On the Hardness of Uniform Random Generation

David Charlton

April 30, 2004

1 Introduction

The focus of this thesis is hardness results for uniform random generation – that is, results indicating how computationally difficult it is to uniformly generate certain sets. In the process, we also discuss results regarding enumerating languages with polynomial delay.

In section 2, we resolve an open question of Goldberg's [1] by providing an algorithm for efficiently enumerating various NP-complete sets, including k-colorable graphs. We also extend a result of Goldberg's by providing evidence that uniform random generation of a language is strictly harder than efficient enumeration of that language.

In section 3, we consider more specifically the problem of uniformly randomly generating certain NP-complete sets. Assuming the hardness of the Quadratic Residue Problem, we show that a broad class of algorithms (specifically, those based on the technique of rejection sampling) are inadequate to efficiently uniformly generate known NP-complete sets.

2 Enumerating with Polynomial Delay

In this section, we introduce the notion of enumerating a language with polynomial delay. We also provide an original algorithm for enumerating some NP-complete sets with polynomial delay, as well as a hardness result contrasting efficient enumeration with random generation.

2.1 Deterministic Enumeration of NP-complete Sets

Definition 2.1.1 *For any language L on binary strings, we say L is* **enumerable with polynomial delay** *if there is a procedure taking in 1^n and outputing the strings of $L \cap \{0,1\}^n$, with at most polynomial time elapsing between each output string. For probabilistic*

enumeration, we require that every string in $L \cap \{0,1\}^n$ is output at least once with exponentially small failure probability, that no string is output more than once, and that no string not in $L \cap \{0,1\}^n$ is output.

Intuitively, though enumeration is not a decision problem, this definition corresponds to a Monte Carlo algorithm for deciding L, since if an x appears in the output then it is definitely in L, while if it does not appear in the output then it probably is not.

In [1], Goldberg provides an algorithm for deterministically enumerating all n-node Hamiltonian graphs with polynomial delay, and leaves open the question of whether there is a polynomial-delay algorithm for enumerating k-colorable n-node graphs. We resolve this question by providing a new, more general, algorithm to enumerate a variety of NP-complete sets, and show how this technique can be used to deterministically enumerate all n-node k-colorable graphs with polynomial delay. ALthough it does not come into our proof, we note that k itself need not be constant, as the performance of our algorithm depends only on n. We also describe, but do not prove in detail, how the same technique can be applied to other languages such as n-node Hamiltonian graphs or n-variable satisfiable 3SAT formulas. While our algorithm is more general than Goldberg's, it does suffer from the drawback that it explicitly requires superpolynomial space (Goldberg's algorithm can be implemented in polynomial space by integrating it with more recent Hamiltonian Path algorithms such as those found in [4]).

In section 2.1.1, we present an explicit algorithm for listing labelled k-colorable graphs. In section 2.1.2, we describe how the technique generalizes to several other combinatorial structures.

2.1.1 Deterministically Enumerating k-colorable Graphs

The intuition behind our algorithm is to examine a small subset of k-colorable graphs and then show that all k-colorable graphs are a subgraph of some element in this subset. In particular, for every possible k-coloring, we can create a graph that includes every valid edge with respect to that k-coloring (that is, a graph that includes every edge going between distinct colors). We can then output all subgraphs of that graph by proceeding recursively, removing one edge at a time.

Our general strategy, then, is to start with the "maximal" graph on each k-coloring and then proceed to generate all subgraphs on those initial graphs (using a lookup table to avoid duplicate outputs). Below we prove that every k-colorable graph is a subgraph of some such maximal graph, and that our algorithm therefore will generate all k-colorable graphs.

To ensure that our algorithm produces outputs with polynomial delay, we keep a queue of graphs to be output. Whenever we remove a graph from the queue in order to output it, we also add all of its subgraphs with one fewer edge back into the queue (if they have not

already been added) before outputting it. In this way, we can continue outputting elements from the queue with polynomial delay until it is empty. If the queue becomes empty, there are two possibilities: either our algorithm is complete, or we have outputted all subgraphs of the current maximal graph. If the latter is the case, then we move on to a new coloring and start over.

Definition 2.1.2 *A* **graph on** n **nodes** *is a pair* (V, E) *where* V *is a set of integers* $\{1 \ldots n\}$ *for some* n *(representing nodes) and* E *is a set of unordered integer pairs* $\{(a_0, b_0), (a_1, b_1), \ldots, (a_{m-1}, b_{m-1})\}$ *(representing edges), where* $\forall k < m, 1 \leq a_k, b_k \leq n$.

Definition 2.1.3 *A* **subgraph** *of a graph* $G = (V, E)$ *is a graph* $G' = (V', E')$ *such that* $V \subseteq V'$ *and* $E \subseteq E'$.

Definition 2.1.4 *An* k**-coloring on** n **nodes** *is a function* C: $\{1 \ldots n\} \rightarrow \{1 \ldots k\}$. *A coloring is* **valid** *for a graph* $G = (V, E)$ *iff for any edge* $(a, b) \in E$, $C(a) \neq C(b)$. *A coloring is* **surjective** *if it is onto its range* $\{1 \ldots k\}$, *that is, at least one node maps to every color.*

Definition 2.1.5 *A* k**-coloring class on** n **nodes** *is an equivalence class of* k*-colorings, where two colorings* C *and* C' *are considered equivalent if they are identical under permutation of colors, that is, if there exists a permutation* p: $\{1 \ldots k\} \rightarrow \{1 \ldots k\}$ *such that* $(\forall j) C(j) = p(C'(j))$.

Note that if a coloring C is valid on a graph G, then any coloring in C's equivalence class is also valid on G.

Definition 2.1.6 *The* **representative** k**-coloring** *of a* k*-coloring class* A *is the coloring* $C \in A$ *such that* $\forall x < n(C(x) = j \Rightarrow \forall i < j \exists y < x C(y) = i)$.

Informally, this is saying that in the representative k-coloring, the first occurrence of any color j is after the first occurrence of all lower-numbered colors – the first node is colored 1, the first node that isn't colored 1 is colored 2, the first node that is neither 1 nor 2 is 3, and so on.

Lemma 2.1.1 *There is exactly one representative* k*-coloring for any* k*-coloring class.*

Proof: We know that every class has at least one representative k-coloring, since we can just go through the coloring changing the labels as described above. We now show that every class has at most one representative coloring.

Suppose we have a k-coloring class A, and two k-colorings $C, C' \in A$ such that both C and C' are representative k-colorings of A. Since C and C' are in the same class, there exists some permutation p: $\{1 \ldots k\} \rightarrow \{1 \ldots k\}$ such that $(\forall j) C(j) = p(C'(j))$, so take some such p, and let j be the lowest-numbered node on which C and C' differ. We now consider two cases.

Case 1: j is the lowest-numbered node which is assigned the color $C(j)$ in C – that is, $(\forall i < j) C(i) \neq C(j)$. Then we also know that j is the lowest-numbered node which is assigned the color $C'(j)$ in C', because otherwise we would have some $i < j$ such that $C'(i) = C'(j) \Rightarrow p(C'(i)) = p(C'(j)) \Rightarrow C(i) = C(j)$, contradiction. However, if j is the first occurrence of both $C(j)$ on C and $C'(j)$ on C', then both colorings must assign the lowest-numbered unused color (by the definition of representative coloring). Since C and C' agree on all lower-numbered nodes, this color will be the same for both colorings, and $C(j) = C'(j)$.

Case 2: $(\exists i < j) C(i) = C(j)$. Then $C'(i) = C(i)$, and since $C'(i) = p(C(i))$ this implies $p(C(i)) = C(i)$. Furthermore, since $C(j) = C(i)$ we have $C'(j) = p(C(j)) = p(C(i)) = C(i) = C(j)$, and the colorings agree.

Thus, we conclude $C = C'$, and there is only one representative coloring for any k-coloring class. ■

Definition 2.1.7 *The* **maximal graph on an n-node k-coloring** C *is the n-node graph G such that $\forall a, b \in \{1, \ldots, n\}, C(a) \neq C(b) \Leftrightarrow (a, b) \in G$.*

Lemma 2.1.2 *Take a maximal graph G on an n-node surjective k-coloring C. Then all valid k-colorings on G are in the same k-coloring class as C.*

Proof: Without loss of generality, take C to be a representative k-coloring (since, as above, all elements of a class are equivalent with respect to validity on graphs, and by lemma 2.1.1 each class has exactly one representative). Now, for contradiction, suppose that $G = (V, E)$ is a maximal graph on C and that $C' \neq C$ is some representative k-coloring that is also valid on G.

Since C is surjective, we can consider the equivalence classes of nodes based on their coloring. Since C and C' are both representative colorings, we know that the equivalence classes in C' are different than those in C, and that therefore (since there are exactly k equivalence classes in C and at most k equivalence classes in C') there exist nodes j and l that are in different equivalence classes in C (i.e. $C(j) \neq C(l)$) but the same class in C' ($C'(j) = C'(l)$). But since G is maximal, this means that $(j, i) \in E$, and therefore C' is not valid on G. ■

Lemma 2.1.3 *Take any k-colorable n-node graph G with $k \leq n$. Then G is a subgraph of some maximal graph on an n-node surjective k-coloring.*

Proof: Take some k-coloring C valid on $G = (V, E)$, and construct a surjective coloring C' from C by arbitrarily setting redundant elements in C's range to unique colors so that C' is a surjective k-coloring valid on G. Now let $G' = (V', E')$ be the maximal graph on C'. Since C' is valid on G, for any $(i, j) \in E$, $C'(i) \neq C'(j)$. Furthermore, by the definition of maximal graphs, for any i, j such that $C'(i) \neq C'(j)$, $(i, j) \in E'$, and we conclude that for any $(i, j) \in E$, $(i, j) \in E'$, and since both graphs are n nodes, implying $V = V'$, we conclude that G is a subgraph of G'. ∎

Lemma 2.1.4 *There is an algorithm to iterate over all n-node surjective representative k-colorings with polynomial delay.*

Proof: We first note that a representative k-coloring can be characterized by the nodes on which a color is used for the first time. For instance, the color 1 is always used for the first time on node 1. Furthermore, for a surjective coloring, there will always be k nodes on which a color is used for the first time, and the value of the coloring on those nodes is uniquely determined for a representative coloring. It is easy to iterate over all placements for these "initial nodes" since this is the same as iterating over all ways of selecting k nodes out of n possibilities (although since color 1 always begins at node 1, we are actually looking at the number of ways to select $k - 1$ nodes out of $n - 1$ possibilities).

Now notice that for a specified placement of initial nodes out of the $\binom{n-1}{k-1}$ choices, listing all representative k-colorings within that placement is just a matter of brute force within the given framework – trying all legal values for all nodes except for the k that were selected. For example, every color up to the first j such that $C(j) = 2$ is uniquely determined. After that, every node has 2 possible colors, until $C(j) = 3$. Then there are three possible colors, and so on. Thus, we can list all possible such colorings by trying every placement of initial nodes and brute-forcing every possibility within each placement. ∎

The intuition behind our algorithm for enumerating 3-colorable graphs is that, since we can iterate over all surjective representative k-colorings, and since every k-colorable graph is a subgraph of a maximal graph on one of those colorings, we start by iterating over each coloring and outputting all subgraphs of its maximal graph, keeping track of which ones we have already output. Since each distinct coloring has at least one unique graph associated with it, we are guaranteed to output a new graph within polynomial time because if we run out of graphs in our current coloring, we just advance to the next one.

Algorithm 2.1.1 *An algorithm to deterministically enumerate all n-node k-colorable graphs with polynomial delay with respect to n.*

Input: $n, k \in \mathbb{N}$
Output: All n-node k-colorable graphs.
Variables: A queue Q of n-node graphs to output (initialized to empty), and a balanced binary tree T of n-node graphs that have already been put in the queue (also initialized to empty).

```
1    For each representative n-node k-coloring C (from Lemma 2.1.4):
2        Add the maximal graph on C to the end of Q
3        While Q is nonempty:
4            Remove G from the front of Q.
5            For each graph G' equal to G with one edge removed:
6                If G' ∉ T, add G' to T and to the end of Q.
7            Output G.
```

Theorem 2.1.5 *Algorithm 2.1.1 deterministically enumerates all k-colorable n-node graphs with polynomial delay with respect to n.*

Proof: It is easy to prove that the algorithm enumerates all k-colorable graphs: we know by Lemma 2.1.4 that we will hit all representative k-colorings, and we know by Lemma 2.1.3 that all k-colorable graphs are a subgraph of at least one of these graphs. Thus by simple induction on number of edges we see that all such graphs will eventually be output. Furthermore, our use of T ensures that each of them will be output exactly once (with the possible exception of the initial maximal graphs, addressed below). Therefore, all we need to show is that some new graph will always be output within polynomial time.

On a particular iteration of the while loop on line 3, suppose Q is nonempty. Then we execute a for loop for at most $\binom{n}{2}$ iterations (that is, at most once for each possible edge we can remove from the current graph), performing simple data structure operations (tree lookup, insert, queue insert) on each iteration, so this loop will terminate in polynomial time, producing an output on line 7. If on the other hand Q is empty, then (assuming the algorithm has not terminated) we add a maximal graph to Q, and by Lemma 2.1.2 this new graph is unique to the current coloring, and thus has not been output before. By the same logic as above, the for loop will terminate in polynomial time and this graph will be output. Therefore there is at most polynomial delay between any subsequent outputs. ■

2.1.2 Generalized Deterministic Enumeration

The general pattern our algorithm for k-colorable graphs followed was: associate some unique instance (maximal graphs) with each representative solution (k-coloring), and then show that all instances are in some way a "subset" of one or more of those instances – then we could look at all subsets of the unique instances, using a lookup table to prevent

duplicate outputs. It is possible to apply those same techniques to other problems, though we do not provide full details.

As one example, consider Hamiltonian graphs: our "solutions" could be the Hamiltonian tour itself, and we could associate "minimal" instances with each possible tour (where the graph associated with a particular tour would be the graph consisting of only the edges in that particular tour). Then every Hamiltonian graph is a supergraph of one of these unique instances. As in the previous section we could iterate over all possible Hamiltonian tours and at each step output the minimal graph on that tour as well as all supergraphs on the minimal graph (using a lookup table to prevent duplicates). As before, we are guaranteed an output in polynomial time since, if we ever run out of graphs to output, we just move on to the next tour and we are guaranteed at least one more unique output. Thus, this yields a new method of enumerating Hamiltonian graphs with polynomial delay. Such an algorithm was already provided in Goldberg [1] as mentioned above. Our algorithm suffers the drawback that it explicitly requires superpolynomial space, but has the advantage of being conceptually simpler.

As another instance, consider n-variable satisfiable 3SAT formulas. There are $2n$ possible literals in such a formula, and thus $\binom{2n}{3}$ possible clauses. We can therefore represent a 3SAT formula as a string of $\binom{2n}{3}$ bits, where a 1 means that the associated clause is present in the formula, and a 0 means it is not. Notice first of all that if a particular formula is satisfiable, then all bitwise subsets of it are also satisfiable (allowing us to perform a subset operation comparable to the one we used on graphs). Notice also that if we fix a particular truth assignment on the variables, then there is a unique maximal formula consisting of all clauses which are consistent with the chosen assignment. Furthermore, all satisfiable formulas are subsets of some such formula. Thus we can iterate over all possible satisfying assignments and output all formulas, starting with the maximal ones, by the same techniques as before, yielding a polynomial-delay algorithm for enumerating satisfiable 3SAT formulas.

We believe it is likely that several other languages could be enumerated with polynomial delay using similar methods. At the very least, this technique resolves an open question in [1] regarding the possibility of efficiently enumerating k-colorable graphs. However, it still leaves open the question of whether such languages can be enumerated with polynomial delay using only polynomial space.

2.2 Hardness of Random Generation

All procedures in this section are probabilistic Monte Carlo unless otherwise noted.

Definition 2.2.1 *For any language L on binary strings, we say L can be* **efficiently randomly generated** *if there is some polynomial-time probabilistic procedure taking in 1^n and outputting uniformly at random an element of $L \cap \{0,1\}^n$.*

Definition 2.2.2 *For any language L and any function $f\colon \mathbb{N} \to \mathbb{N}$, we say L can be* **f-approximated** *if there is some polynomial-time procedure which correctly decides membership in L for $1/2 + \frac{1}{f(n)}$ of all n-bit inputs.*

Definition 2.2.3 *For any language L and any function $f\colon \mathbb{N} \to \mathbb{N}$, we say L can be* **approximately counted to within a factor of** *f if there is a polynomial-time function taking in 1^n and returning some number in the range $[\frac{|L \cap \{0,1\}^n|}{f(n)}, |L \cap \{0,1\}^n| f(n)]$.*

In [1], Goldberg shows that any language that can be randomly generated can also be enumerated with polynomial delay. Under a hardness assumption, we show that there exists a language that can be enumerated with polynomial delay but cannot be randomly generated. Furthermore, given a slightly stronger assumption, we show that for any polynomial p there exists such a language that cannot be p-approximated or approximately counted to within a factor of p.

Definition 2.2.4 *For any complexity class C, we use C_1 to refer to the set of unary languages that are in C.*

Assumption 1 *For some polynomial p, there exists a language $K \in RTIME_1(p(n)2^n)$ such that $K \notin RP_1$.*

Theorem 2.2.1 *Given Assumption 1, there exists a language L that can be enumerated with polynomial delay but cannot be efficiently uniformly generated.*

Proof: Using Assumption 1, take some polynomial p and language K such that $K \in RTIME_1(p(n)2^n)$ but $K \notin RP_1$. We then construct a language L_K where the set of all n-bit strings in L_K is defined to be:

$$L_K \cap \{0,1\}^n = \begin{cases} \{0,1\}^n & \text{if } 1^n \in K \\ \{x \in \{0,1\}^n : x < 2^{n-1}\} & \text{if } 1^n \notin K \end{cases}$$

where the $<$ operator above is the numeric order on n-bit binary integers.

First we note that we can deterministically enumerate L_K with polynomial delay: Construct a machine that spawns two threads whose executions are interleaved. In the first thread, we determine whether $1^n \in K$, taking time polynomial in 2^n. In the second thread, we output the first 2^{n-1} n-bit binary strings, inserting a delay of $\Theta(p(n))$ steps between each output. This takes a total of $\Theta(p(n)2^{n-1}) = \Theta(p(n)2^n)$ steps. By choosing the constant factors in

our algorithms appropriately, we can be assured that the first thread's computation will be finished by the time these outputs are complete. Thus, by the time the second thread is finished, we will know whether $1^n \in K$, and may output the remaining strings or terminate depending on the result.

At the same time, L_K cannot be efficiently uniformly generated: Suppose for contradiction that we have an algorithm to efficiently uniformly generate L_K. Then if $1^n \in K$, our random generator will have probability $1 - \frac{1}{p(n)}$ on each iteration of outputting some $x \geq \frac{2^n}{p(n)}$. Conversely, if we run the random generator k times there is only a $\frac{1}{p(n)^k}$ probability that every output is $< \frac{2^n}{p(n)}$, versus a certainty if $1^n \notin K$, so we have an RP_1 decision algorithm for K, contradicting our hardness assumption. ∎

We consider Assumption 1 to be extremely plausible. Thus, due to Goldberg's result that any language that can be randomly generated can also be probabilistically enumerated, this theorem yields the conclusion that random generation is strictly harder than probabilistic enumeration.

Assumption 2 *For some polynomial p, there exists a language $K \in \mathsf{RTIME}_1(p(n)2^n)$ such that $K \notin \mathsf{BPP}_1$.*

Theorem 2.2.2 *Given Assumption 2, for any polynomial q there exists a language L_q that can be enumerated with polynomial delay but cannot be efficiently uniformly generated. Furthermore, there exists no random procedure to q-approximate L_q or to approximately count L_q to within a factor of smaller than $q^{1/2}$.*

Proof: Using the hardness assumption, take some polynomial p and unary language $K \in \mathsf{RTIME}_1(p(n)2^n)$ such that $K \notin \mathsf{BPP}_1$. We construct the language $L_{K,q}$ such that the set of all n-bit strings in $L_{K,q}$ is defined to be:

$$L_{K,q} \cap \{0,1\}^n = \begin{cases} \{0,1\}^n & \text{if } 1^n \in K \\ \{x \in \{0,1\}^n : x < \frac{2^n}{q(n)}\} & \text{if } 1^n \notin K \end{cases}$$

where the $<$ operator above is the numeric order on n-bit binary integers.

As in the previous theorem, we can deterministically enumerate $L_{K,q}$ with polynomial delay. We again execute two interleaved threads, where the first one computes whether $1^n \in K$ (taking time $O(p(n)2^n)$). Simultaneously, the second thread outputs the first $\frac{2^n}{q(n)}$ n-bit binary strings, inserting a delay of $\Theta(p(n)q(n))$ steps between each output, so that the total time consumed by the second thread is $\frac{2^n}{q(n)}\Theta(p(n)q(n)) = \Theta(p(n)2^n)$. Therefore, as

above, we know whether $1^n \in K$ by the time the second thread terminates, and can either continue outputing strings or terminate depending on that result. Furthermore, $L_{K,q}$ cannot be efficiently uniformly generated for the same reasons as L_K in the previous theorem.

Now, suppose we have a randomized (BPP) procedure q-approximating $L_{K,q}$. Then we can decide K in BPP_1.

First, consider the case where $1^n \in K$. Then our q-approximation will give correct output on at least $1/2 + \frac{1}{q(n)}$ of n-bit inputs, which means that on inputs x such that $x \geq \frac{2^n}{q(n)}$ it will give correct output at least $\frac{1/2}{1-\frac{1}{q(n)}} > 1/2 + \frac{1}{2q(n)}$ of the time, so these inputs will be accepted with probability $> 1/2 + \frac{1}{2q(n)}$.

Now consider the case where $1^n \notin K$. Then the same calculations apply, and inputs x such that $x \geq \frac{2^n}{q(n)}$ will be accepted with probability $< 1/2 - \frac{1}{2q(n)}$. Thus there is a $\frac{1}{q(n)}$ probability gap between the two cases, and we can solve K in BPP_1 by simply testing a polynomial number of random inputs in the range above $\frac{2^n}{q(n)}$ and taking a majority vote. Therefore, there is no such efficient q-approximation algorithm.

Similarly, there cannot be a probabilistic poly-time program that approximately counts $L_{K,q}$ to a better than $q(n)^{1/2}$ factor: Note that the cardinality of $L_{K,q} \cap \{0,1\}^n$ is either $\frac{2^n}{q(n)}$ or 2^n, depending on whether $1^n \in K$. Any procedure that could reach these values to within a better than $q(n)^{1/2}$ factor would be able to distinguish the two values, thus providing a probabilistic poly-time (BPP or better) method of deciding K, and we conclude that there is no such approximate counting method.

■

In closing, we note that if we make the stronger but plausible hardness assumption that there exists a language $L \in \mathsf{TIME}_1(\mathrm{poly}(n)2^n)$ such that $L \notin \mathsf{BPP}_1$, then both claims stands for deterministic as well as probabilistic enumeration by the same reductions as above.

3 Rejection Sampling

In this section, we consider the technique of **rejection sampling**, a method of transforming one random distribution into another. In rejection sampling, we start by generating some set of elements according to a distribution D such that each element x appears with some probability $D(x)$. We wish instead to output elements according to a different distribution D', and achieve this by generating outputs according to D and then "filtering" that distribution by skipping each output with some probability.

3.1 Terminology/Notation

Definition 3.1.1 *Suppose we have some set S. A* **distribution** *on S is a function D that assigns to each $x \in S$ a probability $D(x)$ such that $\sum_{x \in S} D(x) = 1$. A randomized function d is said to* **randomly generate** *S* **according to distribution** *D if for any $x \in S$, $P(d() = x) = D(x)$. We are particularly interested in the* **uniform distribution**, *in which for all $x \in S$, $D(x) = \frac{1}{|S|}$.*

Definition 3.1.2 *Suppose we have a function $f\colon S \to T$ for sets S, T, and a distribution D on S. We use the notation D_f to indicate the distribution on T such that for any $x \in T$, $D_f(x) = \sum_{\{y|f(y)=x\}} D(y)$.*

Intuitively, D_f is the distribution produced by applying the function f to the outputs in the distribution D.

Convention: We will often speak of "distributions" on an infinite set S (such as the set of all graphs). When we say this, it should be understood that we actually mean a sequence of distributions on finite subsets of S. For example, if S is the set of all graphs then we might have the "distribution" $D = \{D_n\}$ such that D_n is a distribution on the set of all n-node graphs.

We model rejection sampling as follows: We treat our input distribution D as a black box B that takes in an integer n and outputs elements of size n from some set such that the probability of getting a particular output x is $D(x)$. (Our definition of "size n" is flexible, allowing for anything polynomially related to n, such as an n-node graph or a 3SAT formula on n variables).

Given such a black box B, we model our rejection sampling as a filtering algorithm which we construct. This algorithm takes some output x from B and outputs either x (if this value is appropriate for our desired output distribution), or "try again", in which case we must try again with another output from B. For output distribution D', the filtering must proceed such that for any output that our filter allows through, the probability of getting a particular x is $D'(x)$.

Finally, we note that the output from our box can be transformed after the fact. For example, suppose our input distributions includes pairs of 3-colorable graphs and valid 3-colorings on those graphs, but we want our output distribution to consist only of the graphs themselves, with no associated coloring. Then we can compose our rejection sampling algorithms with a simple function f that removes the attached coloring from each pair. Therefore we allow simple transformations of this kind to occur after the filtering step where it is clear that they do not fundamentally alter the character of the distribution (see Figure 1).

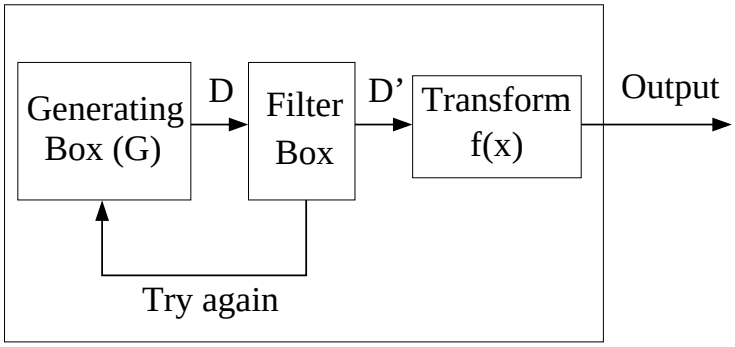

Figure 1: Producing an output distribution through rejection sampling. We are particularly interested in the technique of constructing a filter box.

3.2 3-colorable graphs

In this section, we use the same definitions and terminology for graphs and colorings as in section 2.1.1 above.

Rejection sampling is most effective in situations where the input distribution can be efficiently calculated. In particular, it is easiest to do rejection sampling when we know the frequency of arbitrary inputs and the desired frequency of arbitrary outputs. However, such simple methods will not work on some sets – for example, suppose our input distribution is uniform over all pairs (C, G) such that G is an n-node 3-colorable graph and C is a valid coloring on that graph, and suppose we want to uniformly output all 3-colorable graphs. Then each graph appears in the input with frequency proportionate to the number of colorings on that graph. Calculating this frequency is #P-complete, so if we want to maintain the rejection sampling model then we must use other methods. In this section, we provide formal evidence that there is no efficient rejection sampling algorithm to produce the uniform distribution on 3-colorable graphs from the uniform distribution on (C, G) pairs.

Definition 3.2.1 *We denote the uniform distribution on (C, G) pairs (as described above) as $\mathcal{D}$.*

Definition 3.2.2 *We denote the function which takes in a pair (C, G) and returns G as $\mathfrak{g}$.*

$\mathfrak{g}$ is just a function that allows us to transform a distribution on graph/coloring pairs into a distribution on graphs.

Although we do not provide a proof, we note that there is an efficient algorithm to generate $\mathcal{D}$, and that despite its non-uniformity on the 3-colorable graphs themselves (i.e. $\mathcal{D}_{\mathfrak{g}}$ is not uniform) this algorithm is arguably the easiest way to produce a "random 3-colorable graph".

Definition 3.2.3 *For any graph G, we define the variable χ_G to be the number of valid 3-coloring classes on G.*

Informally, we will refer to χ_G below as the "number of 3-colorings" on a graph, even though strictly speaking it is not. For example, we will refer to graphs which have "only one coloring" or "only four colorings", even though both of these statements are impossible (since, because of permutation of colors, the actual number of 3-colorings will always be a multiple of 3). When we make these statements, it should be understood that we are actually talking about the number of 3-coloring classes, that is, χ_G. This informality is merely shorthand, and does not affect the generality of our results.

Definition 3.2.4 *Given a particular rejection sampling algorithm on graphs, we define the variable ψ_n to be the highest probability with which any n-node graph is accepted by the algorithm.*

Theorem 3.2.1 *There is no efficient rejection sampling algorithm that takes in an input distribution of $\mathcal{D}_{\mathfrak{g}}$ and yields a uniform distribution over all 3-colorable graphs unless NP $\subseteq$ BPP.*

Proof: We show that if there exists such an algorithm, then we can use it to solve the 3-coloring problem in BPP. Suppose we have such an algorithm A. By running A repeatedly on a particular graph G, we can calculate the approximate probability with which G is accepted. We cannot determine the exact ratio for graphs with many colorings, but for any polynomial $p(n)$ we can distinguish between $\frac{1}{p(n)}$ and $\frac{1}{2p(n)}$, which is all we need for the purposes of our reduction.

Model the "efficient algorithm" that performs the rejection sampling as a black box B. Note that B will need to accept a particular n-node graph G with probability proportionate to $\frac{1}{\chi_G}$. Thus, the graph that is most likely to be accepted is one with exactly one coloring (which is accepted with probability $\psi_n \geq \frac{1}{\text{poly}(n)}$, as mentioned above).

Suppose we input into B an n-node graph G that has exactly one coloring (It is easy to construct such a graph – pick any coloring C on n nodes that contains three distinct colors, and construct G such that there is an edge between every pair of nodes with distinct colors). Since this graph is accepted with polynomial probability, we can estimate the probability ψ_n of acceptance to any polynomial factor. We will choose to estimate ψ_n (with high probability) to within a $\pm 1/8$ factor.

Note now that we can perform a similar estimate on any other 3-colorable graph (although our box's behavior is undefined on a graph that is not 3-colorable). In particular, note that for any graph G for which $\chi_G \geq 2$, the probability that G is accepted by B is $\leq \frac{\psi_n}{2}$. Thus,

given such a graph we can estimate its probability of acceptance, also to within a $\pm 1/8$ factor, and if there is more than a $1/4$ distance between that and our estimate of ψ_n, we can say with significant probability that G has more than one coloring.

Now, given *any* graph G on which we wish to solve the 3-coloring decision problem, we proceed as follows: First, reduce G to a circuit X whose satisfying assignments correspond to the valid colorings of G. Then, create a circuit X' that is true if and only if X is true on the given inputs, or the inputs are all zeroes (note that we can easily check whether all zeroes satisfied the original circuit, and if so, we simply accept G). X' now has > 1 satisfying assignment if and only if there is a valid 3-coloring on G. We therefore parsimoniously reduce X' back to a graph, G', which is guaranteed to have at least one valid 3-coloring, and which has more than one if and only if G has at least one. Since (as described above) we can use B to distinguish whether G' has one valid coloring or more than one, we can with high probability distinguish whether G has a coloring, which yields an efficient algorithm for 3-coloring in BPP. ∎

Theorem 3.2.2 *Let Gr_n be the number of 3-colorable graphs on n nodes. Consider the distribution D_{col} such that $D_{col}(C,G) = \frac{1}{Gr_n \chi_G}$. There is no efficient rejection sampling algorithm that takes in an input distribution of $\mathcal{D}$ and outputs D_{col} unless NP $\subseteq$ BPP.*

Proof: We show that if there exists such an algorithm, then we can use it to solve the 3-coloring problem in BPP. Our reduction is the same as in Theorem 3.2.1. We simply note that the only property on which we depended in that theorem was that for any graph, the probability of acceptance was inversely proportionate to the number of colorings on that graph, and that this same property is guaranteed by the premise of our theorem.

The only change we need to make to our reduction is that in the final step, when we reduce from the circuit X' to the graph G', we must with it reduce the satisfying assignment (all zeroes) to a valid 3-coloring C on G', and then input (C, G') into B. ∎

Theorem 3.2.3 *Consider the Quadratic Residue Problem (QRP), that is, for $n = pq$ (p,q unknown) and arbitrary $x \in Z_n^*$, decide whether x is a quadratic residue modulo n. If QRP is hard for BPP, then there is no efficient rejection sampling algorithm that takes in an input distribution of $\mathcal{D}$ and outputs a uniform distribution on 3-colorable graphs.*

Proof: We show that if there does exist an efficient rejection sampling algorithm taking in $\mathcal{D}$ and producing a uniform distribution on 3-colorable graphs, then we can use it to solve the QRP in BPP. For this purpose, take B as a black box from $\mathcal{D}$ to the uniform distribution on 3-colorable graphs.

We note that this theorem is much more general than the preceding ones, since it allows us to reduce using *any* black box, including an adversarial one (i.e. one that is specifically trying to keep us from solving the problem in which we are interested – for example, using our reduction from Theorem 3.2.2 such an adversarial box could simply accept any graph paired with our "easy" coloring C with probability 1 and reject all other colorings on that graph, and thus we would gain no information about whether there were other colorings on G'). The consequence of its generality is that it yields a weaker hardness result, that generating according to the specified distribution is as hard as QRP (although this problem is still widely suspected to be as hard as factoring – see [5], for instance).

First, we define a variant on QRP that we will use in our reduction.

Definition 3.2.5 *The* **Double Quadratic Residue Problem** *(DQRP) is: Given $n = pq$ (p,q unknown) and $(x, y) \in Z_n^* \times Z_n^*$, does either x or y (inclusive) have a root modulo n?*

Motivations: Recall that the behavior of B is undefined on any graph that does not have a valid 3-coloring (and in fact, we are required to include a valid 3-coloring along with the graph as input to B). Therefore, we need a construction in which we can guarantee at least one known solution, while still retaining some additional structure for which we do not know the solution. In this case, we can manufacture a random QR mod n in each pair by simply squaring a random value mod n, thus ensuring that there is always at least one solution.

Given an (x, y) pair as above, we can use parsimonious reduction from SAT as in previous theorems to construct a 3-colorable graph that has one valid coloring for each root of x or y. Assuming that $x \neq y$ then, there will be 4 solutions if exactly one of x or y is a QR mod n, and 8 solutions if both of them are.

Since the input distribution includes graphs with probability proportionate to their number of colorings, and the output distribution includes all 3-colorable graphs with equal probability, we note that the *average* probability with which a graph G is accepted is proportionate to $\frac{1}{\chi_G}$. That is, while a particular coloring may have a high or low probability of acceptance by B, the expected number of acceptances over all colorings of a graph on j nodes should be the same as the expected number of acceptances of a graph G that has only one coloring, that is, ψ_j as defined above. Thus for all n-node 3-colorable graphs G: $\sum_{\text{Colorings } C \text{ of } G} P(B \text{ accepts } (C, G)) = \psi_j$.

A consequence of this formula is that if we consider uniformly all χ_G colorings of a particular graph G, then $E_{\text{Colorings } C \text{ of } G}[P(B \text{ accepts } (C, G)) = \psi_j/\chi_G$. In particular, if we consider the graphs corresponding to DQRP instances as described above, looking uniformly over all pairs (x, y) such that $x \neq y$ are both QR_n and taking uniformly at random one of the eight roots of these values, the expected probability of acceptance is 1/8. However, looking at pairs in which only one element is QR_n and taking the roots of that one value uniformly at

random, the expected probability of acceptance is 1/4. Thus, by taking a sufficient number of samples we should be able to distinguish these two cases. We formalize this below.

Using this information and the black box B, we construct the following algorithm that solves QRP in BPP:

Algorithm 3.2.1 *A probabilistic algorithm for solving QRP, given a rejection sampling black box B taking in distribution $\mathcal{D}$ and outputting the uniform distribution on 3-colorable graphs.*

Input: n, $k \in Z_n^*$
Output: Accepts if k has a root in Z_n^*, rejects otherwise.

1 Iterate some constant number of times:
2 Pick uniformly random $y_1, y_2 \in Z_n^*$
3 Generate pair σ as, with equal probability, $(y_1^2, y_2^2 k)$ or $(y_1^2 k, y_1^2)$.
4 Parsimoniously reduce σ to a j-node 3-colorable graph G with a coloring corresponding to each valid root, and with it reduce the root y_1 to a valid coloring C on G.
5 By a polynomial number of samplings with respect to $\frac{1}{\psi_j}$, approximate the probability p with which (C, G) is accepted by B.
6 Take the average p_a of p across all iterations of the above loop.
7 If p_a is $\geq \frac{3\psi_j}{16}$, reject. Otherwise, accept.

Our first observation about this algorithm is that if k is a quadratic residue, then the pair σ generated in step 1a is uniformly random over all pairs (a, b) of quadratic residues, and the solution is uniformly random over all roots of that pair. This follows because a fixed quadratic residue (k) multiplied by a uniformly random quadratic residue (y_2^2) produces a uniformly random quadratic residue. Furthermore, our root y_1 is uniform over the roots of y_1^2, since y_1 was taken uniformly from all values in Z_n^*, and it is uniform over the roots of σ because both values of σ are uniformly random and we are equally likely to have a uniformly random root of the first or second value in σ.

A consequence of this is that if k is a quadratic residue, we are checking the probability with which B accepts a uniformly random coloring over the space of DQRP-equivalent graphs with 8 colorings. Therefore all probabilities p (from step 1d) are in the range from 0 to ψ_j, and the expected value of p is $\psi_j/8$. (Note that we require a polynomial number of iterations on step 1d because ψ_j may be an inverse polynomial, in which case we need that many iterations to closely approximate a constant multiple of $\psi_j/8$).

If, on the other hand, k is not a quadratic residue, then $y_1^2 k$ is not a quadratic residue either, and we are taking our roots uniformly at random from the space of four possible roots of y_2^2 (since y_2 is chosen UAR). Thus we are looking at values for p (in step 1d) in the range from 0 to ψ_j, and the expected value is $\psi_j/4$.

Since we are taking uniformly random samples from a set of bounded values with averages of either $\psi_j/8$ or $\psi_j/4$, we can distinguish between the two with arbitrarily high fixed probability by taking the average over a fixed number of samples. Therefore, we can efficiently determine whether k is a quadratic residue. Moreover, in the special case where ψ_j is bounded by some constant, we can make this distinction in a constant number of iterations, where the only cost of the reduction is that of generating (a fixed number of) random values in Z_n^* and performing multiplication on them.

■

Theorem 3.2.4 *If QRP is hard for BPP, then there is no efficient rejection sampling algorithm that takes in an input distribution polynomially close to $\mathcal{D}$ and outputs a distribution polynomially close to uniform on 3-colorable graphs.*

Proof: We use a reduction very similar to the one in the previous theorem, but which allows us to generate a polynomial margin between the case where k is a quadratic residue and the case where it is not.

Similar to before, we define the following problem:

Definition 3.2.6 *The* **Multiple Quadratic Residue Problem** *(MQRP) is: Given $n = pq$ (p,q unknown) and $(x_1, x_2, \ldots, x_m) \in (Z_n^*)^m$, does any x_i have a root modulo n?*

We will work with MQRP tuples in which all values are distinct. Given this, we note as before that if exactly one of the values in a given tuple is a quadratic residue then there are 4 valid roots for values in the tuple. If, on the other hand, *all* values in the tuple are quadratic residues, then there are $4m$ valid roots, providing a factor of m between the two cases.

We will use a reduction almost identical to the one used in Algorithm 3.2.1. However, we reduce to 3-coloring from MQRP rather than DQRP (which is a special case of MQRP where $m = 2$).

Consider our input distribution, and let p_1 be a polynomial bound on the worst error ratio in the input (that is, p_1 is the factor by which the frequency of the most – or least – probable input differs from $\mathcal{D}$. Similarly, let p_2 be a polynomial bound on the worst error ratio in the output. Thus, if we use $\mathcal{D}$ as input to the black box B, then $p_1 p_2$ is a polynomial bound on the worst error ratio of the resulting output. To compensate for this error rate, we need only generate an MQRP instance with the properties defined above where m is, for example, $2p_1p_2$, to produce (as before) a ≥ 2 factor difference between the cases where k is a quadratic residue and the cases where it is not. We generate such an MQRP instance as follows:

(a) Uniformly at random generate the values $y_1, y_2, \ldots, y_m \in Z_n^*$.
(b) Generate an m-tuple $\sigma = (y_1^2, y_2^2 k, y_3^2 k, \ldots, y_m^2 k)$, where the value y_1^2 is uniformly at random placed in any location throughout the tuple.

All other aspects of the reduction from the previous theorem are the same, substituting MQRP for DQRP. Note that, as required above, if k is a quadratic residue then we are taking tuples uniformly at random over all m-tuples of quadratic residues, while if it is not then we are taking tuples uniformly at random over all m-tuples with exactly one quadratic residue, producing an arbitrarily large polynomial margin between the two cases which compensates for the non-uniformity of the input and output distributions.

■

Concluding this section, we note that while we used 3-coloring as a specific example, the only property of 3-coloring required for our proofs is that it has an efficient parsimonious reduction from SAT. As a consequence, all our results hold with equal generality for any problem (3SAT, Hamiltonian Path, etc.) which has such a reduction. For instance, it is unlikely that any attempts to uniformly generate satisfiable 3SAT formulas by rejection sampling from the planted assignment distribution will be successful.

4 Conclusion

We have provided a new algorithm for enumerating languages with polynomial delay, as well as evidence about cases where uniform random generation is computationally difficult in a formal sense. Our work leaves several questions open:

Is there a polynomial-space, polynomial-delay algorithm for enumerating NP-complete sets other than Hamiltonian Graph?

Can we prove, with an explicit diagonalization argument (rather than hardness assumptions), that there is a language that can be enumerated with polynomial delay but cannot be uniformly randomly generated?

Can we demonstrate the hardness of uniform random generation of NP-complete sets using rejection sampling under a weaker hardness assumption than Quadratic Residue (for example, it might be that such random generation is as hard as factoring, or even NP-hard)?

Is there an efficient algorithm for uniformly randomly generating an NP-complete set using some technique completely different from the ones we have examined?

Conversely, can we prove that uniformly randomly generating a NP-complete set is computationally difficult regardless of the algorithmic approach employed?

References

[1] Goldberg, L. A.: Efficient Algorithms for Listing Combinatorial Structures, (Cambridge University Press, 1993).

[2] Valiant, L. G.: The Complexity of Computing the Permanent, (Theoretical Computer Science 8, 1979, 189-201).

[3] Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating Satisfiable Problem Instances, (Proceedings of the Seventeenth National Conference on Artificial Intelligence, 2000, 256-261).

[4] Bax, E.: Recurrence-based Heuristics for the Hamiltonian Path Inclusion and Exclusion Algorithm, (California Institute of Technology Collection of Open Digital Archives, 1994).

[5] Menezes, A. J., Van Oorschot, P. C., Vanstone, S. A.: Handbook of Applied Cryptography (CRC Press, 1996).