NTNU
Department of Mathematical Sciences

Master thesis - TMA4900

A multiple-try Metropolis-Hastings algorithm with a tailored number of proposals

Hilde Steinbru Heggstad

2019

Preface

This report is the result of the Master thesis related to the course TMA4900, which is the last course of the master's degree in Industrial Mathematics at the Norwegian University of Science and Technology. The master thesis is meant as an opportunity for the student to investigate a specific topic and do independent research, discuss theory and develop methods related to this topic. It accounts for one semester's work and is written with supervision by a professor at the Department of Mathematical Sciences at NTNU. This thesis investigates the topic of a specific multiple-try Metropolis–Hasting algorithm, which goal is to tailor its number of proposals. I would like to thank professor Håkon Tjelmeland for guidance throughout the semester.

Hilde Steinbru Heggstad
Trondheim, January 2019

Abstract

A multiple-try Metropolis–Hastings(M–H) algorithm is a special case of the M–H algorithm which, instead of proposing one proposal, proposes multiple values in every iteration of the algorithm. The motivation for generating multiple proposals is to make the sampled values from the M–H simulation less correlated. Luo and Tjelmeland (2018) present a multiple-try M–H algorithm which uses an undirected acyclic graph to generate multiple proposals, by first sampling one of the nodes in the graph as a root in a new directed version of the graph. The root node has a value associated with it, which is assumed to be distributed according to a target distribution. The multiple proposals are proposed by using a proposal distribution $q(x_n|x_{pred(n)})$, and propose values in direction of the directed edges, by first conditioning on the value associated with the root node. When all nodes in the directed graph are associated to a value generated through $q(x_n|x_{pred(n)})$, a new node is sampled as the root node, and the value associated with it is accepted as a sample from the target distribution. The new proposals are then generated in a new iteration of the algorithm, with the order determined by the structure of the graph with the sampled node as root.

We present a new version of the algorithm from Luo and Tjelmeland (2018), which instead of using a fixed undirected graph to generate multiple proposals, grows a directed acyclic graph (DAG) in every iteration of the algorithm. The idea is to tailor the number of proposals by making the DAG grow in directions of high density values. Every node in the DAG is associated with a value, and these values are generated through a proposal distribution $q(x_n|x_{pred(n)})$, like in Luo and Tjelmeland (2018). The difference is that when we grow the DAG stochastically, the edges and nodes of the DAG are generated as the proposals are made, making it possible for the DAG to stop growing in directions of bad proposals and expand in the directions of high density values. The derivation of the algorithm is based on the derivations from Luo and Tjelmeland (2018), and are similar in all ways except for that we include the growth of the DAG in the joint distribution used as the target distribution. We demonstrate that the new algorithm does not converge to the specific target distribution of interest, but has a variance dependent on the parameters used in the M–H setup. We try to locate the error through simulating the algorithm for different parameters, and conclude that there must be something wrong in how the probability for a specific DAG being generated is calculated. We also investigate whether the algorithm succeeds in proposing high density values compared to the algorithm from Luo and Tjelmeland, and also compare different strategies for growing the DAG based on their ability to generate high density values and jump between levels when sampling a new node as root.

1 Introduction

Markov chain Monte Carlo (MCMC) algorithms are widely used to generate samples from distributions which are difficult or impossible to sample from directly. The algorithms are powerful, in the sense that it is not necessary to know the normalization constant of the target distribution to be able to sample from it. The target distribution only needs to be evaluated up to a normalization constant. This makes the algorithm ideal for Bayesian inference, which because of the conditional distributions often have normalization constants that are difficult to calculate.

Popular MCMC methods include Metropolis–Hastings (M–H) algorithm (Metropolis et al. 1953; Hastings 1970) and Gibb's sampling (Geman and Geman 1984), where Gibb's sampling is a special case of the Metropolis–Hastings algorithm. The algorithm is built upon the theory of Markov chains and limiting distributions of reversible Markov chains. For proposing a new state for the Markov chain, a proposal distribution is used. To sample from a target distribution $p(x)$, by using a proposal distribution $q(x'|x)$ the probability of accepting a proposed value x' as the new state, given the chain's current state x is

$$\alpha(x'|x) = \min\left\{1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right\}. \tag{1}$$

In the special case of Gibb's sampling, the full conditional distribution for x is used as the proposal distribution $q(x'|x)$, and the M–H acceptance probability reduces to one, because the terms in the M–H acceptance probability will cancel. Liu et al. (2000) present a Multiple-Try Metropolis algorithm, which is a generalization of the M–H algorithm which instead of proposing one value, proposes multiple values as potential new states in every iteration of the algorithm. The motivation of this generalization is to improve the mixing properties of the algorithm.

Luo and Tjelmeland (2018) present a multiple proposal algorithm, which expands the target distribution to sample a directed acyclic graph (DAG) which is based on a sample x from the original target distribution $p(x)$. The joint distribution for the DAG includes both a sample x from the original target distribution $p(x)$, and the joint distribution for proposing all values associated with the nodes in the DAG. By using the full conditional distributions as proposal distributions in the M–H setup, the proposals are defined and accepted with a probability of one. The multiple proposals are proposed by sampling one of the nodes in the undirected acyclic graph as a root node in a directed version of the undirected graph, which has all edges pointing away from the sampled root node. The root node has the value corresponding to the sample from $p(x)$ associated with it, and the multiple proposals are generated through a proposal distribution $q(x_n|x_{pred(n)})$, in direction of the directed edges, by first conditioning on the value associated with the root node. Here the value x_n is associated to a node n, and is proposed conditioned on the value $x_{pred(n)}$, which is the value associated with the predecessor of n in the DAG. When all nodes are associated to a proposed value, one of the nodes in the DAG is sampled as a new root, and the node's associated value is accepted as a sample from $p(x)$. This corresponds to an iteration of the algorithm. In the next iteration, the values are proposed in the order of the directed edges, corresponding to the new node being the root of the DAG.

In every generation of a new DAG, each of the nodes are potential new values to be accepted as a sample from the original target distribution $p(x)$. In Luo et Tjelmeland (2018), the structure of the undirected acyclic graph is fixed and defined in the M–H setup, and what is stochastic is the order the proposed values associated with the nodes, and this order is defined by the sampled root node which determines the structure of the DAG. The values associated with each node is sampled through a tailored proposal distribution, which is a proposal distribution constructed to resemble the original target distribution $p(x)$, and is therefore likely to generate better proposals than for example a Gaussian proposal distribution.

In the M–H setup in Luo et Tjelmeland (2018), the set of values proposed for each DAG is dependent on the fixed structure of the undirected acyclic graph, defined in the M–H setup. This means that if one of the nodes n located at a low level in a DAG, has a low density value $p(x)$ associated with it, the structure of the DAG makes the algorithm likely to propose several low density values, as the nodes at higher levels will condition on the low density point x. In other words, the algorithm does not have the ability to separate between proposing values based on low- or high density points. This motivates for a new version of the algorithm, which makes low density points less likely to be proposed within the DAG.

In this article we present a new version of the algorithm from Luo and Tjelmeland (2018), where the structure of the DAG is stochastic and not given in the M–H setup. The idea is to let the DAG grow dependent on the values associated with its nodes, so that the DAG can expand in the nodes which have high density values associated with them, and avoid expanding in the nodes which are located in the tail of the target distribution. This is done by assigning a probability of a node n having *children*. If a node n gets children, it means that there will be generated nodes in the DAG with associated values conditioning on the value associated with n. The values are generated by a proposal distribution $q(x_n|x_{pred(n)})$, in a similar manner as in Luo and Tjelmeland (2018). The acceptance probabilities are a natural extension of the acceptance probabilities from Luo and Tjelmeland (2018), which uses the full conditional distributions as proposal distributions, and updates the Markov chain by Gibb's sampling. The target distribution and the full conditional distributions differ in the sense that in our new algorithm, the stochastic process of growing the DAG is included. We demonstrate that this intuitive expansion does not converge, meaning that the expressions for the full conditional distributions must be wrong. In our examples, the limiting distributions' variance is dependent on the parameters used in the proposal distribution in the M–H setup.

In the algorithm presented in this work, a Gaussian distribution is used as the proposal distribution $q(x_n|x_{pred(n)})$ instead of the tailored distribution used in Luo and Tjelmeland (2018). When the structure of the DAG is stochastic and dependent of the values associated with the nodes, the proposals generated by the Gaussian distribution will not correspond to a random walk, in the same way it would if a Gaussian distribution was used in Luo and Tjelmeland (2018). This is because in our setup, the DAG should ideally stop expanding in nodes associated to low density values, whilst in the setup in Luo and Tjelmeland (2018) the expansion happens independently of the densities associated with the nodes, leading to proposals corresponding to a random walk. Still, a tailored proposal is also suitable in the M–H presented here, and is likely to increase the probability of generating high density values, but this is not a focus in this work.

Although our algorithm does not converge to the specific target distribution, we expect that a small adjustment to the acceptance probabilities could make the algorithm correct. This is because the algorithm converges to distributions close to the target distribution, making it likely that the acceptance probabilities are close to correct. In Section 7 we demonstrate that the algorithm in general does not converge to the correct target distribution, and try to locate the error by running different simulations with different parameters in the proposal distribution. We also investigate the algorithm's ability to generate high density values, and its ability to jump between levels when sampling a new root. In the closing remarks of this report we present a hypothesis about where in the M–H setup the error is located, based on the simulation experiments from Section 5, and suggest how one could approach the problem of making the acceptance probabilities correct.

2 The Metropolis–Hastings algorithm

The Metropolis–Hastings (M–H) algorithm is used to draw samples from distributions which are difficult to sample from directly. The method is widely used in modern statistics, and is a powerful in the sense that it theoretically can be used to sample from any target distribution $p(x)$, which can the be evaluated up to a proportionality constant. That is, one does not need to know the normalization constant to be able to use the M–H algorithm. These situations typically arise in Bayesian settings, where the normalization constants are difficult or impossible to calculate.

The algorithm is built upon the theory of Markov chains, and is called a Markov chain Monte Carlo method. In principle, the method defines a Markov chain whose limiting distribution is the target distribution $p(x)$. When the Markov chain has converged to its limiting distribution, each state of the Markov chain can be regarded as a sample from the target distribution $p(x)$.

2.1 Constructing a Markov chain with a given limiting distribution

A Markov chain is defined as a stochastic process where each step or decision only depends on the chain's previous step. This is called the Markov property, and any stochastic process which fulfills this property is thereby a Markov chain. By using theory based on Markov chains, it is possible to define a chain which has the target distribution as its limiting distribution.

We want to draw samples from a target distribution $p(x)$ with sample space S by using the M–H algorithm. This is done by using a proposal distribution

$$q(x'|x), \tag{2}$$

together with the M–H acceptance probability

$$\alpha(x'|x) = \min\left\{1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right\} \tag{3}$$

Here (3) is the acceptance probability for accepting the proposed state x' from the current state x, when x' is proposed by (2). By using the acceptance probability in (3), a Markov chain with a limiting distribution equal to $p(x)$ is defined.

If we choose to use a symmetric proposal distribution, meaning that

$$q(x'|x) = q(x|x'), \tag{4}$$

the M-H acceptance probability in (3) reduces to

$$\alpha(x'|x) = \min\left\{1, \frac{p(x')}{p(x)}\right\}, \tag{5}$$

which is now on the form of a Metropolis algorithm, which is a Metropolis–Hastings algorithm that uses a symmetric proposal distribution.

The step by step procedure for using the M–H algorithm, is first to assume that we have a value x distributed according to the target distribution $p(x)$. From here, a value x' is proposed by the proposal distribution in (2). The value x' is either accepted with the probability in (3), or rejected. If x' is accepted, then the Markov chain's state is updated and the current state is x'. If the value is rejected, the chain's state will remain at x. This process is repeated until the chain has converged. When the chain has converged, each state of the Markov chain can be regarded as a sample from the target distribution. It is possible to use the M–H algorithm to draw samples from any target distribution, which is known up to a normalization constant, but in many cases the chain will take too long to converge to the target distribution, making the algorithm impracticable to use.

It is the form of the acceptance probability in (3) that ensures the chain to converge, the proposal distribution does not effect the limiting distribution of the Markov chain. The chain will converge for any choice of proposal distribution $q(x'|x)$, but the proposal distribution has a large impact on how fast the Markov chain converges. For example, a Markov chain with a proposal distribution which resembles the original target distribution is expected to converge faster than a Markov chain with an arbitrary proposal distribution. This is because a proposal distribution which is similar to the target distribution is more likely to propose states that will be accepted by the acceptance probability in (3).

When the shape of the target distribution is relatively unknown, a common choice for the proposal distribution is a Gaussian distribution with mean equal to the Markov chain's current state. The variance of the Gaussian distribution is often referred to as the tuning parameter, meaning that it tunes how the Markov chain jumps. For example, if the tuning parameter is large, the proposal distribution $q(x'|x)$ is likely to propose values that are far away from the chain's current state x in the sample space S. If on the other hand, the tuning parameter is low, the proposal distribution is likely to propose values close to the chain's current state. The tuning parameter effects the ratio between accepted and rejected proposals, and how fast the chain converges. Typically, a too low tuning parameter results in a high acceptance ratio, but slow convergence, whilst a larger tuning parameter has lower acceptance ratio, but may make chain converge faster. This is because a Markov chain which takes ”large steps” is more likely to explore larger parts of the state space in a smaller amount of iterations. Nevertheless, if the tuning parameter gets too large, many of the proposals will end up in the tail of the target distribution, and this will also lead to slow convergence, since almost all proposals gets rejected.

2.2 Gibb's sampling

Recall from the previous section that a Gaussian distribution is a common choice for the proposal distribution when little is known about the target distribution. A special case is when we know the full conditional distributions for the values we wish to sample from a joint target distribution. Then it is possible to use Gibb's sampling (Geman and Geman 1984) for updating the Markov chain. This involves using the full conditional distributions as proposal distributions, and the M–H acceptance probabilities for updating the Markov chain reduces to one. Intuitively this means that the proposal distribution resembles the target distribution up to a normalization constant, which makes the terms in the M–H ratio from (3) cancel, and we are left with an acceptance probability equal to one.

Imagine that we want to sample x from a joint distribution $p(x, y)$, so that this is our target distribution. The full conditional distribution for x is defined as the distribution for x when y is given, and is denoted by $p(x|y)$. The relationship between the joint distribution and the full conditional distribution for x is

$$p(x, y) = p(x|y) \cdot f(y). \tag{6}$$

We wish to update x while keeping y fixed. Inserting for the target distribution $p(x, y)$ and using the full conditional distribution for x as the proposal distribution $p(x'|y)$, the M–H acceptance probability becomes

$$\alpha(x'|x) = \min\left\{1, \frac{p(x', y)p(x|y)}{p(x, y)p(x'|y)}\right\} = \min\left\{1, \frac{p(x'|y) \cdot f(y)p(x|y)}{p(x|y) \cdot f(y)p(x'|y)}\right\} = 1. \tag{7}$$

We then update the chain's state for y, keeping the new state x' fixed, and using the full conditional distribution $p(y'|x')$ in a similar manner as was done to propose the new value x'. The Markov chain is updated in turn for x and y in every iteration. Using the full conditional distribution as the proposal distribution is efficient, since none of the proposed values are rejected. Gibb's sampling is therefore preferred when the full conditional distribution is known, and we are able to draw samples from it.

2.3 Requirements for convergence

A Markov chain is uniquely defined by its transition probabilities, so that if two Markov chains have the same transition probabilities, they are identical. This means that if the transition probabilities described above fulfills the requirements for convergence, it can be used to draw samples from the desired target distribution.

The constructed Markov chain needs to fulfill two requirements for the M–H algorithm to converge to the target distribution. The first requirement is that there must exist a stationary distribution for the Markov chain, and the second requirement is that this stationary distribution is unique. A sufficient condition for a stationary distribution to exist is that the chain is reversible, meaning that the probability for being in a state x and moving to a new state x' is equal to the probability of being in the state x' and moving to the state x. That is, for every pair of states x and x'

$$Pr(x|x')Pr(x') = Pr(x'|x)Pr(x). \tag{8}$$

For the stationary distribution to be unique it is required that each state of the Markov chain is ergodic, which means that it is aperiodic and positive recurrent (Gamerman and Lopes, 2006).

The Markov chain has converged when it has reached its unique stationary distribution, which in this case is equal to the target distribution. Given an initial state x, the chain will converge after a certain amount of iterations. The number of iterations before the chain converges is referred to as the burn-in period. When the chain has converged, the probability of the Markov chain being in a state x equals the density of the target distribution $p(x)$.

To show that a Markov chain with transition probabilities as discussed above converges, it is necessary to show that it is reversible, as expressed in (8). When $x' = x$, the expression in (8) is obviously correct. If $x \neq x'$, the process of moving to a new state involves two steps. The first step is the probability for the new state being proposed, which is given by the density of the proposal distribution in (2). The second step is for the proposed value to be accepted according to the acceptance probability in (3). The probability for moving to a new state x' given the chain's current state x becomes

$$Pr(x'|x) = q(x'|x)\alpha(x'|x), \tag{9}$$

which inserted into the right hand side of the equation for the chain to be reversible from (8) becomes

$$\begin{aligned} q(x'|x)\alpha(x'|x)p(x) &= q(x'|x)\min\left\{1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right\}p(x) \qquad (10)\\ &= \min\{p(x)q(x'|x), p(x')q(x|x')\}, \qquad (11)\end{aligned}$$

and the left hand side becomes

$$\begin{aligned} q(x|x')\alpha(x|x')p(x') &= q(x|x')\min\left\{1, \frac{p(x)q(x'|x)}{p(x')q(x|x')}\right\}p(x') \qquad (12)\\ &= \min\{p(x')q(x|x'), p(x)q(x'|x)\}. \qquad (13)\end{aligned}$$

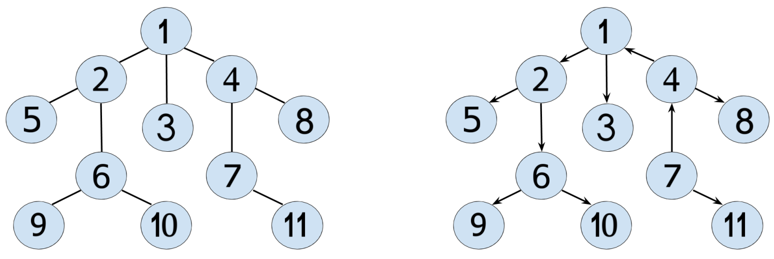
The values of (11) and (13) are equal for every pair x and x', which proves that the Markov chain is reversible, and will have its limiting distribution equal to the desired target distribution $p(x)$.

3 A Multiple-try Metropolis algorithm

Luo and Tjelmeland (2018) present a Multiple-try Metropolis algorithm which proposes several new state values in every iteration of the algorithm, by using an undirected acyclic graph $\mathcal{G}$. In every iteration of the algorithm, one of the nodes in $\mathcal{G}$ is sampled as a root node, leading to a directed version of the graph, with edges pointing away from the sampled root node. Every node in the graph is associated to a value, and the value associated with the root node is assumed to be distributed according to a target distribution $p(x)$ of interest. When a node is sampled as a root, its associated value is accepted as a sample from the target distribution. In the next iteration, new proposals to be associated with the nodes are generated in direction of the directed edges of the graph through a proposal distribution, by first conditioning on the value associated with the root.

3.1 An undirected acyclic graph to generate multiple proposals

In the M–H setup in Luo and Tjelmeland (2018), the undirected acyclic graph $\mathcal{G}$ consists of a set of undirected edges $\mathcal{E}$, and a set of nodes $\mathcal{S}$ is used to generate multiple proposals. The graph



(a) The undirected acyclic graph $\mathcal{G}$ (b) The directed acyclic graph $\mathcal{G}_7$

Figure 1: An example of a undirected graph $\mathcal{G}$ to be used in the M–H setup, and the graph $\mathcal{G}_7$ formed by sampling the node indexed by 7 as root. The nodes are indexed by a number to explain the order in which the values are proposed in this example.

generates the new potential state values from the target distribution $p(x)$, by sampling one of the nodes n in $\mathcal{S}$ as a root r, which results in a directed version of the graph $\mathcal{G}$, that has directed edges pointing away from the root node r. An example of an undirected graph $\mathcal{G}$ and a directed acyclic graph (DAG) resulting from sampling a node in $\mathcal{S}$ is shown in Figure 1. Since the method generates multiple proposals in each iteration, it is called a multiple-try Metropolis algorithm.

The DAG resulting from sampling a node r from $\mathcal{S}$ is denoted $\mathcal{G}_r$, and includes the same set of nodes as $\mathcal{G}$, but a different set of directed edges, denoted $\mathcal{E}_r$. Let each node n in $\mathcal{S}_r$ have a value x_n associated with it. In the directed graph $\mathcal{G}_r$, the values are proposed in direction of the directed edges. For each node n in $\mathcal{S}$, except for the root node r, denote the predecessor of n by $pre(n)$. In the graph $\mathcal{G}_r$, there will be a directed edge for every node $n \neq r$, directed from n's predecessor $pre(n)$ to the node n. The value associated with n is proposed by a proposal distribution

$$q(x_n|x_{pre(n)}), \tag{14}$$

where $x_{pre(n)}$ is the value associated with the predecessor $pre(n)$. For the DAG in Figure 1, denote the nodes n in $\mathcal{G}$ by $1, 2, 3, .., 11$, and their associated values by respectively $x_1, x_2, x_3, .., x_{11}$. When node 7 is the root, the values will be proposed in the order

$$q(x_4|x_7)q(x_{11}|x_7)q(x_8|x_4)q(x_1|x_4)q(x_2|x_1)q(x_3|x_1)q(x_5|x_2)q(x_6|x_2)q(x_9|x_6)q(x_{10}|x_6).$$

Let the set of values associated with each of the nodes in the DAG $\mathcal{G}_r$ be denoted by $\mathcal{X}_r$. Previous to the values in $\mathcal{X}_r$ being generated, each node n in the undirected graph has an equal probability of being the root node of a potential DAG. That is, the probability for a node n in the graph being the root is uniformly distributed.

The idea is to let each of the nodes n in $\mathcal{S}$ be potential root nodes in a directed acyclic graph, corresponding to the graph $\mathcal{G}$ with edges directed away from the root node n of the graph. The structure of $\mathcal{G}$ is deterministic and given in the M–H setup, so its structure will not effect the probability for a node n being the root of the graph previous to a set $\mathcal{X}_r$ being proposed.

3.2 An expansion of the target distribution

Assume that we have a value x sampled from the target distribution $p(x)$, and that x is the value associated with the root node r of a DAG on the form as described above. The values of the other nodes in the DAG are proposed through the proposal distribution in (14), with the proposed values conditioned on the values associated with the predecessor of the different nodes. The distribution of the values $\mathcal{X}_r$, which is the set of values associated with every node in the DAG including the root node, is conditionally dependent on the root node r, and the joint distribution for the set $\mathcal{X}_r$ and the node r being the root equals

$$f(r, \mathcal{X}_r) = f(\mathcal{X}_r|r) \cdot f(r), \tag{15}$$

where

$$f(\mathcal{X}_r|r) = p(x) \cdot \prod_{n\in\mathcal{S}\setminus\{r\}} (x_n|x_{pre(n)}), \tag{16}$$

is the joint probability of all values in $\mathcal{X}_r$. Note that the value x associated with r is independent of r, but the rest of the set $\mathcal{X}_r$ is dependent on the value x and the node's position in the DAG, as this determines the order of the proposals generated through (14). Using that the probability of a node r being the root previous to the set $\mathcal{X}_r$ being proposed is uniformly distributed between the nodes in $\mathcal{S}$, the joint probability of $\mathcal{X}_r$ and $f(r)$ becomes

$$f(\mathcal{X}_r, r) = p(x) \cdot \prod_{n\in\mathcal{S}\setminus\{r\}} (x_n|x_{pred(n)}) \cdot \frac{1}{|\mathcal{S}|} \propto p(x) \cdot \prod_{n\in\mathcal{S}_r\setminus\{r\}} (x_n|x_{pred(n)}). \tag{17}$$

3.3 Sampling from the expanded target distribution

Luo and Tjelmeland (2018) uses the distribution in (17) as the target distribution in the M–H setup. In each iteration of the algorithm, a DAG is proposed as a new state for the Markov chain, and the proposed value x in the first term of (17) is the value which is accepted as a value from the original target distribution $p(x)$. Updating the Markov chain consists of two steps. Consider first that we have a value x which is distributed according to the target distribution, as a starting point for the simulation. This value is associated with a root node r in a DAG resulting from r's position in the undirected graph in the M–H setup. The first step is to propose values for all nodes other than the root node, conditioned on the value x associated with r, using the proposal distribution in (14). This is a Gibb's update for the set $\mathcal{X}_r$ with the full conditional distribution

$$f(\mathcal{X}_r|r, x) = \prod_{n\in\mathcal{S}\setminus\{r\}} (x_n|x_{pred(n)}), \tag{18}$$

given the root node r with the associated value x. Note that even though x is included in the set $\mathcal{X}_r$, $p(x)$ is not a part of the full conditional in (18), since x is given and conditioned on.

The second step is to sample a new root by assuming that given the structure of the undirected graph $\mathcal{G}$, each of the nodes have a probability of being the root. For sampling a new root, we define the sample space as all possible nodes in $\mathcal{S}$ being the root in the DAG. That is, the sample space is as large as the number of nodes in $\mathcal{S}$. Given a DAG with root node r and the values in $\mathcal{X}_r$ associated to every node in $\mathcal{S}$, the full conditional distribution for a node r' being the root equals

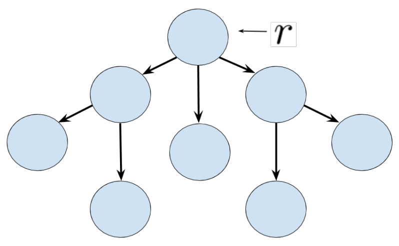
$$f(r'|\mathcal{X}_r) = \frac{p(x_{r'}) \cdot \prod_{n\in\mathcal{S}\setminus\{r'\}} q(x_n|x_{pred(n)})}{\sum_{n^*\in\mathcal{S}} p(x_{n^*}) \prod_{n\in\mathcal{S}\setminus\{n^*\}} q(x_n|x_{pred(n)})}. \tag{19}$$

The expression in (19) can be explained as the density of the joint distribution of the DAG with root node r', divided by the sum of the densities of the joint distributions of all possible choices of root nodes in $\mathcal{S}$. A new root node is chosen by assigning the probability in (19) to each node in $\mathcal{G}$, and applying the standard algorithm for sampling from a discrete distribution, see for example Gamerman and Lopes (2006). This is one of two Gibb's updates for updating the Markov chain, where the full conditional distribution in (19) is used as the proposal distribution.

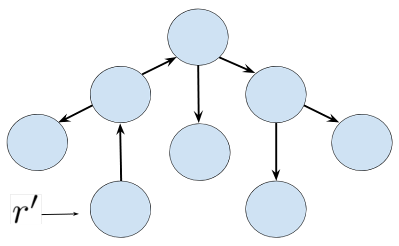
After a new root r' is sampled, new potential values are generated in the next iteration of the algorithm. This is done by using the structure of the DAG resulting from the sampled node r''s position in the undirected acyclic graph given in the M–H setup, and using the proposal distribution in (14), conditioning on the value $x_{r'}$ in the same manner as in the previous iteration of the algorithm. Since the full conditional distributions are used as proposal distributions for $\mathcal{X}_r$ and r', the M–H acceptance probabilities for accepting the proposed values are equal to one.

4 Growing the DAG stochastically

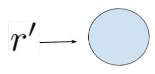
In this section a new version of the algorithm discussed above is presented. In Luo and Tjelmeland (2018) a undirected graph $\mathcal{G}$ is defined in the M–H setup, and its structure determines the order



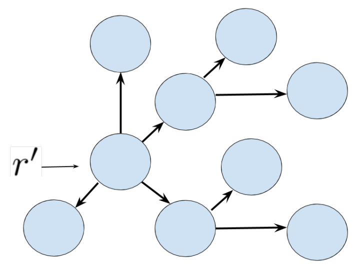
(a) The full grown DAG $\mathcal{G}_r$, with r indicating the root node.



(b) The new DAG $\mathcal{G}_{r'}$ resulting from sampling r' as the new root. The value associated with r' is accepted as a sample from $p(x)$.



(c) The node r' with its associated value is the starting point for growing a new DAG in the next iteration



(d) The full grown DAG $\mathcal{G}'_{r'}$. Here, the next step would be to sample one of the nodes in $\mathcal{G}'_{r'}$, as in (b).

Figure 2: The step by step procedure of growing a DAG and sampling a new root node. This procedure is repeated in every iteration of the algorithm. The value associated with the sampled node r' is accepted as a sample from the original target distribution $p(x)$.

in which the multiple values are proposed when a root node is sampled. We present a version of the algorithm that instead of using the structure of a fixed undirected graph, grows a DAG stochastically conditioned on the densities of the values proposed as the DAG is growing. This is motivated by the favourable effect of increasing the number of high density values, by letting the DAG grow stochastically in directions of high density points.

In each iteration of the algorithm, a DAG is grown conditioned on a root node r with a associated value x, which is assumed to be distributed according to the target distribution $p(x)$. The DAG is denoted by $\mathcal{G}_r$, and has a set of nodes $\mathcal{S}_r$ and a set of directed edges, $\mathcal{E}_r$. The DAG grows stochastically by assigning a probability for every node n in $\mathcal{S}_r$ having a number of children, leading the DAG to expand in n. When a node n samples a number of children, the size of the set $\mathcal{S}_r$ increases accordingly, and the values associated with the children conditions on the value associated with n. The set of values associated with the nodes in $\mathcal{S}_r$ is denoted $\mathcal{X}_r$, as in the setup which uses a fixed graph as presented above. When the DAG is fully grown, a node r' in $\mathcal{S}_r$ is sampled, and its associated value is accepted as a sample from original target distribution $p(x)$, in the same manner as in Luo and Tjelmeland (2018). In the next iteration, a new DAG is grown conditioned on the value associated with the node r' i a similar manner as the DAG $\mathcal{G}_r$ was grown. The procedure of growing a DAG and sampling a new root node is illustrated in Figure 2.

4.1 A target distribution which includes the growth of a DAG

By growing the DAG stochastically depending on the densities of the proposed values, a new target distribution is defined. The new distribution will include the density of the target distribution

$p(x)$, the probability for the specific DAG being grown, and the joint distribution of the set $\mathcal{X}_r$. The multiple proposals will be proposed in the direction of the directed edges of the DAG, in the same way as in Luo and Tjelmeland (2018). The algorithm can be used to generate samples from the original target distribution $p(x)$, by disregarding the parts of the expanded target distribution which are not a part of the factor $p(x)$ when sampling a whole DAG with associated values.

Assume x to be distributed according to the target distribution $p(x)$. Let r be the root node of a DAG, associated with the value x, which will be the starting point for growing the DAG. To grow a full DAG, let the number of children, $c(r)$, for the root node r be stochastic and distributed according to some discrete distribution

$$P(c(r)). \tag{20}$$

We let the number of children, $c(n)$ for all other nodes $n \neq r$ in $\mathcal{S}_r$ be distributed according to another discrete distribution

$$P(c(n)|\mathcal{G}_r^*, \mathcal{X}_r^*), \tag{21}$$

where $\mathcal{G}_r^*$ denotes the part of the graph $\mathcal{G}_r$ grown so far, since the DAG might still be growing when n samples its number of children. The set $\mathcal{X}_r^*$ denotes the set of proposed values associated with the nodes in $\mathcal{G}_r^*$.

Each node n in $\mathcal{S}_r$ is associated with a value x_n. For all nodes n in S_r, except for the root node r, the value x_n is dependent on the value $x_{pred(n)}$, in the same way as in Luo and Tjelmeland (2018). After the graph is fully grown, the joint distribution for the DAG $\mathcal{G}_r$ and the set $\mathcal{X}_r$ becomes

$$f(\mathcal{G}_r, \mathcal{X}_r) = p(x)P(c(r)) \prod_{n \in \mathcal{S}_r \setminus \{r\}} P(c(r)|\mathcal{G}_r^*, \mathcal{X}_r^*)q(x_n|x_{pred(n)}), \tag{22}$$

which is different from the target distribution in (17), in the sense that in addition to include the set of proposed values $\mathcal{X}_r$, it contains the probability of growing the specific DAG $\mathcal{G}_r$.

4.2 Sampling from the expanded target distribution

The procedure for updating the state of the Markov chain with a target distribution equal to (22), consists of two steps, equivalent to updating the Markov chain in the setup from Luo and Tjelmeland (2018). The first step is to grow a DAG and propose multiple values to be associated with it, by conditioning on a root node r with associated value x. The proposal distribution for the set of values $\mathcal{X}_r$ and the DAG $\mathcal{G}_r$ is set to be

$$f(\mathcal{X}_r, \mathcal{G}_r|r, x_r) = P(c(r)) \prod_{n \in \mathcal{S}_r \setminus \{r\}} P(c(n)|\mathcal{G}_r^*, \mathcal{X}_r^*)q(x_n|x_{pred(n)}), \tag{23}$$

which intuitively seems to be the full conditional distribution, and is used to update the Markov chain. Note that this expressions for the full conditional distributions is likely to be wrong, as they do not lead to correct acceptance probabilities, since the Markov chain does not converge to the correct target distribution as will be demonstrated in Section 7.1.

The second step is to propose one of the nodes r' in the grown DAG $\mathcal{S}_r$ as a root in a new DAG, with edges pointing away from the sampled node r'. The value associated with r' is accepted as a value from the original target distribution $p(x)$, and will used as a starting point for growing a new DAG in the next iteration.

The new root node is sampled by calculating the probability of every node n in $\mathcal{S}_r$ being the original root of another DAG, whose structure now is determined by $\mathcal{G}_r$. When a node r' is sampled, the new DAG will have edges pointing away from r', as shown in the process in Figure 2, when the new DAG in (b) is proposed conditioned on the DAG in (a). As a natural exstension of the full conditional distribution from (19), the probability for a node r' in $\mathcal{S}_r$ being the new root, given the set $\mathcal{X}_r$ and the structure of the DAG $\mathcal{G}_r$, is set to be

$$f(r'|\mathcal{G}_r, \mathcal{X}_r) = \frac{p(x_{r'})P(c(r')) \prod_{n \in \mathcal{S}_r \setminus \{r'\}} P(c(n)|\mathcal{G}_{r'}^*, \mathcal{X}_{r'}^*)q(x_n|x_{pred(n)})}{\sum_{n^* \in \mathcal{S}_r} p(x_{n^*})P(c(n^*)) \prod_{n \in \mathcal{S}_r \setminus \{n^*\}} P(c(n)|\mathcal{G}_{n^*}^*, \mathcal{X}_{n^*}^*)q(x_n|x_{pred(n)})}. \tag{24}$$

The expression in (24) resembles the probabilitiy used in Luo and Tjelmeland (2018) from (19), but differs in the sense that it contains the probability for growing the specific DAG resulting from r' being the root of the graph. The numerator represents the density of the joint distribution corresponding to the graph $\mathcal{G}_{r'}$, which has the same set of nodes $\mathcal{S}_r$ with associated values $\mathcal{X}_r$, but a different root node n' leading to the graph having a different set of edges. The denominator represents the sum of the densities of the joint distributions corresponding to every node n^* in $\mathcal{S}_r$ being the root of the graph. Note that the expression in (24) seems like a intuitive extension of the full conditional from Luo and Tjelmeland (2018), but since the algorithm does not converge when using this as the full conditional distribution, this expression is likely to be incorrect.

The expression in (24) is meant to be the full conditional distribution for the new root r', given the structure of the grown DAG $\mathcal{G}_r$ with the belonging set of values $\mathcal{X}_r$, and is used as the proposal distribution in the Gibb's update for the Markov chain. When a new root r' is sampled, the associated value x'_r is accepted as a value from the original target distribution, and the value will be used as a starting point for growing a new graph, according to an equivalent procedure as the graph $\mathcal{G}_r$ was grown. In this step the algorithm differs from the algorithm in Luo and Tjelmeland (2018), in the sense that the new DAG $\mathcal{G}_{r'}$ will grow independently of the structure of the previous DAG $\mathcal{G}_r$. It will only depend on the value x'_r associated with the new root node.

The procedure for growing a DAG and sampling a new root as described in this section is repeated for every iteration of the algorithm. The acceptance probabilities should be equal to one, because we are using the what seems to be the full conditional distributions as a proposal distributions. The acceptance probabilities based on the full conditional distributions presented above will be derived in the next section.

4.3 The acceptance probabilities

The M–H acceptance probability for accepting a proposed value, given the Markov chain's current state is in general expressed as in (3). In this section the acceptance probabilities for this algorithm are presented. Note that the derivations are based on the full conditional distributions presented in the previous section, and that these distributions are likely to be incorrect, since the algorithm does not converge correctly.

Updating the Markov chain in this setup consists of two steps. In the first step, we assume that the Markov chain's current state is a DAG $\mathcal{G}_r$, with associated values $\mathcal{X}_r$, and a root node r associated to the value x, which is distributed according to the target distribution $p(x)$. We wish to update the DAG and associated values, given the current root node r and associated value x. The full conditional distribution for a new DAG with new associated values, given r and x is expressed in (23) and is used as the proposal distribution. Using the target distribution in (22) and the full conditional distribution from (23), the expression for the M–H probability becomes

$$\alpha(\mathcal{G}'_r, \mathcal{X}'_r | \mathcal{G}_r, \mathcal{X}_r) = \min\left\{1, \frac{f(\mathcal{G}'_r, \mathcal{X}'_r) f(\mathcal{G}_r, \mathcal{X}_r | r, x)}{f(\mathcal{G}_r, \mathcal{X}_r) f(\mathcal{G}'_r, \mathcal{X}'_r | r, x)}\right\}, \tag{25}$$

which when inserting the full conditional and the target distribution, gives a M–H probability equal to

$$\alpha(\mathcal{G}'_r, \mathcal{X}'_r | \mathcal{G}_r, \mathcal{X}_r) = \min\left\{1, \frac{p(x) P(c(r)_{\mathcal{G}'_r}) \prod_{n \in \mathcal{S}'_r \setminus \{r\}} P(c(n) | \mathcal{G}'^{\,*}_r, \mathcal{X}'^{*}_r) q(x_n | x_{pred(n)})}{p(x) P(c(r)_{\mathcal{G}_r})) \prod_{n \in \mathcal{S}_r \setminus \{r\}} \left(P(c(n) | \mathcal{G}^*_r, \mathcal{X}^*_r) q(x_n | x_{pred(n)})\right)}\right.$$

$$\left. \cdot \frac{P(c(r)_{\mathcal{G}_r}) \prod_{n \in \mathcal{S}_r \setminus \{r\}} P(c(n) | \mathcal{G}^*_r, \mathcal{X}^*_r) q(x_n | x_{pred(n)})}{P(c(r)_{\mathcal{G}'_r}) \prod_{n \in \mathcal{S}'_r \setminus \{r\}} P(c(n) | \mathcal{G}'^{\,*\prime}_r, \mathcal{X}'^{*}_r) q(x_n | x_{pred(n)})}\right\}, \tag{26}$$

where $c(r)_{\mathcal{G}_r}$ and $c(r)_{\mathcal{G}'_r}$ is the number of children for the root node r in the DAGs $\mathcal{G}_r$ and $\mathcal{G}'_r$, respectively. These two differ because they belong to two different graphs, but condition on the same root node r associated to the same value x. The terms in the M–H ratio in (26) will cancel, and we are left with an acceptance probability equal to one. This is expected since since the full conditional distribution is used as the proposal distribution, meaning that it is a Gibbs update of the Markov chain.

In the second update, the root node is updated in a grown DAG, conditioned on the current DAG's structure and associated values. The expression for the M–H acceptance probability when using the full conditional for r' becomes

$$\alpha(r'|r) = \min\left\{1, \frac{f(\mathcal{G}_{r'}, \mathcal{X}_r) f(r|\mathcal{G}_{r'}, \mathcal{X}_r)}{f(\mathcal{G}_r, \mathcal{X}_r) f(r'|\mathcal{G}_r, \mathcal{X}_r)}\right\}. \tag{27}$$

Notice that the set of proposed values $\mathcal{X}_r$ is fixed in this step, and that the graphs $\mathcal{G}_r$ and $\mathcal{G}_{r'}$ have the same set of nodes. In this step, only the root node changes, and its associated value is accepted as a value from the original target distribution $p(x)$. Inserting the target distribution from (22) and the full conditional distribution from (24) into the expression for the M–H acceptance probability, the expression becomes

$$\alpha(r'|r) = \min\left\{1, \frac{p(x')P(c(r'))\prod_{n\in\mathcal{S}_r\setminus\{r'\}} P(c(n)|\mathcal{G}^*_{r'}, \mathcal{X}^*_{r'}) q(x_n|x_{pred(n)})}{p(x)P(c(r))\prod_{n\in\mathcal{S}_r\setminus\{r\}} P(c(n)|\mathcal{G}^*_{r}, \mathcal{X}^*_{r}) q(x_n|x_{pred(n)})}\right. \tag{28}$$

$$\left. \cdot \frac{\frac{p(x)P(c(r))\prod_{n\in\mathcal{S}_r\setminus\{r\}} P(c(n)|\mathcal{G}^*_{r},\mathcal{X}^*_{r}) q(x_n|x_{pred(n)})}{\sum_{n^*\in\mathcal{S}_r} p(x_{n^*})P(c(n^*))\prod_{n\in\mathcal{S}_r\setminus\{n^*\}} P(c(n)|\mathcal{G}^*_{n^*},\mathcal{X}^*_{n^*}) q(x_n|x_{pred(n)})}}{\frac{p(x')P(c(r'))\prod_{n\in\mathcal{S}_r\setminus\{r'\}} P(c(n)|\mathcal{G}^*_{r'},\mathcal{X}^*_{r'}) q(x_n|x_{pred(n)})}{\sum_{n^*\in\mathcal{S}_r} p(x_{n^*})P(c(n^*))\prod_{n\in\mathcal{S}_r\setminus\{n^*\}} P(c(n)|\mathcal{G}^*_{n^*},\mathcal{X}^*_{n^*}) q(x_n|x_{pred(n)})}}\right\}.$$

The term including the sum over all nodes n^* in $\mathcal{S}_r$ is present in both the numerator and the denominator in the acceptance probability, since the set of nodes $\mathcal{S}_r$ is fixed in this step. The other terms include the expanded target distribution with r and r' as roots in both the numerator and denominator, so this will also cancel. We are left with an acceptance probability equal to one.

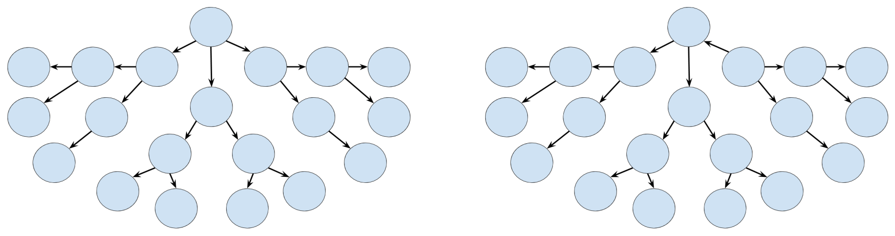
5 Methods for growing the DAG

When growing a DAG stochastically it is important to make sure that it does not grow infinitely large. The size of a DAG can be measured by its number of levels and nodes, so it is natural to use at least one of these quantities in the probability function $P(c(n)|\mathcal{G}^*_r, \mathcal{X}^*_r)$ from (21), for sampling a number of children $c(n)$ for a node n. If we do not include the size of the DAG, we can not prevent the DAG from growing infinitely large, so it is necessary to make the probability function dependent on either the number of nodes or the number of levels. The *level* of a node n in a DAG is defined as the number of edges between the root node r and the node n. The root itself is always located at level 0, and the level of the node located furthest away from the root is defined as the height of the DAG. Nodes that do not not have any children are referred to as *leaf nodes*. For example the DAG in Figure 3 (a) has a height of three, while the DAG in Figure 3 (b) has a height of four, since seven of its leaf nodes are located four edges away from the root. Recall that the motivation for growing the DAG stochastically is to increase the probability of proposing high density values. It is therefore also essential to make the probability function dependent on the density of the values $\mathcal{X}^*_r$ associated with the growing DAG $\mathcal{G}^*_r$.

5.1 Growing the DAG dependent on levels

A naive approach for growing the DAG is to make the probability of a node n having children, dependent on the levels in the DAG. By continuously decreasing the probability for sampling children as the levels in the DAG increase, the DAG will eventually stop growing with a probability of one. As an example, let the number of children for a node n be distributed according to a Poisson distribution

$$P(c(n)|\mathcal{G}^*_r, \mathcal{X}^*_r) = \exp\{-\lambda\} \cdot \frac{\lambda^{c(n)}}{c(n)!}, \tag{29}$$



(a) An illustration of a balanced DAG, meaning that the level of each leaf node is the same. Here the height of the DAG is equal to three

(b) A version of the DAG in (a) which has the originally grown DAG's root's right child as the new root. This makes the height of the DAG equal to four, and sets the probability of this node being sampled equal to zero.

Figure 3: An illustration of a balanced DAG of height three grown by Method 1. If the function for the intensity parameter λ is as in (30), all nodes except the original root node has a zero probability of being sampled.

where the intensity parameter λ is a function of the level $level(n)$ of the node n, its associated value x_n, and the value associated with the predecessor, $x_{pred(n)}$. We set the function for the intensity parameter λ_n equal to

$$\lambda_n(n|x_n, x_{pred(n)}, level(n)) = \max\left\{0, (4 - level(n)) \cdot \min\left\{2, \frac{p(x_n)}{p(x_{pred(n)})}\right\}\right\}. \tag{30}$$

The method for growing the DAG by using the function for the intensity parameter in (30) will be denoted by *Method 1* from here on. A problem with making λ_n a function of the level of a node as in (30), is that most of the nodes in a fully grown DAG might have a probability of zero of being sampled as the new root. This is because if a grown DAG $\mathcal{G}_r$ has three levels, most of the nodes in $\mathcal{S}_r$ will have a probability of zero of being sampled as the new root, since many of the variations of the DAG will have more than three levels. If a DAG has more than three levels, the intensity parameter (30) will be set to zero, which makes the probability of the node having children equal to zero. This leads to the whole numerator in the full conditional distribution in (24) becoming zero. The situation is illustrated in Figure 3.

To prevent the situation discussed above, it seems reasonable to avoid situations in which the DAG will either grow or stop growing with zero or one probabilities, as this has a high risk of making other variations of the DAG impossible. The probability for a node n having children should thus never be set to zero, but continuously decrease as the DAG grows larger. Another approach is to replace the first factor in (30) and get

$$\lambda_n(n|x_n, x_{pred(n)}, level(n)) = \frac{8}{2^{level(n)}} \cdot \min\left\{2, \frac{p(x_n)}{p(x_{pred(n)})}\right\}, \tag{31}$$

which will eliminate the zero probability situations discussed above. The method for growing the DAG by using the function for the intensity parameter in (31) will from now on be denoted by *Method 2*. Even if the adjusted function for the intensity parameter eliminates the zero probability situations, the probability function in Method 2 still favours well balanced DAGs. Variations of the DAGs which have former leaf nodes as roots in a new DAG, are likely to have significantly more levels than the originally grown DAG. For example, a DAG resulting from one of the former leaf nodes in the DAG from Figure 3 (a) being the new root, will have a height equal to six, which makes the probability for this variation of the DAG very small. The computational cost of proposing values associated to the leaf nodes will be useless, if these nodes are likely never to be sampled as new roots. Growing the DAG by Method 2 and Method 3 will make the algorithm likely to resample its old root, or to sample one of the root node's neighbours in most of the iterations.

5.2 Growing the DAG independent on levels

As discussed above, growing the DAG dependent on levels makes leaf nodes unlikely to be sampled as new roots. This motivates for constructing a new probability function independent of the levels, which is more tolerant for different structures. We wish to assign a high probability for nodes having children when the nodes are associated to high density values, at the same time as the DAG needs to stop expanding when it becomes too large. A different approach is to make the probability of a node n having children dependent on the value x_n compared to the densities associated to the other nodes in the growing DAG $\mathcal{G}_r^*$. Let

$$\lambda_n(\mathrm{X}_r) = |\ \{\ \mathrm{x}_{n^*} \in \mathcal{X}_r^* \setminus \{p(x_{n^*}) > p(x_n)\}\}|, \tag{32}$$

where $\mathcal{X}_r^*$ is the set of associated values for all nodes in $\mathcal{G}_r^*$ at lower levels than n. That is, λ_n is the number of proposals in $\mathcal{X}_r^*$ which is of higher density than $p(x_n)$. Using (32) as a function for the intensity parameter, another suggestion for the probability function in (21) is

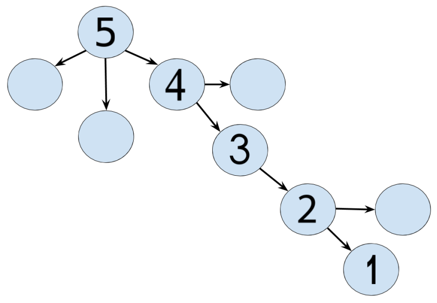
$$P(c(n)|\mathcal{G}_r^*, \mathcal{X}_r^*) = \exp\{-\lambda_n \cdot \alpha - \beta\}, \tag{33}$$

where α and β are parameters to be set in the M–H setup. The expected value for a node having children is not Poisson distributed like it is for Method 1 and Method 2, but gives a probability for a node a having specific number children, given in the M–H setup. We have in our simulation experiments in Section 7 set the number of children equal to three. That is, the probability for a node n having three children is equal to the expression in (33). Consequently, the probability for a node n not having children is equal to the compliment of the probability in (33).

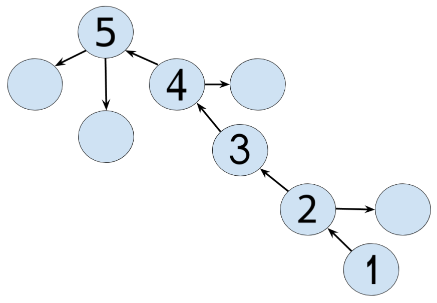
The expression in (33) will be a decreasing function of the size of the DAG. As the number of nodes increases, λ_n will also increase. This is because the target density has its global maxima at the expected value of the target distribution, which means that the density associated to the nodes have an upper bound equal to the expected value of the target distribution. This will ensure that the DAG does not become infinitely large, at the same time as it appears that the probability function does not favour any particular structure. All variations of the grown DAG will have the same number of nodes, so intuitively this method will make DAGs that were not originally grown more likely to be accepted as new states, compared to Method 1 and Method 2. Note that the parameter β is added to avoid zero or one probabilities. This could happen if for example a node n has the temporary largest value in $\mathcal{X}_r^*$ associated with it, which makes the first factor in the exponent of (33) equal to zero. If the constant was not added, this would make the probability for node n having children equal to one, which is not optimal, because other variations of the DAG could have nodes that are assigned a probability of one for having children, but yet does not have any. This happens if the nodes were not the temporary largest nodes when the originally grown DAG was grown.

A weakness with the method discussed above is that high density values are likely to generate few proposals. For example, if the root node is associated with the mean value, the global maxima, the root node is likely to sample children, but the children of the root node will almost certainly have values of lower density than the root node. This will decrease the probability of sampling more nodes in the DAG, which means that a DAG with a high density value as a starting point is likely to consist of few nodes. On the other hand, if a DAG has a low density value as the starting point for growing the graph, the DAG is likely to grow large as the proposed values may be of higher and higher density. These high density points are on the other hand very unlikely to be accepted as new states, since a DAG with a high density point associated with the root node is likely to produce few nodes, making this variation of the DAG unlikely to be accepted when the originally grown DAG is large. An example of this is illustrated in Figure 4, where the subset of two DAG's are indexed by the order of the density associated with each node, ordered from highest to lowest. Figure 4 (a) shows a root node associated with the lowest density of the five indexed nodes. If we compare the probability of proposing values from the root node indexed by 5 to the node associated with the highest density point indexed by 1 in (a), with the proposals being generated in the opposite order as in (b), proposing the values in an increasing order using (33) will add the factor

$$\exp\{-\beta 4\} \tag{34}$$



(a) The sub graph of a DAG with root node indexed by 5. The proposed values are proposed in the order of the directed edges, here the proposed values are proposed in an increasing order.



(b) A version of the sub graph from (a). Here the root node is indexed by 1, indicating that it has the largest density value of the five indexed nodes associated with it. The proposals are proposed in a decreasing order.

Figure 4: An illustration of the conflict of generating high density values and sampling nodes of high density values as new roots. The two sub graphs show the effect of favouring nodes associated with low density points. The nodes that are not indexed can be disregarded, as these are not a part of the subset which creates this effect.

to the joint distribution in (22). On the other hand, proposing the values in the opposite order will add the factor

$$\exp\{-\alpha(1+2+3+4)-\beta 4\} = \exp\{-10\alpha-\beta 4\}, \tag{35}$$

making the structure of the DAG with the lowest value associated with the root $\exp\{\alpha 10\}$ times more likely to be accepted as the new state. As we do not want the target distribution to favour nodes of low density, this is a very undesirable effect. Still the factor $p(x)$ of the expanded target distribution in (22) will still favour high density values, but growing the DAG by using the probability function in (33) is not likely to reinforce the probability of sampling high density values as a new state.

In general it is preferable to grow the DAGs in a way that makes the density associated with the root node dominate the density of the joint distribution in (22), instead of the specific structure of a grown DAG. It will be impossible to make the joint distribution of the DAG independent of its structure, since the growth is stochastic and therefore included in the joint distribution. In most cases, the DAG that was originally grown will be the version of the DAG that has the highest density, since typically the DAG with the highest probability is the DAG that is grown. This is a weakness compared to the algorithm presented in Luo and Tjelmeland (2018), where the structure of the DAG is independent of the expanded target distribution, since the structure is fixed. This makes the proposed values dominate the expanded target distribution to a larger extent. Some strategies have been discussed in this section, but they all have weaknesses. A more optimal strategy is likely to exist, and a combination of the strategies above may also yield better results. Some of the methods are implemented and results are shown in Section 7.

6 Implementation of the algorithm

The algorithm is a Markov Chain Monte Carlo algorithm (MCMC), which is broadly used in Bayesian inference to generate samples from distributions which are difficult sample from directly. As mentioned in the previous section, the algorithm is a multiple proposal algorithm, since it proposes multiple values in every iteration. By using the DAG, the algorithm has the potential to explore larger parts of the original target distribution in each iteration. The DAG should ideally be grown in the direction of high density points, which in practice means that nodes which are associated to high density points have a larger probability of sampling children.

The values associated with each node in the DAG are dependent on the value associated with the predecessor of the node, and the values associated with nodes on lower levels in the DAG. This means that the nodes on each level are independent of each other, which makes the algorithm possible to paralleize. To implement the algorithm the program language C++ was used, which is an object oriented program ideal for making own data structures. Calculations were made on the logarithmic scale to prevent numerical errors.

6.1 Classes and data structures

The data structure is meant to be easy to modify, because calculating the probability for the nodes in the DAG being sampled as the new root includes changing the root node to every node in the grown DAG. This motivates for creating a data structure which is easy to modify and avoids unnecessary copying of objects and values included in the DAG structure. Therefore the relations in the DAG are implemented by pointers (Savitch, 2013, p 448.), so that when the root node changes, we simply change the pointers in a DAG corresponding to a change of root node. A DAG is implemented by using two different classes, one representing a node, and one representing a DAG. A Node object holds the information of the node stored as attributes and is connected to a DAG through pointers to the other nodes. An illustration of how the nodes are connected through pointers is shown in Figure 5. Here an arrow from one node n to another node represents a pointer attribute in n which points to the other node. All nodes in the DAG structure are unique, the nodes are never copied. Only the pointers referring to the allocated memory for a node is copied, meaning that two arrows pointing to the same node in Figure 5, are two copies of the same pointer, referring to the same node, but are stored as attributes in two different Node objects.

The class representing the nodes in the DAG has a large set of attributes. It contains the proposed position and density generated from (14), and the density of the target distribution $p(x)$ at the proposed value. For a Node object n, both the density $q(x_n|x_{pred(n)})$ and the density $q(x_{pred(n)}|x_n)$ is stored. For a node which is the root node, both these attributes are set to zero. The nodes also contain pointers, representing the edges in the graph as shown in Figure 5. Each node except for the root node in a DAG contains a pointer to its predecessor. If a node has children, it will have a pointer to its first child, and this child will have a pointer to the node's next child, so the children of a node are connected through a singly linked list. This also means that if a node has *siblings*, it will have a pointer pointing to the next child in the singly linked list, unless it is the last child that was added. Then this pointer is set to the null pointer

Memory locations for each node is dynamically allocated (Savitch 2013, p. 467) when a Node object gets its number of children sampled, and the children are stored in an array which is kept as an attribute in the parent node. Note that this array is the only place the Node objects are actually stored. All other references to a Node object are pointers referring to the address of the node. Since the structure of the DAG is modified several times through each iteration when the root is changed, it is important to keep track of the allocated memory so it can be freed in a destructor (Savitch 2013, p. 487). A Graph object consisting of nodes needs to be destructed in a top down manner, meaning that the leaf nodes need to be deleted first and the root node last. This is because if nodes at low levels are deleted before the nodes at higher levels, the pointers to the leaf nodes are also deleted, making the memory for the leaf nodes unreachable and impossible to release. This motivates for storing the nodes in the array which is static through the life time of a Graph object, and this is the array stored in every parent node when a node samples its number of children, when the DAG is originally grown at the beginning of an iteration. The pointer relations in Figure 5 are created in addition to these static arrays, to be able to modify the relations within the DAG, without changing the references needed in the destructor at the end of an iteration. The pointers to each Node object is copied in a two dimensional array belonging to a Graph object, and this array is filled when the DAG is originally grown, at the beginning of an iteration of the algorithm. The first index in the array represents the level of the Node object its pointing to in the originally grown DAG. This two dimensional array is what is used in the destructor, by using the indexes representing the levels in the originally grown DAG, and deleting the Node object's allocated array of children in a top down manner.

The Node objects also have some special attributes related to each method for growing

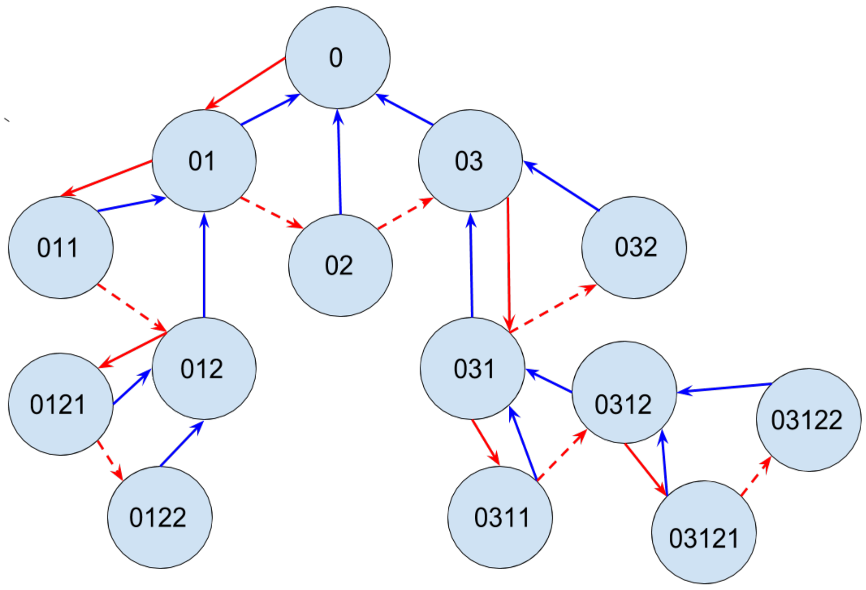


Figure 5: The implemented DAG data structure. The arrows represent the pointers in the DAG. Every arrow pointing out from a node n represents a pointer attribute in n, pointing to the node it is pointing to. A blue arrow represents the pointer from a node to its predecessor. For the root node, the predecessor is set to the null pointer. The solid red arrow represents a pointer from a node to its first child, and a dashed red arrow represents the singly linked list between the children of a node. The indexes are the ID's of every node, which is assigned when a DAG is originally grown at the beginning of an iteration. Starting with assigning the ID of the root node to 0, the children of each node will have its ID's first part identical to its predecessor's ID, followed by the number in which the child was added to its predecessor. For example the node 031's first child will have the ID "031" + "1", which becomes 0311.

the DAG from section 5. For example the methods for growing the DAG based on the density associated with nodes at lower levels than the current node sampling its number of children, uses a linked list ordering the nodes in an increasing order. The list is reconstructed for every change of root in an iteration, with the first element inserted corresponding to the value associated with the new root node. Each node also has a boolean value indicating whether a node is a root node, and an ID representing its position in the originally grown DAG as shown in Figure 5. The target distribution $p(x)$ and proposal distribution $q(x_n|x_{pred(n)})$ for proposing values within the DAG, are objects declared as attributes to every node. These attributes are pointers to specific target- and proposal distributions, which are declared as abstract classes (Savitch 2013, p. 699) to make the code more applicable for different target- and proposal distributions.

6.2 Implementing the growth of the DAG

In each iteration of the algorithm, a DAG is grown based on the current state's root node and associated value. The graph is grown by a method similar to a breadth first search (BFS) (Cormen et al. 2009, pp.594-602), by using the probability function in (21) on all nodes at the current level, before using the probability function on the nodes at the next level. The pseudo code for growing a DAG by BFS is shown in Figure 6. For example to grow the DAG in Figure 5, the root node "0" is the first node to sample its number of children. Since this is the starting point for the DAG, it already has its value from the target distribution associated with it. The children of the node "0" are added to a vector representing the current level of the DAG. For example for the DAG in Figure 5, the nodes labeled "01", "02" and "03" are added to the vector representing the current level. Then values for the nodes at the current level are proposed, based on their predecessor's value through (14). After this, the number of children for each node at the current level is sampled through (21), and memory for these children are allocated. The children sampled for the nodes at the current level are added to a "queue", which is a vector to keep track of all nodes at the next level of the DAG. When "01", "02" and "03" are at the current level, the nodes which are added to the queue becomes "011", "012", "031" and "032". When the values associated with the nodes at the current level have been proposed and each node's number of children has been sampled, the nodes at the current levels are "finished" and we move on to the next level. The procedure repeats itself for the nodes at the next level, by setting the vector representing the current level equal to the queue vector, and emptying the queue. This continues until all nodes at the current level have sampled a number of children equal to zero, meaning that the DAG has stopped growing.

The computational cost of proposing values to be associated with the nodes in the for-loop in Figure 6 and calculating the density corresponding to these values is expensive. Since the DAG is grown by a BFS-strategy, all nodes at a specific level are independent. This means it is possible to propose values and calculate densities for nodes at the same level by parallel programming, which will make the algorithm run faster, although this has not been a focus in this work.

6.3 Changing root node and calculating the acceptance probabilities

Calculating the probabilities for every node in a DAG becoming the new root according to the probability in (24) involves changing the root node to every possible node in the originally grown DAG. To do this, we simply turn the pointers in the DAG corresponding to a change of root node. The pseudo code for changing the root in shown in Figure 7. What is important when changing the pointers in a DAG with relations as in Figure 5, is not to accidentally delete any of the pointers permanently, and not to add a pointer to a node too many times or in wrong relations. For example adding a node as child to more than one node at the time will cause the DAG to be inconsistent, and would result in infinite loops when searching in the DAG. To avoid this, exception handling has been added to reassure that nothing unexpected happens when changing the root of a DAG.

A pseudo code for changing the root of the DAG in 5 from node "0" to node "03122" is shown in Figure 7. Note that the operations for delete child and add child only deletes and adds one pointer at the time. For example the operation delete child for deleting the node "02" as child of node "0" in Figure 5 will only remove "02" from the linked list representing the children of

Grow DAG by BFS

```
Initialize root node = n, empty vector currentLevel, empty vector queue
Sample the number of children for the root node and allocate memory
Add children of the root to currentLevel
while currentLevel not empty {
    for node in currentLevel {
        propose value for node
        sample the number of children for node and allocate memory
        for children of node {
            Add child to queue
        }
    }
    set currentLevel = queue
    clear queue
}
```

Figure 6: Pseudo code for the generalized RAM algorithm.

node "0". That is, the dotted red line from node "01" to "02" will be deleted, but node "0" will still be the predecessor of node 02. When adding a child to a node, the child gets added to the linked list of children for the node, but the predecessor of the child is not changed. Changing the nodes' predcessors are independent operations.

A root change for the DAG in 5 with old root "0" to a new root "03122" according to the pseudo code in Figure 7 is shown in Figure 8. Notice that we need $temp1$ and $temp2$ to store pointers to two nodes at the time, since temporarily changing the edges in the DAG results in a disjointed graph. Also note that a node n has to be deleted as a child from its current predecessor before being added as a child to its new parent. This is because of the linked list structure of the children. If a node n is added as child to another node, before being deleted as child from its predecessor $pred(n)$, the linked structure following n from the linked list belonging to $pred(n)$ will be added to the new parent, causing the old siblings connected to n from $pred(n)$ being added as children to the new parent node.

Calculating the probability of a specific node n being the root node in a DAG from (19) is done by a BFS search through a DAG which has its root node changed by the pseudo code in Figure 7. While searching the DAG, some values are updated and recalculated, and some values remain constant. For example it is not necessary to recalculate the density $p(x)$ in any of the nodes, since this does not change when the root node changes. Recall that for a node n both the value $q(x_n|x_{pred(n)})$ and the value $q(x_{pred(n)}|x_n)$ is stored in the node n. When the root node changes, the nodes with turned edges get these two attributes pushed forward to its new children which used to be their predecessors, and then the values are swapped. This process is shown in Figure 9. This will result in these two attributes being pushed one level forward at the time, starting from the new root node and ending in the old root node. The nodes which are not involved in the change of root node will have these two values unmodified. The result of this process is that the new root node will have these two attributes set to zero. Note that if $q(x_n|x_{pred(n)})$ is symmetric, the same values are pushed one level forward, and the sum of all attributes generated by $q(x_n|x_{pred(n)})$ is constant for every change of root within an iteration, so these values will cancel in the acceptance probability in (28). Other values that need to be recalculated is the probability for every node n^* in the DAG having its new number of children. This is because the number of children might be changed, and the parameters for calculating the probability changes when the root is changed. It is important to use BFS to calculate the probabilities for all the nodes being the root, since the originally grown DAG was grown through BFS. To minimize the risk of bugs in the code, we have implemented the change of root node and recalculations of the probabilities within the nodes inside the BFS-search that was used to grow the DAG. This is so that the code lines which sets

Change root node

```
Set temp1 = newRoot's predecessor's predecessor
Set temp2 = temp1's predecessor
Delete newRoot's predecessor as child of temp1 node
Add newRoot's predecessor as child of newRoot node
Delete newRoot as child for newRoot's predecessor node
Set iterator = newRoot's predecessor
Set newRoot's predecessor's predecessor = newRoot
Set newRoot's predecessor to NULL
while iterator is not the old root{
        Set temp1'predecessor to iterator
        Set iteraor = temp1
        if temp2 is not NULL {
                Delete temp1 as child of temp2 node
                Set temp1 = temp2
                Set temp2 = temp1's predecessor
        }
        Add iterator as child to iterator's predecessor node
}
```

Figure 7: Pseudo for changing the root node of a DAG. The variables newRoot, newRoot's predecessor, temp1, temp2 are pointers to node objects, while newRoot node, newRoot's predecessor node etc are Node objects. The operations delete- and add child only deletes and adds one pointer at the time, and are independent of the predecessor pointer of the involved nodes.

the probabilities in a growing DAG and the probabilities in the BFS for modified DAGs uses the same lines of code.

7 Simulation experiments

In this section we investigate the convergence properties of the algorithm presented in this work. We demonstrate that it in general does not converge to the given target distribution. Instead, it converges to distributions with the same mean as the target distribution, but with a variance dependent on the parameters used in the proposal distribution. We investigate how the algorithm converges for the different parameters by using it to sample from a bivariate Gaussian distribution

$$p(\mathbf{x}) = \frac{\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\}}{\sqrt{(2\pi)^2|\boldsymbol{\Sigma}|}}, \tag{36}$$

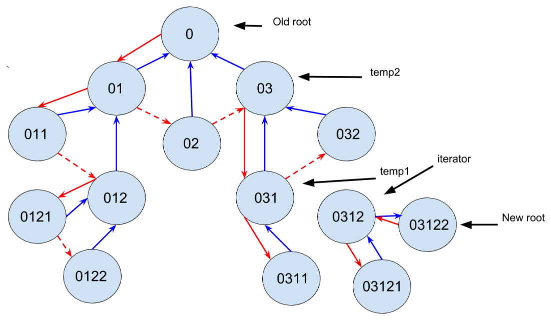
with mean vector,

$$\boldsymbol{\mu} = \left[\begin{array}{c} 1 \\ 2 \end{array}\right]$$

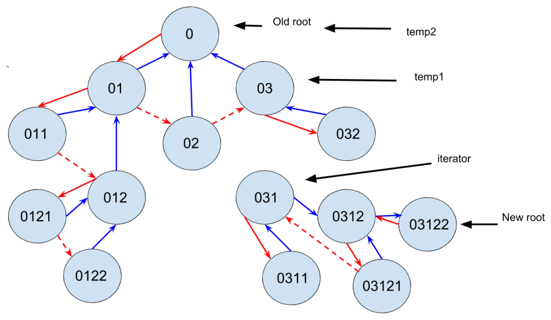
and covariance matrix

$$\boldsymbol{\Sigma} = \left[\begin{array}{cc} 1.0 & 0.5 \\ 0.5 & 4.0 \end{array}\right].$$

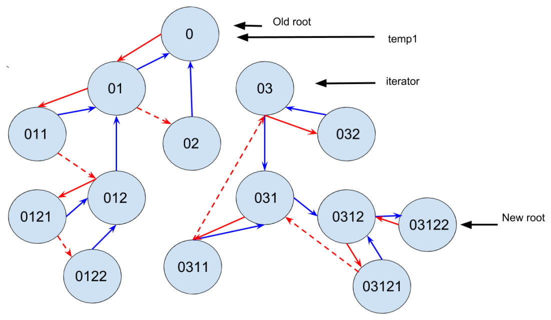
Since all simulations seem to converge to the correct mean vector, we choose to focus on the



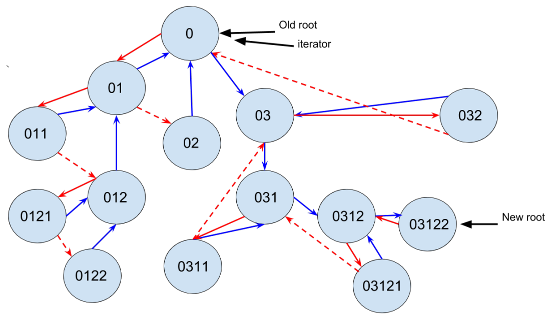
(a) The DAG with one edge turned. This is the DAG's state just before the while loop in Figure 7.



(b) The DAG's state after the first while loop is executed. Here two edges have been turned.

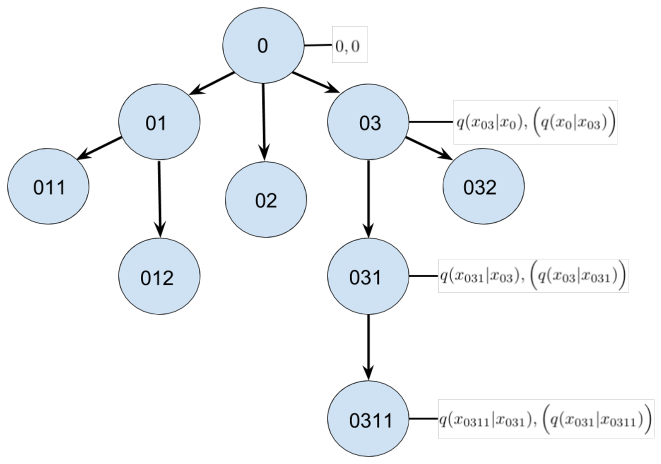


(c) The DAG's state after the second iteration of the while-loop. After this iteration, *temp*2 is set to NULL

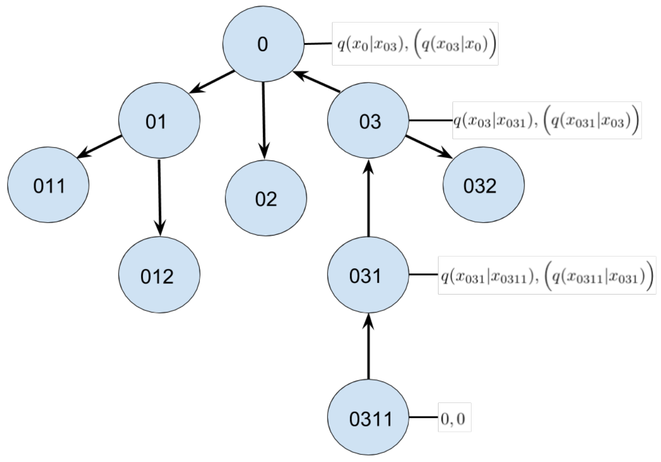


(d) The DAG's state after the third and last iteration of the while loop. Here the change of root node is complete.

Figure 8: The procedure of changing the root node of the DAG in 5 according to the pseudo code in Figure 7.


0
$0, 0$
01
03
$q(x_{03}|x_0), \big(q(x_0|x_{03})\big)$
011
02
032
012
031
$q(x_{031}|x_{03}), \big(q(x_{03}|x_{031})\big)$
0311
$q(x_{0311}|x_{031}), \big(q(x_{031}|x_{0311})\big)$

(a) A DAG with arrows indicating the stored attributes for the density of the proposal distribution to be used in the acceptance probabilities. Here the density $q(x_n|x_{pred(n)})$ is the probability of being proposed in the current DAG, and will be included in the calculation of the acceptance probability for the DAG with "0" as root. The density of "going backwards", $q(x_{pred(n)}|x_n)$, is also stored and shown in paranthesis.


0
$q(x_0|x_{03}), \big(q(x_{03}|x_0)\big)$
01
03
$q(x_{03}|x_{031}), \big(q(x_{031}|x_{03})\big)$
011
02
032
012
031
$q(x_{031}|x_{0311}), \big(q(x_{0311}|x_{031})\big)$
0311
$0, 0$

(b) The DAG from (a) with a new root node, leading to three edges being turned. When an edge is turned, the attributes that represent the densities of the proposed values are swapped, so that the old value for "moving forward" is shown in the paranthesis. The attributes are then pushed one level forward, in the direction of the old root "0".

Figure 9: How the attributes representing the density of the proposal distribution is re-used and moved around when the root node changes. If the proposal distribution is symmetric, the sum of the density of the proposed values will be constant for every change of root node.

estimated covariance matrix for the different simulations. We try to locate the error in the M–H setup by systematically varying the different parameters and investigate how this effects the estimated covariance matrix of the simulations. The target distribution in (36) is used throughout the different simulation experiments, except for one experiment where the target distribution is changed to verify the convergence of simulations with a specific choice of parameters used in the proposal distribution.

We also investigate whether the algorithm succeeds in proposing higher density values within the DAG compared to the algorithm from Luo and Tjelmeland (2018), which is the main motivation for growing the DAG stochastically. The different methods for growing a DAG from Section 5 are tested on their ability to propose high density values within the DAG and their ability to jump between levels when sampling a new root at the end of each iteration.

7.1 A variance dependent on parameters in the proposal distribution

In this section we investigate the estimated covariance matrix for running the algorithm for different sets of parameters. By systematically changing the parameters involved in the M–H setup, we hope to get an understanding for where the error is located. To grow the DAG in this experiment, the strategy independent of levels from Section 5.2 is used. The probability for a node having a certain amount of neighbours is given in a probability function, and each node either has four or one neighbour. The probability for a root node having four neighbours, corresponding to the root node having four children, is set to 0.75. This implies that the probability for the root node having only one neighbour, corresponding to having one child, is set to $1 - 0.75 = 0.25$. For the other nodes, the probability of having three children, corresponding to four neighbours, is equal to the probability in expression (33). To propose values within the DAG by the proposal distribution in (14), a Gaussian distribution with the tuning parameter equal to one is used.

The algorithm was run for different values of the parameters α and β from the probability function in (33). By keeping one of the parameters fixed while varying the other, we try to get an impression of how the different parameters effect the result. Five separate runs where made for each pair of α, β, and the results are shown in Tables 1 to 3 with the standard deviations shown in the paranthesis. Some of the simulations showed particularly interesting results, and five new runs were made with a larger number of iterations, to get a more accurate estimate of the covariance matrix for these special cases. The simulations considered to be of extra interest are the simulations for $\alpha = 10.0$ and $\alpha = 20.0$, and simulations with $\beta > 10$.

The simulations all seem to converge to distributions with the mean equal to the mean of the target distribution in (36), with a standard deviation of 0.02 at most, so this is not included in the tables below. Plots for 20000 iterations of the algorithm for $\alpha = 20.0, \beta = 0.2$ and $\alpha = 0.0, \beta = 30.0$ is shown in Figure 10. Looking at the plots, it seems like the simulations converge to a Gaussian distribution with the same mean, but the variances seem to be different for the two simulations, as the plot in Figure 10 (a) shows a larger spread between the simulated values.

Most of the estimated covariance matrices from the simulations shown in Tables 1 to 3 are significantly different from the covariance matrix of the target distribution, and it is clear that the algorithm in general does not converge correctly, but converges to distributions close to the target distribution. Some pairs of parameters, in particular when the parameter β is large, seem to have the estimated covariance matrix equal to the covariance matrix of the target distribution, so these simulations stand out. Other than that, it seems like the variance is too high when α is large and too low when α is small, and is correct when $\beta > 10$. Although it is difficult to find an exact pattern in the estimated covariance matrices, finding one might indicate where in the M–H setup the error is located.

In Table 1 the parameter β is fixed while α varies. Varying α will effect how large the DAG grows, with high values of α causing smaller DAGs. It also effects how dramatically the probability of having children differs at different levels, since this is likely to increase the exponent in the probability function in (33). The trend for the variance of the limiting distributions in Table 1 is that the variance in general is too large compared to the target distribution. The variance increases as the parameter α increases, but seems to stop growing as α becomes greater than 10, and the estimated covariance matrices seem to stabilize. This is likely because the DAGs grown

Table 1: The estimated covariance matrix for simulations with different values for α while keeping β fixed. Simulations with large values for α it seem to converge to similar target distributions, since the estimated covariance matrices do not differ significantly.

iterations	α	β	$\hat{\mathbf{\Sigma}}_{\mathbf{1,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,2}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$
100 000	0.5	0.2	1.13 (0.03)	0.62 (0.10)	0.62 (0.10)	4.57 (0.11)
100 000	1.0	0.2	1.26 (0.01)	0.67 (0.05)	0.67	5.14 (0.08)
200 000	5.0	0.2	2.06 (0.02)	1.13(0.05)	1.13 (0.05)	8.87 (0.05)
200 000	10.0	0.2	2.64 (0.03)	1.59 (0.13)	1.59 (0.13)	11.68 (0.12)
200 000	20.0	0.2	2.63 (0.00)	1.62 (0.07)	1.62 (0.07)	11.83 (0.31)

Table 2: The estimated covariance matrix for simulations with different values for β while keeping α fixed. The estimated covariance matrices for simulations with $\beta > 10$ seem to converge to the covariance matrix of the target distribution.

iterations	α	β	$\hat{\mathbf{\Sigma}}_{\mathbf{1,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,2}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$
100 000	0.2	0.2	1.02 (0.02)	0.53 (0.02)	0.53 (0.02)	4.13 (0.05)
100 000	0.2	0.5	0.95 (0.03)	0.46 (0.03)	0.46 (0.03)	3.83 (0.04)
100 000	0.2	1.0	0.92 (0.01)	0.46 (0.01)	0.46 (0.01)	3.67 (0.03)
100 000	0.2	5.0	0.95 (0.04)	0.48 (0.01)	0.48 (0.01)	3.80 (0.12)
400 000	0.2	10.0	1.00 (0.01)	0.48 (0.02)	0.48 (0.02)	3.93 (0.04)
400 000	0.2	20.0	1.01 (0.01)	0.51 (0.02)	0.51 (0.02)	3.92 (0.10)
400 000	0.2	30.0	1.00 (0.01)	0.51 (0.01)	0.51 (0.01)	4.06 (0.08)

Table 3: The estimated covariance matrix for simulations with $\alpha = 0$, making the growth of the DAGs independent of the densities of the proposals. The estimated covariance matrices for simulations with $\beta > 10$ seem to converge to the covariance matrix of the target distribution.

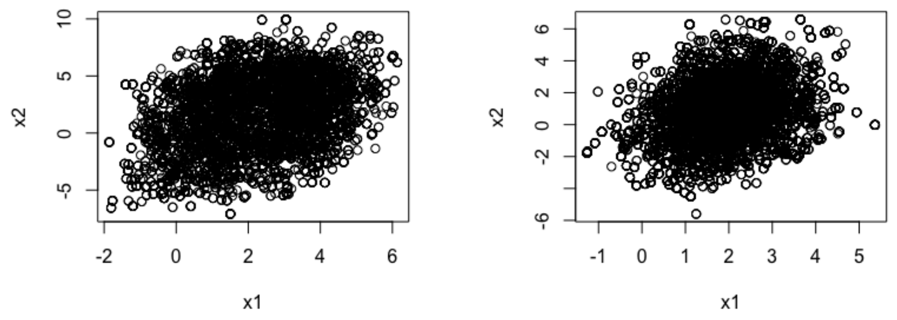
iterations	α	β	$\hat{\mathbf{\Sigma}}_{\mathbf{1,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,2}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$
100 000	0.0	2.0	0.85 (0.01)	0.41 (0.02)	0.41 (0.02)	3.46 (0.12)
100 000	0.0	5.0	0.93 (0.06)	0.45 (0.03)	0.45 (0.03)	3.64 (0.07)
200 000	0.0	10.0	1.01 (0.01)	0.49 (0.02)	0.49 (0.02)	4.08 (0.04)
400 000	0.0	20.0	1.00 (0.01)	0.51 (0.03)	0.51 (0.03)	4.03 (0.05)
400 000	0.0	30.0	1.00 (0.00)	0.49 (0.01)	0.49 (0.01)	3.98 (0.07)

for $\alpha = 10$ and $\alpha = 20$ have similar structures, since they are likely to stop growing in the same situations. To illustrate this, imagine that a node n in a growing DAG $\mathcal{G}_r^*$ at level two has three nodes with associated to densities greater than $p(x_n)$ associated with n. This sets the probability in (33), for the node n having children equal to

$$P(c(n)|\mathcal{G}_r^*, \mathcal{X}_r^*) = \exp\{-10 \cdot 3 - 0.2\}, \tag{37}$$

of having children with $\alpha = 10, \beta = 0.2$, and $\exp\{-20 \cdot 3 - 0.2\}$ if $\alpha = 20$, which both essentially are equal to zero and the DAG will stop growing in both cases. That the DAGs stops growing in similar situations will make structures of the generated DAGs similar, and this could explain why the limiting distributions seem to stabilize for large values of α.

Table 2 shows the variance when α is fixed and β varies. Here the limiting distributions converge closer to the target distribution than the simulations in Table 1, but it is difficult to say whether the resemblance is random, or if the algorithm is closer to correct when the estimates resembles the target distribution. An argument for the convergence being a coincidence is that the estimates go from being to low when $\beta = 0.2$ and too high when $\beta = 0.5$, which might indicate that there exists a parameter, $0.2 < \beta < 0.5$, which makes the algorithm converge. These values for β are not extreme values which make the acceptance probabilities structurally different than for other small choices of β, so it is difficult to argue that the convergence is not coincidental, but a special case in which the algorithm is correct. Another special case is when β is greater than ten,



(a) Plot of 20 000 iterations of the algorithm run for $\alpha = 20.0$ and $\beta = 0.2$. The plot shows that the limiting distribution of this simulation has a large variance compared to the target distribution in (36).

(b) Plot of 20 000 iterations of the algorithm run for $\alpha = 0.0$ and $\beta = 30.0$. This simulation shows simulated values with a variance equal to the one of the target distribution.

Figure 10: Two plots from different simulations for different pairs of α and β. Both simulations have limiting distributions converging to the same mean, located at $x_1 = 1.0$ and $x_2 = 2.0$, but the variances are different. This is seen by the spread of the two plots. The plot in (a) shows a distribution of higher variance than the target distribution. By the shape of the two plots, it seems like the values are sampled from Gaussian distributions.

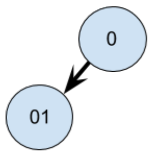
which seem to make the estimated covariance matrix very similar to the covariance matrix of the target distribution. This convergence seems less random, because these choices of β will lead to a specific set of possible DAGs being generated. Extreme values for β will dominate the probability function in (33), making the DAG stop growing at level one in almost all iterations.

The estimates in Table 3 show the estimated covariance matrix when α is fixed and equal to zero, and β varies in a similar manner as in Table 2. Note that when $\alpha = 0$, the probability function in (33) becomes independent of the density of the proposed values associated with the DAG. Although making the growth of the DAG independent of the densities discards the motivation for creating this algorithm, it might give an insight to where the error in the M–H setup is located. That the estimated covariance matrices are wrong, also when the growth is independent of the densities of the proposed values, indicate that the error is related to how we calculate the probability for growing the *structure* of a DAG. This can be seen in the expression for the acceptance probability in (28). When the growth of the DAG is independent of the density of the proposed values, and the proposal distribution $q(x_n|x_{pred(n)})$ is symmetric as in this experiment, the only factors that do not cancel in the joint distributions for the different DAGs included in the acceptance probability in (28), is the density of the target distribution associated with the potential root, and the stochastic growth of the DAG. The stochastic growth of the DAGs is the only difference between the acceptance probabilities from Section 4.3 and the acceptance probabilities used Luo and Tjelmeland (2018). Since the algorithm is wrong also when the growth is independent of the proposed values, the error is likely to be related to some structural properties of a DAG. This gives a strong indication that there is something about the stochastic process of growing a DAG that is not accounted for in the derivations presented in this work.

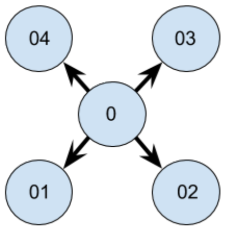
To investigate further the simulations which shows correct convergence properties, we look at the probability function for the parameters used in these simulations. When $\alpha = 0$ and $\beta = 10$, the probability function in (33) for every node n in the DAG, except for the root node, is constant and equal to

$$P(c(n)|\mathcal{G}_r^*, \mathcal{X}_r^*) = \exp\{-10\} = 4.53 \cdot 10^{-5}. \tag{38}$$

The expression in (38) is essentially zero, and is even lower when β is greater than ten. This means that there in practice only will be generated the two types of DAGs shown in Figure 11 in almost



(a) The DAG which will be grown in $\frac{1}{4}$ of the iterations



(b) The DAG which will be grown in $\frac{3}{4}$ of the iterations

Figure 11: Illustration of the two DAGs which will be generated in the simulations with the parameter β larger than ten. This will make the probability of having children for the nodes at level 1 close to zero.

Table 4: Convergence properties for one simulation with $\beta = 30$ and $\alpha = 0$, with a new target distribution. This simulation verifies that the estimated covariance matrix is correct for simulations with large values for β.

$\hat{\boldsymbol{\mu}}_1^{new}$	$\hat{\boldsymbol{\mu}}_2^{new}$	$\hat{\boldsymbol{\Sigma}}_{1,1}^{new}$	$\hat{\boldsymbol{\Sigma}}_{1,2}^{new}$	$\hat{\boldsymbol{\Sigma}}_{2,1}^{new}$	$\hat{\boldsymbol{\Sigma}}_{2,2}^{new}$
8.00	5.01	0.70	0.47	0.47	12.73

all iterations of the algorithm. These two variations are the result from the root node sampling its number of children, which is either one with a probability of 0.25 or four with a probability 0.75. These two structures will dominate the DAGs generated throughout all simulations with high values for β. In this special case, the state of the Markov chain is essentially only updated when the DAG in Figure 11 (a) is grown. This is because changing the root node for the DAG in Figure 11 (b) changes the graph to a DAG with three levels, which has a probability close to zero of being grown, and the result is that the old root will be re-sampled in these iterations.

As a verification of the convergence properties for simulations with β larger than ten, we run a new simulation using a new bivariate Gaussian distribution with mean vector

$$\boldsymbol{\mu}^{new} = \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

and a covariance matrix

$$\boldsymbol{\Sigma}^{new} = \begin{bmatrix} 0.7 & 0.5 \\ 0.5 & 13.0 \end{bmatrix}$$

as the target distribution. The simulation was run one time for $\alpha = 0$ and $\beta = 30$ for 400000 iterations. The result is shown in Table 4. Looking at the estimated values, it seems like the algorithm has the correct convergence properties. to the desired target distribution also for this choice of target distribution, and we conclude that the algorithm converges to the correct target distribution when β is greater than ten.

The tuning parameter used in $q(x_n|x_{pred(n)})$ is also a parameter which might influence the structures of the DAGs generated. A small tuning parameter will make less of the proposed values located in the tail, since "small jumps" is more likely to have high density if it conditions on a high density value, which often will be the case for the value associated with the root node. Tuning this parameter does not effect the acceptance probabilities explicitly, since all terms involving $q(x_n|x_{pred(n)})$ cancels in the acceptance probability when a symmetric distribution is used. To investigate how this tuning parameter effects the convergence, four pairs of α and β showing the most interesting results from the earlier simulations were chosen. The new tuning parameter is

Table 5: The estimated covariance matrix for simulations with the tuning parameter in $q(x_n|x_{pred(n)})$ set to 0.3.

iterations	α	β	$\hat{\mathbf{\Sigma}}_{\mathbf{1,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,2}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$
100 000	0.5	0.2	1.28	0.76	0.76	5.78
400 000	0.0	30.0	1.04 (0.07)	0.51 (0.03)	0.51 (0.03)	4.09 (0.71)
200 000	10.0	0.2	5.57	5.87	5.87	38.51
200 000	20.0	0.2	5.45	5.95	5.95	39.47

set to 0.3. One separate run was done for three of the pairs, but the simulation for $\beta = 30$ was repeated five times to get a more accurate impression to reassure that tuning this parameter does not effect the simulations which seem to converge. The results for the five different pairs are shown in Table 5. It is clear from the results in Table 5 that the tuning parameter involved in the proposal distribution $q(x_n|x_{pred(n)})$ effects the results for the simulations with small values for β, since the estimated convergence probabilities differ significantly from the convergence properties in Table 1. Even if the tuning parameter does not effect the M–H acceptance probabilities directly, it is likely to effect the structures of the DAGs being generated throughout the simulations, and thereby also effect the acceptance probabilities. The trend in Table 5 is that the estimated covariance matrices are even larger than the previous simulations with the same values for α and β, except for the simulation with $\beta = 30$ which still shows the correct convergence properties. That the variance in general has increased in most of the simulations could be because the size of the generated DAGs have increased. The simulation with α greater or equal to ten, has an estimate for $\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$ which is four times as large as in the previous simulations. The DAGs grown with large values for α should result in small DAGs, and that this parameter effects the estimated covariance this much is a bit surprising. To investigate what happens for simulations with large values for α, we do two more runs of 100 000 iterations to estimate the expected number of nodes for a DAG grown with these parameters. The estimate for the simulation with $\alpha = 20.0, \beta = 0.2$, is 20.4 nodes, with the tuning parameter equal to 0.3. Running the simulation with the same set of values for α and β, but with the tuning parameter set to 1.0, gives an estimate of expected nodes equal to 12.11. By these estimates it is clear that this parameter effects the growth of the DAGs for small values of β.

7.2 Comparing the algorithms ability to propose high density values

Recall that the motivation for a new version of the algorithm presented in Luo and Tjelmeland (2018) was to increase the number of high density values proposed within a DAG. In this section we compare the mean density of the values proposed within a DAG for the different methods for growing the DAG from Section 5. We also compare our algorithm to the algorithm presented in Luo and Tjelmeland (2018).

To get an accurate impression of how the different methods perform, it is important that the algorithms compared have roughly the same number of nodes in the DAGs that are used in the simulations. This is because as a DAG grows larger, the values proposed are located further and further away from the root node which has the value distributed according to the target distribution associated with it. Increasing the size of the DAGs is therefore likely to decrease the mean density of the proposed values within each DAG. We try to make the size of the DAGs in each method compared as equal as possible, measured by its expected number of nodes, given the parameters used for growing the DAG. The fixed graph used in each simulation of the algorithm from Luo and Tjelmeland (2018) are initially grown stochastically based on these estimates, and we choose a structure which has approximately the same number of nodes as the expected number of nodes for simulation we wish to compare it to. The graph structure is then kept fixed as described in Section 3.

the Tables 6 and 7 show the expected mean of the logarithm of the densities proposed within one iteration for different methods. This is estimated by calculating the mean of the logarithm of the values proposed within one iteration, and then calculate the mean of these estimates, based

Table 6: The expected mean of the logarithm of the densities proposed within one iteration, together with the expected size of the DAGs measured by its number of nodes. The results are based on 300 iterations. Fixed indicates that the graph is fixed throughout the simulations, while stochastic indicates the method presented in this work.

α	β	fixed/stochastic	mean number of nodes	$E[\overline{\log(p(x_n))}]$
0.0	30.0	stochastic	4.24 (0.07)	-3.98 (0.07)
0.0	30.0	fixed	5	-4.32 (0.09)
0.2	0.1	stochastic	61.88 (1.90)	-4.46 (0.06)
0.2	0.1	fixed	65	-7.07 (0.21)
0.5	0.2	stochastic	22.03 (0.84)	-4.28 (0.07)
0.5	0.2	fixed	20	-5.66 (0.13)
0.4	0.1	stochastic	32.61 (1.15)	-4.33 (0.07)
0.4	0.1	fixed	26	-5.65 (0.13)
30.0	0.2	stochastic	12.34 (0.58)	-5.63 (0.14)
30.0	0.2	fixed	14	-5.25 (0.13)

on all iterations. Table 6 shows the results for the simulations where the DAGs are grown by the method independent of levels from Section 5.2, and Table 7 shows the results for growing the DAG by Method 1 and Method 2 from Section 5.1. Standard deviations are shown in the paranthesis. Note that the "average number of nodes" for the simulations with fixed graphs is constant, because only one graph structure is used corresponding to the undirected acyclic graph presented in Section 3. The simulations marked as "fixed" are simulations of the algorithm from Luo and Tjelmeland (2018), and the initial graph is grown with the method indicated by the other columns of the same row, and kept fixed throughout the iterations. Note that the limiting distributions of the different methods could effect the results, so the estimates presented here are not completely reliable. Simulations with a variance that is very different from the one of the target distribution, will also have proposals of higher variance, which is related to the estimates shown in Table 6 and Table 7.

From the results in Table 6 it seems like the version of the algorithm presented in this article propose values of higher density than the algorithm from Luo and Tjelmeland (2018). When α is set to zero, there should not be any difference, because then the growth is independent of the density of the proposed values. Looking at the results for $\alpha = 0$ on the other hand, the difference seems significant. This could be explained by that the size of the simulations with stochastic graphs has many DAGs of only two nodes, as explained in Section 7.1 and shown in Figure 11, and that it is not accurate to compare the results to a simulation with a graph of five nodes. The results for this experiment are more interesting for larger DAGs, since this will make the proposals in the algorithm of Luo and Tjelmeland (2018) move towards the tail of the distribution, whilst the algorithm presented in this article should be able to stop expanding in direction of bad proposals. Looking at the results for simulations which have its expected number of nodes larger than 20, it seems like our algorithm performs better. The difference in Table 6 is largest for the graphs with average nodes greater than 60 and $\alpha = 0.2, \beta = 0.1$, which is expected since these are the largest DAGs, and we expect the difference to be more significant for large graphs.

When $\alpha = 30$ and $\beta = 0.2$ the densities proposed within the DAG have about the same density on average for both algorithms. In the previous simulation experiment from Section 7.1, we demonstrated that the simulation with $\alpha = 30$ has a very high variance compared to the other simulations, shown in Table 1, and this is likely to effect the result in Table 6. Comparing the simulation for $\alpha = 30$ to the other simulations with stochastic DAGs, we observe that the mean density is significantly lower, even when the expected size of the DAGs is larger in the other simulations. It is very likely that this is because of the high variance of the limiting distribution for these choices of parameters.

That the variances are different is an error in the M–H setup which makes the analysis of the densities of the proposed values for the different parameters unreliable. When the densities are dependent on the limiting distributions of the different simulations, and these limiting distributions

Table 7: The expected mean of the logarithm of the densities proposed within one iteration, together with the expected size of the DAGs measured by its number of nodes. The results are based on 100 000 iterations. Fixed indicates that the graph is fixed throughout the simulations, while stochastic indicates the method presented in this work.

Method 1/2,fixed/stochastic	mean number of nodes	$E[\overline{\log(p(x_n))}]$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{1,2}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,1}}$	$\hat{\mathbf{\Sigma}}_{\mathbf{2,2}}$
Method 1, stochastic	55.84 (2.47)	-4.53 (0.00)	1.25	0.65	0.65	5.39
Method 1, fixed	50	-6.76(0.01)	1.00	0.49	0.49	3.93
Method 2, stochastic	72.02 (3.05)	-4.56 (0.00)	2.06	0.83	0.83	6.68
Method 2, fixed	79	-7.07(0.01)	1.00	0.50	0.50	4.03

differ, we can not know that the results would be the same if the algorithm was correct and converged to the correct distribution. We still expect that the estimates for the simulations that converge close to the target distribution in (36) are reliable, and would be similar if the algorithm was correct, but the simulation for $\alpha = 30.0$ is a special case because of the high variance.

Table 7 shows the results from comparing the algorithm from Luo and Tjelmeland (2018) to the version of our algorithm which grows the DAGs by Method 1 and Method 2 from Section 5.1. In these two methods, the expected number of children for a node n is Poisson distributed, and the intensity parameter is a function of the levels in the DAG. Method 1 uses the function in (30) for the intensity parameter, and Method 2 uses (31). To avoid zero probabilities, we have added a small adjustment to Method 1, so that the intensity parameter is set to 0.5 when the expression in (30) becomes zero, corresponding to the DAG having more than three levels. We do one run of 100 000 iterations for each method, to get an estimate of the variance of the limiting distributions of the different simulations, since this might effect the results, as discussed above. The results are shown in Table 7.

In general, it seems like the density of the values proposed within the algorithm from Luo and Tjelmeland (2018) are significantly lower, and that the difference is larger for the simulations in Table 7 than in Table 6, which is because the DAGs in the simulations from Table 7 are larger. If we compare the methods based on levels from Section 5.1 to the method independent of levels from Section 5.2, we see that the difference is not significant. The average number of nodes in the DAGs for the the simulation with $\alpha = 0.2$ and $\beta = 0.1$ is 61.88, which is about the same as for Method 1 and Method 2 when the DAGs are grown stochastically. The estimates for the mean of the logarithm of the densities fall within the interval $[-4.56, -4.48]$, which is quite small. In general it seems like the estimates for the density of the proposed values in our algorithm is quite robust when the number of nodes in the DAGs increase. Even when the DAGs grow large, the mean density of the proposed values does not decrease as much as in the algorithm from Luo and Tjelmeland (2018). But, since the algorithm is not correct, it is difficult to draw conclusions.

7.3 The algorithm's ability to jump between levels in the DAG.

To make use of the multiple proposals generated within the DAG, it is important that the algorithm has the ability to jump between levels when sampling a new root node, corresponding to an iteration of the algorithm. In this section the different methods for growing the DAG from Section 5 will be tested on this ability, and the results are compared to the algorithm from Luo and Tjelmeland (2018).

To compare our algorithm to the algorithm in Luo and Tjelmeland (2018), we use the same strategy as in the previous section. We initially grow the DAGs by the methods we wish to compare them to, and keep them fixed throughout the simulations. The simulations are based on 300 iterations, and are shown in Table 8 and Table 9 with standard deviations shown in the paranthesis. By "mean levels jumped" we mean the expected number of levels the algorithm jumps in one iteration of the algorithm. The column marked "still" has the estimated proportion of iterations where the algorithm does not move, but re-samples its old root.

From the results it is clear that the algorithm's ability to jump between levels in the DAG

Table 8: The expected number of levels jumped in each iteration, together with the proportion of iterations where the old root was resampled. The DAGs are grown by the method from Section 5.2. The estimates are based on 300 iterations

α	β	fixed/stochastic	mean number of nodes	mean levels jumped	still (%)
0.0	30.0	stochastic	4.26 (0.07)	0.10 (0.02)	89.74%
0.0	30.0	fixed	5	1.02 (0.05)	31.56%
0.2	0.1	stochastic	59.49 (1.80)	0.90 (0.08)	50.75%
0.2	0.1	fixed	53	4.10 (0.12)	6.64%
0.5	0.2	stochastic	21.62 (0.84)	0.88 (0.07)	44.69%
0.5	0.2	fixed	26	2.96 (0.10)	11.96%
0.4	0.1	stochastic	32.61 (1.15)	0.86 (0.05)	49.52%
0.4	0.1	fixed	32	3.39 (0.12)	10.96%
30.0	0.2	stochastic	12.22 (0.56)	0.23 (0.03)	77.77%
30.0	0.2	fixed	17	2.42 (0.09)	12.62%

Table 9: The expected number of levels jumped in each iteration, together with the proportion of iterations where the old root was resampled. The DAGs are grown by the method from Section 5.1. The estimates are based on 300 iterations .

Method	stochastic/fixed graph	average number of nodes	mean levels jumped	still (%)
1	stochastic DAG	58.08 (2.50)	0.30 (0.03)	80.40%
1	fixed graph	48	4.02 (0.15)	7.30%
2	stochastic DAG	78.10(3.70)	0.86 (0.08)	61.11%
2	fixed graph	71	4.41 (0.12)	5.64%

is weak for the setup described in this work. In Table 8, one of best results for growing the DAG stochastically is when $\alpha = 0.5$ and $\beta = 0.2$, and the algorithm jumps 0.88 levels on average, which means that the algorithm is expected to move a bit below one level for every iteration. Since the DAGs generated with these parameters has an expected size of 21.62 nodes, most of these values have approximately zero probability of being sampled as the new root. The proportion of iterations where the original root node is resampled as the new root is 44.69% , which is the best result for the methods growing the DAG stochastically in Table 8. This means that in 44.69% of the iterations, approximately 21 proposals are rejected. This makes the generation of these values useless, and it seems like the algorithm performance does not compensate for the computational cost it takes to generate the multiple proposals, when growing the DAGs by this strategy.

Table 9 shows simulations for our algorithm and the algorithm from Luo and Tjelmeland (2018) when the growth of the DAGs are level based from Section 5.1. Looking at the estimates, the level based methods performance is weak compared to the method independent on levels from Section 5.2. Looking at the DAGs of approxiamtely equal size, the method for $\alpha = 0.2$ and $\beta = 0.1$ resamples its old root in 59.75% of the iterations, compared to Method 1 and 2 which resamples its old roots in respectively 80 and 61.11% of the iterations. The worst performance for growing the DAGs stochastically is for Method 1. Method 2 and the method independent of levels' performance are equal.

The algorithm from Luo and Tjelmeland (2018) jumps easier between levels, with its lowest result for estimated proportion iterations it resamples its old root equal to 5.64%, which is a lot better than the estimated proportions for our algorithm. Growing the DAGs stochastically will always add an extra factor to the probability of resampling the old root, compared to the algorithm in Luo and Tjelmeland, and this is a weakness of this setup. In most cases, the DAG that grows is the DAG which has the highest probability of being grown, which is a consequence of the definition of probability. The best we can do is to reduce this factor, so that the density of the proposed values dominates the acceptance probabilities to a larger extent. When growing the DAGs by the methods presented in Section 5, neither of the strategies perform well enough to compensate for the computational cost of generating multiple proposals.

As discussed earlier, we can not completely rely on the evaluations of the algorithm's performance as long as the acceptance probabilities are wrong, which also makes the estimates related to the acceptance probabilities incorrect. The algorithm's ability to sample a new root node is directly related to the acceptance probabilities from Section 4.3, since this is the probability for sampling the new root. The results discussed in this section is therefore not completely reliable, but is likely to give an indication of how well the methods for growing the DAGs work in practice. What is clear is that finding a good strategy for growing the DAGs is a big challenge in this setup, and is essential for making this algorithm useful.

8 Closing remarks

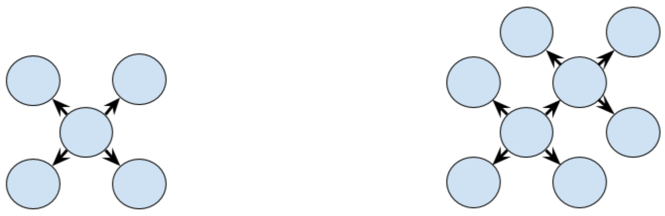
The algorithm discussed in this work is based on the algorithm presented in Luo and Tjelmeland (2018), and is meant to be an improvement of this algorithm in the sense that it tailors the number of proposals generated, to reduce the number of low density values being proposed. Although the derivation of the M–H acceptance probabilities presented in Section 4.3 intuitively seem equivalent to the acceptance probabilities used in Luo and Tjelmeland (2018), the algorithm does not converge to the given target distribution as demonstrated in Section 7.1.

The simulation experiments gave an indication of where in the M–H setup the error was located. Naturally, the error is related to growing the DAGs stochastically, since this is what separates our algorithm from the one presented in Luo and Tjelmeland (2018). That the algorithm does not converge, even when the growth of the DAGs are independent of the densities proposed, indicates that the error is related to the probability of growing a specific structure of a DAG, and that the probability of proposing the values associated within the DAG is independent of this error. The experiment in Section 7.1 showed a special case in which the simulations seemed to have the correct convergence properties. This simulation was based on the method from Section 5.2 which grows the DAG independent of levels, and simulated with extreme values for the parameter β. In this simulation, the DAGs being generated have one of two shapes, shown in Figure 11 in essentially all iterations.

Since the simulation leading to the small set of possible DAGs shown in Figure 11 seem to converge, it could be that the acceptance probabilities are correct in this special case, and that the two DAGs are examples of "legal" structures which makes the algorithm correct. What is special about the two structures in Figure 11 is that they are symmetric, so rotating them would not change how the DAGs appear in the two dimensional illustrations. To understand how we grow a DAG, it is important to keep in mind what we have defined a DAG to be. In Section 4 we defined a DAG $\mathcal{G}_r$ to be a set of nodes $\mathcal{S}_r$ and a set of directed edges $\mathcal{E}_r$, where r is the root node of the DAG which determines the direction of the edges. We also defined a set $\mathcal{X}_r$ which is the set of all proposed values generated through the directed edges of the DAG. What we did not define, was the order of the children for a specific node in the DAG. We have in the implementation assigned ID's for the nodes corresponding to which order the child got its proposed value through (14). However, this order was assigned coincidentally in the implementation, and is not included in the joint distribution from (22). The children of a node are independent of each other, they only depend on their predecessor.

We have associated the set $\mathcal{X}_r$ to a DAG, but how every node is associated to a specific value is not clearly defined in our M–H setup. If a node n has three children, three values will be proposed, conditioned on the value "associated" with n. These three values are assigned to the node n's children, but which child gets what value is not accounted for in our setup. The relationship between the set $\mathcal{X}_r$ and the DAG is that the values are proposed in the direction of the edges of the DAG, but the relationship between one node and one value in $\mathcal{X}_r$ is not clear. We therefore try and look at the special case, when the DAG is grown independently of the proposed values, to get a more defined situation.

The process of growing a DAG independent of the densities, and generating the associated proposals, can be regarded as doing the process in two separate steps. Step one is to grow the full DAG, and step two is to propose values in direction of the directed edges. We set the probability for the root node having four children equal to 0.75, and the probability of having one child equal to 0.25. The probability for all other nodes having three children is equal to 0.2, and the


(a) The root node samples its number of children with a probability of 0.75

(b) One of four nodes samples children. The probability of growing this DAG is estimated to be $0.75 \cdot 0.8^3 \cdot 0.2$. in our setup

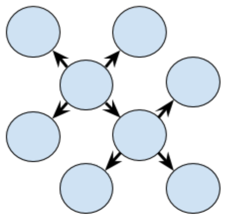

(c) A DAG isomorphic to the DAG in (b).

Figure 12: The process of growing a DAG according to the procedure presented in this work. In the first step, the root node samples its number of children. In the next step, only one of the nodes at level one samples children, leading to the DAG in (b). Isomorphic structures could have been grown in three other ways, corresponding to one of four nodes having children. The DAG in (c) is isomorphic to the DAG in (b).

probability of not having children is consequently 0.8. As an example, imagine that we grow a DAG with these probabilities, and that the process is shown in Figure 12. We try to calculate the probability for the process of growing the DAG in Figure 12 (b) in the same manner as we have calculated the probabilities for growing the DAGs in Section 4.3, by regarding every node sampling its number of children as separate and independent events.

The first node to sample its children, is the root node. The root node gets four children with a probability of 0.75. Then the four children of the root node samples its number of children. One node gets three children, with a probability of 0.2, and the other nodes get zero children, with a probability of 0.8. Calculating the probability for the DAG getting this structure, according to our setup, becomes

$$0.75 \cdot 0.2 \cdot 0.8 \cdot 0.8 \cdot 0.8 = 0.75 \cdot 0.8^3 \cdot 0.2. \tag{39}$$

Recall that what define a DAG by the direction of the edges, and the nodes connected to these edges. The question is whether a different process could lead to the same DAG being generated, or could a different DAG lead to the same order of proposed values? According to our definition of DAG, which not defines the order of the children, there is nothing separating the DAG in Figure 12 (b) and the DAG in 12 (c). Both DAGs lead to the same joint distribution of the proposed values to be proposed in step two of this example. The probability of growing a DAG which leads to proposing the set of associated values in the same order as the DAG in Figure 12 (b) is by these arguments equal to

$$\binom{4}{1} 0.75 \cdot 0.8^3 \cdot 0.2, \tag{40}$$

since one of four nodes having children can happen in four different ways, and they all lead to the same structure of the DAG. This probability is four times the probability we would assign the same DAG in our M–H setup. Since we have not defined the order of the children of a node in a DAG, there is nothing in our definition separating the structure in Figure 12 (b) from the DAG in Figure 12 (c). The two DAGs are said to be isomorphic (Tolley et al. 1995), meaning that there is nothing in our definition of a DAG which separates these two objects. The two objects are structurally identical.

By the arguments presented above, the error in the M–H setup is that several processes for growing a DAG can lead to the same joint density of the proposed values proposed within the DAG. In other words, the same structure can be grown in multiple ways. That this is the error in the M–H setup is difficult to verify, because it includes calculating all possible ways a specific structure can be grown. For the DAG in Figure 12 (b) it is easy to see that there are four different processes for growing this structure, but as the DAGs grows larger, it is difficult to count how many ways a specific structure could have been grown.

To get a correct algorithm according to our hypothesis, the number of ways to grow a DAG isomorphic to the structure that was grown should be multiplied to the expression for joint distribution in (22). Algorithms related to finding isomorphisms are known to be computationally expensive (Tolley et al. 1995). The problem of identifying whether two graphs are isomorphic is known as the "Graph isomorphism problem", which has no known solution in polynomial time (Tolley et al. 1995). To verify if our hypothesis is correct, it is possible to test if an adjusted version of the algorithm converges for small DAGs, where the number of isomorphic DAGs is possible to calculate. If the hypothesis is correct, and the adjusting factor would lead to correct acceptance probabilities, one could try to limit the possible structures to be grown, or find a way to make each DAG unique through defining the order of a node's children and include this in the acceptance probabilities. If the DAGs are grown in the same way as discussed in this work, the algorithm would be too computationally expensive, since counting the set of isomorphic DAGs that can be grown involves verifying if two sub DAGs are isomorphic, which is equivalent to the isomorphism problem (Tolley et al. 1995).

The algorithm fails to converge to the target distribution in the general case, and there are many challenges related to the setup presented in this work. In Section 7 we demonstrated that even if the algorithm succeeds in proposing high density values, it fails to make use of these values, because its ability to jump between levels is too poor. The algorithm resamples its old root in most of the iterations, and when it jumps it is often to its neighbours, making the other proposed values useless. A better strategy for growing a DAG is likely to exist, but growing the DAG stochastically will always add a probability for the root node being resampled to the benefit of the other nodes in most cases, as discussed in Section 7. The algorithm presented in this work is far from complete, and to make a version that converges to the correct target distribution and is able to jump easily between levels, is a demanding task. The algorithm might still have potential, since it succeeds in proposing multiple high density values.

References

Cormen, H., Leiserson, E., Rivest, Stein. (2009) *Introduction to Algorithms,* Third edition. *MIT Press.* 22, 594-602

Gamerman, D., Lopes, H. (2006) *Markov Chain Monte Carlo–Stochastic Simulation for Bayesian inference,* Second edition. *Chapman & Hall/CRC.*

Geman, S. and Geman, D. (1984). “Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images.” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 6, 721–741.

Hasting, W. K (1070). Hastings, W. K. (1970). “Monte Carlo simulation methods using Markov chains and their applications.” *Biometrika,* 57, 97–109.

Liu, J. S., Liang, F. M., and Wong, W. H. (2000). “The multiple-try method and local optimization in Metropolis sampling.” *Journal of American Statistical Association*, 95, 121–134.

Luo, X., Tjelmeland, H. (2018). ”A multiple-try Metropolis-Hastings algorithm with tailored proposals.” Tech. rep., ArXiv e-prints 1807.01914v1, Available from http://arxiv-export-lb.library.cornell.edu/abs/1807.01914

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). “Equation of state calculations by fast computing machines.” *Journal of Chemical Physics,* 21, 1087–1092.

Savitch W. (2013), *Absolute C++*, Fifth edition. *Pearson* 448-699.

Tolley, R., Franceschini W., Petty D. (1995) ”Graph Isomorphism Algorithms: Investigation Of The Graph Isomorphism Problem.” *Institute for Simulation and Training*, Paper 106, Available from http://stars.library.ucf.edu/istlibrary/106