Uniform Generation of Unlabelled Graphs

Ida Pu
Department of Computing
Goldsmiths College, University of London
i.pu@gold.ac.uk

Alan Gibbons
Department of Computer Science
King's College, London
amg@dcs.kcl.ac.uk

January 24, 2006

Abstract

Given the number of vertices n, *labelled graphs* can easily be generated uniformly at random by simply selecting each edge independently with probability 1/2. With $n(n-1)/2$ processors, this takes constant parallel time. In contrast, the problem of uniformly generating *unlabelled graphs* of size n is not so straightforward. In this paper, we describe an efficient parallelisation of a classic algorithm of Dixon and Wilf for the uniform generation of unlabelled graphs on n vertices. The algorithm runs in $O(\log n)$ *expected* time on a CREW PRAM using n^2 processors.

1 Introduction

We consider the problem of generating unlabelled graphs uniformly at random. Very little is known about the uniform parallel generation of combinatorial objects. In [15] the authors describe several RNC algorithms for generating graphs and subgraphs uniformly at random. An RNC algorithms employs randomisation in their design to place the problem being solved in the class NC, which (by the standard definition) is the class of problems that can be solved by efficient parallel algorithms: algorithms that run in polylogarithmic parallel time and use a poynomial number of processors. For example, unlabelled graphs are generated in $O(\log^3 n \log\log n)$ time and $O(n^2)$ work with a polynomially small error probability if their number is known in advance and in $O(\log n)$ time and $O(n^3)$ work otherwise; for an arbitrary graph RNC algorithms are described for the uniform

JCMCC **63** (2007), pp. 209-222

generation of its subgraphs that are either non-simple paths or spanning trees.

The main result of this paper (Section 4) is a Las Vegas style RNC algorithm which runs in $O(\log n)$ *expected* time using n^2 processors of a CREW PRAM and which generates an unlabelled simple graph on n vertices uniformly at random from the set of all such graphs. By a Las Vegas RNC algorithm we mean an algorithm which runs in *expected* polylogarithmic time, uses a polynomial number of processors and which accurately generates the required outcome, in this case *uniform* generation of unlabelled graphs. This result is essentially a parallelisation of a sequential algorithm of Dixon and Wilf [1] which runs in $O(n^2)$ time. The time-complexities stated above for the algorithm of [15] which solves the same problem are *worst-case* running times. What we gain in this paper, at the expense of an *expected* running time, is an algorithm of much greater simplicity. The *expected* running time is also much faster than the *worst-case* running time of [15]. It is also the case that the algorithm described here generates unlabelled graphs *exactly* uniformly, whereas in [15] there are small departures from uniformity.

In section 2 we introduce some preliminary combinatorics. Section 3 recalls Dixon and Wilf's sequential algorithm and describes important variations of some implementation details that lead to our parallelization of the algorithm. In section 4, we describe our parallel algorithm and establish its complexity parameters.

2 Preliminaries

A graph is *simple* if it has no parallel edges nor self-loops. It is *undirected* if no vertex pair (u, v) in the graph is ordered, i.e. if $(u, v) = (v, u)$. We use in this paper the term 'graph' to mean an undirected simple graph unless it is defined otherwise.

A *labelled graph* of size n is a graph whose vertex set is $V = \{1, 2, .., n\}$. An unlabelled graph is an *isomorphism class* of labelled graphs and may be represented by any element of this class.

Given the number of vertices n, *labelled graphs* can easily be generated uniformly at random. For uniform generation of a labelled graph we can simply select each edge independently with probability $1/2$. With $n(n-1)/2$ processors, this takes constant parallel time. In contrast, the problem of uniformly generating *unlabelled graphs* of size n is not so straightforward. If we employ the same algorithm just described for labelled graphs and then remove vertex labels, unlabelled graphs are no longer generated with equal probability. This is because each isomorphism class is generally of different size. When all the labelled graphs are generated with an equal

probability, each unlabelled graph, as an isomorphism class of the labelled graphs, will be generated with a probability that is proportional to the number of labelled graphs in the class.

Two standard sequential approaches can be used to solve the problem of generating unlabelled graphs uniformly at random (see [13] for a survey). One is based on counting formulas using either recurrence relations or asymptotic enumeration. The other is based on the simulation of a suitable Markov chain. In this paper we follow Dixon and Wilf who use the first approach. So the problem of counting unlabelled graphs is essentially the problem of finding the number of distinguished representatives of isomorphic labelled graphs.

Isomorphic labelled graphs form equivalence classes (*called orbits*) under the action of the permutation group on the vertex labels. Burnside's Lemma [4, 3] can be used to express the number of orbits in terms of the number of objects which are fixed by permutations in the group at hand. We briefly discuss the three well-known facts in group theory [14, 10, 11] as follows:

Let $\mathcal{G}$ be a finite group acting on a finite non-empty set Ω. The equivalence relation $\alpha \sim \beta$ $(\alpha, \beta \in \Omega)$ holds iff $\alpha = g\beta$ for some $g \in \mathcal{G}$ and the elements of Ω are partitioned into equivalence classes which are called "orbits of Ω under $\mathcal{G}$". For each $g \in \mathcal{G}$, define $Fix(g) = \{\alpha : g\alpha = \alpha\}$, namely the set of elements fixed by the action of g.

(F1) (Fröbenius-Burnside lemma) The number of orbits: $m = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} |Fix(g)|$.

(F2) For each orbit ω, $|\{(g, \alpha) \in \mathcal{G} \times \Omega : \alpha \in \omega \cap Fix(g)\}| = |\mathcal{G}|$.

(F3) If g, g' lie in the same conjugacy class C of $\mathcal{G}$ then for each orbit ω, $|Fix(g') \cap \omega| = |Fix(g) \cap \omega|$. In particular, $|Fix(g')| = |Fix(g)|$.

Now let Ω be the set of all labelled graphs with n vertices and $\mathcal{G} = S_n$ is the permutation group. The action of $\mathcal{G}$ on Ω is then to relabel the n vertices of every graph $G \in \Omega$ according to each permutation $g \in \mathcal{G}$.

Suppose that we could construct a multi-set S of labelled graphs such that each isomorphism class (i.e.orbit) had the same number of representatives within S. The problem of generating an isomorphism class uniformly at random would then be solved by uniformly choosing an element G from S and then ignoring the labels on G. A set such as S can be constructed as follows: Given Ω, the set of all graphs on n vertices, successively operate on Ω with the elements of S_n and place successively found fixed elements into S. That S would now contain exactly the same number of representatives for each orbit follows from fact (F2) above.

The algorithm just described for uniformly generating an unlabelled graph, although intrinsically simple, is very inefficient. This is because the

cardinality of S_n is exponentially large in the problem size n. This, in turn, provides a sequential running time that is exponential in n. In order to provide a algorithm with an $O(n^2)$ *expected* running time, Dixon and Wilf achieve the same effect by randomly choosing (with the right probability) *one* element of S_n by using fact (F3).

So what they actually do is randomly choose a conjugacy class with weighted probability, and then choose an element within this class with uniform probability. The *weight*, w_j, of a conjugacy class C_j is defined to be the number of fixed elements induced by numbers of C_j when operating on Ω. Thus: $w_j = |Fix(g_j)||C_j|$ where $|C_j|$ is the cardinality of C_j and g_j is any representative of C_j. Note that (F3) allows w_j to be written in this way because it shows that w_j is independent of the choice of g_j. We choose any conjugacy class C_j with probability $Pr(j) = w_j / \sum_{C_i \in \alpha} w_i$, where α is the set of all conjugacy classes. Now $\sum_{C_i \in \alpha} w_i$ is just $|S|$ of our crude algorithm so that, using fact (F1) above, we have $Pr(j) = w_j / m|\mathcal{G}|$. This choice of $Pr(j)$ makes the probability $Pr(w)$ of choosing any orbit w uniform, because $Pr(w) = Pr(j)\frac{|\mathcal{G}|}{w_j} = \frac{1}{m}$, where $\frac{|\mathcal{G}|}{w_j}$ is simply the proportion of fixed elements obtained from C_j that are in w.

3 Dixon and Wilf's Algorithm

Let Ω be the set of all the labelled graphs on n vertices on which the permutation group S_n acts. Let $C_1, C_2, .., C_r$ be the conjugacy classes of S_n and g_j be a representative of C_j $(j = 1, .., r)$. The following algorithm generates an unlabelled graph G with n vertices uniformly at random [1].

Algorithm 1 *RANDGRAPH*

1 **Choose** *a conjugacy class C_j of S_n with probability* $Pr(j) = \frac{|C_j||Fix(g_j)|}{m|S_n|}$, $1 \leq j \leq r$
2 **If** *C_j was chosen in step (1)* **then choose** *uniformly at random an element $G \in Fix(g_j)$*
3 **Output** *the graph G without vertex labels*

The correctness of Algorithm 1, in terms of the uniformity of its generating unlabelled graphs, follows immediately from the group-theoretic considerations of the previous section. We now establish the complexity parameters that [1] obtains for the sequential version of the algorithm. In what follows, we will largely follow [1] but add variations that will aid parallelization of the algorithm in the later section.

3.1 Implementation

We first consider the implementation of Step (1) of Algorithm 1. Let $p(n)$ be the number of partitions of the integer n. From elementary graph theory,

two permutations are conjugate in S_n if and only if they have the same cycle structure (also called cycle pattern). In turn, there is a one-to-one correspondence between the cycle structure of elements in S_n and *partitions* of n. A partition $x_1x_2..x_j$ of n is defined such that $\sum_{i=1}^{j} x_i = n$. But $x_1x_2..x_j$ might also represent a cycle structure in standard representation of a permutation. The implementation of Step 1 conveniently concentrates on the generation of a partition of n. We want to avoid the pretabulation of probabilities of all partitions because $p(n)$ grows about as fast as $\exp(K\sqrt{n})$ with n, for some constant K.

A result by Oberschelp in [8] shows that a few partitions of n, in fact those with many parts of size 1, contribute heavily to the total weight; the other partitions contribute very little. This was shown by a result obtained when deriving an asymptotic formula for g_n, the number of unlabelled graphs on n vertices. (see [8] by Oberschelp and p196-198 of [5]). Following [8], let

$$g_n^{(k)} = \sum_{n-k} w(\pi)/n!$$

where $w(\pi)$ is the weight of the partition π, and the sum is over those partitions π of n with exactly $n-k$ parts of size 1. It is easy to see that $g_n^{(0)} = 2^{\binom{n}{2}}/n!$, $g_n^{(1)} = 0$ (for $n > 1$), and $\sum_k g_n^{(k)} = g_n$. Oberschelp proves that:

$$\sum_{k=r}^{n} g_n^{(k)} = g_n^{(0)} O(n^r 2^{-(nr/2)}) \qquad (r = 0, 1, .., n)$$

Suppose that for each n we fix an ordering $\pi_1, \pi_2, .., \pi_{p(n)}$ of the partitions of n such that for each i, π_i has at least as many parts of size 1 as π_{i+1} does. Define $t_n(\xi)$, for each real ξ, $0 \le \xi \le 1$, as the least integer t such that $\sum_{i=1}^{t} Pr(\pi_i) > \xi$. Finally let $\overline{t_n}$ be the average of $t_n(\xi)$, then Oberschelp's result shows that

$$\sum_{i=2}^{p(n)} Pr(\pi_i) = \sum_{k=2}^{n} g_n^{(k)}/g_n = O(n^2 2^{-n})$$

So $1 \le \overline{t_n} = \sum iPr(\pi_i) \le 1 + p(n)O(n^2 2^{-n})$. Therefore $\overline{t_n} \to 1$ as $n \to \infty$. In particular $\overline{t_n} = O(1)$.

Although this result uses only the fact that π_1 is the partition with all parts of size 1, Dixon and Wilf observed that if the partitions are ordered as described above, then the bound on $\overline{t_n}$ can be made more precise, and direct calculation shows that $\overline{t_n} < 3$ for all n.

We can now describe a top-level, coarse version of randomly generating a partition of n (equivalent to Step 1 of Algorithm 1). The above analysis shows that this algorithm will, on average, terminate after *at most two* partitions have been generated.

Algorithm 2 Generation of a Partition of n: coarse version

```
1  choose a random number ξ, 0 ≤ ξ ≤ 1.
2  for k = 0 to n do
3     for each partition π of n such that π has exactly n − k parts of size 1, do
4        if the total probability of all partitions so far seen > ξ, then exit with π
5        else do next π.
```

We need to refine Algorithm 2 by carefully describing the successive generation of partitions. To produce all partitions π of n with $n-k$ parts of size 1, we produce all partitions of k with no parts of size 1 and adjoin $n-k$ 1s. To produce all partitions of k with no parts of size 1, we generate the partitions of $k-j$ into j parts and add 1 to every part, for each $j = 1, .., k/2$. As we now describe, we depart from [1] in generating partitions of a given number into a fixed number of parts.

An algorithm for generating the partitions of n into exactly m parts is described in [7]. For fixed m, the partitions are generated in lexicographic order. The algorithm is an old one and was discovered by K.F.Hindenburg in 1778 (cf.[9]). We start with an initial partition of $m-1$ parts of size 1 with one part (at the right most position in the standard representation) of size $n-m+1$. To obtain the next partition from the current one, the elements are scanned from right to left, stopping at the rightmost x_i such that $x_m - x_i \geq 2$. Replace x_j by $x_j + 1$ for $j = i, i+1, .., m-1$ and then replace x_m by the remainder $n - \sum_{j=1}^{m-1} x_j$. For example, in the partition 11334, $i = 2$ and the next partition is 12225. This detail is encoded in Algorithm 3.

Algorithm 3 Generation of a Partition of n: fine version

```
Input:  integer n
Output: a partition of n stored in T chosen uniformly at random
1  generate a random number ξ, 0 ≤ ξ ≤ 1, SumProb ← 0
2  for i = 1 to n do T[i] ← 1
3  calculate Prob(partition), SumProb ← SumProb + Prob(partition)
4  if SumProb > ξ then output the partition and stop
5  for k = 2 to n do
     begin {Generate partitions with n − k parts of size 1}
6    for i = 1 to n − k do T[i] ← 1
7    for j = ⌊k/2⌋ downto 1 do
       begin {Generate partitions of k − j with j parts by Hindenburg's method }
8      for i = n − k + 1 to n − k + j − 1 do T[i] ← 2
9      T[n − k + j] ← k − j + 2
10     calculate Prob(partition), SumProb ← SumProb + Prob(partition)
11     if SumProb > ξ then output the partition and stop
12     finish ← 0
13     repeat     for i = n − k + 1 to n − k + j do T[i] ← T[i] − 1
14       p = n − k + j − 1
15       while T[n − k + j] − T[p] ≤ 2 and p > n − k do p ← p − 1
16       if p − 1 > n − k then
17         begin     T[p] ← T[p] + 1
18         for q = p + 1 to n − k + j − 1 do T[q] ← T[p]
19         s ← 0, for q = n − k + 1 to n − k + j − 1 do s ← s + T[q]
20         T[n − k + j] ← k − s, for i = n − k + 1 to n − k + j do T[i] ← T[i] + 1
21         calculate Prob(partition), SumProb ← SumProb + Prob(partition)
22         if SumProb > ξ then output partition and stop
```

```
            end
23        else finish ← 1
24      until finish = 1
        end
    end
```

We need to derive the running time of Algorithm 3 which uniformly at random generates a partition of n. So far we have not considered how to compute the probability of choosing a partition (lines 3, 10 and 21) which represents a conjugacy class C_j. Recall from line 1 of Algorithm 1 that we choose C_j with probability

$$Pr(j) = \frac{|C_j||Fix(g_j)|}{m|S_n|}$$

Now [1]:

$$|C_j| = \frac{n!}{\prod_i (i^{k_i} k_i!)} \text{ and } |Fix(g_j)| = 2^{c(g)}$$

where $c(g)$ is the number of *edge* cycles induced by the vertex permutation g_j. For example, if g_j is (1)(2 3), then $c(g) = 2$ corresponding to the edge cycles (1 2) → (1 3) → (1 2) and (2 3) → (2 3). We note that $|S_n| = n!$ and $m = g_n$, the number of unlabelled graphs on n vertices. For the moment, we shall assume that the universal constant g_n is pre-computed and therefore known, but we return to this question in the next section. We now have that:

$$Pr(j) = \frac{2^{c(g)}}{g_n \prod_i (i^{k_i} k_i!)}$$

From [1]:

$$c(g) = \frac{1}{2}\{\sum_{i=1}^{n} (l(i))^2 \phi(i) - l(1) + l(2)\}$$

where $l(i) = \sum_{i|j} k_j$, the conjugacy class (partition of n) has k_j parts of size j and the sum is over all j that are divisible by i. Also $\phi(i)$ is the Euler phi-function, that is the number of positive integers that are less than and *co-prime* to i, i.e.

$$\phi(i) = |\{n : 1 \leq n < i \text{ and } gcd(n, i) = 1\}|$$

We need to pre-compute, for $1 \leq i \leq n$:

$$\phi(i) = i \prod_{p|i} (1 - p^{-1}) \tag{1}$$

where the product is taken over all primes p that divide i. The identity (1) is provided by [6].

For given i, we can compute the primes less than i in $O(\log i)$ time using $O(i/\log i)$ processors [12]. The subset of primes that divide i can then be found easily with $(i/\log i)$ processors in a further $O(\log i)$ time. For all $1 \leq i \leq n$ we can therefore pre-compute the $\phi(i)$'s in $O(\log n)$ parallel-time using $\sum_{i=1}^{n} i/\log i = O(n^2)$ processors. It follows that for any conjugacy class, $Pr(j)$ can be computed in $O(n^2)$ sequential time. Therefore, *on average* (since only a constant number of partitions need to be computed on average), the total time for computing the probabilities with which partitions are chosen is $O(n^2)$. It is easy to see now that overall, the *expected* running time of Algorithm 3 is also $O(n^2)$.

From Step 1 we have a partition in standard representation $i_1 i_2 .. i_j$ from which we must construct a *representative* of the corresponding conjugacy class. Such a representation can be obtained by writing down the symbols $1, 2, .., n$ and then inserting brackets so as to obtain the correct sequence of cycle lengths. For example the partition 11134 would yield the representative

$$(1)(2)(3)(4\ 5\ 6)(7\ 8\ 9\ 10).$$

For computational purposes, a more convenient representative would be to record, for each $1 \leq i \leq n$, the next integer in its cycle. For our example, this gives 1 2 3 5 6 4 8 9 10 7 where the ith position records the integer following i on the cycle. In general, this may be achieved by the following linear time code in which $REP[a]$ stores the integer following a in its cycle:

```
Algorithm 4
1  a ← 1
2  for b = 1 to j do
       begin
3      if i_b = 1 then begin REP[a] ← a, a ← a + 1 end
4      else
5        begin      d ← a
6        for c = 1 to i_b − 1 do begin REP[a] ← a + 1, a ← a + 1 end
7        REP[a] ← d, a ← a + 1
         end
       end
```

In order to choose a graph uniformly at random from $Fix(g)$, we compute the edge cycles associated with g and, for each cycle independently, choose with probability $1/2$ whether all or none of the edges of that cycle will appear in H, the edge set of the graph generated. In this way each graph in $Fix(g)$ has equal probability of being chosen. We now use the array REP to compute the edge cycles induced by the vertex permutation. In the process we construct the adjacency matrix A of the graph constructed. The following code achieves this in $O(n^2)$ time because each location $A[i, j]$ is visited not more than twice.

```
Algorithm 5
1  for i = 1 to n do
2      for j = 1 to n do A[i, j] ← *
```

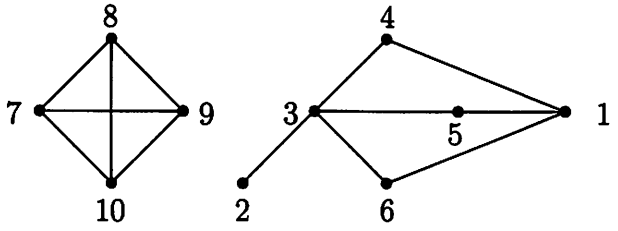

Figure 1: The graph generated

```
3  for i = 1 to n do
4    for j = 1 to n do if A[i,j] = * then
       begin
5      if i = j then A[i,j] ← 0
6      else begin
7           choose z to be 0 or 1 uniformly at random
8           x ← i, y ← j, A[i,j] ← A[j,i] ← z, i ← REP[i], j ← REP[j]
9           while not (i = x and j = y) do A[i,j] ← A[j,i] ← z, i ← REP[i], j ← REP[j]
            end
       end
```

We can illustrate the implementation of step 2 with the following representative of a conjugacy class: 1 2 3 5 6 4 8 9 10 7. The edge cycles induced by the vertex permutation are as follows, where the subscript indicates a random choice that the cycle is in (1) or not in (0) the graph: $\{(1\,2)\}_0$, $\{(1\,3)\}_0$, $\{(2\,3)\}_1$, $\{(1\,4)(1\,5)(1\,6)\}_1$, $\{(2\,4)(2\,5)(2\,6)\}_0$, $\{(3\,4)(3\,5)(3\,6)\}_1$, $\{(1\ 7)(1\ 8)(1\ 9)(1\ 10)\}_0$, $\{(2\ 7)(2\ 8)(2\ 9)(2\ 10)\}_0$, $\{(3\ 7)(3\ 8)(3\ 9)(3\ 10)\}_0$, $\{(4\,5)(5\,6)(4\,6)\}_0$, $\{(4\,7)(5\,8)(6\,9)(4\,10)(5\,7)(6\,8)(4\,9)(5\,10)(6\,7)(4\,8)(5\,9)(6\,10)\}_0$ $\{(7\ 8)(8\ 9)(9\ 10)(7\ 10)\}_1$, $\{(7\ 9)(8\ 10)\}_1$. The graph generated is shown in Figure 1.

3.2 Dixon and Wilf's Result

The foregoing Implementation Section has shown that steps 1 and 2 of Algorithm 1 may be made to run in $O(n^2)$ sequential expected time. Step 3 trivially takes $O(n^2)$ time. Thus provided (discussion of implementing Step 1 of the algorithm) the universal constant g_n, the number of unlabelled graphs on n vertices, is known, we have the following theorem.

Theorem 3.1 *(Dixon and Wilf) There exists an $O(n^2)$ time sequential algorithm to generate the unlabelled graphs on n vertices uniformly at random.*

From a purely practical point of view, we may consider that efficient computation of g_n is not a problem. As [1] points out, the formula of

Oberschelp (given earlier) for g_n will compute g_n in $O(1)$ time provided that a *fixed* accuracy (say D digits) is all that is required. Within the computation we have to take successive partitions in the same order that make Algorithm 3 efficient. Tiny errors in the probability of graphs will then be of no consequence. From a theoretical point of view, it is not known if there is a polynomial time algorithm to compute g_n exactly.

4 Parallel Algorithm for Uniform Generation of Unlabelled Graphs

In this section we establish the existence of a Las Vegas style RNC algorithm for the uniform generation of unlabelled graphs by parallelising Algorithm 1 of the previous section.

Consider Algorithm 3 which realises Step 1 of Algorithm 1. This algorithm merely finds a constant number of partitions of n and keeps a running total of the probabilities with which they should be chosen. It takes $O(n^2)$ time to compute such a probability and $O(n)$ time to generate each successive partition from the previous one. The bulk of the code of Algorithm 3 is just to ensure that partitions are generated in the right order so that on average only a constant number of them are required. In order to establish polylogarithmic running time for Algorithm 1 using a polynomial number of processors it is therefore sufficient to achieve these parameters for the computation of a single probability and for the generation of one partition from another.

Recall that there exists a parallel algorithm running in $O(\log n)$ time with $O(n/\log n)$ processors [12] that can evaluate the Euler phi-function $\phi(i)$ for all $1 \leq i \leq n$. We therefore pre-compute these $\phi(i)$ in readiness for the computation of $c(g)$ and then the probability $Pr(j)$ defined in the previous section. For fixed i, we can compute the $l(i)$ contained in the expression for $c(g)$ using the standard balanced binary tree technique [2]) in $O(\log n)$ time with $n/\log n$ processors. For all i we therefore need $O(\log n)$ time with $n^2/\log n$ processors to compute the $l(i)$. With the $\phi(i)$ and $l(i)$ pre-computed one more application of the balanced binary tree technique will evaluate $c(g)$ in a further $O(\log n)$ time with $n/\log n$ processors. Similar computations will compute $\prod_i (i^{k_i} k_i!)$ and $2^{c(g)}$ in $O(\log n)$ time with $n/\log n$ processors. It follows that overall, $Pr(j)$ may be computed by a CREW PRAM in $O(\log n)$ time with $n^2/\log n$ processors.

Now consider the generation of one partition from another. It is in fact very easy to efficiently parallelise this operation. We need just consider how Hindenburg's algorithm generates a next partition of n into m parts from a similar partition. If the current partition $x_1 x_2 .. x_m$ is stored in an array X, $X[i]$ containing x_i, then the first task is to find the largest i such

that $x_m - x_i \geq 2$. This is easily achieved by assigning a single processor to each $1 \leq i \leq m$ which then places a 0 in $Y[i]$ and overwrites this with i iff $x_m - x_i \geq 2$. This takes constant time with $m < n$ processors, or we can reschedule the work on $m/\log n < n/\log n$ processors in $O(\log n)$ time. The x_i with largest i such that $x_m - x_i \geq 2$ is then at the address in X corresponding to the largest number stored in array Y. The maximum of a set of m numbers is found in $O(\log m) < O(\log n)$ time using $m/\log n < n/\log n$ processors by the standard balanced binary tree technique. Let x_j denote the corresponding element in the partition. Hindenburg's algorithm now requires the following assignments to generate the new partition:

for *all* i, $j \leq i \leq m-1$ **in parallel do** $x_i \leftarrow x_j + 1$
$x_m \leftarrow n - \sum_{i=1}^{m-1} x_i$

The summation in the assignment to x_m can be evaluated in $O(\log m) < O(\log n)$ time with $m/\log n < n/\log n$ processors using the balanced binary tree technique. The assignments to x_i, $j \leq i \leq m-1$ may be scheduled to run on the same number of processors in the same time. Overall we, therefore, have the following lemma:

Lemma 4.1 *Step 1 of Algorithm 1 can be realised on a CREW PRAM in* $O(\log n)$ *time using* $n^2/\log n$ *processors.*

Now consider the parallel implementation of Algorithm 4 which generates a suitable representative (in the array REP) of a representative of the conjugacy class corresponding to a randomly generated partition. If the partition is $x_1 x_2 .. x_j$ stored in array $X[1..j]$, then the following code generates REP (in the following algorithm, array $Z[i]$ stores the index of the cycle that i belongs to):

1 *perform a parallel prefix computation on the* x_i, *placing* $\sum_{i=1}^{k} x_i$, $1 \leq k \leq j$, *in* $Y[k]$.
2 **for** *each* i, $1 \leq i \leq n$ **in parallel do**
3 *find* $Z[i]$ *such that* $Y[Z[i]-1] < i \leq Y[Z[i]]$
4 **if** $X[Z[i]] = 1$ **then** $REP[i] \leftarrow i$
5 **else if** $i = Y[Z[i]]$ **then** $REP[i] \leftarrow Y[Z[i]-1] + 1$
6 **else** $REP[i] \leftarrow i + 1$

Here Step 1 takes $O(\log n)$ time using $n/\log n$ processors of a CREW PRAM by the standard prefix sum algorithm, see [2]. Step 2 takes a constant time with n processors or, by rescheduling, $O(\log n)$ time on $n/\log n$ processors.

To complete the description of the parallel implementation of Algorithm 1 we must parallelise the action performed by Algorithm 5. That is, we must generate the adjacency matrix A of the graph produced by the algorithm from the array REP. The method we adopt is different from that of Algorithm 5.

For the matrix elements $A[i,j]$, $i < j$, we first assign the edge following (i,j) on the edge cycle induced by the corresponding vertex permutation.

If (k,l) is this edge then we adopt the convention of writing the edge such that $k < l$. In this way, if we trace edge cycles within the adjacency matrix, then we stay within the upper right triangular portion of the matrix. The following assignments achieve our objective:

for *all* (i,j), $2 \leq i \leq n$, $i \leq j \leq n$ **in parallel do**
  $A[i,j] \leftarrow$ **if** $REP[i] < REP[j]$ **then** $(REP[i], REP[j])$ **else** $(REP[j], REP[i])$

This takes constant time with $O(n^2)$ processors or $O(\log n)$ time with $n^2/\log n$ processors. Each $A[i,j]$ now contains a pointer to the next edge in the edge cycle and a number of edge cycles are formed. We then assign the value of 0 or 1 uniformly to each such cycle and replace the corresponding location on a cycle with the cycle value. As we shall see, this can be done efficiently by the pointer doubling method [2]. We proceed in two stages:

1. Each location $A[i,j]$ needs to acquire an additional pointer such that every location on the *same* cycle has a pointer to the *same* location. This location can conveniently be that location on the cycle which has the lexicographically smallest address (i,j). For any (i,j) denote this by $\overline{(i,j)}$. Initially set $(i,j) \leftarrow *$ for all $2 \leq i \leq n$, $i \leq j \leq n$. Now each time we double pointers we compare the current value of $\overline{(i,j)}$ with that of the item pointed to and update $\overline{(i,j)}$ if a smaller value is found. After $O(\log n^2) = O(\log n)$ time using $O(n^2)$ processors, each edge (i,j), $i < j$, will have (perhaps indirectly) "seen" every other edge on its cycle and will have found its $\overline{(i,j)}$ value.

2. On each edge cycle there will be a unique edge such that $\overline{(i,j)} = (i,j)$. If we assign a processor to each (i,j) we then execute:

   for *all* (i,j), $2 \leq i \leq n$, $i \leq j \leq n$ **in parallel do**
     if $\overline{(i,j)} = (i,j)$ **then** *uniformly at random assign* 0 *or* 1 *to* $A[i,j]$

   We now copy the value of $A[i,j]$ assigned to in the last step to every other edge on the same cycle:

   for *all* (i,j), $2 \leq i \leq n$, $i \leq j \leq n$ **in parallel do**
     if $\overline{(i,j)} \neq (i,j)$ **then** $A[i,j] \leftarrow A[\overline{i,j}]$

   Notice that this involves *concurrent reads*, so that our PRAM model is the CREW variant.

The rest of the elements of the adjacency matrix are assigned to as follows:

for *all* (i,j), $1 \leq i \leq n$, $1 \leq j \leq i$ **in parallel do**
  if $i = j$ **then** $A[i,j] = 0$ **else** $A[i,j] \leftarrow A[j,i]$

This takes constant time with $O(n^2)$ processors or $O(\log n)$ time with $n^2/\log n$ processors. Taking the most work costly stage in the above description we have the following Lemma.

Lemma 4.2 *Step 2 of Algorithm 1 can be realised on a CREW PRAM in $O(\log n)$ time using n^2 processors.*

Combining the last two lemmas we can now state the main result of this paper.

Theorem 4.1 *There exists an algorithm running in $O(\log n)$ expected time using n^2 processors of a CREW PRAM which uniformly at random generates an unlabelled graph on n vertices from the set of all such graphs.*

5 Summary and Open Problems

The main result of this paper is the design of a Las Vegas style algorithm which runs in $O(\log n)$ *expected* time on a CREW PRAM using n^2 processors which uniformly generates an unlabelled graph from the set of all such graphs on n vertices. The work measure of our parallel algorithm is a logarithmic factor larger than the sequential algorithm of Dixon and Wilf. Whilst this is no great penalty to pay to get efficient parallel computation it does pose the question as to whether there is an *optimal* parallel algorithm for this problem.

A problem closely related to the one solved in this paper is the uniform generation at random of *connected* graphs on n vertices efficiently either by a sequential or a parallel algorithm (of course, the former would follow from the later). In fact the algorithm of Dixon and Wilf and our parallelisation of it provide solutions to this problem. We can run either algorithm repeatedly until a connected graph is produced. There are well-known algorithms for testing connectivity that run in $O(n^2)$ sequential time or $O(\log^2 n)$ parallel time on a CREW PRAM using $n^2/\log^2 n$ processors. Since the proportion of connected graphs in the set of all unlabelled graphs is always at least $1/2$ we will need, on the average, to test at most two graphs. Thus we have an algorithm running in $O(\log^2 n)$ parallel time on a CREW PRAM using $n^2/\log^2 n$ processors which uniformly generates *connected* graphs on n vertices.

One interesting problem is that of generating uniformly at random the graphs on n vertices with *exactly* e edges. It is an open question as to whether the methods described in this paper would provide an effective solution either by sequential or by parallel computation.

References

[1] J.D. Dixon and H.S. Wilf. The random selection of unlabelled graphs. *Journal of Algorithm*, 4:205–213, 1983.

[2] A. Gibbons and P. Spirakis. *Lectures in Parallel Computation.* Cambridge University Press, 2005.

[3] C. Godsil and G. Royle. *Combinatorial Theory, Martin Aiger-Mathematics.* Springer, 2004.

[4] J. Gross and J. Yellen. *Graph Theory and Its Applications.* CRC Press, 1999.

[5] F. Harary and E.M. Palmer. *Graphical Enumeration.* Academic Press New York and London, 1973.

[6] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* Oxford, 5th edition, 1979.

[7] L.D.H. Lehmer. *Applied Combinatorial Mathematics*, chapter The Machine Tools of Combinatorics. Beckenback (ed.), Wiley NY, 1964.

[8] W. Oberschelp. Kombinatorische anzahlbestimmungen in relationen. *Mathematics Annul*, 174:53–58, 1967.

[9] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[10] J.J. Rotman. *The Theory of Groups - an Introduction.* Allyn and Bacon, Inc., 1965.

[11] C.C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, 1994.

[12] J. Siorenson and I. Parberry. Two fast parallel prime number sieves. *Information and Computation*, 114, 1994.

[13] M.G. Tinhofer. Generating graphs uniformly at random. *Computing*, 7:235–255, 1990.

[14] D.B. West. *Introduction to Group Theory.* Prentice-Hall Inc., 2nd edition, 2001.

[15] M. Zito, I. Pu, M. Amos, and A. Gibbons. RNC algorithms for the uniform generation of combinatorial structures. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Atlanta, Georgia, 28–30 January 1996.