Partial rejection sampling

Mark Jerrum

Queen Mary, University of London

Markov Processes, Mixing Times and Cutoff
Durham, 29th June 2017

Joint work with Heng Guo (QMUL)
and Jingcheng Liu (UC, Berkeley)

Lovász Local Lemma

Suppose $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is a formula in variables $X_1, \ldots, X_n$. (The variables may be Boolean, and the clauses might be disjunctions of literals, but this is not essential.) Assume that

- each clause shares variables with at most d other clauses;
- under a uniform random assignment to the variables $X_1, \ldots, X_n$ each clause is false with probability at most p.

Then the Lovász Local Lemma (LLL) asserts that, if $4pd \leqslant 1$, then Φ is true with non-zero probability.

The LLL guarantees only an exponentially small probability, so simple rejection sampling will not, in general, find a satisfying assignment to Φ efficiently.

Moser-Tardos resampling algorithm

A remarkable breakthrough is due to Moser and Tardos (2010), who found an algorithmically efficient version of LLL:

1. Initialize variables $X_1, \ldots, X_n$ independently at random.
2. While there exists an unsatisfied clause:
   pick one and resample all its variables.

Moser and Tardos showed that this algorithm is efficient under the same condition as LLL.

Searching versus sampling

Question

Instead of simply finding a satisfying assignment, can we generate one uniformly at random?

Consider the problem of sampling independent sets in graph. Variables correspond to vertices. Clauses correspond to edges. A typical clause has the form $\neg X_i \vee \neg X_j$.

The Moser-Tardos algorithm selects an edge with both endpoints in the current independent set and re-randomises the variables corresponding to the two endpoints.

For a path of length two (i.e., with three vertices) the empty independent set is generated with probability $\frac{2}{9}$ and not $\frac{1}{5}$ as required.

In fact, any efficient algorithm ought to fail, as sampling independent sets uniformly at random is a computationally hard problem (NP-hard).

A second example: spanning trees

Goal: Given a graph G with distinguished vertex r, sample a uniform spanning tree with root r.

A second example: spanning trees

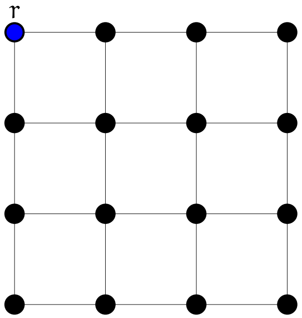
Goal: Given a graph G with distinguished vertex r, sample a uniform spanning tree with root r.

In this example,

- there is a variable for each vertex v other than r; it points out the "parent" of r;
- there is a clause for every cycle in the graph; it asserts that the pointers don't align themselves with that cycle.

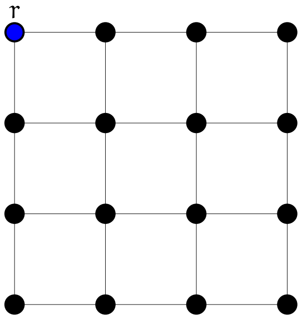
Note that the variables do not have to be Boolean, and there may be a lot of clauses!

Apply Moser-Tardos



Apply Moser-Tardos

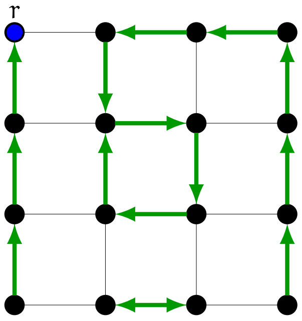
$\rightarrow$ 1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

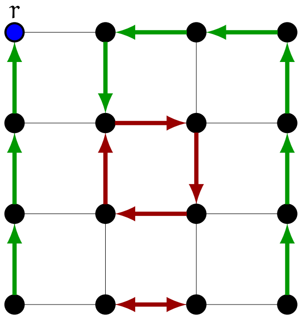
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, resample all vertices along all cycles.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

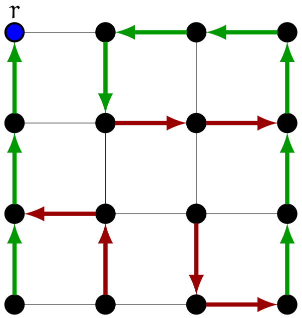
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, <span style="color:red">resample</span> all vertices along all cycles.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

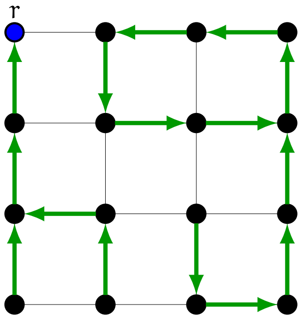
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, resample all vertices along all cycles.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

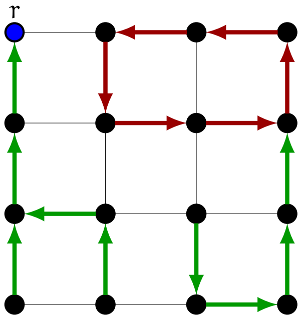
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, resample all vertices along all cycles.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

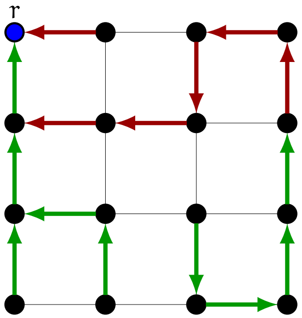
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, <span style="color:red">resample</span> all vertices along all cycles.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.

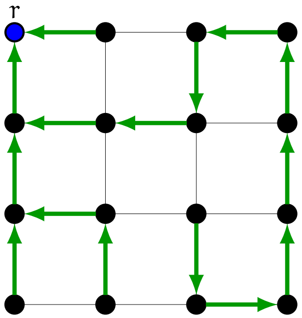
$\rightarrow$ 2. While there is a (directed) cycle in the current graph, resample all vertices along all cycles.



Apply Moser-Tardos

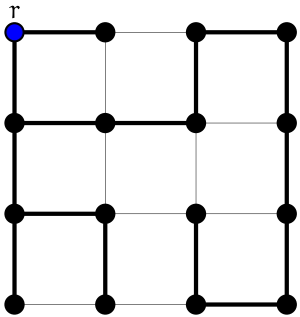
1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.
2. While there is a (directed) cycle in the current graph, <span>resample</span> all vertices along all cycles.

$\rightarrow$ 3. Output.



Apply Moser-Tardos

1. For each $v \neq r$, assign a random arrow from v to one of its neighbours.
2. While there is a (directed) cycle in the current graph, resample all vertices along all cycles.
3. Output.



When this process stops, there are no cycles and the result is a spanning tree.

Wilson's "cycle-popping" algorithm

Wilson (1996) showed that the output from the above procedure is uniform.

What is it about this particular application that caused the output to be uniform? How does it differ from the independent set example?

Extremal instances

Definition

We call an instance (formula) $\Phi = C_1 \wedge \cdots \wedge C_m$ extremal if every pair of distinct clauses C_i and C_j are either independent (C_i and C_j have no variables in common) or disjoint (C_i and C_j cannot both be false).

- Extremal instances Φ (in some precise sense) minimize the probability that the formula is true [Shearer 85].
- Moser-Tardos is slowest on extremal instances.
- However, slowest for searching is best for sampling!

Theorem (Guo, Jerrum and Liu 2017)

For extremal instances, the output of Moser-Tardos is uniform.

Extremal instances: a second example

Another example which leads to extremal instances is sampling sink-free orientations of a graph.

In this case, there is a Boolean variable encoding the orientation of each edge, and there is a clause for each vertex v, enforcing the condition that at least one edge is oriented away from v.

Moser-Tardos specialises to the "sink-popping" algorithm of Cohn, Pemantle and Propp (2002).

Extremal instances: running time

For extremal instances, the expected number of (sequential) resampling steps is proportional to

$$\frac{\mathrm{Pr}(\text{Exactly one clause in } \Phi \text{ is false})}{\mathrm{Pr}(\Phi \text{ is true})}.$$

(The upper bound is due to Kolipaka and Szegedy (2011).)

For sink-free orientations and spanning trees, this expression yields upper bounds on (sequential) running time of $O(n^2)$ and $O(nm)$, respectively, where $n = |V(G)|$ and $m = |E(G)|$.

Beyond extremal instances

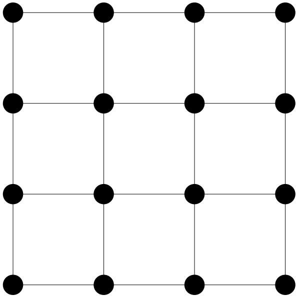
So far we have seen partial rejection sampling for extremal instances.

We cannot expect many sampling problems to correspond to extremal instances.

The way we extend the range of partial rejection sampling is suggested by the following example.

Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.
3. $\text{Resample} = \text{Bad} \cup \partial\text{Bad}$.
4. Resample variables in set Resample. Check independence.



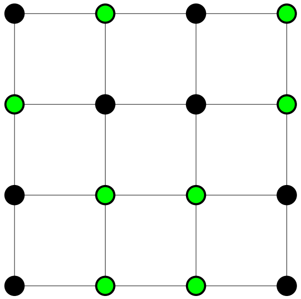
Sampling independent sets

$\rightarrow$ 1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.

2. Let Bad be the set of vertices in connected components of size at least 2.

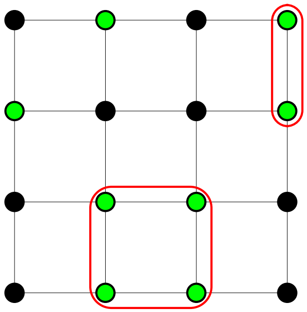
3. $\mathrm{Resample} = \mathrm{Bad} \cup \partial\mathrm{Bad}$.

4. Resample variables in set Resample. Check independence.



Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
$\rightarrow$ 2. Let Bad be the set of vertices in connected components of size at least 2.
3. Resample $=$ Bad $\cup\, \partial$Bad.
4. Resample variables in set Resample. Check independence.

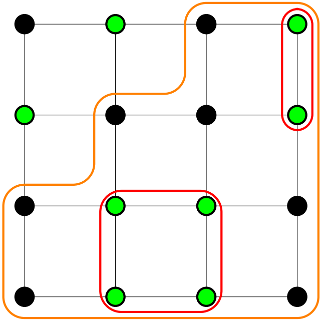


Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.

$\rightarrow$ 3. Resample $=$ Bad $\cup\, \partial$Bad.

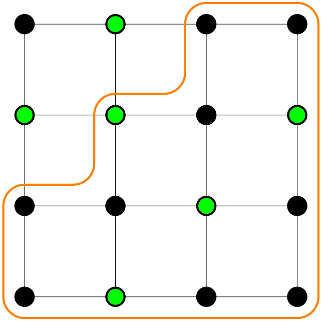
4. Resample variables in set Resample. Check independence.



Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.
3. Resample $=$ Bad $\cup\, \partial$Bad.

$\rightarrow$ 4. Resample variables in set Resample. Check independence.



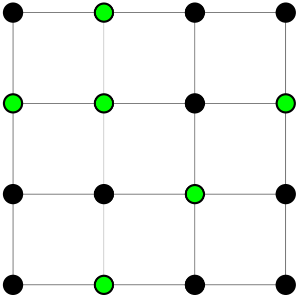
Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.

2. Let Bad be the set of vertices in connected components of size at least 2.

3. $\mathrm{Resample} = \mathrm{Bad} \cup \partial\mathrm{Bad}$.

$\longrightarrow$ 4. Resample variables in set $\mathrm{Resample}$. Check independence.



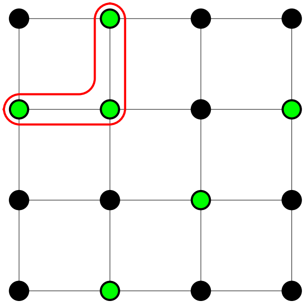
Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.

$\rightarrow$ 2. Let Bad be the set of vertices in connected components of size at least 2.

3. Resample $=$ Bad $\cup\ \partial$Bad.

4. Resample variables in set Resample. Check independence.

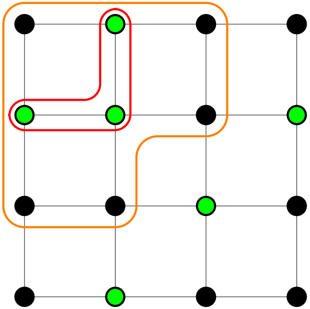


Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.

$\rightarrow$ 3. Resample $=$ Bad $\cup\, \partial$Bad.

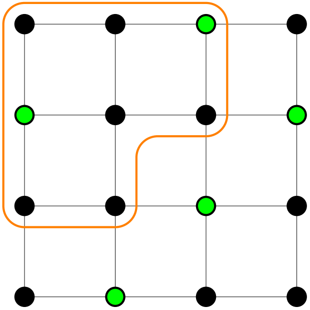
4. Resample variables in set Resample. Check independence.



Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.
3. Resample $=$ Bad $\cup\, \partial$Bad.

$\longrightarrow$ 4. Resample variables in set Resample. Check independence.



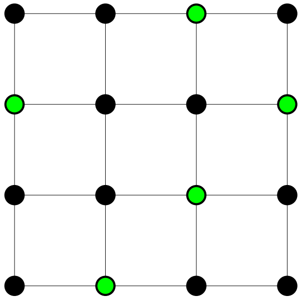
Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.

2. Let Bad be the set of vertices in connected components of size at least 2.

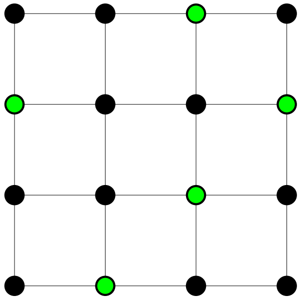
3. Resample $=$ Bad $\cup\, \partial$Bad.

$\rightarrow$ 4. Resample variables in set Resample. Check independence.



Sampling independent sets

1. Randomize each vertex (in/out). Consider the connected components induced by the in-vertices.
2. Let Bad be the set of vertices in connected components of size at least 2.
3. $\mathrm{Resample} = \mathrm{Bad} \cup \partial \mathrm{Bad}$.
4. Resample variables in set Resample. Check independence.



When the algorithm stops, it yields a uniform independent set.

Unblocking sets

The key property of the set Resample is that it is unblocking under the current assignment σ to the variables.

Definition

A set $\mathcal{U}$ of variables is unblocking under σ if $\sigma[\mathcal{U}]$ determines the truth value of all clauses that *share* variables with $\mathcal{U}$ (and not just the clauses containing *only* variables from $\mathcal{U}$

The resampling set from the independent set example was unblocking.

In applications we also require that $\mathcal{U}$ is “adapted” to σ.

Partial rejection sampling

Algorithm 1 Partial Rejection Sampling

```
$\mathrm{PRS}(V, \Phi)$ // $\Phi$ is a formula on variable set $V$
Sample, from the product distribution, an assignment $\sigma$ to the variables in $V$
while $\mathrm{Bad}(\sigma) \neq \emptyset$ do
    $S \leftarrow \bigcup\{\mathrm{var}(C) : C \in \mathrm{Bad}(\sigma)\}$
    Resample all variables in $U = \mathrm{Resample}(S; \sigma)$
end while
```

Note 1. The procedure Resample must not probe variables outside of U while computing U. This is the condition of being adapted.

Note 2. A previous approach to rejection sampling that tries to preserve randomness is the "Randomness Recycler" of Fill and Huber.

Partial rejection sampling: correctness

Let σ_t = (assignment after t iterations), and T = #iterations.

Prove by induction on t that, for all $t \leqslant T$, and all subsets $B \subseteq \Phi$ of clauses satisfying $\Pr(\mathrm{Bad}(\sigma_t) = B) > 0$,

$$\mathcal{D}(\sigma_t \mid \mathrm{Bad}(\sigma_t) = B) = \mathcal{D}(\pi \mid \mathrm{Bad}(\pi) = B).$$

where π denotes a sample from the product distribution.

Note that $\mathcal{D}(\sigma_0) = \mathcal{D}(\pi)$, so the induction hypothesis holds when $t = 0$. Note also that when $t = T$ the induction hypothesis expresses correctness of the algorithm.

Theorem (Guo, Jerrum and Liu, 2017)

When PRS halts, its output is uniform over satisfying assignments.

Sampling satisfying assignments to a k-CNF formula

Suppose we run PRS on a k-CNF formula Φ of degree d. (There are k variables per clause and each variable occurs at most d times.)

Theorem (Guo, Jerrum and Liu, 17)

PRS has linear expected running time if $d \leqslant \frac{1}{6e} \cdot 2^{k/2}$, and any two dependent clauses share at least $\min\{\log dk, k/2\}$ variables.

Sampling satisfying assignments to a k-CNF formula

Suppose we run PRS on a k-CNF formula Φ of degree d. (There are k variables per clause and each variable occurs at most d times.)

Theorem (Guo, Jerrum and Liu, 17)

PRS has linear expected running time if $d \leqslant \frac{1}{6e} \cdot 2^{k/2}$, and any two dependent clauses share at least $\min\{\log dk, k/2\}$ variables.

In contrast, uniform sampling of satisfying assignments is NP-hard even if $d \geqslant 5 \cdot 2^{k/2}$ and dependent clauses share $k/2$ variables [Bezáková, Galanis, Goldberg, Guo, Štefankovič, 2016].