Algorithmica (2014) 68:916–939
DOI 10.1007/s00453-012-9703-x

Fast Sequential Importance Sampling to Estimate the Graph Reliability Polynomial

David G. Harris · Francis Sullivan · Isabel Beichl

Received: 23 September 2011 / Accepted: 17 October 2012 / Published online: 8 November 2012

Abstract The reliability polynomial of a graph counts its connected subgraphs of various sizes. Algorithms based on sequential importance sampling (SIS) have been proposed to estimate a graph's reliability polynomial. We develop an improved SIS algorithm for estimating the reliability polynomial. The new algorithm runs in expected time $O(m \log n\ \alpha(m,n))$ at worst and $\approx m$ in practice, compared to $\Theta(m^2)$ for the previous algorithm. We analyze the error bounds of this algorithm, including comparison to alternative estimation algorithms. In addition to the theoretical analysis, we discuss methods for estimating the variance and describe experimental results on a variety of random graphs.

Keywords Reliability polynomial · Graph · Fully-polynomial relative approximation scheme · fpras · Network reliability · Sequential importance sampling · On-line algorithm · Incremental algorithm

1 Introduction

Let G be a connected undirected graph with vertex set V and edge set E, with $|E| = m$, $|V| = n$. We define $\mathrm{Rel}(p)$, the all-terminal reliability polynomial of G, to be the probability that the graph remains connected when edges are removed independently

D.G. Harris (✉)
United States Department of Defense, Washington DC, USA
e-mail: davidgharris29@hotmail.com

F. Sullivan
IDA/Center for Computing Sciences, Bowie MD, USA

I. Beichl
National Institute of Standards and Technology, Gaithersburg MD, USA

with probability p. This function is a polynomial which can be factored as

$$\mathrm{Rel}(p) = \sum_{i=0}^{m} N_i (1-p)^i p^{m-i}$$

where N_i is the number of connected subgraphs of G with i edges. Note that $N_i = 0$ for $i < n-1$. This polynomial has various physical applications, for example determining the reliability of a computer network or power grid.

It will be more convenient to work with a factored form of the reliability polynomial. We define the *reverse reliability generator*

$$r_G(x) = \sum_{i=0}^{m} N_{m-i} x^i$$

This polynomial has degree $K = m - n + 1$. (This notation is not standard, but makes exposition of our algorithms which compute in turn from N_m down to N_{m-n+1}, much simpler.)

In particular, we use the terms "low" and "high" order coefficient to refer, respectively, to ways of removing just a few edges from G and to graphs that have a few edges added to a spanning tree of G. In this paper, we develop algorithms to estimate the coefficients N_i individually. The reverse reliability generator can then be used to evaluate the reliability polynomial itself via

$$\mathrm{Rel}(p) = (1-p)^m r\Big(\frac{p}{1-p}\Big).$$

Exactly computing the reliability polynomial is known to be #P-complete [12]. A variety of algorithms have been proposed to estimate the reliability polynomial, or fragments of it, for a graph. Some of these algorithms seek to compute $\mathrm{Rel}(p_0)$ for a fixed probability p_0, such as [5] or [9]. The problem of estimating $\mathrm{Rel}(p_0)$ is related to the problem of estimating the reliability polynomial coefficients N_i, but they are not equivalent especially in terms of evaluating the relative error. For example, Karger's algorithm [9] gives a fully-polynomial relative approximation scheme (fpras) for the problem of estimating $1 - \mathrm{Rel}(p_0)$. As we will discuss, this does not provide a fpras for the coefficient N_i themselves and in fact no fpras for these is known.

There is no known polynomial-time algorithm to estimate arbitrary coefficients, but a variety of special cases have been considered. Reference [2] has described an algorithm to transform a given dense graph G into a relatively sparse graph G', with only $O(n \log n)$ edges, which has approximately the same reliability. An algorithm of [10] computes the high-order coefficients of planar graphs in polynomial time; this is of limited application for more general graphs. References [12] and [13] describe polynomial-time algorithms for computing low-order coefficients (corresponding to small cutsets) exactly. These algorithms, while polynomial time, are still quite expensive as they require multiple network flow computations; for very large-scale graphs they may not be practical.

Conversely, algorithms such as [3] estimate high-order coefficients accurately and low-order coefficients poorly. This algorithm also computes N_{n-1}, the number of spanning trees of G, exactly in polynomial time. However, the time complexity of that algorithm is again super-linear, which means that on very large-scale graphs it may not be feasible.

In this paper, we will examine an algorithm of [1] to estimate the generator coefficients using sequential importance sampling (SIS). This algorithm emphasizes accuracy in low-order coefficients. (For many physical problems, such as computer networks or electrical grids, the low-order coefficients—which correspond to most of the edges remaining present—are the most relevant.) This algorithm, which we denote Algorithm 1, produces an estimate $\hat{r}$ as follows:

Algorithm 1

1. `Set` $m_0 \leftarrow 1$.
2. `For` $k = 1, \ldots, K$ `do`
   3. `Determine` D, `the set of non-bridges of` G_k.
   4. `Set` $m_k \leftarrow m_{k-1}|D|$
   5. `Choose an edge` $e_k \in D$ `uniformly`.
   6. `Set` $G \leftarrow G - e_k$
7. `Return the estimate`

$$\hat{r}(x) = 1 + m_1 x + \frac{m_2}{2!}x^2 + \cdots + \frac{m_K}{K!}x^K$$

This algorithm returns a single estimate $F_1(G)$. We emphasize that we are estimating the entire polynomial, not simply evaluating the polynomial at a particular value of x, and so our estimate actually consists of all K coefficients. Since the algorithm is randomized, for any fixed G the resulting estimate $\hat{r} = F_1(G)$ is a random variable. Each estimate is a polynomial; we use the notation $\hat{r}[k]$ to mean the kth coefficient of this polynomial. The estimated $\hat{r}[j]$ and $\hat{r}[k]$ are, in general, statistically dependent.

Since the estimates $F_1(G)$ are unbiased, the variance can be reduced by taking T independent samples and averaging the results. Let us denote this average $\overline{F}_1$; this statistic $\overline{F}_1$ remains unbiased while the variance decreases as $1/T$. This procedure is embarrassingly parallel; hence we will focus on the single-processor time complexity of F_1 without any attempt to parallelize it; all of the timings will scale perfectly with multiple processors.

Note that the probability distribution of $F_1(G)$ may be complicated, but we can bound the relative error of $F_1(G)$ by the Chebyshev bound in terms of the relative variance. As we take T samples, the relative variance decreases as $1/T$. For this reason, we will summarize the accuracy of these algorithms in terms of a single parameter, the relative variance of a single sample

$$\mathrm{rv}_k = \frac{\mathbf{V}[F_1(G)[k]]}{\mathbf{E}[F_1(G)[k]]^2}.$$

This estimate $\hat{r}$ can be used, in turn, to provide an unbiased estimate of the reliability polynomial at an arbitrary value of p:

$$\widehat{\mathrm{Rel}}(p) = (1-p)^m \hat{r}\left(\frac{p}{1-p}\right).$$

The main cost of Algorithm 1 is step (3), the search for bridges (also known as cut-edges): at each iteration, this costs a full $\Theta(m)$ work, for a total cost of $\Theta(m^2)$. Note the use of Θ bounds here; there is no hope that on typical graphs the cost will be any less than the worst case.

Ideally, we would detect all bridges in total time $O(m)$. Unfortunately, this top-down algorithm modifies the graph by removing edges, and in general it is difficult to maintain graph structures, such as bridges, under edge deletion. Certain algorithms, such as [6] and [8], can in fact maintain such structures relatively efficiently. These algorithms, though polylogarithmic, still have high complexities (as many $O(\log^4 m)$ operations per query).

In contrast, when edges are added, various algorithms (known as "incremental algorithms") can keep track of many graph properties very efficiently. These incremental algorithms tend to have sublogarithmic or even constant time complexity per operation, which is much better than the decremental algorithms.

In this paper, we will discuss how to transform Algorithm 1 (and variants), which appear to require deleting edges, into an effectively incremental algorithm with expected running time $O(m\alpha(m,n)\log n)$. With some heuristics, we achieve very fast run time of $\approx m$ in practice. This makes this algorithm much faster, both asymptotically and practically, than prior approaches such as [1, 5, 9]. This algorithm should be practical on truly large-scale graphs.

Rather than attempt to describe a single monolithic "master algorithm," we will break this down into a series of relatively smaller improvements, which we can discuss and analyze separately.

In addition to analyzing the algorithm theoretically, we also developed a C implementation which we applied to various types of random graphs. We will frequently discuss the "practical" performance of these algorithms, both in terms of variance and timing, based on these experiments. These experiments are discussed in Sect. 8. We believe that both the empirical and theoretical points of view are necessary to understand the behaviors of this algorithm.

2 Allowing Bridges

The first alternative to Algorithm 1 ignores bridges entirely, and focuses solely on graph connectivity instead. It simply removes edges one by one from G until arriving at a disconnected graph. We will describe this as Algorithm 0, estimating the reverse reliability generator in time $O(m\alpha(m,n))$ where α is the inverse Ackermann function. This is based on efficient algorithms for disjoint sets, a classical computer-science topic [14].

Algorithm 0

1. Choose K edges $e_1, \ldots, e_K$ uniformly without replacement from G.
2. Remove $e_1, \ldots, e_K$. Traverse the resulting graph to find its connected components; this can be represented as a partition of the vertex set V.
3. For $k' = K, \ldots, 1$ do the following:
   4. If there is a single vertex-class remaining, terminate the loop.
   5. Let $e_k = \langle v, w \rangle$. Merge the vertex-classes $[v]$ and $[w]$.
6. Return the estimate

$$\hat{p}(x) = 1 + \binom{m}{1}x + \binom{m}{2}x^2 + \cdots + \binom{m}{k'}x^{k'}$$

By using appropriate disjoint-set data structures, we can implement step (5) in at most $O(m\alpha(m, n)) \approx m$ operations [14].

Note that this algorithm tracks graph connectivity, and it does so in reverse order. The obvious way of implementing Algorithm 1 is to check that the graph remains connected as each edge is successively removed. Algorithm 0 turns this around, however, re-inserting edges and checking when the graph becomes connected. As such, this algorithm behaves like an incremental graph algorithm.

Although Algorithm 0 is much faster than Algorithm 1, its variance is also much worse. Essentially, Algorithm 0 is a Monte Carlo sampling among all subgraphs with k edges removed, counting how many are connected. For high-order coefficients it is extremely improbable to reach a connected subgraph and the resulting estimates have exponentially high variance. As we will see, this increase in variance completely counteracts the decrease in runtime. However, Algorithm 0 is a proof of concept demonstrating how incremental algorithms may be used to compute Algorithm 1.

A simple induction on G shows that, for any graph G and any coefficient k, we have

$$\mathbf{V}[F_0(G)[k]] \geq \mathbf{V}[F_1(G)[k]].$$

This gives another important role to Algorithm 0: as Algorithm 0 is particularly simple to analyze, it will allow us to easily upper-bound the variance of Algorithm 1.

3 Backtracking to Avoid Bridges

Algorithm 0 removes edges one by one. When a bridge is selected, the graph becomes disconnected. In this case, Algorithm 0 estimates the corresponding coefficient of the reliability polynomial to be zero, which inflates variance. There is an alternative when this occurs: to back-track to the last iteration k at which the graph became disconnected and select a different edge e_k.

This back-tracking Algorithm 2 has two phases. In the first Phase, we choose a series of edges $e_1, \ldots, e_K$ with the same probability distribution as Algorithm 1. Namely, the edges are chosen so that e_i is uniformly distributed among the non-bridges of $G - e_1 - \cdots - e_{i-1}$. (To simplify the notation, we write $G_i = G - e_1 - \cdots -$

e_{i-1}.) Note that, while this process samples spanning trees, the resulting distribution on spanning trees is far from uniform. Hence algorithms such as Wilson's algorithm will not be suitable for this task [16].

Instead of explicitly listing the non-bridges of the graphs, as in Algorithm 1, we employ a kind of batch rejection sampling. We select edges e_i uniformly among all possible edges, and only later do we test if these edges were bridges. Edges which turned out to be bridges are rejected and sampled again. This process ensures that e_i is still uniformly distributed among the non-bridges. In the second Phase, once all the edges have been chosen, we count the number of non-bridges available at each graph G_i.

The Algorithm 2 Phase I involves rather complex looping and data structures. Before explaining the full, efficient version of this algorithm, we give an (inefficient) sketch of the process. To begin, we select edges $e_1, \ldots, e_K \in G$ uniformly without replacement. If the resulting graph G_K is a spanning tree, then these edges were valid, and were selected with the desired probability distribution. Otherwise, for some $k_1 < K$, the edge e_{k_1} was a bridge of G_{k_1}. In this case, the edges $e_1, \ldots, e_{k_1-1}$ were selected with the desired probability distribution, so we can output them. We still need to select a valid $e_{k_1}, \ldots, e_K$. We now backtrack to the graph G_{k_1} and mark edge e_{k_1} as a bridge — so we will not select it henceforth. We repeat this process, choosing new edges $e_{k_1}, \ldots, e_K$ from the graph G_{k_1}. If this gives a spanning tree, we are done. Otherwise, we backtrack to some $k_2 > k_1$, and so on.

This process can be efficiently implemented as follows:

Algorithm 2 Phase I

1. Set $k = 1$. Set the FOUND-TREE flag to be FALSE.
2. Repeat the following steps while the FOUND-TREE flag is FALSE:
   3. Choose edges $e_k, \ldots, e_K \in G_k$ uniformly without replacement among the edges which have not been marked as bridges of G_k.
   4. Enumerate the connected component structure of G_K, which can be regarded as a partition on the vertex set.
   5. If G_K is connected, set FOUND-TREE flag to be true. Output $e_1, \ldots, e_K$ and terminate Algorithm 2 Phase I.
   6. Otherwise, for $k' = K, \ldots, 1$ do the following:
      7. If there is a single vertex-class remaining, terminate the loop.
      8. Merge the vertex-classes corresponding to the end-points of $e_{k'}$.
   9. At this point, we have found the $k' \leq K$ which is maximal such that $G_{k'}$ is connected.
   10. Perform a depth-first search of $G_{k'}$, detecting any bridges (note that $e_{k'}$ must be a bridge of $G_{k'}$ by maximality of k').
   11. Set $k \leftarrow k'$ and continue the loop.

Although we have stated this algorithm as if each iteration k had its own separate graph G_k, in reality we maintain a single data structure which stores G_k for the *current* iteration k alone. This requires a data structure which remembers all the changes made to the graph and can reverse them.

The second Phase of this algorithm counts how many non-bridges were available at each iteration. We do this starting from the spanning tree $T = G_K$ and using an incremental algorithm to keep track of its 2-edge-connectivity. Reference [11] describes an incremental algorithm for maintaining this information. The basic idea of this algorithm is to keep track of a partition $\mathcal{V}$ of the vertex set. Initially, every vertex is in its own class. When an edge $e = \langle i, j \rangle$ is added, the vertices along the tree-path from i to j are merged.

Algorithm 2 Phase II

1. Let $T = G_K$, and interpret this graph as a rooted tree on V (i.e. each vertex but the root node has a parent). We will keep track of a related vertex structure $\mathcal{V}$, which is a partition of V, and an associated tree-structure $\mathcal{T}$ on $\mathcal{V}$.
   Initially, $\mathcal{V}$ consists of singleton sets, and $\mathcal{T}$ is the tree corresponding to T.
2. Set $d_K \leftarrow 0$. This records the number of non-bridges available at each iteration.
3. For $k = K, \ldots, 1$ do the following:
   4. Let $e_k = \langle i, j \rangle$.
   5. Determine $[i], [j]$, the classes in $\mathcal{V}$ containing i and j.
   6. Find $[v_1], \ldots, [v_l]$, the tree-path in $\mathcal{T}$ connecting $[i], [j]$ through their nearest common ancestor. Merge $[v_1], \ldots, [v_l]$ and update $\mathcal{T}$ appropriately.
   7. Set $d_{k-1} \leftarrow d_k + l$.
8. For $k = 0, \ldots, K$ set $m_k = \prod_{i<k} d_i$
9. Return the estimate

$$\hat{p}(x) = 1 + m_1 x + \frac{m_2}{2!} x^2 + \cdots + \frac{m_K}{K!} x^K$$

Each iteration of the loop in Phase I steps (3)–(11) takes $O(m)$ time. If step (5) does not terminate the loop, then a bridge has been detected; since there are at most n bridges, the maximum number of iterations is n and the maximum runtime of the entire Phase I is $O(mn\alpha(m, n))$. The Phase II costs $O(m\alpha(m, n))$.

4 Expected Complexity of Algorithm 2

In this section, we demonstrate that the expected run-time of Algorithm 2 is $O(m \log n\ \alpha(m, n))$, which is essentially linear time. This is much better than the worst-case complexity $O(mn\alpha(m, n))$. We have already seen that Phase II has worst-case time $O(m\alpha(m, n))$, so it suffices to show this expected time bound for Phase I alone.

Before introducing the formal proof of the run-time of Algorithm 2 Phase I, we describe the intuitive reason for its surprising speed. The critical step is in step (10), where we perform a full depth-first search. This way, whenever a bridge is removed from the graph, eventually causing back-tracking, we search for other bridges (other than the one which caused the back-tracking). If there are few bridges compared to the number of edges, then we will not be likely to choose a bridge. Hence if we choose an edge at random and it turns out to be a bridge, this likely means there are many bridges present. Although in the worst case we might back-track once for each bridge, it is more likely that the bridges are detected in large clusters. Hence the expected number of back-trackings is more like $O(\log n)$.

This is a completely combinatorial phenomenon; in the analysis of this algorithm, it is essentially irrelevant that we are dealing with graphs with bridges. The key to the analysis is that, conditional on selecting an edge uniformly from the graph which turns out to be a bridge, it is likely there are many other bridges also present.

Theorem 1 *For any graph G, the expected run-time of Algorithm* 2 *is* $O(m \log n\, \alpha(m, n))$.

Proof To simplify the analysis of Algorithm 2 Phase I, we introduce a slightly modified algorithm, which is deliberately inefficient but easier to analyze.

As before, we start with the initial graph G. We keep track of two classes of edges: the *tree* edges T, which will ultimately form the resulting tree; and the *marked edges* M, which have been detected to be bridges but not removed from eligibility for future selection. Note that in Algorithm 2 Phase I itself, these two classes are not distinguished: as soon as a bridge is detected it is immediately removed. This is much more efficient, as all marked edges will be removed eventually, but it also introduces unwanted dependencies between the edges.

1. To begin, set $k \leftarrow 1$, $T_k \leftarrow \emptyset, M_k \leftarrow \emptyset$. Set FOUND-TREE to be FALSE:
2. Repeat the following while FOUND-TREE is FALSE:
   3. Repeat while $G_k \neq T_K$:
      4. Select an edge $e \in G_k - T_k$ uniformly at random.
      5. If this edge $e \in M_k$, set $T_k \leftarrow T_k \cup e$ and continue the loop.
      6. Otherwise, set $e_k \leftarrow e, T_{k+1} \leftarrow T_k, M_{k+1} \leftarrow M_k$. Set $k \leftarrow k+1$.
   7. At this point, $G_k = T_k$. Test if the resulting graph G_k is connected. If so, we have the found desired tree T_k. Set FOUND-TREE to be TRUE, output $e_1, \ldots, e_K$, and terminate.
   8. Otherwise, re-insert edges to find the maximal k' such that $G_{k'}$ is connected. Traverse the graph $G_{k'}$, searching for bridges. Set $T_{k'} \leftarrow T_{k'} \cup e_{k'}$. Any other bridges of $G_{k'}$ which are not already in $T_{k'}$ are *marked* (added to $M_{k'}$) but not added to $T_{k'}$. Set $k \leftarrow k'$.

This algorithm generates edges $e_1, \ldots, e_K$ with the same probability distribution as Algorithm 1. Furthermore, it is always slower than Algorithm 2. So it will suffice to bound the running time of this modified algorithm.

Let T denote the final spanning tree produced by this algorithm, $T = G - e_1 - \cdots - e_K$. The key difference that makes this algorithm easier to analyze is that, at the end of this process, for each tree-edge $t \in T$, there is a unique k such that $t \in T_{k+1} - T_k$. We call this the *selection time* $s(t)$ for t. Note that there are two ways such an edge can be selected. If the edge t was marked and chosen as the edge of step (5), then no back-tracking cost was incurred. If the edge t was discovered to be the disconnecting edge in step (8), then a back-tracking cost of $O((m - s(t))\alpha(m - s(t), n))$ was incurred. In this latter case we say that this edge t *caused back-tracking*.

It suffices to analyze the run-time of this algorithm *conditional on a fixed choice of outputs* $e_1^*, \ldots, e_K^*$. In this case, let T^* be the corresponding tree-edges, which are also fixed. For each edge $t \in T^*$, denote the *available time* $a(t)$ as the minimal k such that t is a bridge of $G - e_1^* - \cdots - e_k^*$. Sort the tree-edge $t_1^*, \ldots, t_{n-1}^* \in T^*$ in increasing order of their available time, so that $a(t_1^*) \le a(t_2^*) \le \cdots \le a(t_{n-1}^*)$. To simplify the notation we write a_i for $a(t_i^*)$.

Note that, again conditional on the fixed outputs e^*, the available times a are also fixed. However, the selection times $s_i = s(t_i^*)$ remain random variables. The selection times are almost the sole remaining sources of randomness, and determine the run-time of this algorithm. We must have $s_i \ge a_i$ for all i; for if t_i^* is removed earlier than a_i, the algorithm would include t_i^* among the eventual outputs instead of e^*. For each $i \le n - 1$, s_i is uniformly distributed among $a_i, \ldots, m$ and these s_i are independent.

There is one remaining source of randomness which is not captured by the selection times. This is if two edges have identical selection times, there is still uncertainty as to which was selected first. Note that, among edges selected at any given time, the precise ordering of the edges is uniformly distributed among all possible permutations of those edges. Dealing with edges with identical selection time would require keeping track of this ordering, which is notationally cumbersome. To simplify the exposition, we will assume that all selection times s are distinct.

We now claim that if for some $i > j$ we have $s_i < s_j$, then edge t_j^* will not incur back-tracking. There are two ways the edge t_i^* was selected. In the first case, edge t_i^* itself incurred back-tracking. Then in step (8) of this algorithm, we would perform a depth-first search of the graph G_{s_i}. Because the edges t^* are sorted by their available times, t_j^* is a bridge of G_{a_j} and hence of G_{s_i} as well. So edge t_j^* becomes marked (if it was not already), and when it is actually selected will not incur back-tracking. In the second case, suppose that edge t_i^* was already marked when it was selected. Then, it must have been marked by selection of edge t_k^* for $k < i$. At this point, edge t_i^* must have been a bridge of the resulting graph G_{s_k}, so $s_k \ge a_i \ge a_j$. Hence, again edge t_j^* would be marked by edge t_k^*.

For any position ℓ, let n_ℓ denote the number of tree-edges t with $a(t) \le \ell$. Now, for any i and $\ell \ge a_i$, the probability that $s_i \ge \ell$ is $\frac{m-\ell+1}{m-a_i+1}$. Conditional on this event, the probability of $s_i < \min(s_{i+1}, \ldots, s_{n_\ell})$ is at most $\frac{1}{n_\ell - i + 1}$. These are both necessary conditions for edge t_i^* to incur back-tracking with $s_i \ge \ell$. If back-tracking does occur at $s_i = \ell$, then the cost of this back-tracking is $\alpha(m - \ell, n)(m - \ell)$.

Hence the expected back-tracking incurred by selecting edge t_i is

$$\mathbf{E}[\text{Back-tracking due to } t_i] \le \sum_{\ell \ge a_i} \alpha(m - \ell, n)(m - \ell) P(s_i = \ell, s_i < s_{i+1}, \ldots, s_{n_\ell})$$

$$\leq \alpha(m,n) \sum_{\ell \geq a_i} P(s_i \geq \ell, s_i < s_{i+1}, \ldots, s_{n_\ell})$$

$$\leq \alpha(m,n) \sum_{\ell \geq a_i} \frac{m-\ell+1}{m-a_i+1} \frac{1}{n_\ell - i + 1}$$

Summing over all i and $l \geq a_i$ we have

$$\begin{aligned}
\mathbf{E}[\text{backtracking work}] &\leq \alpha(m,n) \sum_{i=1}^{n} \sum_{\ell=a_i}^{m} \frac{m-\ell+1}{m-a_i+1} \frac{1}{n_\ell - i + 1} \\
&= \alpha(m,n) \sum_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \frac{1}{n_\ell - i + 1} \\
&= \alpha(m,n) \sum_{\ell} \log(n_\ell + 1) \\
&= O(\alpha(m,n) m \log n)
\end{aligned}$$

Note that these constant terms are all independent of e^*. Hence, conditional on any fixed $e_1^*, \ldots, e_K^*$ being output by this algorithm, the expected run-time is bounded by some universal constant times $m\alpha(m,n)\log n$. Integrating over e^*, we see that the expected run time of this algorithm is $O(m\alpha(m,n)\log n)$. □

5 Heuristics for Finding Easy Bridges

Although Algorithm 2 has near-linear worst case behavior, in practice it wastes a lot of work back-tracking and removing bridges. We can avoid much of this work, in practice, by searching for some easy-to-find bridges (bridges which can be found in just $O(1)$), while avoiding the complete graph traversal. The key idea of these heuristics is that many types of disconnections occur for local reasons, and do not require a full traversal of the graph. This is especially true for Erdős-Rényi random graphs, which have no global structure, but occurs also in more structured types of random graphs such Barabasi-Albert graphs. For example, a graph becomes disconnected when a vertex has degree zero. If we focus solely on this local structure, which does not require a search of the full graph, then we can find nearly all of the bridges. If we fail to detect a bridge, we will still have our $O(m\alpha(m,n)\log n)$ worst-case behavior.

Our experiments with random graphs show that the single most common type of bridge occurs when a vertex has degree one. These bridges may be easily detected; whenever we remove an edge $\langle v, w \rangle$, we simply check whether the degrees of vertices v, w have been reduced to one. If so, we contract the resulting singleton edge (and check whether this produces any other degree-one vertices, and so forth). This $O(1)$ check detects the vast majority of bridges.

Another easy method of detecting bridges is to remove degree-two vertices. With regards to graph connectivity, a vertex v connected to w_1 and w_2 is equivalent to a single edge connecting w_1 to w_2. We must keep track of the fact that the new edge

has "weight two", i.e. that it represents two simple edges. The main advantage of this technique is that it can discover loops: if $w_1 = w_2$, then the new edge is a free-floating loop, which has no effect on the graph connectivity. This free-floating loop can, in effect, be treated as an edge without vertex endpoints; this simplifies the graph and leads to new bridges.

A final method for detecting bridges, which is somewhat more expensive than these other techniques, performs a breadth-first search of the graph starting at a given vertex, limiting the search to radius two. To ensure that this step remains bounded, we only perform this check for vertices of degree $\leq d_{\max}$; a good choice seems to be $d_{\max} = 5$. Such vertices with low degree are more likely to have bridges, while also being faster to search. See Appendix A for more details.

As we have noted, there are decremental algorithms such as [8] that are capable of detecting all bridges, without heuristics, in time $O(\log^4 m)$ per edge deletion. We have not considered such algorithms for two reasons. First, these algorithms are quite complicated and expensive, much worse than the $O(1)$ time for heuristics. Second, if the heuristics fail and a bridge is selected, our algorithm will maintain a good asymptotic run time of $O(m \log n\, \alpha(n))$, which is better than could be obtained using the algorithm of [8].

It would be cumbersome to describe a combined algorithm, which simultaneously performs the back-tracking of Algorithm 2 and the bridge-heuristics. Rather, we describe the bridge heuristics as a kind of oracle which is accessed during Algorithm 2. We then discuss what conditions are needed of this oracle to ensure it preserves the run-time guarantees of Algorithm 2.

We suppose we have a data structure B (for bridge-detection) that supports insertion and deletion of edges. This data structure may identify certain edges of G as being bridges. If B identifies an edge e as a bridge, then e must be indeed a bridge; however, B need not discover all bridges. B may also identify G as being disconnected.

Algorithm 2a Phase I

1. Set $k = 1$. Initialize the data structures for B.
2. For $i = k, \ldots, K$ repeat the following:
   3. Select an edge $e_i \in G_i$ uniformly among the edges which have not been marked as bridges of G_i. Update B.
   4. If B identifies G as disconnected, terminate this loop prematurely.
5. Use disjoint-set structures, find k' maximal such that $G_{k'}$ is connected. Each time an edge is re-inserted, update B.
6. If $k = K$, terminate the loop.
7. Otherwise, perform a depth-first search of G_k, detecting any bridges (note that e_{k+1} must be a bridge of G_k by maximality of k). Contract any bridges found. Update B.

Algorithm 2a Phase I maintains correctness, as B is guaranteed to have only false negatives. Furthermore, as long as B is fast enough, it maintains the optimal running time.

Theorem 2 *Suppose B is a data structure whose running time, over any sequence of n edge deletions, is $O(m + n\alpha(m, n))$. Then Algorithm 2a Phase I has expected running time $O(m \log n\, \alpha(m, n))$, the same as Algorithm 2 Phase I.*

Proof First, note B, as given, supports only edge deletions. However, by using the rollback method, we can also support edge re-insertions. That is, given $e_1, \ldots, e_k$, B updates its data structure as e_i are successively removed. If the edges $e_k, e_{k-1}, \ldots, e_t$ are re-inserted, we simply undo all the changes that were made between e_k and e_t. With appropriate journaling, this can also be done in time $O(m\alpha(m, n))$.

In each iteration of step (2), Algorithm 2a performs a depth-first search of the graph, removes k edges, and back-tracks $k' < k$ edges. The cost of these operations is $O(m\alpha(m, n))$. The cost of the B data structure is also $O(m\alpha(m, n))$, including edge re-insertions. Hence, the entire cost of each iteration remains $O(m\alpha(m, n))$. The proof of Theorem 1 remains valid of Algorithm 2a as well as Algorithm 2.

It is important to note that this proof works only with true upper bounds on running time, *not* amortized bounds. (Because the costly operations may be replayed multiple times during the rollback, while the operations that payed for them are not replayed.) Hence data structure B must repay all its amortizations within the sequence of n edge deletions. However, these costs may be hidden by step (7), which entails a full traversal of the graph. □

We will now describe a concrete data structure B which detects easy bridges and has run-time $O(m\alpha(m, n))$ for a sequence of edge deletions and re-insertions.

This data structure B maintains a stack of vertices which must be checked for adjacent bridges. Every time an edge is transformed, the incident vertices are added to this active vertex stack. This data structure may be considered to manipulate a "shadow graph" H, which is essentially a labeled copy of G. Every edge $e \in H$ corresponds to a set of edges $e \in G$. We maintain a disjoint-set structure on the edges of G to their corresponding edge of H; we let $[e]$ denote the edge of H corresponding to the appropriate edge-class containing $e \in G$.

`Data structure` B

0. `Suppose` e `is removed from` G. `Update the structure as follows;`
1. `Remove` $[e]$ `from` H. `Push its endpoints` v, w `onto the stack.`
2. `While the vertex stack is non-empty, do:`
   3. `Pop the top-most vertex` x `from the stack.`
   4. `If` x `has degree zero in` H, `output DISCONNECTED and halt.`
   5. `If` x `has degree one in` H, `with incident edge` $t = \langle x, y\rangle$, `contract` $t \in H$ `and push` y `onto the stack.`
   6. `If` x `has degree two in` H, `connected to two edges` $t_i = \langle x, y_i\rangle$, `do the following:`
      i. `Remove these edges` t_1, t_2 `from` H.
      ii. `Insert a new edge` t `into` H `with endpoints` $\langle y_1, y_2\rangle$.
      iii. `Merge the edge-classes of` t_1, t_2, `and let` $t \in H$ `be the edge of` H `which corresponds to this merged class.`

iv. Push y_1, y_2 onto the stack.
v. If $y_1 = y_2$, then this new edge t is inserted with null vertices (i.e. as a free-floating loop).

7. If $\deg(x) \leq d_{\max}$, search the radius-two neighborhood of x. Contract any bridges found and push their endpoints onto the stack.

There are two ways vertices can be added to the stack: when an edge is removed, and when a contraction occurs. The first type of insertion occurs at most $O(m)$ times. Whenever the second type occurs, a vertex becomes contracted from the graph, hence the second type occurs at most $O(n)$ times. Hence steps (3)–(7) collectively occur at most $O(m)$ times. Each iteration of these steps costs at most $\alpha(m, n)$, as it requires a disjoint set operation. Hence, data structure B satisfies the requirements of Theorem 2.

Note that once the graph G is fixed, the choice of algorithm for detecting bridges in Algorithm 2a is a purely pragmatic one. Any bridge-detection method leads to the same probability distribution. Hence there is ultimately a single metric, total expected run-time (which includes the run-time of the bridge detection plus the time for backtracking when bridge-detection fails). We can use heuristics of various complexity, exact decremental algorithms such [6] and [8], or any other scheme. Furthermore asymptotic bounds on the run-time of algorithms are not relevant—we simply sample various algorithms and use the one that is fastest for G.

In practice, we have found that searching the radius-two neighborhood usually costs more time than it saves in Algorithm 2a. For certain graphs, such as Erdős-Rényi random graphs, merging degree-two vertices is also not cost-effective in Algorithm 2a. We have omitted this heuristic because it complicates the code significantly and the advantage is relatively minor even in the best case.

6 Bounds on Relative Variance

In this section, we examine upper bounds on the relative variance of Algorithms 0 and 1. We will make a few remarks about our approach. First, all of these bounds are reported in terms of a single structural parameter m. If one develops formulas in terms of other parameters such as n, m together or n alone, the resulting relative variance has a relatively complicated form, and can exhibit super-exponential growth or non-monotonicity. However, the bounds in terms of m are simply exponential.

Second, we will focus solely on the ultimate coefficient $\hat{r}[K]$, which is the estimate for the number of spanning trees $\kappa(G)$. As we have noted, algorithms such as [3] can compute the high-order coefficients with polynomial relative variance, and in fact can compute $\kappa(G)$ exactly in polynomial time. Hence the algorithms of this paper are not the method of choice for estimating $\kappa(G)$.

In general, for Algorithm 0, for any graph G the ultimate coefficient $\hat{r}[K]$ always has the highest relative variance. For Algorithm 1, the relative variance of coefficient i tends to increase with i, and also the gap between Algorithms 0 and 1 increases with i. It is possible to develop bounds on the relative variance of coefficient i. If i remains fixed, then these bounds are no longer exponential. If i is allowed to grow

asymptotically, then the exact form of these bounds depends on the growth rate of i, for example, if $i = \Theta(n), \Theta(m), K - \Theta(m), K - \Theta(n)$. To simplify the discussions, we will focus on the worst case, namely $i = K$.

These decisions allows us to summarize the behavior of the algorithm in terms of a single simple parameter, which is the rate of this exponential growth. To describe this we use O^* bounds, which ignore sub-exponential functions.

Theorem 3 *The relative variance of coefficient of Algorithm 0 on any graph G is $O^*(2^m)$.*

Proof Algorithm 0 is a Bernoulli random variable. Its relative variance is $1/p$, where p is the probability of selecting a spanning tree of the original graph. As Algorithm 0 chooses its $n-1$-edge subgraphs uniformly, this probability is

$$p = \frac{\kappa(G)}{\binom{m}{n-1}}.$$

where $\kappa(G)$ denotes the number of spanning trees of G.

For a fixed value of $m = m$, this is minimized at $n = m/2$, yielding $1/p = \binom{m}{m/2} = O^*(2^m)$. □

Although in practice Algorithm 1 tends to have exponentially lower variance than Algorithm 0, in theory it is possible for it to have relative variance $\Theta^*(2^m)$ as well. Thus, the estimate of 2^m relative variance is tight for Algorithms 0 and 1.

Theorem 4 *There exists a family of simple graphs G for which the relative variance of Algorithm 1 is $\Theta^*(2^m)$.*

Proof Consider the following graph on $2n$ edges and n vertices. We take a complete graph on $k = \sqrt{n}$ distinguished vertices, and a loop connecting the remaining $n - k$ vertices. (We ignore any quantization effects if $\sqrt{n}$ is non-integral; the error committed is polynomial.) See Fig. 1.

We call the first class of vertices "complete vertices" and the second "loop vertices." To estimate the relative variance, we need to estimate

$$\mathbf{E}_{e_1,\ldots,e_K} 1/p$$

where p is the probability of choosing edges $e_1, \ldots, e_K$ and the expectation is taken over the *uniform* distribution on $e_1, \ldots, e_K$ such that the resulting G_K is connected.

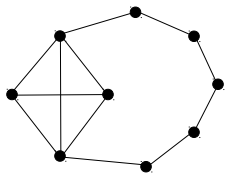

Fig. 1 Graph includes a complete graph on k vertices and a loop connecting $n - k$ vertices

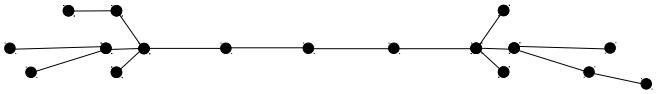

Fig. 2 Spanning trees involving all $n-k$ loop vertices, and which include exactly $k/2$ complete vertices on each end of the chain

We only seek to estimate the relative variance up to polynomial bounds. Hence it suffices to show that there is a subset E_0 of all possible $\langle e_1, \ldots, e_K \rangle$ such that $\mathbf{E}_{E_0} 1/p = O^*(2^m)$ and $\binom{m}{K} K!/|E_0|$ is bounded by a polynomial.

Consider the choice of $\langle e_1, \ldots, e_K \rangle$ such that the resulting spanning tree contains all of the loop-edges. The resulting spanning tree contains two clusters of complete vertices at each end of a long chain. We further restrict our attention to trees such that there are exactly $k/2$ complete vertices at each end of this chain. (See Fig. 2). The number of spanning trees of this form is $\binom{k}{k/2}((k/2)^{k/2-2})^2$, which is a polynomial fraction of the total number of spanning trees $\sim nk^k$.

Now, consider any tree T of this form. For this fixed tree, let us analyze

$$\mathop{\mathbf{E}}_{\substack{e_1,\ldots,e_k \\ \text{such that } G_k = T}} 1/p = \mathbf{E}_\sigma \frac{1}{P(\sigma(1))} \cdots \frac{1}{P(\sigma(K)|\sigma(1), \ldots, \sigma(K-1))}$$

where $\sigma \in S_K$ is a permutation of the edges $e_1, \ldots, e_K$.

All of the edges e_i connect a complete vertex to another complete vertex. Suppose that the edge $\sigma(K)$ connects one of the distinguished vertices on the left of the tree to one on the right of three. In fact half of such edges have this form, so this event has polynomial probability. In this case, all of the loop vertices are non-bridges of $G_1, \ldots, G_{K-1}$. In this case, $P(\sigma(i)|\sigma(1), \ldots, \sigma(i-1)) \geq 1/(2n-k-i+1)$, so

$$\begin{aligned}
\text{rv Algorithm 1} &= \frac{1}{\kappa(G)} \mathbf{E}_u(1/p) \\
&= \Omega^*\left(\frac{1}{nk^k} \frac{1}{P(\sigma(1)) \cdots \frac{1}{P(\sigma(K)|\sigma(1),\ldots,\sigma(K-1))}} \right) \\
&\geq \Omega^*\left(\frac{1}{nk^k} \frac{1}{2n-k} \cdots \frac{1}{n}\right) \\
&= \Omega^* \frac{1}{nk^k} \frac{n!}{(2n-k)!} \\
&= \Omega^*(4^n) \\
&= \Omega^*(2^m)
\end{aligned}$$

as claimed.

This shows that Algorithm 1 has relative variance $\Theta^*(2^m)$. □

7 The Karger fpras for Failure Probability

In [9], Karger introduced a fpras for estimating the unreliability $1 - \mathrm{Rel}(p_0)$. Note that this is a point estimate, given a fixed value of p_0, rather than an estimate for the entire polynomial. Karger's scheme essentially consists of two unrelated algorithms, depending on the size of $\mathrm{Rel}(p_0)$. When $\mathrm{Rel}(p_0)$ is small, we use Algorithm K1, which simply draws a random subgraph of G in which edges are removed independently and identically with probability p_0, and test if the resulting graph is connected. When $\mathrm{Rel}(p_0)$ is big, we use an alternate Algorithm K2 which counts the near-minimal cut-sets of G.

Although the values $1 - \mathrm{Rel}(p)$ are linear combinations of the coefficients N_i, this does not give a fpras for counting the number of connected subgraphs. The reason is that obtaining N_i from $1 - \mathrm{Rel}(p)$ would require inverting a Vandermonde matrix, which is exponentially ill-conditioned [7]. Hence, to obtain polynomial precision in N_i would require exponential precision in $1 - \mathrm{Rel}(p)$, which in turn would require exponential time-complexity.

We can use our SIS algorithm to improve Karger's scheme however. Although Algorithm K1 is polynomial-time, in practice it has two main problems. First, to estimate $\mathrm{Rel}(p_0)$ using K1 requires drawing many random subgraphs in which edges fail with probability p_0. If we wish to estimate $\mathrm{Rel}(p_1)$ for $p_1 \neq p_0$, we need a completely new set of samples. In practice, it is common to seek $\mathrm{Rel}(p)$ for multiple value of p, for example if we seek to estimate a value of p at which $\mathrm{Rel}(p) = 1/2$. Hence, if we seek to evaluate T samples of $\mathrm{Rel}(p_i)$ at $i = 1, \ldots, k$ we incur time $O(kTE)$.

When we run Algorithm 1 for T samples, we obtain in time $O(Tm \log n\, \alpha(n))$ an estimate $\widehat{\mathrm{Rel}}$ for the entire reliability polynomial. This estimated $\widehat{\mathrm{Rel}}$ can be evaluated at any value of p in $O(m)$ time. Hence, the total time complexity of evaluating multiple points reduces to $O(Tm \log n\, \alpha(n) + mk)$, which can be significantly smaller when k is large. In fact, the probability p_i need not be available initially; we have $O(Tm \log n\, \alpha(n))$ precomputation, after which we may estimate, for any p, $\mathrm{Rel}(p)$ in time $O(m)$.

The second main problem with Karger's scheme is that each sample of K1 draws a single random subgraph H, in which the number of edges actually removed is binomially distributed. Hence each sample uses information about N_i for only a single value of i. Algorithm 1 removes multiple edges sequentially, and obtains information about all the coefficients N_i. Hence, on a sample by sample basis, Algorithm 1 is more accurate than Algorithm K1. In fact, we can state this even for Algorithm 0:

Theorem 5 *Let G any graph and $p \in [0, 1]$. Then Algorithms 0 and* K1 *are both unbiased estimators of $\widehat{\mathrm{Rel}}(p)$. Algorithm 0 has variance which is no greater than Algorithm* K1.

Proof First, consider the expected square of Algorithm K1. With probability $N_{m-i}\, p^i (1-p)^{m-i}$, Algorithm K1 obtains a connected subgraph, in which case it estimates 1; otherwise it estimate 0. Hence the expected square of K1 is

$$\mathbf{E}[K1(p)^2] = 1 - \sum_i N_{m-i}\, p^i (1-p)^{m-i} = \mathrm{Rel}(p)$$

Next, consider the Algorithm 0. Algorithm 0 removes edges in turn until the resulting subgraph becomes disconnected. We abuse notation so that $F_0(p)$ is the estimated $\widehat{\mathrm{Rel}}(p)$ when $\widehat{\mathrm{Rel}}(x)$ is estimated by Algorithm 0.

Let X_i denote the event that graph obtained after removing i edges remains connected, so that

$$F_0(p) = \sum_i \binom{m}{i} p^i (1-p)^{m-i} X_i$$

Note $\mathbf{E}[X_i X_j] = \mathbf{E}[X_{\max i,j}]$. This allows to expand the sum

$$\begin{aligned}
\mathbf{E}[F_0(p)^2] &= \sum_{i,j} \binom{m}{i} p^i (1-p)^{m-i} \binom{m}{j} p^j (1-p)^{m-j} \mathbf{E}[X_i X_j] \\
&= \sum_i \binom{m}{i} p^i (1-p)^{m-i} \left(\sum_{j \le i} \binom{m}{j} p[j] \mathbf{E}[X_i] \right. \\
&\quad \left. + \sum_{j>i} \binom{m}{j} p^j (1-p)^{m-j} \mathbf{E}[X_j] \right) \\
&= \sum_i \binom{m}{i} p^i (1-p)^{m-i} \left(\sum_{j \le i} \binom{m}{j} p^j (1-p)^{m-j} N_{m-i} / \binom{m}{i} \right. \\
&\quad \left. + \sum_{j>i} N_{m-j} p^j (1-p)^{m-j} \right) \\
&= \sum_i N_{m-i} p^i (1-p)^{m-i} \left(\sum_{j \le i} \binom{m}{j} p^j (1-p)^{m-j} \right. \\
&\quad \left. + \sum_{j>i} \frac{N_{m-j}}{N_{m-i}} \binom{m}{i} p^j (1-p)^{m-j} \right) \\
&\le \sum_i N_{m-i} p^i (1-p)^{m-i} \left(\sum_{j \le i} \binom{m}{j} p^j (1-p)^{m-j} \right. \\
&\quad \left. + \sum_{j>i} \binom{m}{j} p^j (1-p)^{m-j} \right) \\
&= \sum_i N_{m-i} p^i (1-p)^{m-i} \\
&= \mathrm{Rel}(p) = \mathbf{E}[K1^2]
\end{aligned}$$

□

Hence Algorithm 1 has better variance than Algorithm K1 and in addition it can be precomputed without specifying an exact value of p.

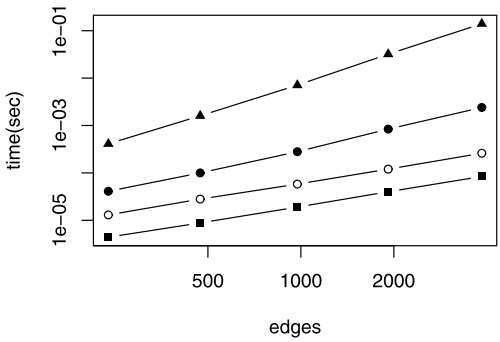

Fig. 3 Average runtime. Note the logarithmic scale for times. Key:
solid square = Algorithm 0;
solid triangle = Algorithm 1;
solid circle = Algorithm 2;
open circle = Algorithm 2a

8 Results

To test these algorithms empirically, we generated random graphs of various sizes and types. For each graph, we ran our various algorithms for a large number of iterations, tabulating the running time and program outputs.

Let us begin with the running times of these algorithms. We generate Erdős-Rényi random graphs with average edge density 10 but varying vertex counts. Figure 3 depicts the running times of the algorithms on these graphs, as a function of the edge count. Similar dynamics are seen on other types of random graphs.

All these timings must be taken with a grain of salt, as the different codes have very different profiles and could all be optimized to a greater or lesser extent. Nevertheless, we can reach some conclusions. Algorithm 0 is fastest, about three times faster than the others. The runtime of Algorithm 2a appears to scale linearly in m. Algorithm 2, while much faster than Algorithm 1, is also much slower than the heuristic algorithms Algorithm 2a. Its rate of growth appears to be slightly super-linear. Algorithm 1, implemented in the $\Theta(m^2)$ method, is as expected much slower than the incremental algorithms.

The running time is only half of the story, however. The other half is the accuracy of the estimates produced by these algorithms. To determine this, we analyzed a variety of test cases—random graphs generated according to various schemes. Note that straight-forward extraction of sample means and variances is not suitable for these types of statistics, which have exponentially large variance. Appendix B describes the statistical methodology used to estimate the relative variances of these algorithms.

Our first test case is a Barabasi-Albert random graph, with 100 vertices and 491 edges. We chose this type of random graph because it roughly approximates the dynamics seen in real-life graphs. Figure 4 depicts an experiment measuring the variance of Algorithms 0 and 1 as described in Appendix B.

Despite the fact that they have the same worst-case behavior, in practice Algorithm 1 is much better than Algorithm 0.

Test Case 2 is a sparse Erdős-Rényi random graph with 50 vertices and 238 edges. Figure 5 depicts the variance of Algorithms 0 and 1.

We observe the same type of behavior: Algorithm 1 has much better variance than Algorithm 0.

To investigate the way in which these algorithms scale with n, we generated Barabasi-Albert random graphs, of average degree 10, and computed the relative vari-

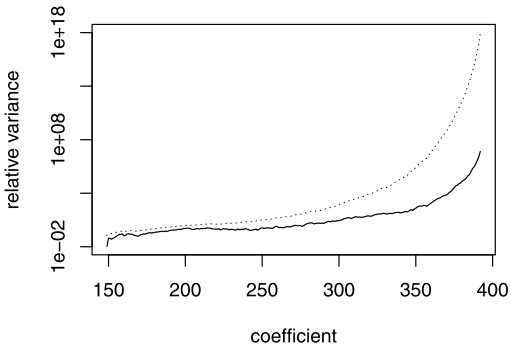

Fig. 4 Relative variance on a Barabasi-Albert random graph. Key: *dotted* = Algorithm 0; *solid* = Algorithm 1

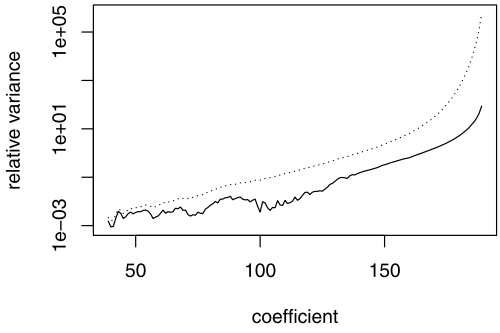

Fig. 5 Relative variance on an Erdős-Rényi random graph. Key: *dotted* = Algorithm 0; *solid* = Algorithm 1

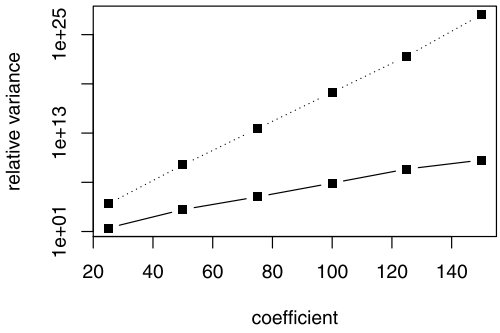

Fig. 6 Relative variance on Barabasi-Albert graphs of degree 10. Key: *dotted* = Algorithm 0; *solid* = Algorithm 1

ance of the ultimate coefficient, the number of spanning trees. Note that the number of spanning trees can be counted in polynomial time using the Matrix-Tree theorem, so this is not an appropriate use of these algorithms. However, this illustrates the scaling behavior as shown in Fig. 6.

Both of these algorithms have relative variances which increases exponentially, but for Algorithm 1 the rate of increase is far lower than for Algorithm 0.

For our final test case, we examined a real-world network, a representation of the power grid of the Western United States [15]. After removing bridges, this graph contained 3330 vertices and 4984 edges. Figure 7 shows the relative variance of Algorithms 0 and 1 on this graph.

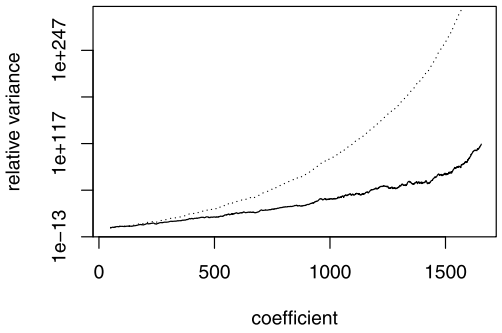

Fig. 7 Relative variance on the Western Power Grid data. Key: *dotted* = Algorithm 0; *solid* = Algorithm 1

Both of these algorithms are essentially useless for the highest-order coefficients, but it is clear that Algorithm 1 has dramatically lower variance than Algorithm 0. If we set a threshold of say 10^9 samples as the upper end of feasible, and we wish to attain a relative error of approximately 1, then we set a threshold of roughly $10^{9/2}$ for the maximum relative variance of a coefficient. In this case, Algorithm 0 allows us to estimate up to coefficient ~ 180, while Algorithm 1 allows us to estimate up to coefficient ~ 250.

9 Conclusion

We have improved the SIS algorithm of [1] for estimating the graph reliability polynomial. The runtime has been dramatically reduced from $\Theta(m^2)$ to $O(m \log n\alpha(m, n))$ in the worst case and with appropriate heuristics $\approx m$ in practice. On even moderately-sized graphs, this translates into orders of magnitude speedup. We believe that this is the first usable linear-time algorithm for estimating the reliability polynomial.

We have also described a method for estimating the accuracy of these algorithms on a specific graph. This is important both theoretically (allowing us to analyze these algorithms in a wide range of settings) and practically (allowing us to quantify our uncertainty, especially in cases where the uncertainty hugely outscales the estimate itself.)

We have demonstrated that, in the worst case, these algorithms may exhibit relative variance up to 2^m for high-order coefficients of the reliability polynomial. Empirically as well we see these algorithms exhibit exponentially increasing relative variance (though at a much smaller rate). This means that this algorithm is not the ultimate solution for the reliability polynomial in general. Nevertheless, the advantages of this algorithm, particularly its dramatically fast runtime, mean that this should be the method of choice in many cases, including physical problems in which the early coefficients of the generator are most relevant. The linear runtime allows estimates on much larger-scale graphs than previously possible—this should scale well into thousands of edges or more.

Acknowledgements Thanks to Antonio Possolo, for helpful conversations about the measurements of the algorithms' probability distributions, and for supporting Mr. Harris's research at NIST. Thanks to Brian Cloteaux for helping to edit and revise this paper.

Appendix A: Radius-Two Search for Bridges

In addition to detecting degree-one and degree-two vertices, Algorithm 2 conducts, in some circumstances, a breath-first search around a modified vertex v to find any easy bridges. The key idea of the search is that we assume that, if the vertex v is connected to a radius-two vertex, then it is connected to the entire graph. We search for edges which disconnect v from the radius-two vertices.

To begin, we let V_1 be all vertices connected to v, and let $V_2 = V - V_1 - v$. A bridge can arise if there is a unique $w \in V_1$ that is connected to V_2. If furthermore this w is not connected to any vertex in V_1 and if w is connected by a single edge to v, then the edge $\langle v, w\rangle$ is a bridge. If, on the other hand, w has a unique edge e connecting it to V_2, then e is a bridge.

The following algorithm finds such bridges:

1. Initially set $w = \emptyset$
2. For each $x \in V_1$:
   3. For each distinct vertex y connected to x:
      4. If $y \in V_2$ and $w = \emptyset$, set $w = x$ and continue with the next x.
      5. If $y \in V_2$ and $w \neq \emptyset$, we find no bridges. Halt.
6. If $w = \emptyset$, then the graph is disconnected. Return.
7. If there is a single edge connecting v to w, it is a bridge. Contract it and return.
8. If there is a single edge $\langle w, u\rangle$ with $u \in V_2$, then it is a bridge; contract it and return.
9. No bridges have been found. Terminate.

This code requires a data structure which supports efficient enumeration of *distinct* vertices. A mere list of edges (which may be multi-edges) is not enough. With this data structure, this code runs in time $O(V_1^2)$. Because Algorithm 2 only conducts this search if the degree of v is bounded by a constant $d_{\max}$, the total cost of these searches can be regarded as $O(1)$. In practice, we did not implement this data structure; this abandons the guarantee on maximum running time but in practice is faster.

There is one further improvement we can make here. If in either step (8) or (9) there is a double edge e_1, e_2 such that simultaneously removing e_1 and e_2 disconnects the graph, then we can simplify the graph by contracting the two edges e_1 and e_2 and inserting a loop of weight $w(e_1) + w(e_2)$.

Appendix B: Estimating Variance

In Sect. 6, we examined worst-case variance across all possible graphs of a given size. In this section, we consider variance from a different angle. Suppose we are given a graph G and use the algorithms we have described to estimate its reliability generator. How reliable are these estimates? In this section, we describe a method for estimating relative variance.

To begin, let us consider top-down algorithms such as Algorithm 0 or 1. In such algorithms, we may remove an edge e with probability $P(e)$; if the edge is removed,

then the corresponding estimate is multiplied by $1/P(e)$. $P(e)$ is essentially the importance function attached to each edge. For Algorithm 0, $P(e) = 1/m$ for all edges; for Algorithm 1, $P(e) = 1/|D(G)|$ if e is a non-bridge, and $P(e) = 0$ otherwise.

In such cases, the straight-forward way to estimate $\mathbf{E}[F(G)]$ and $\mathbf{E}[F(G)^2]$ would be to take T samples $F(G)$ and extract the sample mean and variance. Let $\hat{\mu}$ and $\hat{\sigma}^2$ be the sample estimates obtained in this way. We have that

$$\mathbf{V}[\hat{\mu}] = \frac{\sigma^2}{T}$$

$$\mathbf{V}[\hat{\sigma}^2] = \sigma^4 \left(\frac{2}{T-1} + \frac{\kappa}{T} \right) \qquad \kappa = \text{kurtosis of } F(G)$$

so these estimates may be very inaccurate when σ is large. For the high-order coefficients of the reverse reliability generator, this makes it exponentially expensive to estimate μ, σ accurately. This is a very dangerous situation: when the estimates $F(G)$ are very inaccurate, it is also very difficult to determine how inaccurate they are.

A better method of estimating accuracy is to estimate $\mathbf{E}[F(G)]$ and $\mathbf{E}[F(G)^2]$ using importance sampling. Instead of choosing the edge e to remove with the indicated probability $P(e)$, we use an alternative importance-sampling probability distribution

$$P'(e) = \frac{\#\text{ of spanning trees of } G - e}{E - V + 1}.$$

We emphasize that there are two distinct probability distributions in play here. $P(e)$ is the probability distribution used by algorithm F itself; $P'(e)$ is the probability distribution we are imposing in order to measure algorithm F.

This gives us the following SIS algorithm $F^{(d)}$ for estimating $\mathbf{E}[F(G)^d]$:

Algorithm $F^{(d)}$

1. `Set` $m_0 = 1$
2. `For` $k = 1, \ldots, K$
   3. `Choose an edge` $e \in G$ `with probability` $P'(e)$.
   4. `Set` $m_k \leftarrow m_{k-1} \frac{1}{P(e)^{d-1} P'(e)}$. `Set` $G \leftarrow G - e$.
5. `Return the estimate`

$$\hat{r}^{(k)} = \sum_{k=0}^{K} \frac{m_k x^k}{(k!)^d}$$

It is easy to verify that $\mathbf{E}[F^{(d)}(G)] = \mathbf{E}[F(G)^d]$.

The importance function $P'(e)$ ensures that algorithm $F^{(1)}$ estimates the high-order coefficient of $\mathbf{E}[F(G)]$ (the number of spanning trees of G) exactly (variance zero), although the estimates for the lower-order coefficients have higher variance. The estimate of the high-order coefficients of $\mathbf{E}[F(G)]^2$ also has greatly improved variance, although the low-order coefficients are worse. We believe it is much safer to have an error estimate which is accurate when the error is large.

The estimate of σ^2 provided by $F^{(2)}$ is typically the right order of magnitude, but for low-order coefficients it is not very precise. The sample variance is a better estimator for these low-order coefficients. To use the sample variance safely, we first estimate σ^2 using $F^{(2)}$ for all coefficients. For those coefficients for which $\hat{\sigma}^2$ is sufficiently small, we replace our estimate of σ^2 with the sample variance.

Note that to use this method, we must compute the importance function $P'(e)$ for all edges $e \in G$ at every stage of the recursion. To describe how this is done, we begin by recalling the Kirchoff matrix-tree theorem used to count the number of spanning trees of a graph G. Let A_G be the adjacency matrix of G, and let D be a diagonal matrix whose ith entry is the degree of vertex v_i. The Kirchoff formula states that $\kappa(G) = \det L$, where $L = (D - A_G)_{11}$ denotes the minor of $D - A_G$ obtained by removing the first row and column.

The SIS algorithm $F^{(d)}$ would appear to be very expensive if it computed κ naively from the Kirchoff formula. There are m iterations, in each of which we must $\kappa(G-e)$ for each edge. Evaluating a determinant costs $O(n^3)$ work and $O(n^2)$ memory. In all, naive computations would cost $O(m^2n^3)$ work and $O(n^2)$ memory.

Reference [4] describes a variety of algorithms to compute κ and to update this structure as edges of G are removed. We will discuss one variant of those algorithms. The main idea is to keep track of $\lambda(e) = \kappa(G-e)/\kappa(G)$ for each edge $e \in G$. These are precomputed for the original graph G. Next, when we update G by removing an edge e, we must update λ to the new λ'. We will find the following notation convenient. If $e = \langle i, j \rangle$ is any edge in G, we define the column vector δ_e to be the vector which is $+1$ in coordinate i, is -1 in coordinate j, and is zero elsewhere. Observe that when edge e is removed from G, the matrix L changes by $\delta_e \delta_e^T$:

$$L_{G-e} = L_G + \delta_e \delta_e^T$$

Now let us examine how to update λ':

$$\begin{aligned}
\lambda'(d) &= \kappa(G-e-d)/\kappa(G-e) \\
&= \det(L_{G-e-d})/\det(L_{G-e}) \\
&= \det(L_G + \delta_d\delta_d^T + \delta_e\delta_e^T)/\det(L_G + \delta_e\delta_e^T) \\
&= \det\Big(I + (L_G + \delta_e\delta_e^T)^{-1}\delta_d\delta_d^T\Big) \\
&= 1 + \delta_d^T(L_G + \delta_e\delta_e^T)^{-1}\delta_d \\
&= 1 + \delta_d^T\big(L_G^{-1} - \frac{uu^T}{1-\delta_e^T u}\big)\delta_d \quad \text{where } u = L_G^{-1}\delta_e \\
&= 1 + \delta_d^T L_G^{-1}\delta_d - \frac{(\delta_d^T u)^2}{1-\delta_e^T u} \\
&= \lambda(d) - \frac{(\delta_d^T u)^2}{\lambda(e)}
\end{aligned}$$

As described in [4], this leads to the following technique for updating λ. Whenever we remove edge e from G, we use a sparse matrix inversion algorithm such as

conjugate gradient to compute $u = L_G^{-1}\delta_e$. We then update $\lambda(d)$ for each edge $d \in G$ by

$$\lambda(d) \leftarrow \lambda(d) - (\delta_d^T u)^2/\lambda(e)$$

In total, this technique allows to reduce the memory requirement to $O(m)$ and the complexity to $O(mn)$. Furthermore, in practice the conjugate gradient method needs far less than $m\,n$ iterations to compute an adequate approximation, so in practice the work requirement per iteration is more like $\approx m \log n$. This is still much more expensive than computing F itself, however it allows us to achieve accurate estimates for $\mathbf{E}[F(G)]$ and $\mathbf{E}[F(G)^2]$ on graphs of interest. In many cases, straightforward sampling of F itself would have required infeasible computation to obtain accurate results.

References

1. Beichl, I., Cloteaux, B., Sullivan, F.: An approximation algorithm for the coefficients of the reliability polynomial. Congr. Numer. **197**, 143–151 (2010)
2. Chechik, S., Emek, Y., Patt-Shamir, B., Peleg, D.: Sparse reliable graph Backbones. In: ICALP, pp. 261–272 (2010)
3. Colbourn, C., Debroni, B., Myrvold, W.: Estimating the coefficients of the reliability polynomial. In: Proceedings of the Seventeenth Manitoba Conference on Numerical Mathematics and Computing, Congressum Numerantium, vol. 62, pp. 217–223 (1988)
4. Colbourn, C., Myrvold, W., Neufeld, E.: Two algorithms for unranking arborescences. J. Algorithms **20**, 268–281 (1996)
5. Fishman, G.: A Monte Carlo sampling plan for estimating network reliability. Oper. Res. **34**, 581–594 (1986)
6. Galil, Z., Italiano, G.: Fully dynamic algorithms for 2-edge connectivity. SIAM J. Comput. **21**, 1047–1069 (1992)
7. Gautschi, W., Inglese, G.: Lower bounds for the condition number of Vandermonde matrices. Numer. Math. **52**, 241–250 (1988)
8. Henzinger, M., King, V.: Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation. SRC Technical Note (1997)
9. Karger, D.: A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. SIAM J. Comput. **29**, 11–17 (1996)
10. Myrvold, W.: Counting k-component forests of a graph. Networks **22**, 647–652 (1992)
11. La Poutre, J.: Maintenance of 2- and 3- connected components of graphs, Part II: 2- and 3-edge-connected components and 2-vertex-connected components. Tech. Rep. RUU-CS-90-27, Utrecht University (1990)
12. Provan, J., Ball, M.: The complexity of counting cuts and the probability that a graph is connected. SIAM J. Comput. **12**, 777–788 (1983)
13. Ramanathan, A., Colbourn, C.: Counting almost minimum cutsets with reliability applications. Math. Program. **39**, 253–261 (1987)
14. Tarjan, R.: Efficiency of a good but not linear set union algorithm. J. ACM **22**, 215–225 (1975)
15. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature **393**, 440–442 (1998)
16. Wilson, D.: Generating random spanning trees more quickly than the cover time. In: Proceedings of the Twenty-Eighth Annual ACM Symposium of Theory of Computing, pp. 296–303 (1996)