the vertices of the graph according to the permutation. In this case m is the number of unlabelled graphs g_n. The number of conjugacy classes in S_n (cf. [20]) is the number $p(n)$ of integer partitions of n. Throughout the paper the notation $[k_1, k_2, \ldots, k_n]$ will be used equivalently for conjugacy classes and partitions: $g \in [k_1, k_2, \ldots, k_n]$ has k_1 cycles of length 1, k_2 cycles of length 2 and so on. Sometimes π will be used to denote conjugacy classes. The cardinality of $[k_1, k_2, \ldots, k_n]$ is $\frac{n!}{\Pi_i (i^{k_i} k_i!)}$ while $|Fix(g)| = 2^{q(g)}$ where $q(g)$ is the number of cycles of the permutation on pairs g^* definable in terms of g.

Step (2) can be easily solved in $O(n^2)$ sequential time. The probabilities in Step (1) have the form $Pr(\pi) = \frac{2^{q(g)}}{g_n \Pi_i (i^{k_i} k_i!)}$. The selection of $\pi \subseteq S_n$ is involved because of the large number of conjugacy classes. The solution is obtained by noticing that classes of permutations moving only a few elements are much more likely to be chosen. This leads to an $O(n^2)$ expected time sequential algorithm.

In deriving a parallel solution the number of partitions has to be fixed in advance (processors have to be allocated for enumerating them). In what follows we prove that a polynomial number of partitions can be enumerated fast in parallel and that, using them, it is possible to devise an RNC algorithm for generating unlabelled graphs whp.

Partitions. Integer partitions of n in descending lexicographic order can be enumerated with constant delay in parallel using $O(n)$ processors [1]. For the purposes of this paper partitions are to be generated fast in parallel in the following *weak* ascending lexicographic order (walo for short): π_j follows π_i if the number of ones in π_i is not less than the number of ones in π_j.

Let $s_i(n)$ be the number of partitions of n whose smallest element is i; let $u(n,k)$ (resp. $\underline{u}(n,k)$) be the number of partitions of n with smallest element equal to one and having exactly (resp. at least) $n-k$ occurrences of it. The following lemma describes some relationships of these numbers.

Lemma 2.1 *For all $n, k \in N^+$ with $1 \leq k \leq n$,*

1. $p(n) \sim \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$;
2. $p(n) = 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} s_i(n)$;
3. $s_k(n) = s_k(n-k) + s_{k+1}(n+1)$;
4. $p(n) = s_1(n+1)$;
5. $u(n,k) = 1 + \sum_{i=2}^{\lfloor k/2 \rfloor} s_i(k)$.
6. $\underline{u}(n,k) = 1 + \sum_{i=2}^{k} u(n,i)$.

7. $\underline{u}(n,k) = p(k)$.

Proof. See [23] for detailed combinatorial proofs. □

Results *5* and *6* of Lemma 2.1 give a way to enumerate all partitions of n having $n-k$ ones by generating all partitions of k with smallest part at least 2 and then padding them out with $n-k$ ones on the left. The algorithm for enumerating all partitions of n having at least $n-k$ ones in walo simply iterates this procedure.

Lemma 2.2

1. *If* $k \in \Theta(\ln^2 n)$ *then there exist two constants* $c_1 < c_2$ *such that* $n^{c_1} \leq \underline{u}(n,k) \leq n^{c_2}$.
2. *In particular if* $k = A\ln^2 n$ *then* $\underline{u}(n,k) \in \Theta(n^{\pi\sqrt{2A/3}})$.

Proof. If we choose $1 \leq A\ln^2 n \leq k \leq B\ln^2 n$ then we can claim (using Lemma 2.1)

$$\underline{u}(n,k) = p(k) \leq (1+\varepsilon)\frac{e^{\pi\sqrt{2k/3}}}{4k\sqrt{3}} \leq \frac{(1+\varepsilon)}{4\sqrt{3}} n^{\pi\sqrt{2B/3}}$$

and

$$\underline{u}(n,k) = p(k) \geq (1-\varepsilon)\frac{e^{\pi\sqrt{2k/3}}}{4k\sqrt{3}} \geq \frac{(1-\varepsilon)}{4\sqrt{3}} n^{\left(\pi\sqrt{2A/3}-1\right)}$$

Notice that if we impose $\pi\sqrt{2B/3} \geq \pi\sqrt{2A/3} - 1$ we get the following constraint on B,

$$B \geq \left(1 - \frac{3}{\pi}\sqrt{\frac{2}{3}}\right) A + \frac{3}{2\pi^2}.$$

The second part of the proof is a simple special case. □

The next result describes a parallel algorithm for enumerating all partitions of a given number n.

Theorem 2.1 *All partitions of any given number* n *can be enumerated in* $O(n^{1.5})$ *time on an EREW PRAM using* $O(p(n))$ *processors.*

Proof. Partitions of n in descending lexicographic order can be grouped w.r.t. their largest element i. The enumeration problem then is solved recursively by concatenating this i with all the partitions of $n-i$ with largest element at most i.

Since for any given partition $\pi \equiv [k_1, \ldots, k_n]$ the identity $\sum_i ik_i = n$ holds then only at most $\lfloor\sqrt{2n}\rfloor$ terms k_i are non-zero so that the resulting computation tree has depth $O(\sqrt{n})$ in each path and has $p(n)$ paths. The result follows. □

Using Theorem 2.1 and the value for k given by Lemma 2.2 it is possible to prove the following result:

Theorem 2.2 *If* $k = \ln^2 n$ *then the first* $\underline{u}(n,k)$ *partitions of* n *in walo can be enumerated in* $O(\lg^3 n)$ *on an EREW PRAM using* $O(\underline{u}(n,k))$ *processors.*

Proof. By Lemma 2.2 and Lemma 2.1 if $k = \frac{3}{32\pi^2}\ln^2 n$ the task of enumerating in parallel $\underline{u}(n,k)$ partitions of n in walo can be reduced to the parallel enumeration of all partitions of i (for all $i \in \{2, \ldots, k\}$) with smallest element at least 2 (padded with $n-i$ leading ones). A slight variation of the algorithm in Theorem 2.1 can be applied. The overall algorithm runs in $O(k^{1.5})$ steps on an EREW PRAM with $n^{O(1)}$ processors. □

Selecting a Cycle Type. The parallel algorithm is described below.

```
enumerate first u(n,k) partitions π_j in walo;
for all u(n,k) partitions compute Pr(π_j);
choose u.a.r. ξ ∈ [0,1);
Parallel Prefix on Pr(π), results in P;
for all j, 1 ≤ j ≤ u(n,k) in parallel do
    if ((P_{j-1} ≤ ξ) and (P_j > ξ))
        then output(π_j) and stop;
output(ERROR);
```

Notice that the algorithm is parameterized by the number of partitions used.

Lemma 2.3 *If* g_n *is given and* $k \in \Theta(\lg^2 n)$ *then all probabilities* $Pr(\pi_j) = \frac{2^{q(n)}}{g_n \Pi_i \left(i^{k_i} k_i!\right)}$ *for* $j = 1, \ldots, \underline{u}(n,k)$ *are computable in* $O(\lg^3 n)$ *steps using* $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ *processors on an EREW PRAM.*

Proof. $q(g) = \frac{1}{2}\left\{\sum_{i=1}^{n} l(i)^2\varphi(i) - l(1) + l(2)\right\}$ (see [9]) where φ is the Euler totient function while $l(i) = \sum_{i|j} k_j$. In particular $q(g) \leq \binom{n}{2}$ for all $g \in S_n$. The algorithm for computing the probabilities above requires the following preprocessing steps:

(1) Compute 2^j for all $j = 1, \ldots, \binom{n}{2}$. Using parallel prefix this requires $O(\lg n)$ steps using $O(n^2/\lg n)$ processors.
(2) Compute $j!$ for all $j = 0, 1, \ldots, n$ (again $O(\lg n)$ steps using $O(n^2/\lg n)$ processors).
(3) Compute $\varphi(j)$ for all $j = 1, \ldots, n$. Using results in [24] this requires $O(\lg n)$ steps using $O((n/\lg n)^2)$ processors.

The steps above can all be slowed down to run in $O(\lg^3)$ time on $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ processors. Once the preprocessing above has been done $l(i) = \sum_{i|j} k_j$, $q(g)$ and $\Pi_i \left(i^{k_i} k_i!\right)$ can be computed in $O(\lg^3 n)$ steps using $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ processors (using again the fact that only $O(\sqrt{n})$ components of each partition are non-zero) and the well known Brent Scheduling principle. □

Let $t_n(\xi)$ be the random variable denoting the least integer n_j such that $P_j > \xi$ in the algorithm. The following lemma gives a useful bound on the i-th moment of $t_n(\xi)$.

Lemma 2.4 *For all n and $j \geq 1$ there exist constants $0 < c_2 \leq 2$ and $c_3, c_4 > 0$ such that*

$$E(t_n(\xi)^i) \leq c_2 + 2^{(c_3 i\sqrt{n}+\lg^2 n)-(c_4 n \lg^2 n+\lg n)}.$$

Proof. It is worth remembering that $E(t_n(\xi)^i) = \sum_{l=1}^{p(n)} l^i Pr(\pi_l)$. The proof refines a claim in [9] and follows from enumeration results in [13]. □

Now we can state and prove the main theorem concerning the algorithm above.

Theorem 2.3 *There is an $m \in \Theta(n)$ such that if g_n is given then the algorithm above runs in $O(\lg^3 n)$ time with $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ processors on an EREW PRAM with error probability less than m^{-1}.*

Proof. We can enumerate a polynomial number of partitions using the algorithm of Theorem 2.2 in $O(\lg^3 n)$ parallel steps using $O(\underline{u}(n,k))$ processors. For all $j = 1, \ldots, m$ we can compute $Pr(\pi_j)$ in $O(\lg^3 n)$ time using $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ processors (by Lemma 2.3). The thresholds $P_j = \sum_{i=1}^{j} Pr(\pi_i)$ for all $j = 1, \ldots, \underline{u}(n,k)$ can be evaluated by running parallel prefix and the appropriate partition can be chosen accordingly in $O(\lg^3 n)$ time again using $O\left(\frac{n^{3/2}\underline{u}(n,k)}{\lg^3 n}\right)$ processors.

Using the i-th moment expression of the Chebishev inequality (cf. [19]), Lemma 2.4 and the hypothesis $\underline{u}(n,k) \geq n^c$, the error probability $Pr(t_n(\xi) > \underline{u}(n,k))$ of the algorithm can be bounded above by $\frac{c_2}{n^{ci}}$ and well known powering techniques can then be applied to reduce the error probability below any polynomial factor. □

Notice that smaller error probability is traded off for a larger number of processors used. Moreover by using $\Theta(\sqrt{n})$ partitions we can make this algorithm work efficiently (in the sense of [11]).

Selecting a Graph. Given $\pi \equiv [k_1, k_2, \ldots, k_n]$ obtained in phase one, the second phase is accomplished by choosing $g \in \pi$ and then selecting u.a.r. $\alpha \in Fix(g)$.

The permutation g can be represented by an array A of n elements such that A_i contains the address (i.e. the index) of the element following i in its cycle (e.g. $A \equiv [1,3,2,5,6,4]$ corresponds to $g = (1)(2\ 3)(4\ 5\ 6)$). The array A can be built in $O(\lg n)$ steps using $O(n^2/\lg n)$ processors.

The construction of g^* and the random choice of its cycles to form the selected $\alpha \in Fix(g)$ involves the following steps:

(1) build the two-dimensional matrix A^* whose entries are pairs of integers such that for all $i, j = 1, 2, \ldots, n$ with $i \neq j$, $A^*_{i,j} = (A_i, A_j)$ means $g^*(\{i,j\}) = \{A_i, A_j\} = \{g(i), g(j)\}$ (a single parallel step using n^2 processors).
(2) compute the cycles of A^* (this requires $O(\lg n)$ steps and $O(n^2/\lg n)$ processors by standard pointer doubling techniques).
(3) define α selecting A^*'s cycles with probability 1/2.

Thus we have the following result.

Theorem 2.4 *Given $\pi \subseteq S_n$ and $g \in \pi$, a graph in $Fix(g)$ can be selected u.a.r. in $O(\lg n)$ parallel time on an EREW PRAM with $O(n^2/\lg n)$ processors.* □

Putting together results 2.3 and 2.4 in this section we get an RNC algorithm for generating u.a.r. unlabelled graphs of given order if their number is part of the input.

2.2 Avoiding the Counting.

In [27] an alternative sequential algorithm is described which doesn't require the number g_n. The basic idea is to use a so-called *restarting procedure*, i.e. a procedure which accepts a result and outputs it only with a certain probability (otherwise it restarts the whole probabilistic process). If an upper bound B is known for g_n a restarting procedure can be devised to generate unlabelled graphs u.a.r.. A second feature of this method is the use of a different (much smaller) decomposition of S_n. This implies a simplified selection procedure: probabilities can be tabulated and equivalence classes selected accordingly. A minor drawback is that, once a particular class has been selected, the representative g of this class also has to be chosen u.a.r. and this in turn will introduce some inefficiency in the algorithm.

Define R_1 containing only the identical permutation and, for $2 \leq i \leq n$, $R_i = \{g : g \in S_n \text{ moving } i \text{ objects}\}$. Clearly $|R_1| = 1$, $|R_i| = n!/(n-i)!$ for all $i \geq 2$ and $S_n = \bigcup_{i=1}^{n} R_i$.

Lemma 2.5 *Let R_i be defined as above, and let $B_1 = 2^{\binom{n}{2}}$ and for $2 \leq i \leq n$ let $B_i = 2^{\binom{n}{2}-H(n,i)}\frac{n!}{(n-i)!}$ where $H(n,i) = i\left(\frac{n}{2} - \frac{(i+2)}{4}\right)$. Then $|R_i| \leq B_i$ for all $i = 1, \ldots, n$.*

Proof. See [27]. □

Given the following definitions

$$\Phi_g = g \times Fix(g), \quad \Phi = \bigcup_{g \in S_n} \Phi_g, \quad B = \sum_{i=1}^{n} B_i$$

the sequential generation algorithm is described by the following steps:

(1) select R_i with probability $\frac{B_i}{B}$;
(2) select u.a.r. $g \in R_i$;
(3) select u.a.r. $\alpha \in Fix(g)$ and output its orbit with probability $\frac{|R_i||Fix(g)|}{B_i}$ otherwise back to step (1).

In what follows the probabilistic behaviour of this algorithm is described in detail. The expected number of restarts of this procedure is bounded above by a constant K. Let T be the random variable which counts the number of restarts before termination.

Lemma 2.6 *The random variable T has geometric distribution with parameter $|\Phi|/B$.*

Proof. The random variable T assumes values $0, 1, 2, \ldots$, moreover any iteration of the algorithm is performed independently from all the others and the success probability at each iteration is

$$Pr(Su) = \sum_{\Phi} Pr(Su|s)P(s) = \sum_{\Phi} \frac{P(s)}{B \cdot P(s)} = \frac{|\Phi|}{B}$$

Su is shorthand for "success" and s for any compound object $(g, \alpha) \in \Phi$ generated by the algorithm above. $Pr(Su|s)$ is the expression given in step (3) of algorithm above whereas $Pr(s) = \frac{B_i}{B|R_i||Fix(g)|}$. From the previous remarks it follows that T has geometric distribution with parameter $|\Phi|/B$. □

The parallel algorithm is based again on a conversion from sequential expected time to worst case parallel time whp. There is a preprocessing stage in which all the B_i, for $i = 1, \ldots, n$ are computed. Then ρ classes $R_{i_1}, \ldots, R_{i_\rho}$, permutations $g_j \in R_{i_j}$ and numbers $\xi_j \in [0, 1]$ are selected. If ρ is sufficiently large the probability that, for all j, $\xi_j > \frac{|R_{i_j}||Fix(g_j)|}{B_{i_j}}$ is less than ρ^{-c} for any $c > 0$. Let $\hat{j} = min\left\{j : \xi_j \leq \frac{|R_{i_j}||Fix(g_j)|}{B_{i_j}}\right\}$; the algorithm returns the orbit corresponding to an α randomly chosen in $Fix(g_{\hat{j}})$. The following paragraphs describe the three main phases of the algorithm.

Notice that calculations involved in Step (3) above can be easily performed using results of the preceding steps and a subroutine for computing $|Fix(g)| = 2^{q(g)}$ (which takes $O(\lg n)$ steps using $O(n^2/\lg n)$ processors since $q(g) \leq n^2$). A graph can be output using the second phase of algorithm in 2.1.

Preprocessing stage. B_1 and for all $i = 2, \ldots, n$ the numbers $2^{H(n,i)}$ can be computed in $O(\lg n)$ steps using $O(n^2/\lg n)$ processors. Moreover for $i = 2, \ldots, n$ the numbers $n!/(n-i)! = \Pi_{j=n-i+1}^{n} j$ can be found using a modified parallel prefix computation (starting from the list of numbers in reverse order) in $O(\lg n)$ steps using $O(n/\lg n)$ processors. From these remarks it follows that all B_i and $\sum B_i$ can be computed in $O(\lg n)$ steps using $O(n^2/\lg n)$ processors. Finally parallel prefix is run on the probabilities $B_i/\sum B_i$ to define the threshold values P_i for the sampling procedure in the next paragraph.

Selecting a class. Once the probabilities involved have been computed the selection of R_i can be done by choosing a real number in $[0, 1]$, copying it into an array of size n and comparing it in parallel with the thresholds P_i (one step using n processors).

Selecting an element. There are several algorithms for generating permutations in parallel u.a.r. (see for instance [1, 4, 25]). In [27], Wormald outlines a method for uniformly generating permutations in R_i (i.e. permutations which move exactly i elements out of n). Combining these two facts we obtain the following result.

Lemma 2.7 *Elements of R_i can be generated u.a.r. in $O(\lg n)$ parallel steps using $O(\delta n)$ processors whp if $\delta \geq c \lg n$ for all $c > 0$.*

Proof. δ groups of $O(n)$ processors are allocated. Each of them selects a random permutation of $1, 2, \ldots, i$ (in $O(\lg n)$ time steps using a result in [4]). Let the random variable X denote the number of derangements in m_1 trials. X is binomially distributed with success probability $p = i¡/i! \sim e^{-1}$ (where $i¡$ is the *sub-factorial* of i as defined for example in [12, pp. 194–196]). Then the failure probability is $(1-p)^{\delta}$ which is less or equal than n^{-c} for any $c > 0$ if $\delta > c \lg_{1/(1-p)} n$. □

Theorem 2.5 *If $\rho \in \Omega(n)$ then there exists a randomized parallel algorithm for generating unlabelled graphs u.a.r. in time $O(\lg^2 n)$ using $O\left(\rho \frac{n^2}{\lg n}\right)$ processors on an EREW PRAM whp.*

Proof. The algorithm listed after this proof solves the problem. The time to compute every iteration of the inner loop is dominated by the time to generate $g \in R_i$ u.a.r. (which is $O(\lg n)$ and $O(\delta n)$ processors by Lemma 2.7) and the time to compute $|Fix(g)| = 2^{q(g)}$. To compute $|Fix(g)|$ the cycle type of g has to be determined first and then $2^{q(g)}$ can bew computed using a simplified version of the algorithm in Lemma 2.3 in $O(\lg n)$ time using $O(n^2/\lg n)$ processors. The final selection of α can be performed again as in the algorithm in 2.1. So the overall running time is $O(\lg n)$ using at most $O\left(\rho \frac{n^2}{\lg n} + \frac{n^2}{\lg n}\right)$ processors.

Let M and Y be the random variables denoting respectively the number of successful selections of $g_j \in R_{i_j}$ and the number of times $\xi_j \leq |R_{i_j}||Fix(g_j)|/B_{i_j}$. The failure probability of the whole algorithm is:

$$Pr(Y = 0) = \sum_{x=0}^{\rho} Pr(Y = 0|M = x)Pr(M = x)$$

If $\rho \in \Omega(n)$ then, using Lemma 2.6

$$\begin{aligned} Pr(Y = 0) \leq & \\ \leq & \sum_{x=0}^{\lg n} Pr(M = x) + \left(1 - \frac{|\Phi|}{B}\right)^{\lg n} \\ = & \sum_{x=0}^{\lg n} \binom{\rho}{x} A^x (1-A)^{\rho - x} + \left(1 - \frac{|\Phi|}{B}\right)^{\lg n} \end{aligned}$$

where $A = 1 - (1-p)^{\delta}$ is given by Lemma 2.7 and the last expression is less than n^{-c} for some $c > 0$.

An improved algorithm with polynomially small error probability can be obtained by using the powering technique again. □

$B_1 \leftarrow 2^{n(n-1)/2}$;
for all i, $2 \leq i \leq n$ **in parallel do**
    Compute B_i;

$B \leftarrow \sum B_i$;
for all j, $1 \le j \le \rho$ **in parallel do**
  Choose $i_j \in \{1, \ldots, n\}$ with $Pr(i) = \frac{B_i}{B}$;
  Generate a random permutation $g_j \in R_{i_j}$;
  Choose $\zeta_j \in [0, 1]$ u.a.r.;
  if (g_j generated) $\wedge \left(\zeta_j \le \frac{|R_{i_j}||Fix(g_j)|}{B_{i_j}}\right)$
    then $A_j \leftarrow 1$;
    else $A_j \leftarrow 0$;
$\hat{j} = \min\{j \mid A_j \neq 0\}$;
if defined($\hat{j}$) **then**
  Choose $\alpha \in Fix(g_{\hat{j}})$;
  Return the orbit of α; Stop;

3 Generation of Subgraphs.

In what follows a graph $G = (V, E)$ with n vertices is given. We describe how to generate u.a.r. non-simple paths and spanning trees of G. The generation of spanning trees in this way requires the parallel generation of random walks, this also will be described.

3.1 Uniform Generation of Non-Simple Paths.

A non-simple path in a graph is a path which may visit any vertex or any edge more than once. In this section an RNC algorithm for uniformly generating polynomial length non-simple paths based on a counter is given. Crucially the problem is shown to be *self-reducible* (in the sense of [21]).

Let A be the *adjacency matrix* of G. It is well known (see [10] for example) that the (i, j)-th entry of A^k, the k-th power of A, is the number of non-simple paths from vertex i to vertex j of length k (A^k may be found in time $S(n) \lg k$ using $n^3 / \lg n$ processors where $S(n)$ is the time required to square an $n \times n$ matrix).

Let $s, t \in V$ and let $k \in n^{O(1)}$. Moreover, for the moment at least, assume that $k = 2^m$ for some positive integer m. A non-simple path P_{st} of length k from s to t can be thought of as being recursively constructed from two paths P_{su} and P_{ut}, each of length $k/2$ for some $u \in V$. Clearly such a self-reducible description of P_{s-t} has depth logarithmic in k and thus, for $k = n^{O(1)}$, has depth $O(\lg n)$. Such a description of P_{st} may be used to uniformly generate the paths of length k between s and t. The midpoint u of P_{st} is chosen with probability $\frac{A^{k/2}_{s,u} A^{k/2}_{u,t}}{A^k_{s,t}}$ which, of all paths of length k from s to t, is the proportion that have u as a midpoint. P_{su} and P_{ut} are recursively computed in parallel.

In what follows an array p stores the path output by the algorithm: for $0 \le i \le k$, p_i is the i-th vertex in the path (initially $p_0 = s$ and $p_k = t$). Before an arbitrary step vertices p_a and p_b may have been identified without yet having identified p_i for $a < i < b$. Let the algorithmic action of choosing $p_i = u$ be $URV(a, i, b)$. Notice that, if the powers of A are given, using prefix sums $URV(a, i, b)$ takes $O(\lg n)$ parallel steps by an optimal algorithm on a PRAM allowing concurrent reads.

The following recursive program $URP(x, y)$ computes the path using $URV(a, i, b)$ as a subroutine (notice x and y are always powers of 2).

$URP(x, y)$
  if $(y - x) > 1$ **then**
    $URV(x, (x + y)/2, y)$;
    in parallel
      $URP(x, (x + y)/2)$;
      $URP((x + y)/2, y)$;

$URP(0, k)$ is merely a formulation of the algorithm described earlier.

It is easy to generalise the algorithm for any integer value of $k \in n^{O(1)}$. In $O(\lg n)$ time a single processor may express k as a sum of (at most a logarithmic number) of powers of 2. In the following description the array R stores these powers of 2 for any input value of k.

input k;
$K \leftarrow k$; $i \leftarrow 1$; $M_i \leftarrow 1$; $j \leftarrow 0$;
repeat
  $i \leftarrow i + 1$; $M_i \leftarrow 2M_{i-1}$
until $K < M_i$;
repeat
  $i \leftarrow i - 1$;
  if $K \ge M_i$ **then**
    $j \leftarrow j + 1$; $R_j \leftarrow M_i$; $K \leftarrow K - M_i$;
until $K = 0$;
Parallel prefix on R result in Q for $1 \le i \le j$;
for $i = 1$ **to** $\lg R_1$ **do**
  $A^{2^i} \leftarrow A^{2^{i-1}} A^{2^{i-1}}$;
for $i = 2$ **to** j **do**
  $A^{Q_i} \leftarrow A^{Q_{i-1}} A^{P_i}$;
$Q_0 \leftarrow 0$;
for $i = j - 1$ **downto** 1 **do**
  $URV(0, Q_i, Q_{i+1})$
for all i, $0 \le i \le j - 2$ **in parallel do**
  $URP(Q_i, Q_{i+1})$

The first two loops compute P. The array Q stores the positions in the array p which divide its length into portions each a power of 2 in length. The vertices that are located at these positions are chosen sequentially (these choices require the computation of a number of A^p for which p is not a power of 2 as shown). Finally, the last for loop deals, in parallel, with all subsections of the generated path which are a power of 2 in length.

The running cost of the above algorithm is dominated by the evaluation of the A^p. This takes $O(\lg^2 n)$ time using $n^3 / \lg n$ processors. If these quantities are precomputed the complexity parameters are $O(\lg n)$ time and

n^2 processors. This is yet another example in which the so-called *transitive closure bottleneck* is dominant.

Theorem 3.1 *A non-simple path of length $k \in n^{O(1)}$ between two vertices $s, t \in V$ can be generated u.a.r. in $O(\lg^2 n)$ time using $n^3/\lg n$ processors on an CREW PRAM.*

Proof. The algorithm above solves the problem. Let $p_0 = s$ and $p_k = t$. The probability that any path $(p_0, \ldots, p_k)$ is generated is given by the product of the probabilities that each of the vertices on the path is chosen to occupy its position. After cancelling many terms, this is easily seen to be: $\frac{\Pi_{i=0}^{k-1} A_{p_i,p_{i+1}}}{A^k_{p_0,p_k}} = \frac{1}{A^k_{p_0,p_k}}$ and the result follows. □

The algorithm may be easily modified to generate a non-simple path of length k without specified endpoints. The endpoints are then chosen with probability based on the number of paths of length k that have these endpoints. By similar means we can obtain an RNC uniform generation algorithm for non-simple paths of *maximum* length k.

3.2 Random Walks and Spanning Trees.

In [2, 6] it is shown that a spanning tree T of an undirected graph is obtained by generating a random walk starting at a vertex s which covers the graph and including in T, for each node i but s, the edge $\{j, i\}$ used for the first entry to i during the walk. Using Markov chain analysis it is possible to show that if the starting point of the walk is chosen according to the stationary distribution of the walk then T is uniformly distributed among all undirected spanning trees of G. This approach can be parallelized as described below.

3.2.1 Random Walks.

Two algorithms for the parallel generation of random walks are described. One is based on a counter-like approach while the other is not.

In each step of a *random walk* in a graph, if v is the currently visited vertex, the next vertex visited is adjacent to v and is chosen with probability d_v^{-1}, where d_v is the degree of v.

The same algorithmic approach taken for non-simple paths in the previous section can be adopted if the adjacency matrix, A, is replaced by the so-called *transition matrix*, T with $T_{i,j} = 1/d_i$ if j is adjacent to i (zero otherwise). It is well known from Markov Chain theory (see [18] for example) that the (i, j)-th entry of the k-th power of T is the probability that a random walk starting at i terminates at j after k steps. The following result is immediate.

Theorem 3.2 *There exists an RNC algorithm for generating a random walk of given polynomial length between two specified endpoints.* □

Notice that the probability that any random walk $(v_0, v_1, v_2 \ldots v_k)$ starting at v_0 and ending at v_k is generated is given by the product of the probabilities that each of the vertices is chosen to occupy its position. After cancelling many terms, this is easily seen to be $\frac{\Pi_{i=0}^{k-1} T_{v_i,v_{i+1}}}{T^k_{v_0,v_k}}$ which is analoque to the expression in 3.1. As was the case for non-simple paths, the algorithm may be easily modified to generate a random walk of length k without specified endpoints. The endpoints are chosen with probability based on the number of random walks of length k that have these endpoints (in this case $Pr((v_0, v_1, v_2, \ldots, v_k)) = \Pi_{i=0}^{k-1} T_{v_i,v_{i+1}}$ as required by using Markov chain theory). It is also very easy to modify the algorithm so as to generate random walks of *maximum* length k. Clearly the above algorithm works within the same complexity constraints as the parallel uniform generator of non-simple paths: $O(\lg^2 n)$ time using $n^3/\lg n$ processors.

An alternative algorithm which generates random walks without recourse to a counter-like facility is described next. The algorithm generates a random walk of fixed specified length from each vertex of the graph. As described later, the uniform generation of a spanning tree follows from the generation of a random walk which covers the graph. Thus a counter-like approach is not needed for the generation of spanning trees. While the following algorithm is good enough for applications like spanning tree generation it is not as flexible as the previous approach. For example, it does not allow the prior specification of both endpoints of the random walk.

For a graph with n vertices, we can obtain a random walk of length k in constant time using nk processors as follows. In parallel, each processor $P_{i,j}$, $1 \le i \le n$, $1 \le j \le k$, chooses a neighbour of vertex i with probability d_i^{-1} and stores this in the (i, j)-th location of the array N. For any starting vertex i visited at time $t = 0$, the array N stores a random walk such that if the vertex visited at time t, $0 \le t < k$ is v_{it} (note: $v_{i0} = i$), then the vertex visited at time $t+1$ is given by $N_{v_{i,t},t+1}$. This immediate and natural algorithm was employed in [17].

3.2.2 Uniform Generation of Spanning Trees.

Given a graph $G = (V, E)$ and a random walk starting at $s \in V$ which covers the graph, a spanning tree T of G can be obtained by selecting for each $v \in V - \{s\}$ the first entry edge for v in the walk. If p is an array storing successive vertices in the walk, the first entry edges are obtainable by sorting lexicogaphically pairs (p_i, i) so that if $i < j$ but $p_i = p_j$ then (p_i, i) precedes (p_j, j). Then the leftmost occurence of each vertex is selected and the corresponding entry edge can be found in W using the index component. The following instructions give a more detailed algorithm.

input p;
for all i, $1 \le i \le k$ **in parallel do**

```
    W_{i,1} ← (p_{i-1}, p_i);
for all i, 0 ≤ i ≤ k in parallel do
    S_{i,1} ← p_i;
    S_{i,2} ← i;
Sort (S_{i,1}, S_{i,2});
S_{-1,1} ← 0; S_{-1,3} ← 0;
for all i, 0 ≤ i ≤ k in parallel do
    if S_{i,1} = S_{i-1,1} + 1 then S_{i,3} ← 1
    else S_{i,3} ← 0;
for all i, 0 ≤ i ≤ k in parallel do
    if S_{i,3} = 1 then W_{S_{i,2},2} ← 1
    else W_{S_{i,2},2} ← 0;
Prefix computation on W_{i,2};
for all i, 1 ≤ i ≤ k in parallel do
    if (W_{i,2} > 0) ∧ (W_{i,2} = W_{i-1,2} + 1)
       then T_{W_{i-1,2}} ← W_{i,1};
```

The dominant cost arises from the sorting operation. This can be carried out in $O(\lg k)$ time with k processors by the optimal algorithm of [8]. The essential question from the complexity point of view is: how large should k be in order that every vertex of the graph has a high chance of being visited and such that the complexity resources required are not too large? Several authors have studied the so-called *cover-time*, C_G (for example [3, 7, 19]) which is the expected time to visit all vertices. The value for C_G is $O(n \lg n)$ for almost all graphs and $O(n^3)$ for the worst graphs. By choosing k sufficently larger than C_G the algorithm above will generate a spanning tree u.a.r. whp. The result is summarized in the following theorem.

Theorem 3.3 *Given an undirected graph G there exists an RNC algorithm running in $O(\lg^2 n)$ time using $O(n^3/\lg n)$ processors which generates a spanning tree of G u.a.r. whp.*

Proof. We choose the starting point s of the walk with probability $d_s / \sum_{i=1}^{n} d_i$ (i.e. the stationary distribution of the walk, see [19, p. 56] for details). Let the length of the walk be $k \in \Omega(nC_G)$ and, using the required powers of the transition matrix T, select the end point t of the walk with probability equal to the proportion of random walks of length k starting at s which end up at t. Finally a random walk of length k from s to t and a spanning tree T are generated as described in the previous part of this section.

Let T_G be the number of steps required to cover the graph; by the Chebishev inequality, the error probability of this algorithm is $Pr(T_G \geq k) \leq \frac{C_G}{k} \leq \frac{1}{n}$ since $k \in \Omega(nC_G)$. The result follows by powering again. □

4 Conclusion and Open Problems.

In this paper two RNC algorithms for generating unlabelled graphs u.a.r. were described. The first provides an efficient solution to the problem if the number of graphs is known. The second algorithm uses an upper bound on the number of unlabelled graphs and a simplified selection procedure but, if an arbitrarily polynomially small error probability is sought, it is a linear factor away from efficiency. An interesting open problem is whether the efficiency of the algorithms can be improved by a better approximation for g_n.

Moreover we showed how, given an arbitrary graph, the following associated combinatorial structures can be generated u.a.r. by RNC algorithms: labelled subgraphs, Eulerian subgraphs, non-simple paths and spanning trees. In general, although we obtained RNC algorithms, the number of processors required is too large to be really satisfactory. Any improved algorithm would have to avoid the transitive closure bottleneck incurred by repeated matrix multiplication. On the other hand, the work measure for our algorithms is little different from known sequential algorithms.

One specific problem for which we were unable to find an RNC solution is that of generating uniformly (or nearly uniformly) at random a *simple* path between a pair of specified vertices. This task might have many uses as a subprocedure in randomised algorithms for (for example) determining the connectivity of a graph or for flow augmentation in a flow network.

References

[1] S.G. Akl and I. Stojmenovic. Parallel Algorithms for Generating Integer Partitions and Compositions. *Journal of Comb. Math. and Comb. Computing*, 13:107–120, April 1993.

[2] D.J. Aldous. The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees. *SIAM Journal on Discrete Mathematics*, 3(4):450–464, November 1990.

[3] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovasz, and C. Rackoff. Random Walks, Universal Traversal Sequences and the Complexity of Maze Problems. *20th Annual Symp. on Foundations of Computer Science*, pages 218–223. 1979.

[4] M.D. Atkinson and J.R. Sack. Uniform Generation of Combinatorial Objects in Parallel. Technical Report SCS-TR-185, Carleton University, Ottawa, Canada, January 1991.

[5] M.D. Atkinson and J.R. Sack. Uniform Generation of Binary Trees in Parallel. *Journal of Parallel and Distributed Computing*, 23:101–103, 1994.

[6] A.Z. Broder. Generating Random Spanning Trees. *30th Symp. on Foundations of Computer Science*, pages 442–447. 1989.

[7] A.Z. Broder and A.R. Karlin. Bounds on the Cover Time. *Journal of Theoretical Probability*, 2(1):101–120, 1989.

[8] R. Cole. Parallel Merge Sort. *27th Symp. on Foundations of Computer Science*, pages 511–516. 1986.

[9] J.D. Dixon and H.S. Wilf. The Random Selection of Unlabelled Graphs. *Journal of Algorithms*, 4:205–213, 1983.

[10] A.M. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[11] A.M. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[12] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[13] F. Harary and E.M. Palmer. *Graphical Enumeration.* Academic Press, 1973.

[14] M. Jerrum. Uniform Sampling Modulo a Group of Symmetries Using Markov Chain Simulation. Technical Report ECS-LFCS-94-288, University of Edinburgh, 1994.

[15] M. Jerrum and A. Sinclair. Fast Uniform Generation of Regular Graphs. *Theoretical Computer Science*, 73:91–100, 1990.

[16] M.R. Jerrum, L.G. Valiant, and V.V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science*, 43(2–3):169–188, 1986.

[17] D.R. Karger, N. Nisan, and M. Parnas. Fast Connected Components Algorithms for the EREW PRAM. *4th Symp. on Parallel Algorithms and Architectures*, 1992.

[18] J.G. Kemeny and J.L. Snell. *Finite Markov Chains.* Springer-Verlag, 1976.

[19] S. Naor. Probabilistic Methods in Computer Science. Lecture Notes - Technion (Israel), 1992.

[20] J.J. Rotman. *The Theory of Groups: An Introduction.* Advanced Mathematics. Allyn and Bacon, 1965.

[21] C.P. Schnorr. Optimal Algorithms for Self-Reducible Problems. *3rd International Colloquium on Automata, Languages and Programming*, pages 322–337, 1976.

[22] A. Sinclair. *Algorithms for Random Generation and Counting: a Markov Chain Approach.* Birkhäuser, 1993.

[23] A. Slomson. *An Introduction to Combinatorics.* Chapman and Hall, 1991.

[24] J. Sorenson and I. Parberry. Two Fast Parallel Prime Number Sieves. *Information and Computation*, 114:115–130, 1994.

[25] I. Stojmenovic. On Random and Adaptive Parallel Generation of Combinatorial Objects. *Internat. Journal of Computer Mathematics*, 42:125–135, 1992.

[26] G. Tinhofer. Generating Graphs Uniformly at Random. *Computing*, 7:235–255, 1990.

[27] N.C. Wormald. Generating Random Unlabelled Graphs. *SIAM Journal on Computing*, 16(4):717–727, 1987.