2013

Sampling from the Hardcore Process

William C. Dodds
Claremont McKenna College

CLAREMONT McKENNA COLLEGE

SAMPLING FROM THE HARDCORE PROCESS

SUBMITTED TO

PROFESSOR MARK HUBER

AND

DEAN GREGORY HESS

BY

WILLIAM DODDS

FOR

SENIOR THESIS

SPRING 2013

APRIL 29TH, 2013

Abstract

Partially Recursive Acceptance Rejection (PRAR) and bounding chains used in conjunction with coupling from the past (CFTP) are two perfect simulation protocols which can be used to sample from a variety of unnormalized target distributions. This paper first examines and then implements these two protocols to sample from the hardcore gas process. We empirically determine the subset of the hardcore process's parameters for which these two algorithms run in polynomial time. Comparing the efficiency of these two algorithms, we find that PRAR runs much faster for small values of the hardcore process's parameter whereas the bounding chain approach is vastly superior for large values of the process's parameter.

Acknowledgments

First and foremost, I want to thank my reader, Professor Mark Huber, for his tremendous support throughout the entire process of writing this paper. In addition to guiding me to a topic and helping me through the many problems I encountered, your advice and comments have been an invaluable resource. Many, many thanks for working through this paper with me. A final thank you to my family and friends who supported me throughout the semester.

Contents

1 Introduction

An independent set I is a collection of nodes on state space $\{0,1\}^V$ such that no two nodes colored 1 are adjacent to each other. The hardcore process is a probability distribution π

$$\pi(I) = \lambda^{|I|}/z$$

where λ is the parameter of the problem and z is the normalizing constant for this distribution. A number of protocols have been suggested for sampling from π, two of which are Partially Recursive Acceptance Rejection (PRAR, see [2]) and bounding chains (see [4]) in conjunction with coupling from the past (CFTP, see [5]). This paper investigates experimentally which of these two algorithms is more efficient to sample from the hardcore process on a $k \times k$ lattice for a given value of λ. Implementing both of these protocols, we find that for relatively small values of λ, PRAR is vastly superior to using bounding chains; however, for large values of λ, bounding chains become far more efficient than PRAR.

This paper proceeds as follows: Section 2 discusses PRAR, Section 3 implements PRAR on a Binary Tree, Section 4 implements PRAR on a $k \times k$ lattice, Section 5 discusses using bounding chains, Section 6 implements the bounding chain and CFTP protocol on the $k \times k$ lattice, Section 7 compares the two protocols and presents the study's conclusions.

2 Partially Recursive Acceptance/Rejection

Acceptance/Rejection (AR) is a method of sampling from an unnormalized target distribution. Consider a discrete state space, Ω, with the property that $\mathbb{P}(X = x) \propto f(x)$, for some unnormalized weight function $f(x)$. Hence:

$$\mathbb{P}(X = x) = f(x)/z$$

for some normalizing constant z. Acceptance/Rejection eliminates the need to explicitly calculate z by using an enveloping distribution $g(x)$ on Ω such that $f(x) > 0 \Longrightarrow g(x) > 0$ and $(\forall x \in \Omega)(f(x) \leq g(x))$. Further, it must be possible to draw $X \sim g$. In its simplest form AR then draws $X \sim g$ and accepts this draw with probability $f(X)/g(X)$, thus avoiding the need to explicitly calculate the normalizing constant, z.

However, when using AR to sample from Markov random fields, the probability of acceptance can be very low; the running time of the algorithm can grow exponentially in the dimension of the problem. A better approach is to use a recursive form of AR. In essence, this form of AR allows us to draw each individual node one at a time and accept or reject it based solely upon the values of adjacent nodes.

Consider using AR on the hardcore process on a Markov random field over a graph $G = (V, E)$ with state space $\Omega = C^V$, where C is a set of colors, and target density of the form:

$$f(x) = \left[\prod_{v\in V} a(x(v))\right]\left[\prod_{\{i,j\}\in E} b(x(i),x(j))\right]$$

Definition The hardcore process is a Markov random field with $C = \{0,1\}$ with parameter $\lambda \geq 0$ such that λ is the weight factor given to each 1. In the notation introduced above:
$a(0) = 1, a(1) = \lambda, b(0,0) = b(0,1) = b(1,0), b(1,1) = 1.$

Consider then the problem of determining the probability that the root node in our hardcore process is a 1. We begin by drawing a value for the root node with distribution Unif$(\{0,1\})$. To use the recursive form of AR, start with a node set V; it is possible to partition V into $\{v\}$ and $V\backslash\{v\}$ for a particular node (the root node in the hardcore process), v. Using the notation introduced above:

$$f_{\{v\}} = a(x(v))$$

and

$$f_{V\backslash\{v\}} = \left[\prod_{v\in V\backslash\{v\}} a(x(v))\right]\left[\prod_{\{i,j\}\in E, i,j\in V\backslash\{v\}} b(x(i),x(j))\right].$$

If $g_{\{v\},V\backslash\{v\}} = f_{\{v\}} \cdot f_{V\backslash\{v\}}$, we can sample $X_1 \sim f_{\{v\}}$ and $X_2 \sim f_{V\backslash\{v\}}$ and then accept $X = (X_1, X_2)$ as a draw from f with probability:

$$\frac{f(X)}{g(X)} = \prod_{\{v,j\}\in E, j\in V\backslash\{v\}} b(x(v),x(j)).$$

We can do this process recursively by drawing $V\backslash\{v\}$ using the same algorithm.

However, to save time, in some circumstances it may be possible to determine the color of the root node by only building out a fraction of the recursive structure. For example, if the root node is a 0, we will accept it regardless of the adjacent nodes, so we're done. If, on the other hand, our root node is a 1, it is necessary to build out the process further using recursive calls. However, if we draw a 0 for any given node, we know we will accept it regardless of the values of adjacent nodes so we stop the recursive calls on that node. Because the method only uses a partial set of the recursive calls, this technique is named partially recursive acceptance/rejection (PRAR).

3 PRAR on a Binary Tree Hardcore Process

We first consider the hardcore process on a binary tree. In essence, each node on the infinite tree is colored either 0 or 1, but configurations with two adjacent 1's are rejected. The goal is to determine the probability that the root node of the tree is a 1. An algorithm to simulate this process is presented below:

Algorithm 1 PRAR1

```
 1: Input: $\lambda$
 2: Output: $X(v)$ from $f_{\{v\}}$
 3: repeat
 4:     $acceptflag \leftarrow$ false
 5:     $U \leftarrow$ Unif$([0,1])$
 6:     $X(v) \leftarrow 0$
 7:     if $U \leq \frac{\lambda}{(\lambda+1)}$ then
 8:         $X(v) \leftarrow 1$
 9:     end if
10:     if $X(v) = 0$ then
11:         $acceptflag \leftarrow$ true
12:     else
13:         $left \leftarrow PRAR1(\lambda)$
14:         if $left = 0$ then
15:             $right \leftarrow PRAR1(\lambda)$
16:         end if
17:     end if
18:     if $left = 0$ and $right = 0$ then
19:         $acceptflag \leftarrow$ true
20:     end if
21: until $acceptflag =$ true
```

Implementing this algorithm (code included in Appendix), we find estimates of $\mathbb{P}(Rootnode = 1)$ for various values of λ. See Figure 1. Notably, we find the algorithm runs in finite time for $\lambda \leq 0.82$, i.e., the critical value of λ is ~ 0.83.

Furher, one can analytically calculate the probability that the root node is a 1 in the binary tree setting. First, let $p = \lambda/(\lambda+1)$, the probability that a node is rolled a 1 and let $\mathbb{P}(Acceptance)$ be the probability that a node colored 1 is accepted.

Note that there are an infinite number of "ways" one can have a root node with a value of 1: You can first roll a 1 and accept it; you can roll a one, reject it, roll another one, accept it; you can roll a one, reject it, roll a one, reject it, roll a one, accept it; etc. Hence:

$$\begin{aligned}\mathbb{P}(Root = 1) = p \cdot \mathbb{P}(Acceptance) + \\ p \cdot [1 - \mathbb{P}(Acceptance)] \cdot p \cdot \mathbb{P}(Acceptance) + \\ p^2 \cdot [1 - \mathbb{P}(Acceptance)]^2 \cdot p \cdot \mathbb{P}(Acceptance) + ...\end{aligned}$$

Simplifying this geometric sequence yields:

$$\mathbb{P}(Root = 1) = \frac{p \cdot \mathbb{P}(Acceptance)}{1 + p\,(\mathbb{P}(Acceptance) - 1)} \tag{1}$$

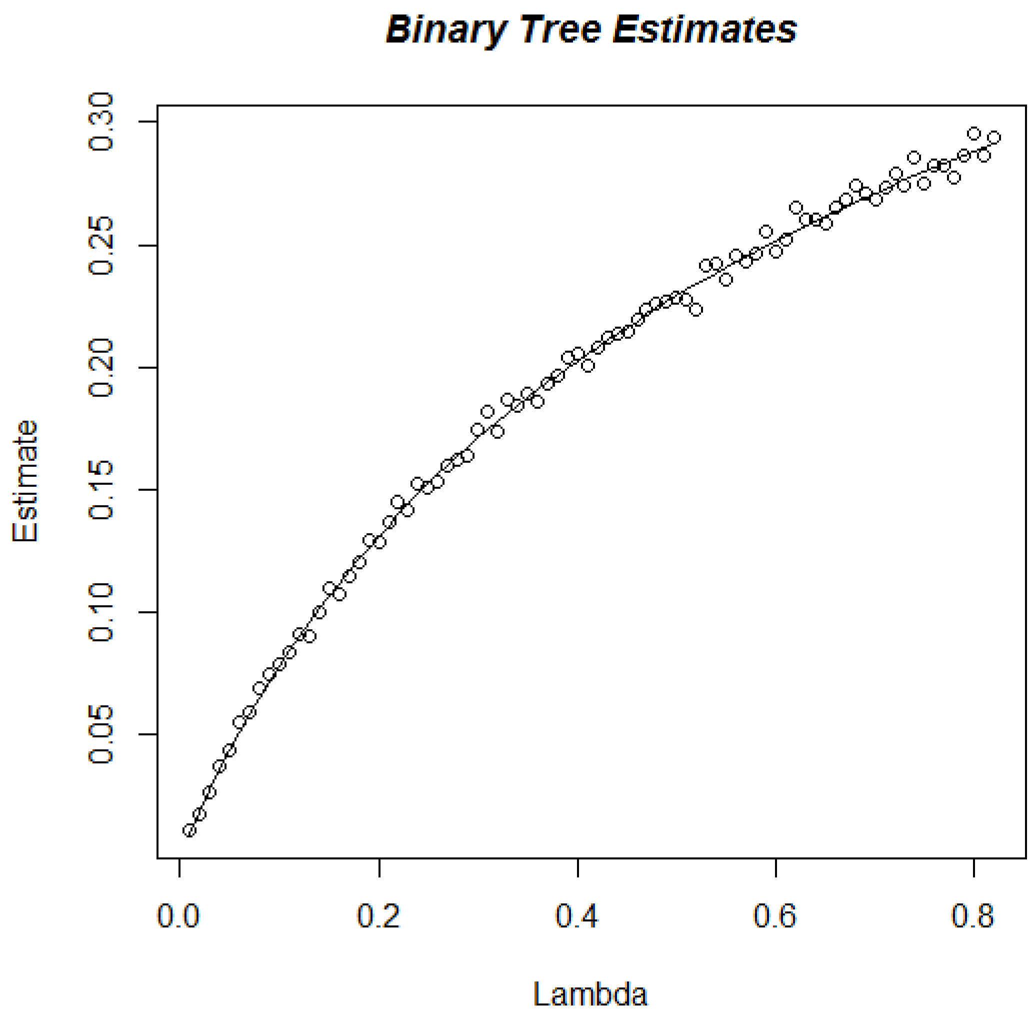

Figure 1: Points in the figure indicate the percentage of time the root node was a 1 for a given value of λ. The line shows the predicted percentages based on the equations derived below.

Furthermore, we can calculate $\mathbb{P}(\mathit{Acceptance})$ as for a node colored 1 to be accepted, both of its children nodes must be 0. So:

$$\mathbb{P}(\mathit{Acceptance}) = [1 - \mathbb{P}(\mathit{LeftChild} = 1)] \cdot [1 - \mathbb{P}(\mathit{RightChild} = 1)]$$

From the fact that $\mathbb{P}(\mathit{Root} = 1) = \mathbb{P}(\mathit{LeftChild} = 1) = \mathbb{P}(\mathit{RightChild} = 1)$, we get (2):

$$\mathbb{P}(\mathit{Acceptance}) = \left[1 - \frac{p \cdot \mathbb{P}(\mathit{Acceptance})}{1 + p\,(\mathbb{P}(\mathit{Acceptance}) - 1)}\right]^2 \tag{2}$$

From (2), (1), and $p = \lambda/(\lambda+1)$, we can calculate $\mathbb{P}(\mathit{Root} = 1)$ entirely in terms of the parameter, λ. This is plotted in the above figure as a straight line through the estimated probabilities.

To find the critical value of λ then, we look at the expected number of tosses, denoted $\mathbb{E}(T)$ as a function of p and $\mathbb{P}(\mathit{Root} = 1)$:

$$\mathbb{E}(T) = (1-p) + p\,[\mathbb{E}(T) + \mathbb{P}(\mathit{Root} = 1) \cdot \mathbb{E}(T) + (1 - \mathbb{P}(\mathit{Root} = 1))\,[\mathbb{E}(T) + \mathbb{P}(\mathit{Root} = 1) \cdot \mathbb{E}(T)]]$$

If $[1 - p\,(2 + \mathbb{P}(\mathit{Root} = 1) \cdot (1 - \mathbb{P}(\mathit{Root} = 1)))] > 0$, we can simplify the above equation to:

$$\mathbb{E}(T) = \frac{1-p}{1 - p\,(2 + \mathbb{P}(\mathit{Root} = 1)\,(1 - \mathbb{P}(\mathit{Root} = 1)))} \tag{3}$$

Numerically solving for the critical value of λ, i.e., the value of λ for which the root node is either accepted or rejected in finite time with probability 1, I find $\lambda_{crit} \sim 0.8284$. A graph of the number of tosses taken by the algorithm can be seen in Figure 2.

4 PRAR on a Lattice Hardcore Process

In this section we turn to look at the same hardcore process described above on a $k \times k$ lattice. Again, the end goal here is to describe the probability that the root node of the lattice is a 1. The PRAR algorithm on the $k \times k$ lattice is slightly more involved (see Algorithm 2).

Implementing this algorithm, the first thing we can discover is the probability that our root node is a 1 as a function of λ for various values of k. Graphs of this probability vs. λ for $k = 4, 5$, and 10 can be seen in Figure 3, Figure 4, and Figure 5, respectively. Furthermore, as a check to determine whether the output from this algorithm is actually $X(v)$ from $f_{\{v\}}$, we can use a brute force method for a low dimensional field to determine exactly the probability of the root node being colored 1 for all values of λ.

The next phenomenon that we're interested in looking at is the run-time, or number of tosses required as a function of λ. Figure 6 shows the number of tosses required, per trial, to generate a sample from the hardcore process.

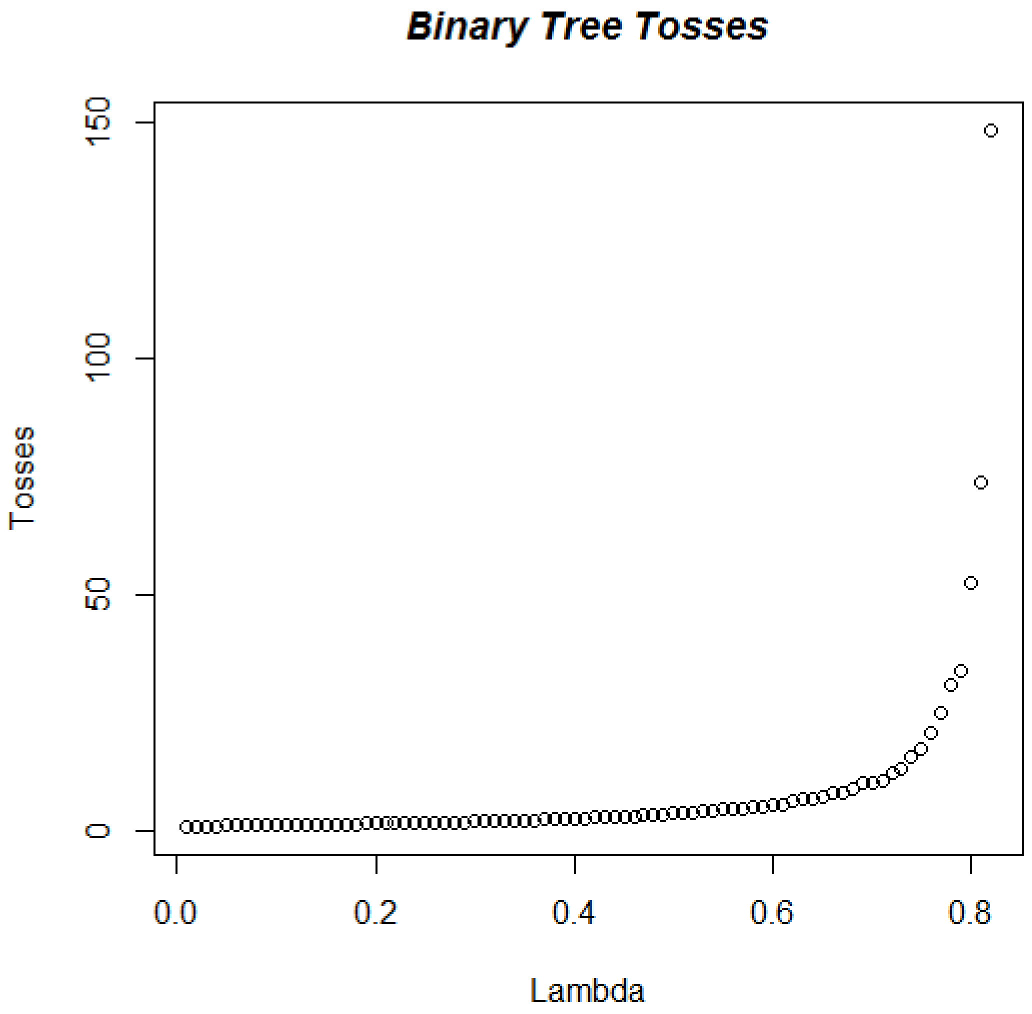

Figure 2: Each point represents the number of tosses per trial, for a specific value of λ that Algorithm 1 needed to determine the color of the root node.

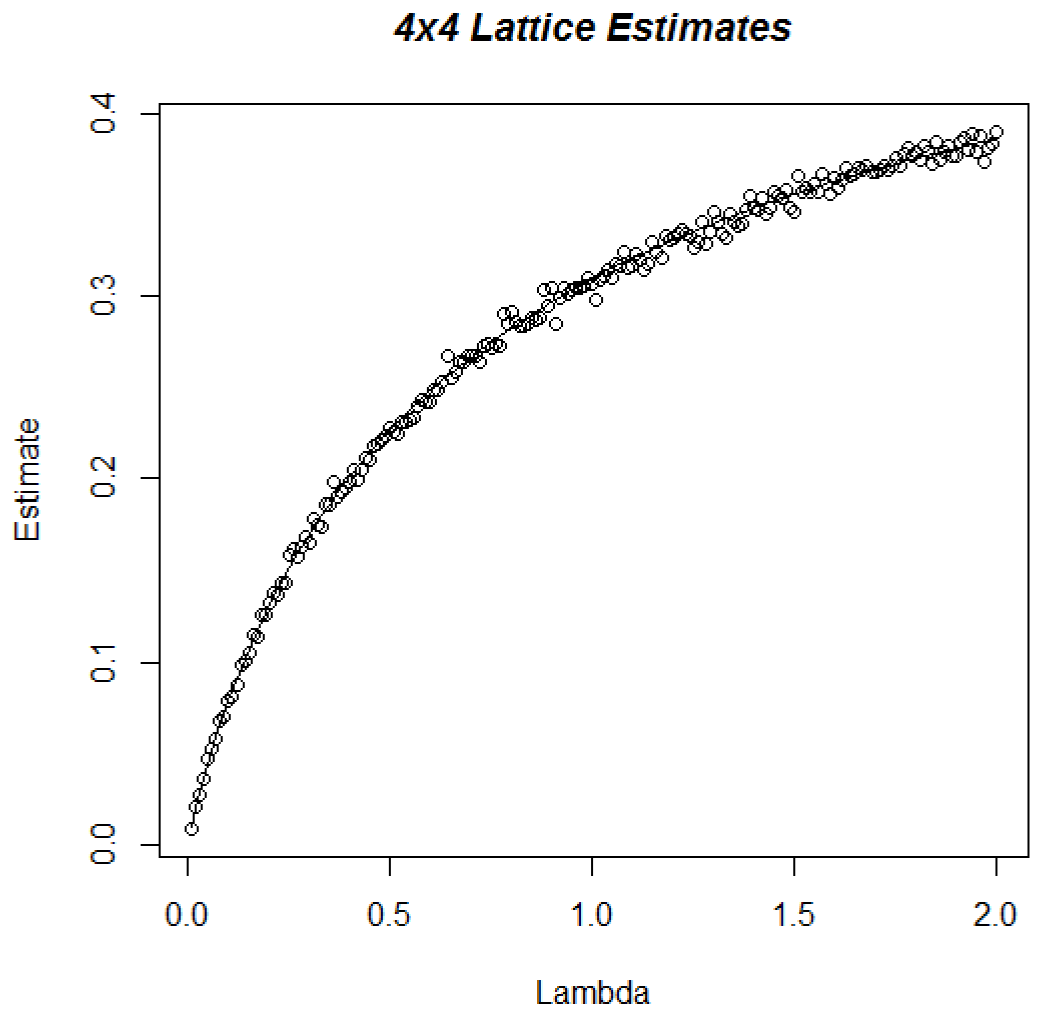

Figure 3: Each point represents the probability that the root node is a 1 for a specific value of λ on the 4×4 lattice. The line through the points denotes the actually probability that the root node is a 1 for each value of λ calculated by generating all possible 4×4 lattices.

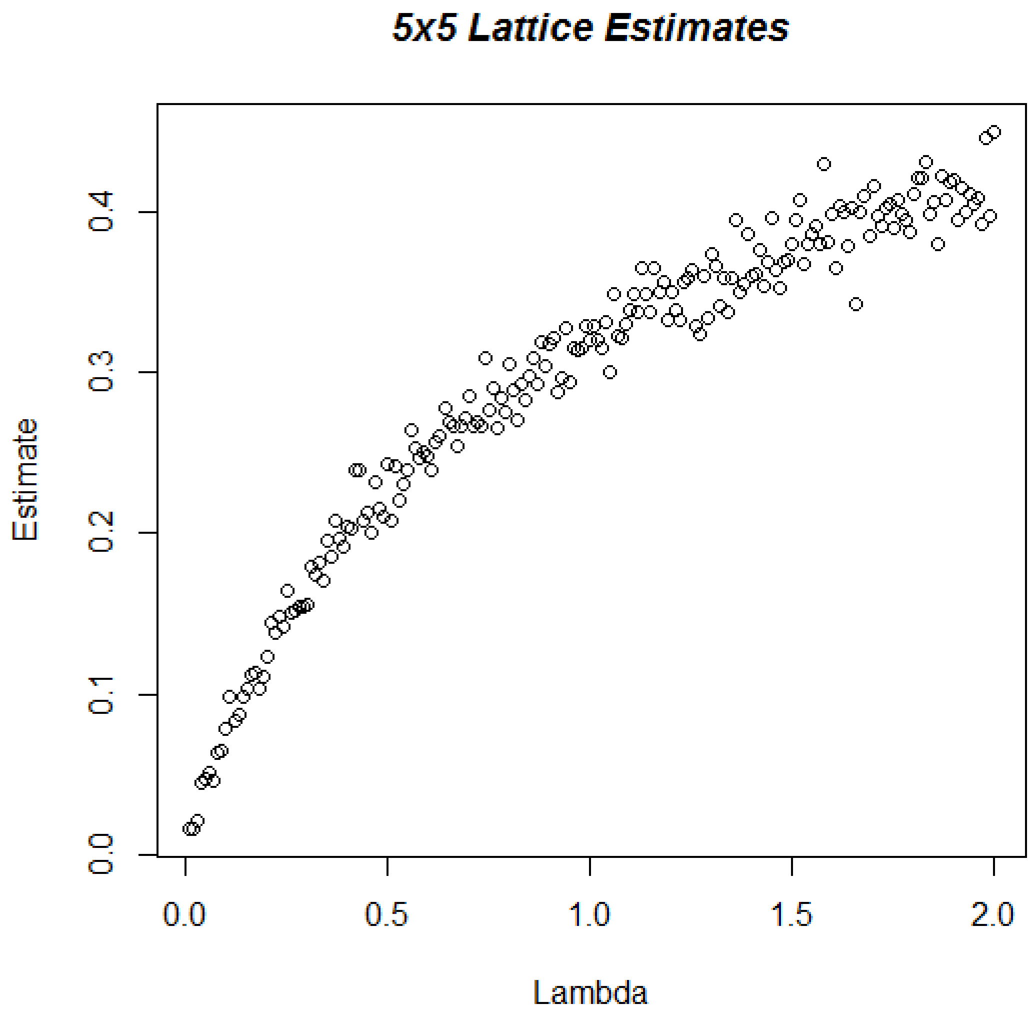

Figure 4: Each point represents the probability that the root node is a 1 for a specific value of λ on the 5×5 lattice.

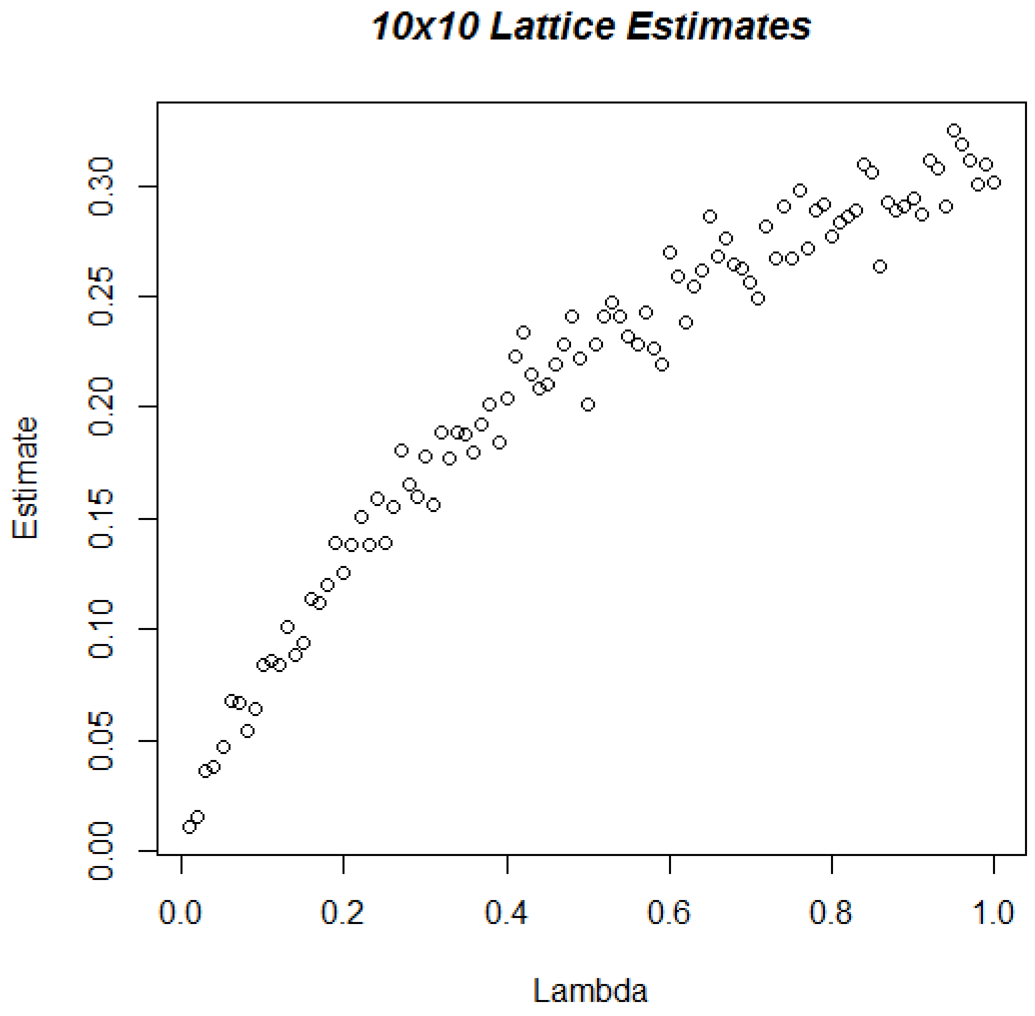

Figure 5: Each point represents the probability that the root node is a 1 for a specific value of λ on the 10×10 lattice.

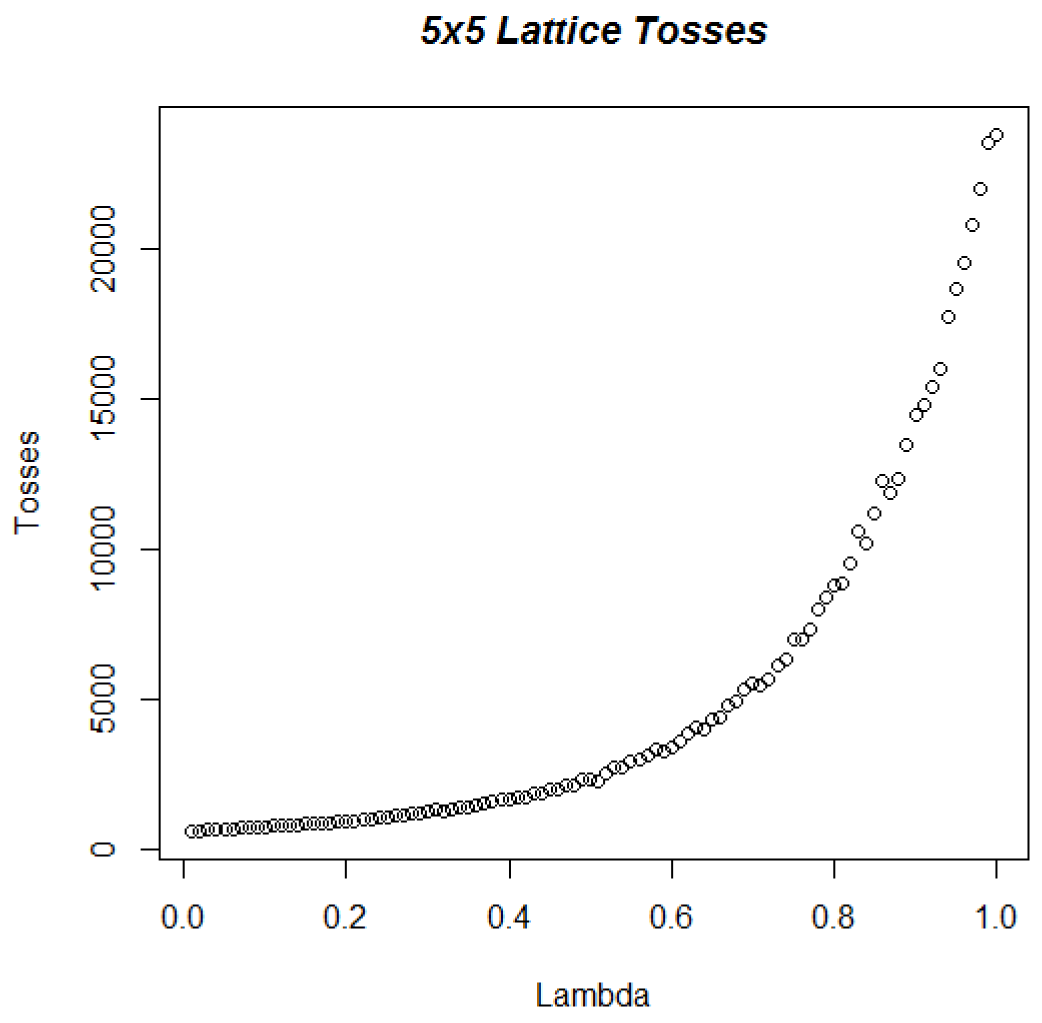

Figure 6: Each point represents the number of tosses per trial, for a specific value of λ, that Algorithm 2 needed to sample from the hardcore process on the 5×5 lattice.

Algorithm 2 PRAR2

```
 1: Input: $\lambda$, $Location$
 2: Output: $X(v)$ from $f_{\{v\}}$
 3: repeat
 4:    $acceptflag \leftarrow$ false
 5:    $U \leftarrow \textbf{Unif}([0,1])$
 6:    $X(v) \leftarrow 0$
 7:    if $U \leq \frac{\lambda}{(\lambda+1)}$ then
 8:       $X(v) \leftarrow 1$
 9:    end if
10:    if $X(v) = 0$ then
11:       $acceptflag \leftarrow$ true
12:    else
13:       repeat
14:          Draw $Adjacent_i$ conditional on $Location$
15:          $Adjacent_i \leftarrow PRAR2(\lambda, Location)$
16:       until ($Adjacent_i = 1$ or no more adjacent nodes exist)
17:    end if
18:    if $Adjacent_1 = Adjacent_2 = ... = Adjacent_n = 0$ then
19:       $acceptflag \leftarrow$ true
20:    end if
21: until $acceptflag$ = true
```

However, the more pertinent question surrounding run-times is that of determining the set of λ for which run-time grows polynomially, rather than exponentially, in the dimension of the problem (i.e., the size of the lattice). To get an idea of an upper bound on this critical λ, we can look at the average required number of tosses, per trial, to get an accepted draw from the hardcore process. For values of λ below ≈ 0.5, the relationship between the dimension of the problem and the required number of tosses is roughly linear. This is illustrated in Figure 7 below. For values of λ between 0.5 and ≈ 0.67, the relationship between the dimension of the lattice and the run time is polynomial. This is illustrated in Figure 8 below. Finally, for λ above approximately 0.7, the required number of tosses is growing faster than polynomially in the dimension of the problem. This can be seen in Figure 9. While it is somewhat difficult to tell whether or not PRAR is running in polynomial time when $\lambda = 0.7$, it is certainly no longer polynomial when $\lambda = 0.75$ (see Figure 10). Hence, we can conclude that for values of $\lambda \leq 0.67$, the PRAR algorithm, implemented on the hardcore process, runs in time polynomial in the dimension of the problem.

5 Bounding Chains

Another way to sample perfectly from the hardcore process is to use bounding chains in conjunction with coupling from the past (see [3] and [4]). To use a

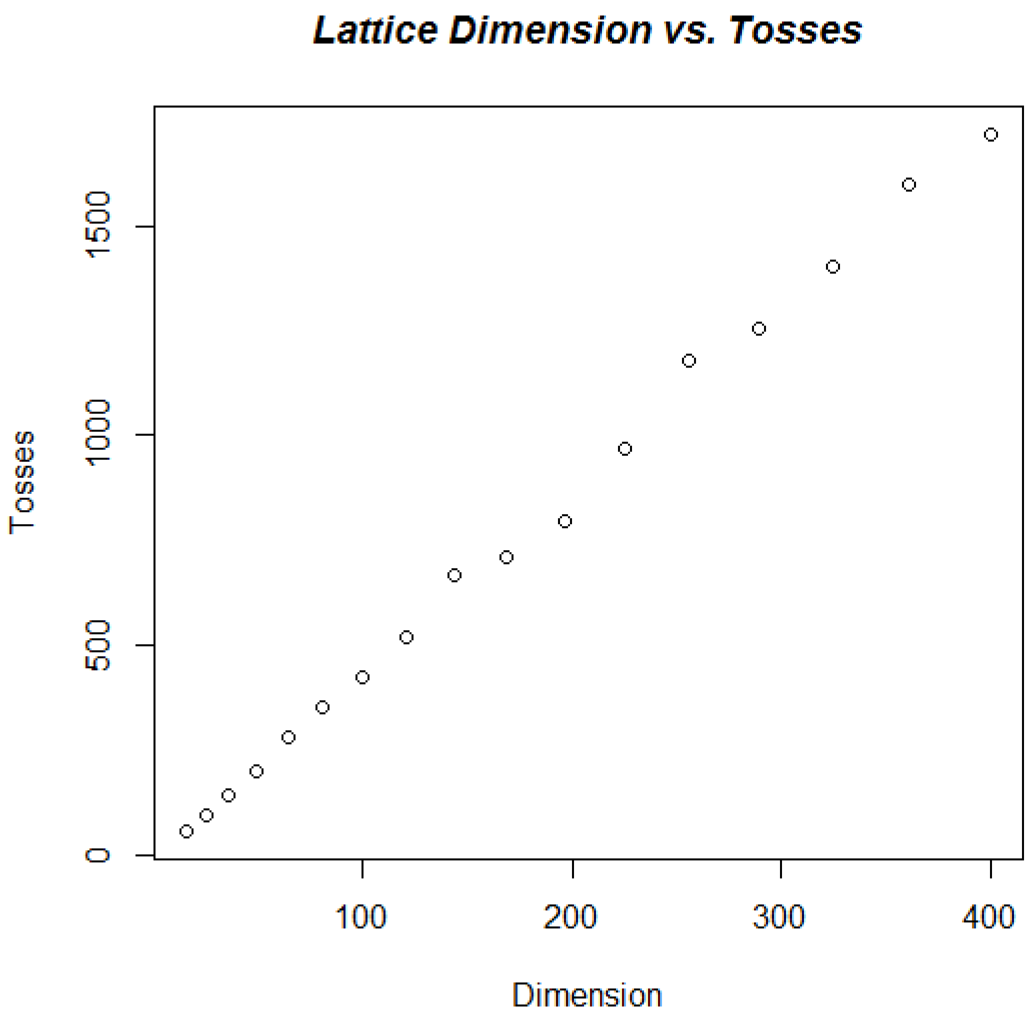

Figure 7: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice that Algorithm 2 is working on for $\lambda = 0.5$. Note that the relationship is linear.

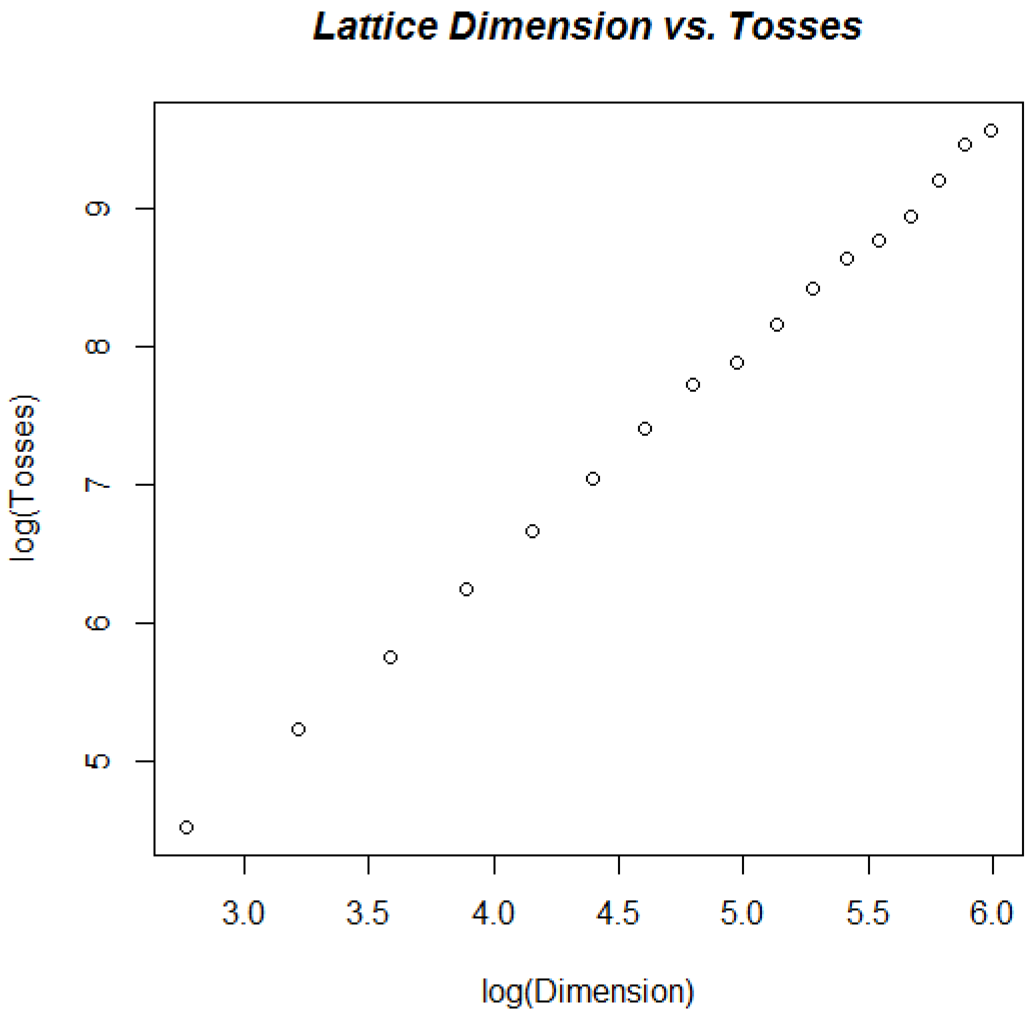

Figure 8: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice that Algorithm 2 is working on (both axes are plotted on a log scale) for $\lambda = 0.67$. As the log-log plot of dimension vs. required tosses is linear, the relationship between dimension and tosses is polynomial.

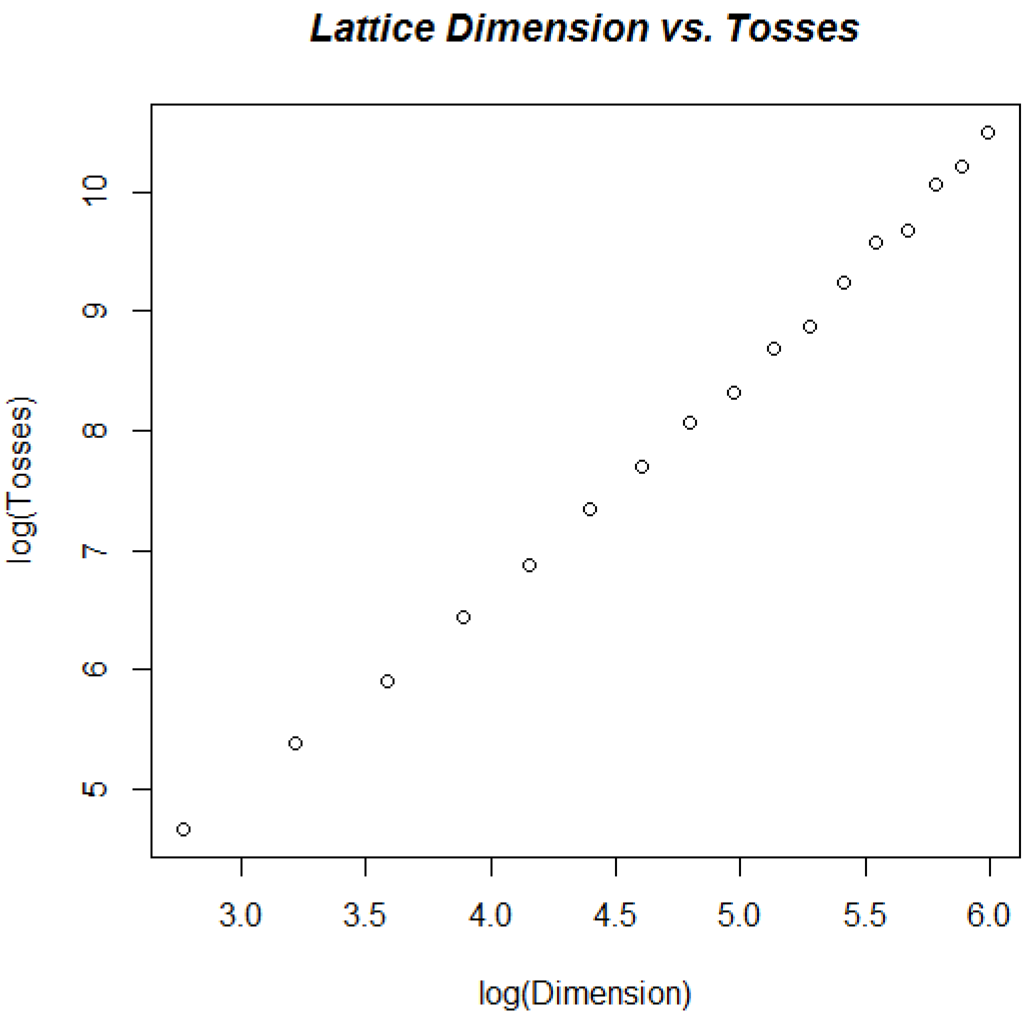

Figure 9: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice that Algorithm 2 is working on (both axes are plotted on a log scale) for $\lambda = 0.7$. As the log-log plot of dimension vs. required tosses is no longer linear, the relationship between dimension and tosses is not polynomial.

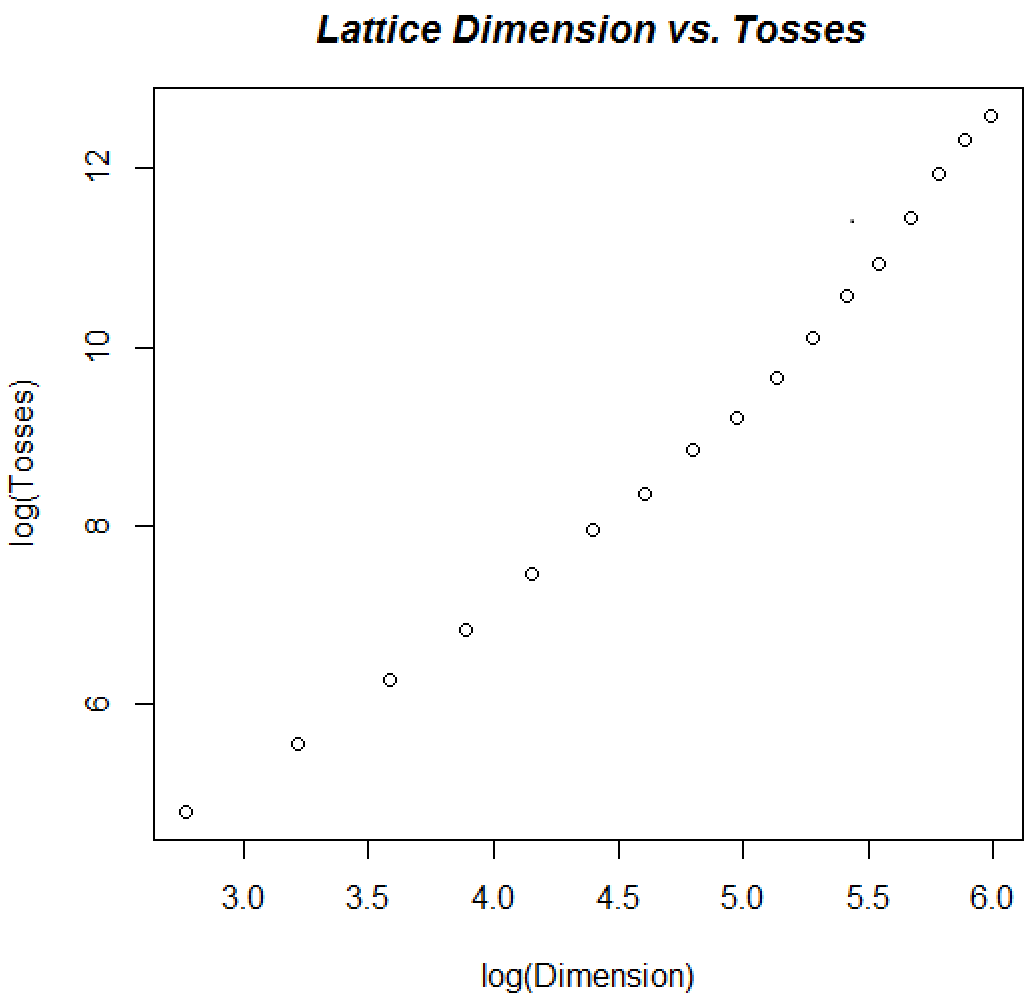

Figure 10: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice that Algorithm 2 is working on (both axes are plotted on a log scale) for $\lambda = 0.75$. As the log-log plot of dimension vs. required tosses is no longer linear, the relationship between dimension and tosses is not polynomial.

bounding chain to sample from the hardcore process, we first need a Markov chain that is stationary over the hardcore process. While both a Gibbs sampler or a Metropolis-Hastings algorithm could be used, Dyer and Greenhill present a new Markov chain, hereafter referred to as the Dyer-Greenhill chain, which mixes faster than either a Gibbs sampler or a Metropolis-Hastings algorithm for the hardcore process. First, we present the algorithm to take a step in the Dyer-Greenhill chain (Algorithm 3), then we present the corresponding bounding chain.

For a lattice with node set V, the Dyer-Greenhill chain, letting N_v denote the neighbors of a node v:

Algorithm 3 Dyer-Greenhill Chain

```
1: Input: State $X$, $\lambda$, $p_{swap}$
2: Output: New State $X$
3: $v \leftarrow$ Unif$(V)$
4: $U \leftarrow$ Unif$([0,1])$
5: if $U > \frac{\lambda}{(\lambda+1)}$ then
6:     $X(v) \leftarrow 0$
7: else if $U < \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains no elements colored 1 then
8:     $X(v) \leftarrow 1$
9: else if $U < p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains exactly one element colored
   1 (call it $w$) then
10:    $X(v) \leftarrow 1$ and $X(w) \leftarrow 0$
11: end if
```

Essentially, the Dyer-Greenhill chain is very similar to a regular Gibbs sampler, in that it picks a node v at random and changes it based on probabilities that depend on the neighbors, N_v, of v. The difference, however, lies in the fact that the Dyer-Greenhill chain invokes a new parameter, p_{swap}, which can take on any value between 0 and 1, that determines the probability that the chain makes a new move - swapping two adjacent nodes. However, this new move can only occur if exactly one adjacent node is colored 1, at which point the chain changes the color of v to 1 and the color of the adjacent node to 0.

The idea behind creating a bounding chain for a process X on the Dyer-Greenhill chain is to create a new process Y, such that $X_t(v) \in Y_t(v) \implies X_{t+1}(v) \in Y_{t+1}(v)$. In other words, $Y(v)$ contains all the possible values that $X(v)$ could possibly be at any given time step in the chain. For the Dyer-Greenhill chain, each node v is colored 0 or 1, so each element of our bounding chain Y is colored 0, 1, or $\{0,1\}$. The chain begins such that $Y_0(v) = \{0,1\}\ \forall v \in V$, denoting essentially that all the values of X_0 are unknown. If it is the case after t steps in the bounding chain that $|Y_t(v)| = 1\ \forall v$, it must be the case that $X_t = Y_t$, and hence X_t is a draw from the stationary distribution as Y starts in the stationary distribution. The bounding chain for the Dyer-Greenhill chain (Algorithm 4) considers six possible cases, determining how to take a step in the chain for a given set of a neighbors of a node v.

Algorithm 4 Bounding Chain for the Dyer-Greenhill Chain

```
1: Input: State $Y$, $\lambda$, $p_{swap}$
2: Output: New State $Y$
3: $v \leftarrow$ Unif$(V)$
4: $U \leftarrow$ Unif$([0,1])$
5: if $U > \frac{\lambda}{(\lambda+1)}$ then
6:     $Y(v) \leftarrow 0$
7: else
8:     if $N_v$ contains no elements colored $1$ and $N_v$ contains no elements colored $\{0,1\}$ then
9:         $Y(v) \leftarrow 1$
10:    else if $U \leq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains exactly one element colored $1$ (call it $w$) and $N_v$ contains no elements colored $\{0,1\}$ then
11:        $Y(v) \leftarrow 1$ and $Y(w) \leftarrow 0$
12:    else if $U \geq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains exactly one element colored $1$ (call it $w$) and $N_v$ contains no elements colored $\{0,1\}$ then
13:        $Y(v) \leftarrow 0$
14:    else if $N_v$ contains more than one element colored $1$ then
15:        $Y(v) \leftarrow 0$
16:    else if $U \leq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains exactly one element colored $\{0,1\}$ (call it $w$) and $N_v$ contains no elements colored $1$ then
17:        $Y(v) \leftarrow 1$ and $Y(w) \leftarrow 0$
18:    else if $U \geq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains exactly one element colored $\{0,1\}$ (call it $w$) and $N_v$ contains no elements colored $1$ then
19:        $Y(v) \leftarrow \{0,1\}$
20:    else if $U \leq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains at least one element colored $\{0,1\}$ (call it $w$) and $N_v$ contains 1 element colored $1$ then
21:        $Y(v) \leftarrow \{0,1\}$ and $Y(w) \leftarrow \{0,1\}$
22:    else if $U \geq p_{swap} \times \frac{\lambda}{(\lambda+1)}$ and $N_v$ contains at least one element colored $\{0,1\}$ (call it $w$) and $N_v$ contains 1 element colored $1$ then
23:        $Y(v) \leftarrow \{0,1\}$
24:    else if $N_v$ contains more than one element colored $\{0,1\}$ and $N_v$ contains no elements colored $1$ then
25:        $Y(v) \leftarrow \{0,1\}$
26:    end if
27: end if
```

First, note that if all neighbors of v are known, we are simply moving according to the Dyer-Greenhill chain. If no neighbors are colored 1, but one is colored $\{0,1\}$, then if we roll to swap, v is a 1 and the unknown neighbor is now a known 0. If no neighbors are colored 1, but one is colored $\{0,1\}$, and we roll for v to be a 1, we must change it to $\{0,1\}$. If two or more neighbors are colored $\{0,1\}$, then we do not know the value of v, so it must be changed to $\{0,1\}$. The worst case scenario is if one neighbor is colored 1 and at least one neighbor is colored $\{0,1\}$. If we roll to swap, both v and the neighbor colored 1 are switched to $\{0,1\}$, and if we do not roll to swap v changes to $\{0,1\}$.

An algorithm using this bounding chain to sample exactly from the hardcore process using coupling from the past can be found in Section 6.

6 Bounding Chain on a Lattice Hardcore Process

In order to use the theory developed in the previous section to sample perfectly from the hardcore process, we use the bounding chain in conjunction with coupling from the past (CFTP). In essence, what this algorithm does is start with $Y(v) = \{0,1\}\ \forall v$, and then use t uniform random variates to run the bounding chain forward t steps. If $|Y_t(v)| = 1\ \forall v$, we are done and we return Y as our draw from the hardcore process. If, however, $|Y_t(v)| > 1$ for some v, we recursively call our CFTP protocol to generate a new Y_0, essentially going back in time to generate our new initial state. Then, we use the same original t uniforms to run our new Y_0 forward to Y_t, finally returning Y_t as our draw from the hardcore process. This is presented in Algorithm 5.

Algorithm 5 CFTP for Dyer-Greenhill Bounding Chain

```
1:  Input: $t$
2:  Output: $Y$ from the hardcore process
3:  $U_1, U_2, \ldots, U_t \leftarrow$ Unif$([0,1])$
4:  $Y_0(v) \leftarrow \{0,1\}\ \forall v$
5:  Use Algorithm 4 and $U_1, U_2, \ldots, U_t$ to run $Y_0$ forward to $Y_t$
6:  if $|Y_t(v)| = 1\ \forall v$ then
7:      Return $Y_t$
8:  else
9:      $Y_0 \leftarrow$ CFTP$(2t)$
10:     Use Algorithm 4 and $U_1, U_2, \ldots, U_t$ to run $Y_0$ forward to $Y_t$
11:     Return $Y_t$
12: end if
```

The first thing that we can look at by implementing this algorithm on a $k \times k$ lattice is the probability that the root node is a 1 for various values of λ and a given value of k. Graphs of this probability vs. λ for $k = 4, 5$, and 10 can be seen in Figure 11, Figure 12, and Figure 13, respectively. Finally, to determine whether the output from this algorithm is actually $X(v)$ from $f_{\{v\}}$, we use a brute force method on the 4×4 lattice to determine exactly the probability of the root node being colored 1 for all values of λ.

Potentially, the next phenomenon we are interested in around Algorithm 5 is the number of uniform random variates required to generate a draw from hardcore process. This is shown below in Figure 14 for the 5×5 lattice.

Finally, the most pertinent question surrounding run-times is that of determining which values of λ run-time grows polynomially, rather than exponentially, in the dimension of the problem (i.e., the size of the lattice). To find out where run-time is growing polynomially, we can look at the average required number of tosses, per trial, to get an accepted draw from the hardcore process. In [4], Huber proves that for $\lambda \leq 1$, Algorithm 5 will run in polynomial time. We experimentally verify this as well, noting that for $\lambda \leq 1.5$ Algorithm 5 runs in polynomial time. This is shown below in Figure 15 and 16.

7 Conclusion

Quite possibly the most important experimental finding of our work is the determination of the values of λ for which PRAR is faster than using a bounding chain in tandem with coupling from the past to sample from the hardcore process. The general finding here is that for small values of λ, PRAR is faster, often by many orders of magnitude, than a bounding chain approach to sampling from the hardcore process on a $k \times k$ lattice. However, at around the point that PRAR appears to be no longer polynomial in the dimension of the lattice, the bounding chain protocol seems to become more efficient. Finally, for large values of λ, the bounding chain approach is significantly more efficient, often by many orders of magnitude, than using PRAR. Figure 17 displays the required number of tosses to get a sample from the hardcore process on a 20×20 lattice using both Algorithm 2 and Algorithm 5.

For values of λ below roughly 0.67, PRAR is significantly faster than using a bounding chain. For example, on a 20×20 lattice, PRAR used $\sim 11,000$ uniforms to generate a draw from the hardcore process whereas the bounding chain algorithm used $\sim 32,000$ uniforms. The difference is even starker if λ is slightly lower. By the time that $\lambda = 0.7$, the bounding chain approach is slightly faster than PRAR. On a 20×20 lattice, PRAR required approximately 37,000 uniforms to generate a draw from the hardcore process whereas the bounding chain approach needed only 32,000 uniforms. The difference becomes even more pronounced for slightly larger values of λ. Again on the 20×20 lattice, when $\lambda = 0.8$, PRAR requires just over 8,000,000 uniforms to generate a single draw from the hardcore process whereas using a bounding chain requires roughly 36,000 uniforms. Finally, when $\lambda = 1.5$, the bounding chain approach required only 66,000 uniforms to generate a sample from the hardcore process on the 20×20 lattice, demonstrating that it is vastly more efficient at higher values of λ than PRAR. Hence, we conclude that for small values of λ, using PRAR is faster than using a bounding chain protocol, whereas for large values of λ a

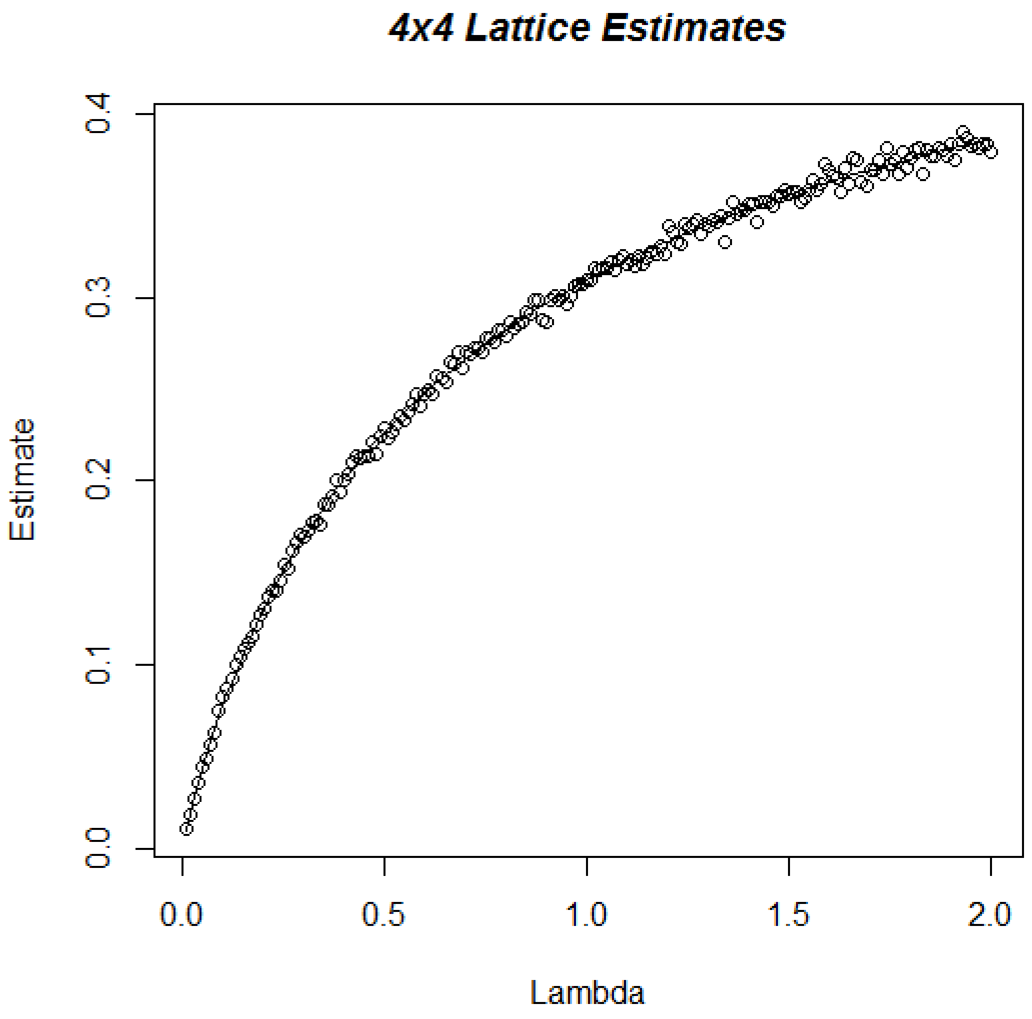

Figure 11: Each point represents the probability that the root node is a 1 for a specific value of λ on the 4×4 lattice. The line through the points denotes the actually probability that the root node is a 1 for each value of λ calculated by generating all possible 4×4 lattices.

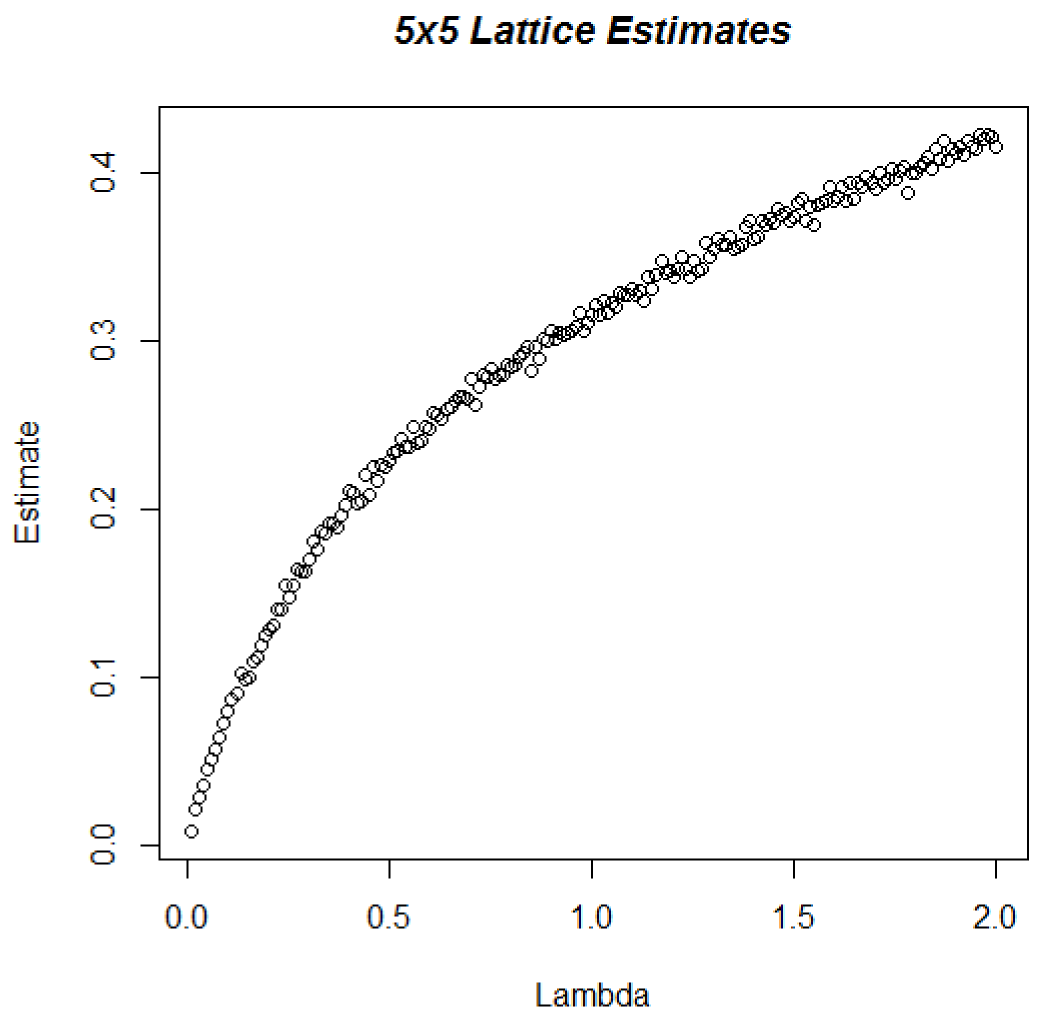

Figure 12: Each point represents the probability that the root node is a 1 for a specific value of λ on the 5×5 lattice.

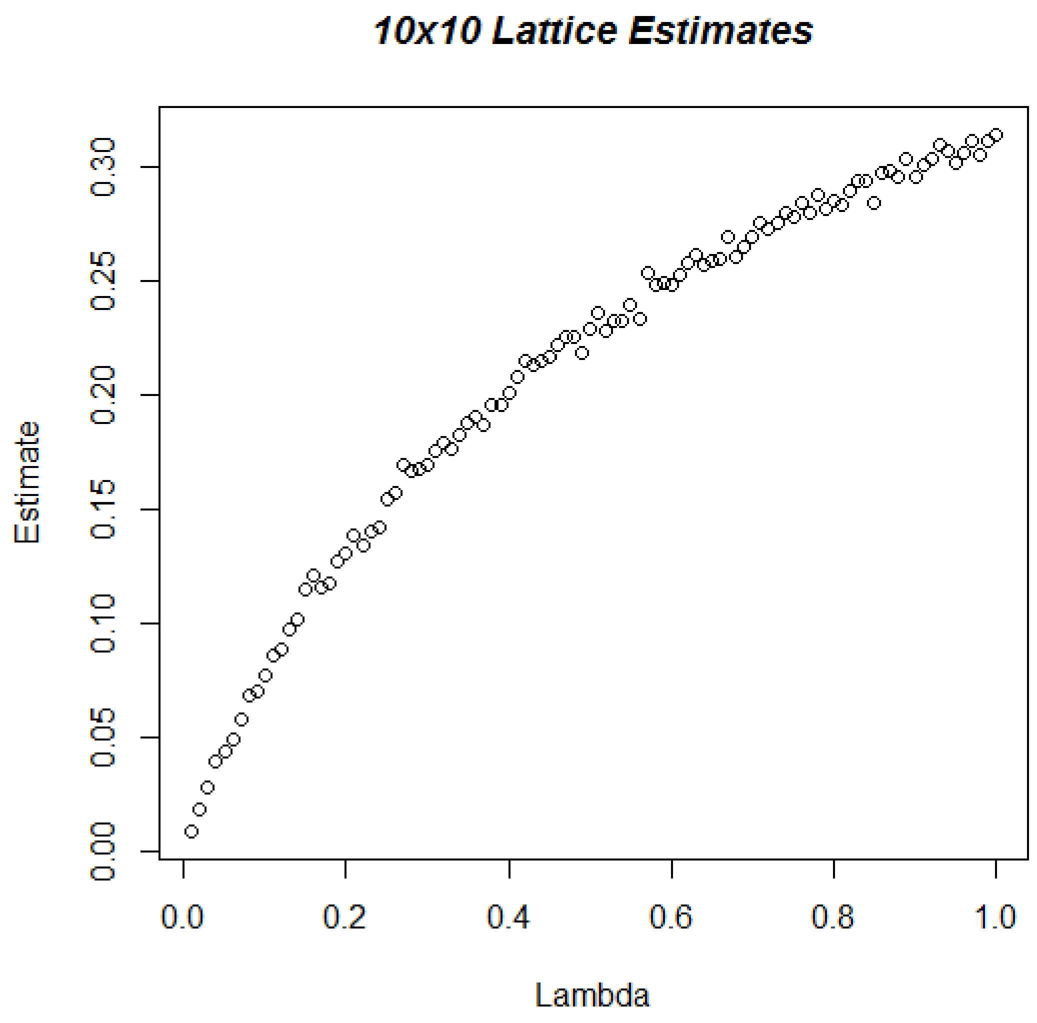

Figure 13: Each point represents the probability that the root node is a 1 for a specific value of λ on the 10×10 lattice.

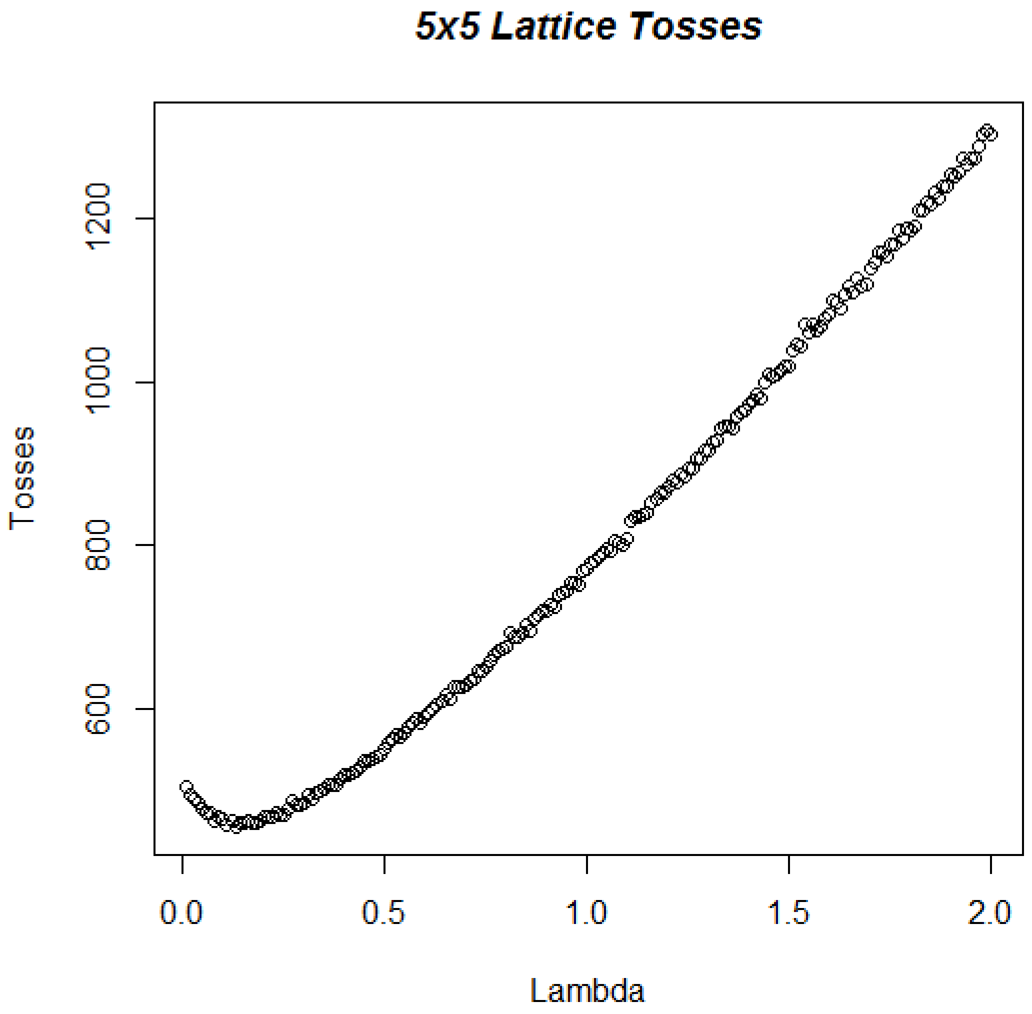

Figure 14: Each point represents the number of tosses per trial, for a specific value of λ, that Algorithm 5 needed to sample from the hardcore process on the 5×5 lattice.

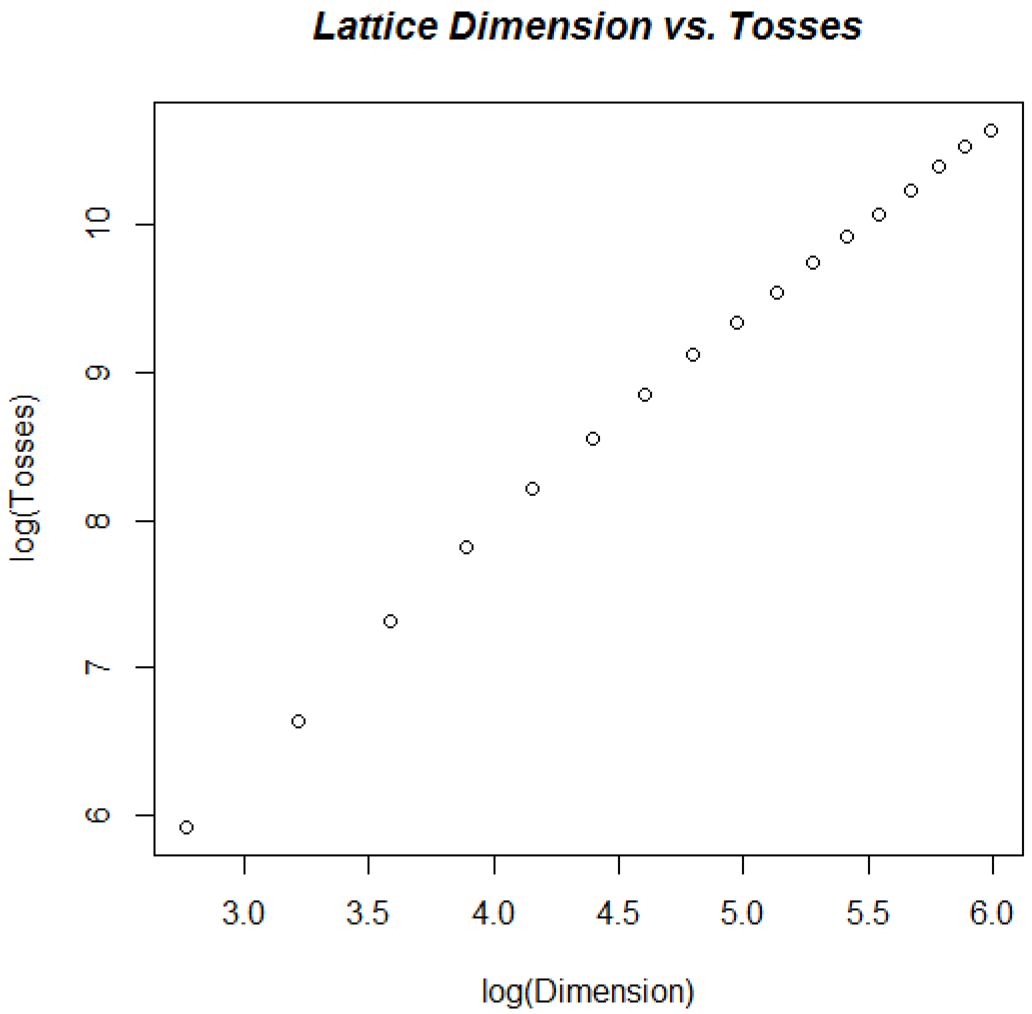

Figure 15: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice (both axes are plotted on a log scale) for $\lambda = 1$ using Algorithm 5. As the log-log plot of dimension vs. required tosses is linear, the relationship between dimension and tosses is polynomial.

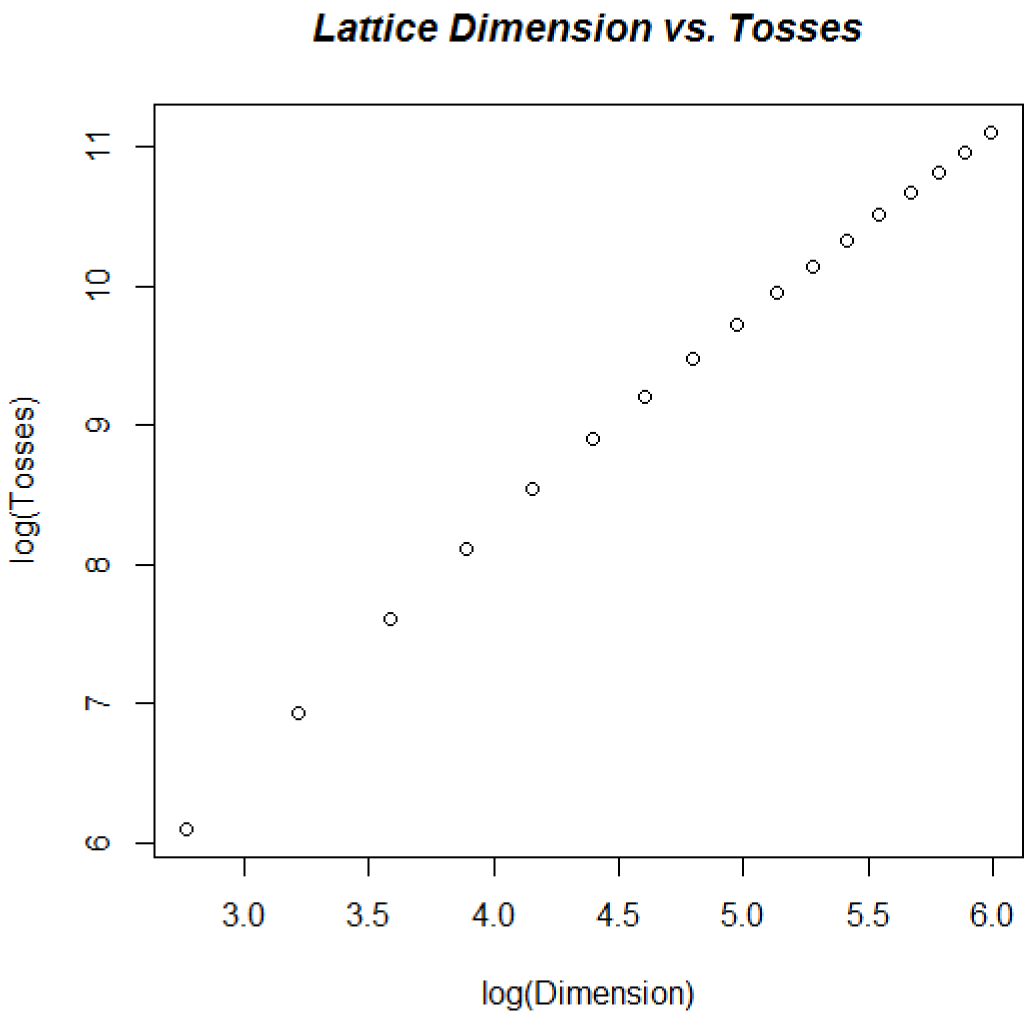

Figure 16: This figure describes the relationship between the required number of tosses, per trial, to get a sample from the hardcore process vs. the dimension of the lattice (both axes are plotted on a log scale) for $\lambda = 1.5$ using Algorithm 5. As the log-log plot of dimension vs. required tosses is linear, the relationship between dimension and tosses is polynomial.

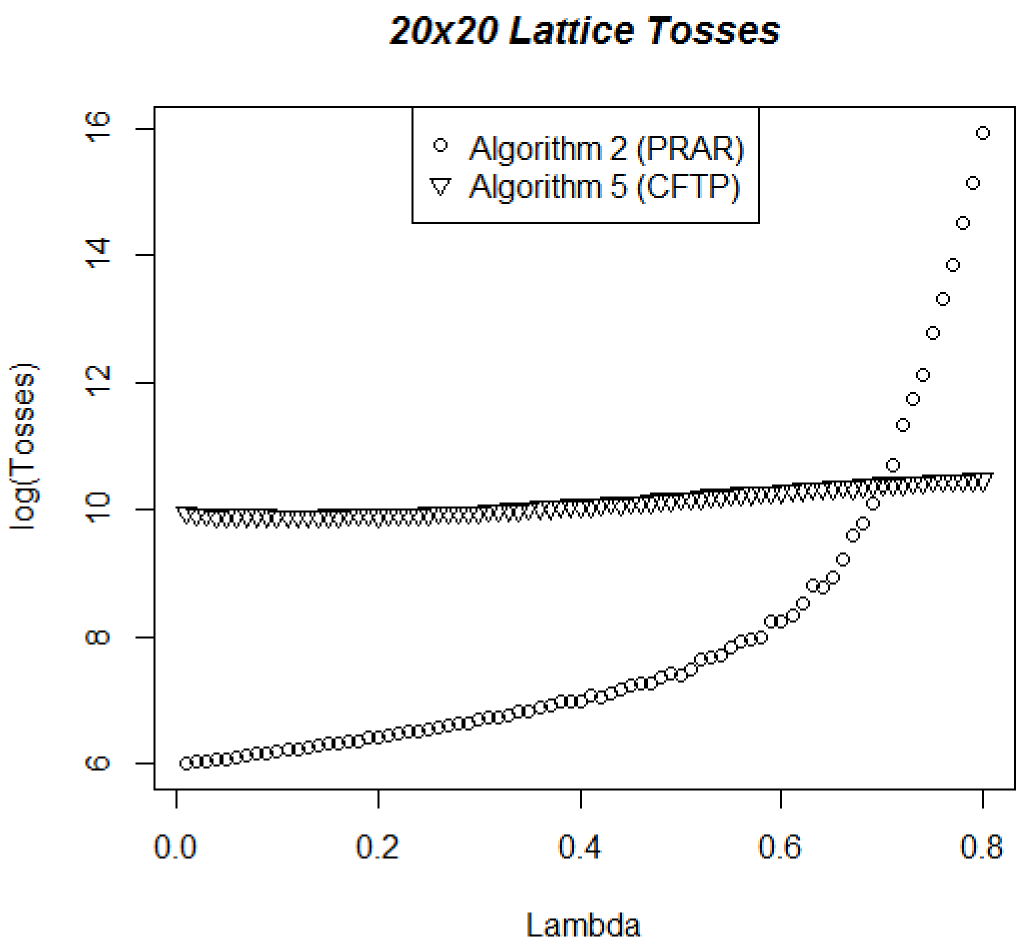

Figure 17: This figure presents a comparison of Algorithm 2 and Algorithm 5. Each point represents the number of tosses per trial, for a specific value of λ, needed to sample from the hardcore process on the 20×20 lattice.

bounding chain algorithm is significantly more efficient.

References

[1] M. Dyer and C. Greenhill. *On Markov chains for independent sets.* J. Algorithms, 2000.

[2] M. Huber. *Perfect simulation using partially recursive acceptance/rejection.* 2012.

[3] M. Huber. *A faster method for sampling independent sets.* 1999.

[4] M. Huber. *Perfect Sampling Using Bounding Chains.* Annals of Applied Probability, 1999.

[5] J.G. Propp and D.B. Wilson. *Exact Sampling with coupled Markov chains and applications to statistical mechanics.* Random Structures Algorithms, 1996.

A R Code for Algorithms

A.1 PRAR on Binary Tree Code

```
# This function creates the binary tree and returns the value of the root node. (Algorithm 1)

f <- function(root,lambda) {
  x <- c(0,1);
  prob <- c(1/(lambda+1),lambda/(lambda+1));
  repeat {
  tosscounter <<- tosscounter + 1;
    if(root == 0) {
      break;
          }
    else {
      child1 <- sample(x,1,replace=TRUE,prob);
        child1 <- f(root=child1,lambda);
          if(child1[1] == 0) {
            child2 <- sample(x,1,replace=TRUE,prob);
          child2 <- f(root=child2,lambda)
          }
    }
    if(child1[1] == 0 && child2[1] == 0) {
    break;
    }
    if(child1[1] == 1 || child2[1] == 1) {
    # Calls the function recursively if either of the children nodes are 1.
    root <- sample(x,1,replace=TRUE,prob);
    }
  }
  output <- matrix(c(root,tosscounter),2,1);
  return(output);
}
```

```
# This function replicates the function f a specified number of times
# And returns the probability the root node is a 1, the standard deviation,
# And the average number of tosses required to determine the value of the root node

simulation <- function(lambda,trials=10000) {
  x <- c(0,1);
  prob <- c(1/(lambda+1),lambda/(lambda+1));
  PVector <- rep(0,trials);
  TVector <- rep(0,trials);
  for (i in 1:trials) {
    tosscounter <<- 0;
    run <- f(sample(x,1,replace=TRUE,prob),lambda);
    PVector[i] = run[1];
    TVector[i] = run[2];
  }
  results <- (PVector == 1) + 0;
  return(c(mean(results),sd(results)*trials^(-1/2),mean(TVector)));
}

# This function calls simulation for a specified set of lambdas

results <- function(lambdas) {
  output <- rep(0,length(lambdas));
  output <- sapply(lambdas, simulation);
  return(output);
}

# These next two functions are used to calculate the exact probability
# That the root node is 1 as a function of lambda

predict.fun <- function(lambda,x){((1-(lambda/(lambda+1))*x/(1-(lambda/(lambda+1))*(1-x)))^2 - x)}

acceptprob <- function(lambdas,l=10000) {
x<-seq(.05,.99,length=l);
predicted = rep(0,length(lambdas));
dist <- rep(9,10000)
for(i in 1:length(lambdas)) {
  for(j in 1:length(x)) {
    dist[j] <- abs(predict.fun(lambdas[i],x[j]))
  }
predicted[i] <- x[which.min(dist)]*(lambdas[i]/(lambdas[i]+1))/(1-(lambdas[i]/(lambdas[i]+1))*(1-x[w
}
return(predicted);
}

# This function numerically finds the critical value of lambda as a function
# Of the probability of acceptance(pred) and lambda

critval <- function(lambdas,pred) {
  dist <- 9;
  criticalval <- 0;
  for(i in 1:length(lambdas)) {
    tempdist <- (-1/(pred[i]^2-pred[i]-2) - lambdas[i]/(lambdas[i]+1));
    if (tempdist >= 0 && tempdist < dist) {
      dist <- tempdist;
      criticalval <- lambdas[i]
    }
  }
return(criticalval);
}
```

A.2 PRAR on Lattice Code

```
# This function f implements PRAR on the lattice, returning the value of the root node. (Algorithm 2

f <- function(root,lambda,row,column,k=4) {
  modified <- matrix(nrow = k, ncol = k);
  right <- 0;
  down <- 0;
  up <- 0;
  left <- 0;
  x <- c(0,1);
  prob <- c(1/(lambda+1),lambda/(lambda+1));
  repeat {
    tosscounter <<- tosscounter + 1;
    mat[which(modified==1)] <<- NA;
    mat[row,column] <<- root;
    if(root == 0) {
      break;
         }
    else {
        # If the root node is not 0, we draw it's children
      if(column<k && is.na(mat[row,column+1])==TRUE) {
        right <- sample(x,1,replace=TRUE,prob);
        modified[row,column+1] <- 1;
          right <- f(root=right,lambda,row,column+1,k)[1];
      }
        if(right == 0 && row<k && is.na(mat[row+1,column])==TRUE) {
          down <- sample(x,1,replace=TRUE,prob);
        modified[row+1,column] <- 1;
          down <- f(root=down,lambda,row+1,column,k)[1];
        }
        if(right == 0 && down ==0 && row>1 && is.na(mat[row-1,column])==TRUE) {
          up <- sample(x,1,replace=TRUE,prob);
        modified[row-1,column] <- 1;
        up <- f(root=up,lambda,row-1,column,k)[1];
        }
        if(right == 0 && down==0 && up==0 && column>1 && (is.na(mat[row,column-1])==TRUE)) {
          left <- sample(x,1,replace=TRUE,prob);
          modified[row,column-1] <- 1;
          left <- f(root=left,lambda,row,column-1,k)[1];
        }
    }
    # If all children are 0, we return the value of the root node
    if(right==0 && down==0 && up==0 && left==0) {
      break;
    }
    # If not, we redraw the root node and start over
    if(right == 1 || down == 1 || up == 1 || left == 1) {
      root <- sample(x,1,replace=TRUE,prob);
    }
  }
  mat[which(modified==1)] <<- NA;
  output <- matrix(c(root,tosscounter),2,1);
  return(output);
}

# simulation runs f a specified number of times, returning the probability
# The root node is a 1, the standard deviation, and the average number of
# Tosses used to get a draw

simulation <- function(lambda,trials=10000,k=4) {
  x <- c(0,1);
  prob <- c(1/(lambda+1),lambda/(lambda+1));
  PVector <- rep(0,trials);
  TVector <- rep(0,trials);
  for (i in 1:trials) {
    mat <<- matrix(nrow = k, ncol = k);
    tosscounter <<- 0;
    run <- f(sample(x,1,replace=TRUE,prob),lambda,1,1,k=dim(mat)[1]);
    PVector[i] = run[1];
    TVector[i] = run[2];
  }
  results1 <- (PVector == 1) + 0;
  print(lambda);
  return(c(mean(results1),sd(results1)*trials^(-1/2),mean(TVector)*k^2));
}

# results runs simulation for a vector or different lambdas

results <- function(lambdas,trials=10000,k=4) {
  output <- matrix(NA,nrow=3,ncol=length(lambdas));
  for(i in 1:length(lambdas)) {
    output[,i] = simulation(lambdas[i],trials,k);
  }
  return(output);
}

# check1 determines the number of independent sets with a given number of
# 1's for all possible 4x4 lattices

check1 <- function(k=4) {
  list1 <- list(0:1);
  tmp <- expand.grid(rep(list1,k^2));
  count <- rep(0,k^2);
  Vhardcore <- rep(0,k^2);
  for (i in 1:dim(tmp)[1]) {
    hardcore <- 1;
    dummymat <- matrix(tmp[i,],k,k);
    dummymat2 <- t(dummymat);
    for(j in 1:(k^2-k)) {
      if(dummymat[[j]]==1 && (dummymat[[j]]==dummymat[[j+k]])){
        hardcore <-0;
      }
      if(dummymat2[[j]]==1 && (dummymat2[[j]]==dummymat2[[j+k]])) {
        hardcore <-0
      }
    }
   if(hardcore == 1) {
     count[sum(dummymat==1)] <- count[sum(dummymat==1)] + 1;
     if(dummymat[[1]] == 1) {
       Vhardcore[sum(dummymat==1)] <- Vhardcore[sum(dummymat==1)] + 1;
     }
   }
  }
output <- matrix(data=NA, 2,k^2);
output[1,] = count;
output[2,] = Vhardcore;
return(output);
}

# This function gives the exact probability the root node is a 1 for each value
# of lambda

pred.fun <- function(lambdas,k=4) {
  x <- check1(k);
  x1 <- x[1,];
  x2 <- x[2,];
  output <- rep(0,length(lambdas));
  z <- 0 + 1;
  # The one accounts for the 0 matrix which isn't counted in check1()
  pr <- 0;
  for(j in 1:length(lambdas)) {
    z <- 0 + 1;
    pr <- 0;
    lambda <- lambdas[j];
    for(i in 1:length(x1)) {
      z <- z+x1[i]*lambda^(i);
      pr <- pr+x2[i]*lambda^(i);
    }
  output[j] = pr/z;
  }
return(output);
}
```

A.3 CFTP on Lattice Code

```
# step takes a single step in the bounding chain for the Dyer-Greenhill Chain
# (Algorithm 4)

step <- function(X, lambda, U, d1, d2, pswap) {
  if(U>(lambda/(lambda+1))) {X[d1,d2] <- 0}
  else {
    if(d1>1) {up <- X[d1-1,d2]}
    else {up <- 0}
    if(d2>1) {left <- X[d1,d2-1]}
    else {left <- 0}
    if(d1<dim(X)[1]) {down <- X[d1+1,d2]}
    else {down <- 0}
    if(d2<dim(X)[2]) {right <- X[d1,d2+1]}
    else {right <- 0}

    if(sum(up,left,down,right, na.rm = TRUE) > 1) {X[d1,d2] <- 0}
## Case 3
    if(sum(is.na(c(up,left,down,right))) == 0 && (up == 0 && left == 0 && right == 0 && down == 0))
##Case 1
    if(sum(is.na(c(up,left,down,right))) == 0 && (sum(up,left,down,right, na.rm = TRUE) == 1)) {
```

```
##Case 2
      if(U>(pswap*lambda/(lambda+1))) {X[d1,d2] <- 0}
      else {
      if(up == 1) {
          X[d1,d2] <- 1;
          X[d1-1,d2] <- 0;
      }
      else if(down == 1) {
          X[d1,d2] <- 1;
          X[d1+1,d2] <- 0;
      }
      else if(left == 1) {
          X[d1,d2] <- 1;
          X[d1,d2-1] <- 0;
      }
      else if(right == 1) {
          X[d1,d2] <- 1;
          X[d1,d2+1] <- 0;
      }
      }
    }
    if((sum(is.na(c(up,left,down,right))) > 1) && (sum(up,left,down,right, na.rm=TRUE) == 0)) {X[d1,
##Case 6
    if((sum(is.na(c(up,left,down,right))) == 1) && (sum(up,left,down,right, na.rm=TRUE) == 0)) {
##Case 4
      if(U>(pswap*lambda/(lambda+1))) {X[d1,d2] <- NA}
      else {
        if(is.na(up) == 1) {
          X[d1,d2] <- 1;
          X[d1-1,d2] <- 0;
        }
        else if(is.na(down) == 1) {
          X[d1,d2] <- 1;
          X[d1+1,d2] <- 0;
        }
        else if(is.na(left) == 1) {
          X[d1,d2] <- 1;
          X[d1,d2-1] <- 0;
        }
        else if(is.na(right) == 1) {
          X[d1,d2] <- 1;
          X[d1,d2+1] <- 0;
        }
      }
    }
     if((sum(is.na(c(up,left,down,right))) >= 1) && sum(up,left,down,right, na.rm=TRUE) == 1) { ##Ca
      if(U>(pswap*lambda/(lambda+1))) {X[d1,d2] <- NA}
      else {
      if((is.na(up)== FALSE) && (up == 1)) {
          X[d1,d2] <- NA;
          X[d1-1,d2] <- NA;
      }
      else if((is.na(down)== FALSE) && (down == 1)) {
          X[d1,d2] <- NA;
          X[d1+1,d2] <- NA;
      }
      else if((is.na(left)== FALSE) && (left == 1)) {
          X[d1,d2] <- NA;
          X[d1,d2-1] <- NA;
      }
      else if((is.na(right)== FALSE) && (right == 1)) {
          X[d1,d2] <- NA;
          X[d1,d2+1] <- NA;
      }
      }
     }
   }
return(X);
}

# This is the CTTP Algorithm for the above bounding chain algorithm. (Algorithm 5)

CFTP <- function(t, k, lambda, pswap) {
  UVector <- runif(t);
  d1Vector <- sample(1:k, t, replace = TRUE, rep(1/k, k));
  d2Vector <- sample(1:k, t, replace = TRUE, rep(1/k, k));
  tosscounter <<- tosscounter + 3*t;
  X <- matrix(NA,nrow=k,ncol=k);
  for(i in 1:t) {
    X <- step(X, lambda, UVector[i], d1Vector[i], d2Vector[i], pswap);
  }
  if(length(which(is.na(X))) == 0) {
    return(X);
  }
  else {
  X0 <- CFTP(2*t, k, lambda, pswap);
  X <- X0;
  for(j in 1:t) {
    X <- step(X, lambda, UVector[j], d1Vector[j], d2Vector[j], pswap);
  }
  return(X);
  }
}

# simulation runs the above algorithm a specified number of times, returning
# the probability the root node is a 1, the standard deviation, and the
# number of uniforms required to get a draw

simulation <- function(lambda,pswap,t,k,trials=10000) {
  PVector <- rep(0,trials);
  TVector <- rep(0,trials);
  for(i in 1:trials) {
    tosscounter <<- 0;
    run <- CFTP(t,k,lambda,pswap);
    PVector[i] <- run[1,1];
    TVector[i] <- tosscounter;
  }
  return(c(mean(PVector),sd(PVector*trials^(-1/2)),mean(TVector)));
}

# results repeats simulation for a vector or lambdas

results <- function(lambdas,pswap,t,k,trials=10000) {
  output <- matrix(NA,nrow=3,ncol=length(lambdas));
  for(i in 1:length(lambdas)) {
    output[,i] <- simulation(lambdas[i],pswap,t,k,trials)
  print(lambdas[i]);
  }
  return(output);
}
```