Approximate recursive calculations of discrete Markov random fields

Petter Arnesen

Master of Science in Physics and Mathematics
Submission date: June 2010
Supervisor: Håkon Tjelmeland, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

The candidate should develop and implement an approximate recursive algorithm for simulating realisations from discrete Markov random fields. He should also evaluate the effectiveness and accuracy of the algorithm for common discrete Markov random fields.

Assignment given: 22. January 2010
Supervisor: Håkon Tjelmeland, MATH

Preface

This thesis completes my Master of Science degree at NTNU. First of all I would like to thank Håkon Tjelmeland for his excellent supervision. His advises and feedbacks have been very much appreciated.

I would also like to than my fiancée Hanne Aune Ulen for much appreciated support. Finally I would like to thank all my fellow classmates for five great years, and especially Daniel Høyer Iversen whom I have worked close with on many occasion.

Petter Arnesen, 14.06.2010, Trondheim.

Abstract

In this thesis we present an approximate recursive algorithm for calculations of discrete Markov random fields defined on graphs. We write the probability distribution of a Markov random field as a function of interaction parameters, a representation well suited for approximations. The algorithm we establish is a forward-backward algorithm, where the forward part recursively decomposes the probability distribution into a product of conditional distributions. Next we establish two different backward parts to our algorithm. In the first one we are able to simulate from the probability distribution, using the decomposed system. The second one enables us to calculate the marginal distributions for all the nodes in the Markov random field. All the approximations in our algorithm are controlled by a positive parameter, and when this parameter is equal to 0, our algorithm is by definition an exact algorithm. We investigate the performance of our algorithm by the CPU time, and by evaluating the quality of the approximations in various ways. As an example of the usage of our algorithm, we estimate an unknown picture from a degenerated version, using the marginal posterior mode estimate. This is a classical Bayesian problem.

Contents

1 Introduction

In this thesis we present an approximate algorithm for calculations of discrete Markov random fields defined on graphs. Markov random fields is a well investigated topic in the literature, see for instance Besag (1974) or Kindermann & Snell (1980). However, exact calculations of such fields are very limited by an intractable normalizing constant. Reeves & Pettitt (2004) and Friel & Rue (2007) manages to do exact calculations of this normalizing constant for binary Markov random fields defined on smaller lattices. Their algorithm is a forward-backward algorithm (Scott 2002). To do calculations of larger binary Markov random fields defined on graphs, Tjelmeland & Austad (2010) introduce an approximate recursive forward-backward algorithm. They express the probability distribution of the field in terms of interaction parameters between subsets of the nodes in the graph, which is a convenient representation for approximations. This thesis can be viewed as a generalization of their work, as we let our Markov random fields be discrete, and it is a continuing of the work done in Arnesen (2009).

As in Tjelmeland & Austad (2010), we present a recursive forward-backward algorithm. This algorithm enables us to do exact calculations of Markov random fields defined on small graphs and with a small discrete sample space. For more complex problems we introduce approximations to our algorithm. The approximations are defined by approximating sufficiently small interaction parameters to 0. The level of approximation is controlled by a parameter, and by setting this parameter equal to 0 we get the exact version of our algorithm. In the forward part of our algorithm we sequentially decompose the probability distribution into a product of conditional distributions, one for each node in the graph. For each conditional distribution we obtain, one distribution for the remaining nodes must also be determined. Note that this representation is in fact a partially ordered Markov model (Cressie & Davidson 1998). We present two alternative backward parts to the algorithm. The first one enables us to simulate from the decomposed system, giving us realisations from the approximate probability distribution. This can be done without further approximations. As an alternative, we are able to calculate the marginal distributions for all the nodes in the graph, but for complex problems we need to introduce approximations also in these calculations. We also present two examples of Markov random fields, namely the Potts model and a generalized version of the Potts model, see Wu (1982). As an example of the usage of our algorithm we estimate an unknown underlying picture in a Bayesian setting, adopting a discrete Markov random field as our prior model. This is a classical problem in the literature and many solutions have been proposed. See for instance the simulated annealing procedure in Geman & Geman (1984) or the ICM estimate in Besag (1986). As it is easily accessible to us, we use the marginal

$$\begin{tikzcd}
1 \arrow[r, dash] \arrow[d, dash] & 2 \arrow[r, dash] \arrow[d, dash] & 3 \arrow[r, dash] \arrow[d, dash] & 4 \arrow[r, dash] \arrow[d, dash] & 5 \arrow[d, dash] \\
6 \arrow[r, dash] \arrow[d, dash] & 7 \arrow[r, dash] \arrow[d, dash] & 8 \arrow[r, dash] \arrow[d, dash] & 9 \arrow[r, dash] \arrow[d, dash] & 10 \arrow[d, dash] \\
11 \arrow[r, dash] \arrow[d, dash] & 12 \arrow[r, dash] \arrow[d, dash] & 13 \arrow[r, dash] \arrow[d, dash] & 14 \arrow[r, dash] \arrow[d, dash] & 15 \arrow[d, dash] \\
16 \arrow[r, dash] & 17 \arrow[r, dash] & 18 \arrow[r, dash] & 19 \arrow[r, dash] & 20
\end{tikzcd}$$

Figure 1: A 4×5 lattice with a first order neighbourhood system.

posterior mode estimate in our examples.

We start out by giving a short introduction to Markov random fields defined on graphs, see Section 2. Here we present important concepts as neighbourhood systems and cliques, formulate the famous Hammersley-Clifford theorem, and give two examples of discrete Markov random fields. In Section 3 we explain how to represent the probability distribution of a discrete Markov random field as a DAG (directed acyclic graph). We use this DAG representation when we present the exact version of our forward-backward algorithm in Section 4. In Section 5 we introduce our approximations and discuss how the quality of these approximations can be evaluated. In Section 6 we carefully explain how we chose to implement our approximate forward-backward algorithm, before we, in Section 7, present some results from running our algorithm. Finally, in Section 8, we give a closing discussion on the work done in this thesis and make suggestions for further work.

2 Markov random fields

In this section we give a brief introduction to Markov random fields. Markov random fields are well investigated in the literature, and for a more complete introduction the reader is referred to Besag (1974) and Kindermann & Snell (1980). All our Markov random fields will be defined on graphs, so in the first section of this introduction we define a graph and the important concept of neighbourhoods. In the second section we move on to define Markov random fields. Finally we present the most important theorem concerning this theory, namely the Hammersley-Clifford theorem.

2.1 Graphs and neighbourhoods

To define a graph we start out with a set of nodes S, and we label these nodes such that $S = \{1, 2, ..., n\}$. Next we define a neighbourhood system for the nodes in S.

Definition 1: A collection $N = \{N_1, ..., N_n\}$ is a neighbourhood system for the set S if, and only if, $N_k \subset S$, $k \notin N_k \; \forall \; k \in S$ and $k \in N_l \Leftrightarrow l \in N_k$ for all distinct pairs of nodes $k, l \in S$.

This definition tells us that for all nodes $k \in S$ we define a neighbourhood of other nodes N_k. These neighbourhoods are constructed such that if a node $k \in S$ is in the neighbourhood of a different node $l \in S$, then l must be in the neighbourhood of k as well. If so, we say that k and l are neighbours. This neighbourhood system creates connections between the nodes, and these connections form the set of edges E in the graph. We write $E = \{(k, l) | k \in N_l, k, l \in S\}$, and denote the graph by $G = (S, E)$. This completes our graph definition, and Figure 1 shows an example of such a graph. In this graph we have $S = \{1, 2, ..., 20\}$, and the edges are shown as lines between the nodes. This graph is nothing but a 4×5 lattice, and we will continue to use this figure to illustrate the rest of the definitions in this section.

For the lattice in Figure 1 we can easily find the neighbourhood of each node. The neighbourhood of for instance node 8 is $N_8 = \{3, 7, 9, 13\}$. We observe that all the interior nodes in this lattice have four neighbours each, namely the four nodes closest to them. This is called a first order neighbourhood system (Besag 1986). If we instead define the eight closest nodes as the neighbourhood of each interior node, we get what is called a second order neighbourhood system. Both a first and a second order neighbourhood of an interior node in a lattice are shown in Figure 2. For nodes on the boundary we need to be more specific about the neighbourhoods. One intuitive option is to define the neighbourhood of these nodes as in Figure 1. Here the four nodes on the corners of the lattice have two neighbours, for instance $N_5 = \{4, 10\}$, while the rest of the boundary nodes have three neighbours, for instance $N_{18} = \{13, 17, 19\}$. This is called a free boundary condition. An alternative to this is the torus boundary condition. In that case we define the boundary nodes to have neighbours on the opposite boundary, such that all the neighbourhoods are of the same size. For instance we get $N_{18} = \{3, 13, 17, 19\}$ and $N_5 = \{1, 4, 10, 20\}$ if we again assume a first order neighbourhood system to our example.

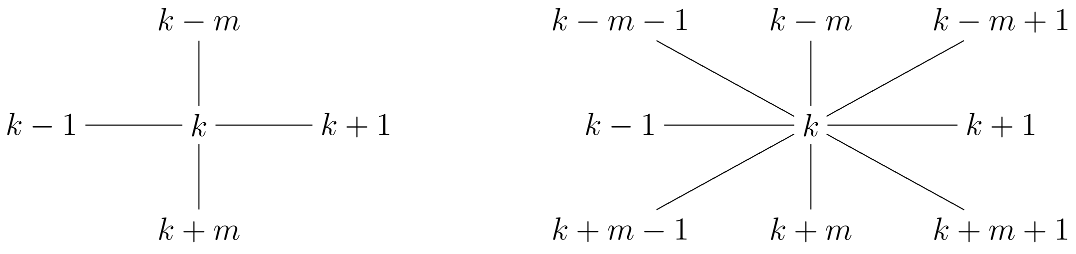


Figure 2: A first and a second order neighbourhood of an interior node k in a lattice. The neighbours are indicated with lines and m is the horizontal dimension of the lattice.

2.2 Markov random fields defined on graphs

In this section we define a Markov random field, and we start out by associating a discrete stochastic variable z_k to each $k \in S$. We assume that $z_k \in \Omega = \{0, 1, ..., K-1\} \ \forall \ k \in S$, which means that Ω is the discrete sample space for all z_k. Next we let these variables form the stochastic vector $z = (z_1, z_2, ..., z_n)$, and define a probability distribution $\pi(z) \ \forall \ z \in \Omega^n$. For notational convenience we introduce $z_{-k} = (z_i | i \in S \setminus \{k\})$ and $z_\Lambda = (z_i | i \in \Lambda)$, where $\Lambda \subseteq S$. We define a Markov random field as follows.

Definition 2: A random field z is a Markov random field with respect to a neighbourhood system N if, and only if, its distribution $\pi(z) > 0 \ \forall \ z \in \Omega^n$ and the full conditional $\pi(z_k | z_{-k})$ fulfils the Markov property

$$\pi(z_k | z_{-k}) = \pi(z_k | z_{N_k}) \ \forall \ k \in S, \tag{1}$$

where N_k is the neighbourhood of node k.

That is, a stochastic vector z is a Markov random field if the full conditional distribution for $z_k \ \forall \ k \in S$ only depends on the nodes in its neighbourhood. To complete this section we define a clique in the following way.

Definition 3: A set $\Lambda \subseteq S$ is a clique if $k \in N_l$ for all distinct pairs $k, l \in \Lambda$. Let $\mathcal{C}$ denote the set of all cliques.

This means that a subset of S is a clique if, and only if, every node in this subset is in the neighbourhood of all the other nodes in the subset. We notice that the empty set, $\emptyset$, and all subsets containing just one element, $|\Lambda| = 1$, are cliques according to this definition. If we again take look at Figure 1, we see that the set of all cliques in this situation is $\mathcal{C} = \{\emptyset, \{1\}, \{2\}, ..., \{20\}, \{1, 2\}, \{1, 6\}, \{2, 3\}, \{2, 7\}, ..., \{19, 20\}\}$. The cliques will be important when we present the Hammerley-Clifford theorem in Section 2.3, but first let us take a look at two examples of Markov random fields.

2.2.1 The Potts model

As an example of a discrete Markov random field we look at the Potts model (Wu 1982). The Potts model is a generalization of the Ising model used in the binary case. We assume our random field to be defined on a 2-dimensional lattice, and write the Potts model as

$$\pi(z) = c \cdot \exp\left\{ -\frac{\alpha}{2} \sum_{k=1}^{n} \sum_{l \in N_k} I(z_k \neq z_l) \right\}, \tag{2}$$

where c is a normalizing constant, $I(\cdot)$ is an indicator function, and α is a parameter. The model is called ferromagnetic if $\alpha>0$ and antiferromagnetic if $\alpha<0$. We will focus on the ferromagnetic model. This model favours equal values on sites close to each other, or clustering of the values in Ω. If we assume a Potts model to the lattice in Figure 1, we see that two neighbours $k, l \in S$ contribute with a value α to the sums in (2) if $z_k \neq z_l$.

An important feature of the Potts model is the critical temperature α_c. It is shown that for values of α below this critical temperature, $\alpha < \alpha_c$, the effect of the boundary nodes on the nodes in the centre of the lattice vanishes as the lattice grows. However, for values above this critical temperature, $\alpha > \alpha_c$, this effect does not vanish no matter how large the lattice is. See Kindermann & Snell (1980) or Hurn, Husby & Rue (2003) for a more detailed discussion concerning the critical temperature. The clustering of the values in Ω increases as the value of α gets close to and above this critical temperature. Wu (1982) gives a formula for calculation of the critical temperature for Potts models defined on lattices

$$\alpha_c = \ln(1 + \sqrt{K}), \tag{3}$$

where K again is the size of Ω. As a second example of a Markov random field we introduce our generalized Potts model.

2.2.2 The generalized Potts model

We would like to generalize the Potts model so that we can discriminate between the different values in Ω. This can be done by writing

$$\pi(z) = c \cdot \exp\left\{ -\frac{1}{2} \sum_{k=1}^{n} \sum_{l \in N_k} \alpha(z_k, z_l) \right\}, \tag{4}$$

where $\alpha(i,i) = 0$ and $\alpha(i,j) = \alpha(j,i) > 0$ for all $i, j \in \Omega$ and $i \neq j$. We easily see that this is a generalization of (2) by setting $\alpha(i,j) = \alpha$ for $i \neq j$. With this model we are able to discriminate between the different values in Ω. For instance we can make it unlikely that a value $i \in \Omega$ is in the neighbourhood of a different value $j \in \Omega \setminus \{i\}$ by choosing a high value on $\alpha(i,j)$. If we assume a generalized Potts model to the lattice in Figure 1, we see that two neighbours $k, l \in S$ contribute with a value $\alpha(z_k, z_l)$ to the sums in (4).

2.3 The Hammersley-Clifford theorem

The Hammersley-Clifford theorem from 1971 is an important theorem in the theory concerning Markov random fields. Here it is formulated as in Hurn et al. (2003).

Theorem 1 (Hammersley-Clifford): The stochastic vector $z \in \Omega^n$ is a Markov random field if, and only if, its probability distribution can be written as

$$\pi(z) = \frac{1}{C} \exp\left\{ - \sum_{\Lambda \in \mathcal{C}} U_\Lambda(z_\Lambda) \right\}, \tag{5}$$

where C is the normalizing constant

$$C = \sum_z \exp\left\{ - \sum_{\Lambda \in \mathcal{C}} U_\Lambda(z_\Lambda) \right\} < \infty.$$

This theorem gives a general form of the probability distribution $\pi(z)$ of a Markov random field z. It states that the probability distribution only depends on the set of energy functions $\{U_\Lambda(z_\Lambda) | \Lambda \in \mathcal{C}\}$. We will not prove this theorem, but the proof can be found in for instance Besag (1974) or Clifford (1990). From this theorem we also see that computation of the normalizing constant C becomes impossible when the number of nodes in the graph increases. This is one of the main problems concerning Markov random fields.

3 The DAG representation

Tjelmeland & Austad (2010) established a recursive forward-backward algorithm for simulation of binary Markov random fields. Our task in the next two sections is to generalize this algorithm so that we can simulate from discrete Markov random fields. In addition we explain how to use the algorithm to calculate all the marginal distributions in such a field. As in the previous section, we assume that $\Omega = \{0, 1, ..., K-1\}$ and that the stochastic field is defined by the vector $z = (z_1, z_2, ..., z_n)$ where $z_k \in \Omega \; \forall \; k \in S$. We start out by defining a general form of the probability distribution of a Markov random field. Next we define a

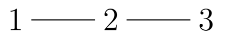


Figure 3: Illustrative graph with $\Omega = \{0, 1, 2\}$.

vertex-weighted DAG, from now on just termed DAG, to represent this distribution. In Section 4 we finally establish our exact forward-backward algorithm. In these two sections we use the graph in Figure 3 for illustration of the concepts that are presented, and we assume that $\Omega = \{0, 1, 2\}$ for this illustrative example.

3.1 The probability distribution

First we write the probability distribution of a Markov random field as

$$\pi(z) = c \cdot \exp\{-U(z)\}, \tag{6}$$

where c is a normalizing constant and $U(z)$ is an energy function. This energy function will be defined in a moment, but first we define the set

$$\mathcal{P}(S) = \{(\Lambda, u) | \Lambda \subseteq S, u \in \mathcal{Z}^{|\Lambda|}\}, \text{ where } \mathcal{Z} = \Omega \setminus \{0\} = \{1, ..., K-1\}.$$

This set contains all pairs (Λ, u), where Λ is a subset of the nodes in S and where u holds the values of these nodes. Note that the value 0 is never present in u. Using the definition of $\mathcal{P}(S)$ for the graph in Figure 3, we obtain the set $\mathcal{P}(S) = \{\emptyset, (\{1\}, (1)), (\{1\}, (2)), (\{2\}, (1)), ..., (\{1, 2, 3\}, (2, 2, 1)), (\{1, 2, 3\}, (2, 2, 2))\}$.

Next we define a one-to-one relation χ between $\mathcal{P}(S)$ and the sample space Ω^n by

$$z = \chi(\Lambda, u) \Leftrightarrow \begin{cases} \Lambda & = & \mathcal{L}(z) \\ u & = & \mathcal{U}(z), \end{cases} \tag{7}$$

where $\mathcal{L}(z)$ and $\mathcal{U}(z)$ are two functions that in the following will be explained. The function $\mathcal{L}(z)$ returns a vector Λ that contains all the indices of z where $z_k \neq 0 \; \forall \; k \in S$. The elements of Λ is sorted in increasing order, which means that if $\Lambda = \{k_1, k_2, ..., k_m\}$, we know that $k_1 < k_2 < ... < k_m$ where $m \leq n$. The function $\mathcal{U}(z)$ returns a vector u that contains the values of z where $z_k \neq 0 \; \forall \; k \in S$. If $\Lambda = \{k_1, k_2, ..., k_m\}$, we order u such that $u = (z_{k_1}, z_{k_2}, ..., z_{k_m})$. Note that when we write z_Λ, we use Λ as an operator that gives us the elements of z with indices from Λ, whereas the function $U(z)$ gives us the non-zero elements of z. The operator Λ and the function $\mathcal{U}(z)$ do therefore not represent the same property.

The energy function $U(z)$ is finally defined as

$$U(z) = \sum_{(\Lambda,u)\in\mathcal{P}(S)} \beta(\Lambda,u) I(u = z_\Lambda), \tag{8}$$

where $\beta(\Lambda,u)$ is an interaction parameter for the nodes $\Lambda \subseteq S$ when these nodes have values according to $u \in \mathcal{Z}^{|\Lambda|}$. The indicator function $I(\cdot)$ is equal to 1 if $(\chi(\Lambda,u))_k = z_k \ \forall \ k \in \Lambda$. Whenever an energy function is expressed this way, we say that it is on canonical form. Next we construct our DAG representation of the probability distribution presented in this section.

3.2 The probability distribution as a DAG

First of all we define the elements of $\mathcal{P}(S)$ to be the nodes in our DAG representation. Next we need to define an edge in this DAG. Let $a = (\Lambda,u)$ and $b = (A,v)$ be two elements from $\mathcal{P}(S)$. There exist an edge from node a to node b if $A \subset \Lambda$, $|\Lambda \setminus A| = 1$, and $(\chi(\Lambda,u))_A = v$. We say that a is a parent of b, and that b is a child of a. The vertex-weights in the DAG is the set of all interaction parameters $\beta[\mathcal{P}(S)] = \{\beta(\Lambda,u)|(\Lambda,u) \in \mathcal{P}(S)\}$, and where each weight $\beta(\Lambda,u)$ is stored in its corresponding node (Λ,u). Finally the DAG is denoted by $\mathcal{G}(\mathcal{P}(S),\beta[\mathcal{P}(S)])$. The DAG for our illustrative example is shown in Figure 4, but note that not all the nodes are displayed. However, the number of edges going in and out from each of the nodes is correct. In this figure we see how an element $(\Lambda,u) \in \mathcal{P}(S)$ has $|\Lambda|$ number of children, and that these children are $(\Lambda \setminus \{k\}, (\chi(u,\Lambda))_{\Lambda\setminus\{k\}}) \ \forall \ k \in \Lambda$. For instance, the children of the node ({1,2,3},(2,1,1)) are $(\{1,2\},(2,1))$, $(\{1,3\},(2,1))$, and $(\{2,3\},(1,1))$.

In this section we have seen how to represent the probability distribution of a Markov random field as a vertex-weighted DAG. We established a one-to-one relation between the sample space of the random field and the set $\mathcal{P}(S)$. The elements of $\mathcal{P}(S)$ were defined to be the nodes in our DAG representation. Using the set $\mathcal{P}(S)$, we defined an energy function containing a sum of interaction parameters, and these parameters were defined to be the weights in our DAG representation. In the next section we establish a recursive algorithm that enables us to efficiently calculate these parameters, given an energy function.

3.3 Calculation of the interaction parameters

So far we have established the weighted DAG representation of a Markov random field. However, given an energy function $U(z)$, we still need to calculate all the weights. In the binary case, Tjelmeland & Austad (2010) gave an efficient recursive formula for calculation of the interaction parameters. In this section we generalize

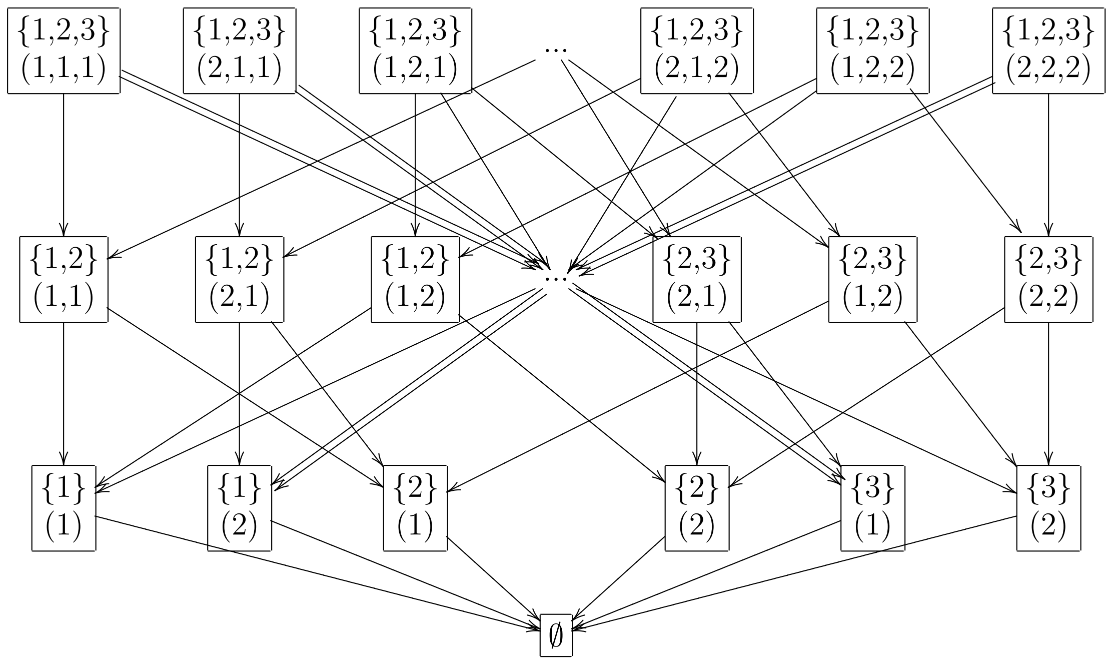


Figure 4: The DAG $\mathcal{G}(\mathcal{P}(S), \beta[\mathcal{P}(S)])$ for our illustrative example.

this formula to the discrete case. Note that it is not obvious at this point that (8) is a unique representation of a given $U(z)\ \forall\ z \in \Omega^n$. It will, however, become clear through this section that this is in fact true. First we look at the number of interaction parameters that needs to be calculated. Next we present an intuitive, but naive way of calculating these parameters. Finally we present our recursive formula.

3.3.1 The interaction parameters given the energy function

For a discrete Markov random field we know that for every $k \in S$ we have $z_k \in \Omega$, where $|\Omega| = K$. However, for a node $(\Lambda, u) \in \mathcal{P}(S)$ in our DAG representation, every element in u takes its value from $\mathcal{Z}$, where $|\mathcal{Z}| = K - 1$. For every value of $|\Lambda|$ we count

$$\binom{|S|}{|\Lambda|} \cdot (K-1)^{|\Lambda|}$$

nodes in the DAG. If we sum this expression for every possible value of $|\Lambda|$, the total number of nodes in the DAG becomes

$$\sum_{|\Lambda|=0}^{|S|} \binom{|S|}{|\Lambda|} \cdot (K-1)^{|\Lambda|} = |\Omega|^n. \tag{9}$$

This represents the number of interaction parameters that we need to calculate from the energy function. Note that the number of unknown interaction parameters is equal to the sample space of z. Keeping this in mind, we move on to calculate the interaction parameters.

Assume the energy function $U(z)$ to be known. For a given element $(\Lambda, u) \in \mathcal{P}(S)$ we write

$$U(\chi(\Lambda, u)) = \beta(\Lambda, u) + \sum_{(A,v)\in\mathcal{P}(S)\setminus\{(\Lambda,v)\}} \beta(A, v)I(v = z_A),$$

where $\chi(\Lambda, u) = z \in \Omega^n$. In this expression we have separated the term involving the element (Λ, u) from the rest of the sum. In this way we obtain the following expression for the interaction parameter in node (Λ, u)

$$\beta(\Lambda, u) = U(\chi(\Lambda, u)) - \sum_{(A,v)\in\mathcal{P}(S)\setminus\{(\Lambda,u)\}} \beta(A, v)I(v = z_A). \quad (10)$$

As we can see from this expression, an interaction parameter for a node in the DAG is a function of the given energy function and every interaction parameters that can be found by a recursive search through the nodes children. This recursive search leads us every possible way down to the node $\emptyset$. We observe that by (10) we have actually obtained a formula for calculating all the interaction parameters in the DAG. Starting with the node $\emptyset$, we can calculate all the interaction parameters by first calculating all the parameters where $|\Lambda| = 1$, then calculate the parameters where $|\Lambda| = 2$, and so on.

The method presented above is an easy and intuitive way of calculating the interaction parameters. However, it is an inefficient way. When we calculate the interaction parameters where $|\Lambda| = g$, we need to search through all the nodes down to $\emptyset$, but the nodes from level $|\Lambda| = g - 2$ down to $\emptyset$ have already been searched through when the interaction parameters on level $|\Lambda| = g - 1$ were calculated. We want to take advantage of this, and that will be our focus in the next section.

3.3.2 The recursive formula

To obtain our recursive formula we start out by defining the parameter $\gamma_0(\emptyset) = \beta(\emptyset)$ in the node $\emptyset$. For the rest of the nodes in the DAG we define the parameters

$$\gamma_g(\Lambda, u) = \frac{1}{g} \sum_{t\in\Lambda} \gamma_g(\Lambda\setminus\{t\}, (\chi(\Lambda, u))_{\Lambda\setminus\{t\}}) \text{ for } g = 1, ..., |\Lambda|, (\Lambda, u) \in \mathcal{P}(S)\setminus\emptyset. \quad (11)$$

We see that such a parameter is calculated strictly from the nodes one level below in the DAG. It was shown in Arnesen (2009) that these parameters where nothing but

$$\gamma_g(\Lambda, u) = \sum_{t(1),...,t(g)\in\Lambda} \beta(\Lambda \setminus \{t(1),...,t(g)\}, (\chi(\Lambda, u))_{\Lambda\setminus\{t(1),...,t(g)\}}), \tag{12}$$

which means that for a node (Λ, u) the parameter $\gamma_1(\Lambda, u)$ is the sum of the interaction parameters of the children of the node, $\gamma_2(\Lambda, u)$ is the sum of the interaction parameters of the children's children of the node, and so on. Whenever these parameters are available the interaction parameters can be calculated by

$$\beta(\Lambda, u) = U(\chi(\Lambda, u)) - \sum_{g=1}^{|\Lambda|} \gamma_g(\Lambda, u). \tag{13}$$

This is obtained by inserting (12) into (10). It is obvious that the recursive formula presented here is much more efficient than the intuitive way of calculating the interaction parameters. It is also clear from the construction of this formula that (8) is in fact a unique representation of the energy function. In Section 4.1.1 we use this recursive formula to calculate all the interaction parameters of a generalized Potts model defined on a graph.

4 Exact recursive algorithm

In the previous section we presented our DAG representation for the probability distribution of a Markov random field. In addition we derived a recursive formula for calculation of the weights in this DAG. We observed that the number of weights, and the number of nodes in the DAG, very quickly became large as a function of $|S|$ and $|\Omega|$. However, in the following section we will explain why many of these interaction parameters are equal to 0, and how we can use this to construct a much smaller DAG. As an example of this property we calculate all the interaction parameters for a Markov random field distributed according to the generalized Potts model. In the rest of this section we finally present our recursive forward-backward algorithm. By the forward part of this algorithm we will be able to decompose the distribution $\pi(z)$ according to

$$\pi(z) = \left[\prod_{k=1}^{n-1} \pi(z_{t(k)}|z_{t(l)}, l = k+1, ..., n)\right] \pi(z_{t(n)}), \tag{14}$$

where $t(1), ..., t(n)$ represents a chosen permutation of the elements in S. We will show how each of the distributions in this expression can be represented by an individual DAG. Next we use this representation to define two different backward parts to our algorithm. Through the first one we are able to simulate from $\pi(z)$, while the second one enables us to calculate the marginal distributions for all the nodes in S. Finally we explain how the algorithm can be applied on a Bayesian problem.

In this section we establish an exact algorithm. However, as $|S|$, $|\Omega|$, and the size of the neighbourhoods grows, we get computational problems with this algorithm. This motivates the use of the approximations presented in Section 5.

4.1 Reduced DAG representation

In the general DAG representation of Section 3.2 we obtained a total of $|\Omega|^n$ nodes, where an equally large number of interaction parameters had to be computed. However, as we have assumed a Markov random field, we can prove by the Hammersley-Clifford theorem of Section 2.3 that all interaction parameters representing non-cliques are equal to 0. We formulate this property in the following theorem.

Theorem 2: Let z be a Markov random field with respect to a given neighbourhood system N, and let $\mathcal{C}$ be the set of all cliques. Then $\beta(\Lambda, u) = 0 \ \forall \ \Lambda \notin \mathcal{C}$ and $u \in \mathcal{Z}^{|\Lambda|}$.

The proof of this theorem can be found in Arnesen (2009). Using this theorem, we write the energy function of a Markov random field as

$$U(z) = \sum_{(\Lambda,u)\in\mathcal{B}} \beta(\Lambda, u) I(u = z_\Lambda), \tag{15}$$

where $\mathcal{B}$ is the set defined by

$$\mathcal{B} = \bigcup_{\substack{(\Lambda,u)\in\mathcal{P}(S) \\ \beta(\Lambda,u)\neq 0}} \left\{(A, v) | A \subseteq \Lambda, (\chi(\Lambda, u))_A = v\right\}. \tag{16}$$

We see that a node $(\Lambda, u) \in \mathcal{P}(S)$ does not belong to this set if, and only if, $\beta(\Lambda, u) = 0$ and all other nodes of higher order that can reach this node by the edges in the DAG also has interaction parameters equal to 0. Using Theorem 2, we are able to reduce the number of nodes in our DAG representation. We denote this reduced DAG by $G = \mathcal{G}(\mathcal{B}, \beta[\mathcal{B}])$, where $\beta[\mathcal{B}]$ represents the weights as before. To illustrate the set $\mathcal{B}$, we take another look at the example in Figure 3. As discussed in Section 2.2, the cliques in such a Markov random field is the empty set, sets containing just one element, and sets containing two nodes where these two nodes are neighbours. The DAG for this situation is shown in Figure 5, but note that we

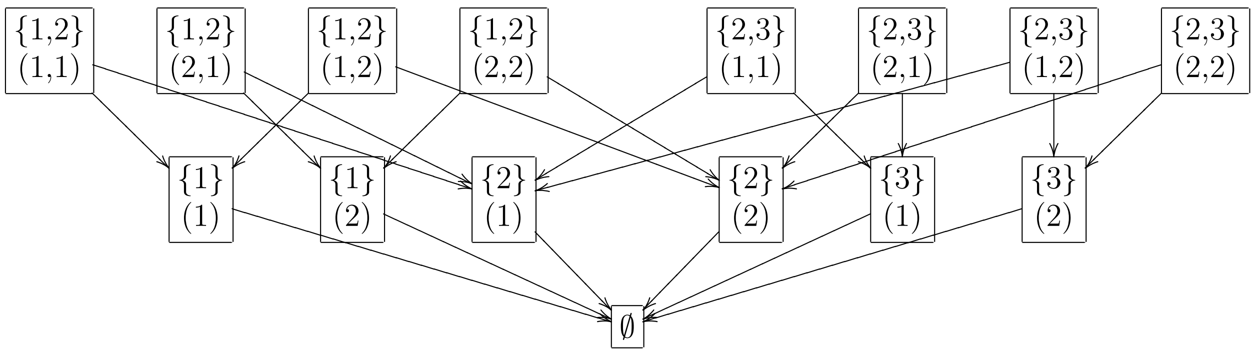


Figure 5: The DAG $\mathcal{G}(\mathcal{B}, \beta[\mathcal{B}])$ for our illustrative example.

have assumed that $\beta(\Lambda, u) \neq 0$ for $\Lambda = \{1,2\}, \{2,3\}$ and $u \in \mathcal{Z}^2$. In this figure we see that all third order interactions are gone compare to Figure 4. Because of this and because $\{1,3\} \notin \mathcal{C}$, the nodes $(\{1,3\}, (z_1, z_3))$ for all $z_1, z_3 \in \mathcal{Z}$ are gone as well.

The point of this section was to reduce the number of nodes in our DAG representation. By this, much more complex problems can be evaluated. Later we will also see that this reduction makes our algorithm much more efficient. But first, let us look at an example where we calculate all the interaction parameters of a generalized Potts model, using the reduced DAG representation and the recursive formula of Section 3.3.2.

4.1.1 The interaction parameters of a generalized Potts model

Assume that our Markov random field is defined on a lattice with a first order neighbourhood system, and that it is distributed is according to (4). We label the nodes in lexicographical order, and to write the distribution on the form of (6) and (15), we need to determine the interaction parameters. We start out by setting (6) equal to (4), and obtain

$$U(z) = \sum_{(\Lambda,u)\in\mathcal{B}} \beta(\Lambda, u) I(u = z_\Lambda) = \frac{1}{2} \sum_{k=1}^{n} \sum_{l \in N_k} \alpha(z_k, z_l). \tag{17}$$

We will solve this system of equations with respect to the β-parameters, using the recursion formula of Section 3.3.2. Because of the one-to-one relation (7), we can obtain the interaction parameter $\beta(\emptyset)$ from the vector $z = (0, 0, ..., 0)$. We find that

$$\beta(\emptyset) = \frac{1}{2} \sum_{k=1}^{n} \sum_{l \in N_k} \alpha(0, 0) = 0,$$

and that $\gamma_0(\emptyset) = \beta(\emptyset) = 0$. Moving on, we calculate $\gamma_1(\{k\}, (z_k)) = \gamma_0(\emptyset) = 0$ for all $k \in S$ and $z_k \in \mathcal{Z}$. From the vector $z = (0, ..., z_k, 0, ..., 0)$ and (17) we find that $U(z) = |N_k|\alpha(0, z_k)$. Using (13), we obtain the first order interaction parameters

$$\beta(\{k\}, (z_k)) = |N_k|\alpha(0, z_k) \ \forall \ k \in S \text{ and } z_k \in \mathcal{Z},$$

where N_k as usual is the neighbourhood of node k. Finally we need to determine the second order interactions $\beta(\{k, l\}, (z_k, z_l))$ for all $k \in S$, $l \in N_k$, and $z_k, z_l \in \mathcal{Z}$. From (11) we find

$$\begin{aligned}
\gamma_1(\{k, l\}, (z_k, z_l)) &= \gamma_0(\{k\}, (z_k)) + \gamma_0(\{l\}, (z_l)) \\
&= \beta(\{k\}, (z_k)) + \beta(\{l\}, (z_l)) \\
&= |N_k|\alpha(0, z_k) + |N_l|\alpha(0, z_l) \text{ and,} \\
\gamma_2(\{k, l\}, (z_k, z_l)) &= \frac{1}{2}(\gamma_1(\{k\}, (z_k)) + \gamma_1(\{l\}, (z_l))) \\
&= \frac{1}{2}(0 + 0) = 0,
\end{aligned}$$

and from the vector $z = (0, .., z_k, 0, .., z_l, 0, ..., 0)$ and (17) we find that

$$U(\chi(\{k, l\}, (z_k, z_l))) = (|N_k| - 1)\alpha(0, z_k) + (|N_l| - 1)\alpha(0, z_l) + \alpha(z_k, z_l).$$

Using (13) one last time, we get the second order interaction parameters

$$\beta(\{k, l\}, (z_k, z_l)) = \alpha(z_k, z_l) - \alpha(0, z_k) - \alpha(0, z_l) \ \forall \ k \in S, l \in N_k, \text{ and } z_k, z_l \in \mathcal{Z}.$$

Because this is a generalized Potts model with a first order neighbourhood system, all the interaction parameters for all the nodes in $\mathcal{B}$ are obtained by these formulas. From these formulas we are able to represent the energy function of any generalized Potts model on the form of (15). This representation enables us to use our recursive forward-backward algorithm presented in the following sections.

4.2 The forward part, decomposition

In first step of the forward part of our algorithm we decompose the probability distribution $\pi(z)$ according to

$$\pi(z) = \pi(z_r|z_{-r})\pi(z_{-r}), \tag{18}$$

where r is a chosen element from S, the first variable to be decomposed from the system. By iterating this procedure, decomposing $\pi(z_{-r})$ in the next iteration,

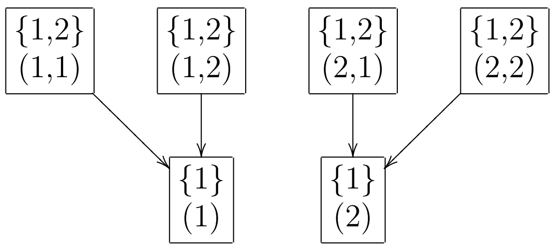


Figure 6: The DAG $G_1 = \mathcal{G}(\mathcal{B}_1, \beta[\mathcal{B}_1])$ for our illustrative example.

we obtain (14). In the next three sections we will therefore focus on establishing the distributions $\pi(z_r|z_{-r})$ and $\pi(z_{-r})$ from $\pi(z)$. These two distributions will be represented by DAGs, in the same way as $\pi(z)$ is represented.

4.2.1 Splitting

First of all we define the set $\mathcal{B}_r = \{(\Lambda, u) \in \mathcal{B} | r \in \Lambda\}$. In words, this is a subset of $\mathcal{B}$ containing all elements where node r is present in Λ. We can use this set to determine the distribution $\pi(z_r|z_{-r})$ in the following way

$$\pi(z_r|z_{-r}) \propto \exp\left\{ -\sum_{(\Lambda,u)\in\mathcal{B}_r} \beta(\Lambda, u) I(u = z_\Lambda) \right\}, \tag{19}$$

where $z = (z_1, ..., z_{r-1}, z_r, z_{r+1}, ..., z_n)$. The set $\mathcal{B}_r$, and therefore the distribution $\pi(z_r|z_{-r})$, can also be represented as a DAG. We denote this DAG by $G_r = \mathcal{G}(\mathcal{B}_r, \beta[\mathcal{B}_r])$, where $\beta[\mathcal{B}_r] = \{\beta(\Lambda, u)|(\Lambda, u) \in \mathcal{B}_r\}$ denotes the weights. This way we obtain a one-to-one relation between the interaction parameters in $\pi(z_r|z_{-r})$ and the nodes in G_r. Let us again look at our example for illustration. By assuming $r = 1$, we establish the DAG G_1 shown in Figure 6. This DAG is easily obtained by finding all the nodes in the DAG representing $\mathcal{B}$ where $1 \in \Lambda$, and separate these nodes from the rest of the DAG. The DAG G_1 represents the distribution $\pi(z_1|z_2, z_3) = \pi(z_1|z_2)$, where the last equality is valid because z_1 is independent of z_3 given z_2, according to the Markov property (1). Note that G_1 consists of two separate parts. Generally we get one separate part for each element in $\mathcal{Z}$.

So far we have obtained the DAG representing $\pi(z_r|z_{-r})$, and in the following we move on to determine the distribution $\pi(z_{-r})$. This is a distribution for the stochastic field z_{-r}, with an energy function $U_*(z_{-r})$ that needs to be determined. To do this we first define the set $\mathcal{B}_{-r} = \mathcal{B} \setminus \mathcal{B}_r$, such that $\mathcal{B}_r$ and $\mathcal{B}_{-r}$ is a partition of $\mathcal{B}$. The set $\mathcal{B}_{-r}$ can also be represented as a DAG, and this DAG is denoted by $G_{-r} = \mathcal{G}(\mathcal{B}_{-r}, \beta[\mathcal{B}_{-r}])$. The DAG G_{-1} for our illustrative example is shown in Figure 7, and note how this DAG contains all the nodes in Figure 4 that are not

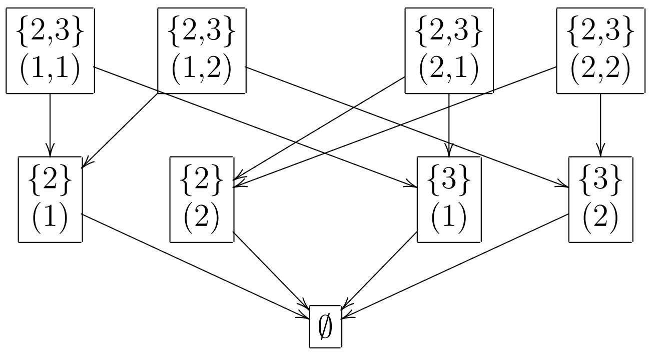


Figure 7: The DAG $G_{-1} = \mathcal{G}(\mathcal{B}_{-1}, \beta[\mathcal{B}_{-1}])$ for our illustrative example.

in Figure 6.

Next we show how to use both these DAGs to determine the energy function $U_*(z_{-r})$, and thereby the probability distribution $\pi(z_{-r})$.

4.2.2 Calculating $U_*(z_{-r})$

To determine the energy function $U_*(z_{-r})$ for the distribution $\pi(z_{-r})$, we start by summing out the variable z_r from $\pi(z)$,

$$\begin{aligned}
\pi(z_{-r}) &= \sum_{z_r} \pi(z) = \sum_{z_r} c \cdot \exp\left\{ - \sum_{(\Lambda,u)\in\mathcal{B}} \beta(\Lambda,u) I(u = z_\Lambda) \right\} \\
&= c \cdot \exp\left\{ -U_*(z_{-r}) \right\} \\
&\Downarrow \\
U_*(z_{-r}) &= -\ln\left(\sum_{z_r} \exp\left\{ - \sum_{(\Lambda,u)\in\mathcal{B}} \beta(\Lambda,u) I(u = z_\Lambda) \right\} \right).
\end{aligned}$$

Next we use that $\mathcal{B}_r$ and $\mathcal{B}_{-r}$ is a partition of $\mathcal{B}$ to write the energy function as

$$U_*(z_{-r}) = U_*^1(z_{N_r}) + U_*^2(z_{-r}),$$

where we obtain

$$\begin{aligned}
U_*^1(z_{N_r}) &= -\ln\left(1 + \sum_{z_r\in\mathcal{Z}} \exp\left\{ - \sum_{(\Lambda,u)\in\mathcal{B}_r} \beta(\Lambda,u) I(u = z_\Lambda) \right\} \right), \text{ and} \\
U_*^2(z_{-r}) &= \sum_{(\Lambda,u)\in\mathcal{B}_{-r}} \beta(\Lambda,u) I(u = z_\Lambda),
\end{aligned}$$

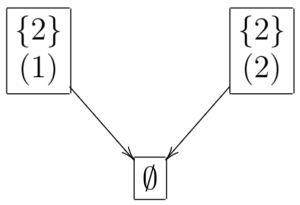


Figure 8: The DAG $\mathcal{G}(\mathcal{P}(N_1), \Delta\beta[\mathcal{P}(N_1)])$ for our illustrative example.

where again $z = (z_1, ..., z_{r-1}, z_r, z_{r+1}, ..., z_n)$. The right hand side of the expression for $U_*^2(z_{-r})$ may look like it is dependent on the value of z_r since it is present in the vector z. However, the value of z_r is unimportant since $r \notin \Lambda$ for any $(\Lambda, z_\Lambda) \in \mathcal{B}_{-r}$. This is just a convenient notation.

So far we have seen that the energy function $U_*(z_{-r})$ can be expressed as a sum of two separate energy functions, one concerning interaction parameters of the elements in $\mathcal{B}_r$, and one of the elements in $\mathcal{B}_{-r}$. We also observe that $U_*^2(z_{-r})$ is on canonical form, and that $U_*^1(z_{N_r})$ is not. We wish to express $U_*^1(z_{N_r})$ on canonical form as well, and this can be done by observing that for all $(\Lambda, u) \in \mathcal{B}_r$, we know that $\Lambda \setminus \{r\} \subseteq N_r$. Because the inner sum in the expression for $U_*^1(z_{N_r})$ is over the elements of $\mathcal{B}_r$, we introduce $\mathcal{P}(N_r) = \{(\Lambda, u) | \Lambda \subseteq N_r, u \in \mathcal{Z}^{|\Lambda|}\}$ and write

$$U_*^1(z_{N_r}) = \sum_{(\Lambda,u)\in\mathcal{P}(N_r)} \Delta\beta(\Lambda, u) I(u = z_\Lambda), \tag{20}$$

where the interaction parameters $\Delta\beta(\Lambda, u)$ can be determined by the recursive formula presented in Section 3.3.2. The set $\mathcal{P}(N_r)$ is represented by the DAG $\mathcal{G}(\mathcal{P}(N_r), \Delta\beta[\mathcal{P}(N_r)])$, where $\Delta\beta[\mathcal{P}(N_r)]$ represents the weights as usual. Let us continue our illustrative example by finding the DAG representing $\mathcal{P}(N_1)$. The neighbourhood N_1 is easily obtained by traversing the nodes in the DAG representing $\mathcal{B}_1$. In this case we have $N_1 = \{2\}$, and we construct the DAG $\mathcal{G}(\mathcal{P}(N_1), \Delta\beta[\mathcal{P}(N_1)])$ as shown in Figure 8.

Now only one step separates us from determine the DAG representing the distribution $\pi(z_{-r})$, and it will be explained in the next section. We denote this DAG by $G_* = \mathcal{G}(\mathcal{B}_*, \beta[\mathcal{B}_*])$, where $\beta[\mathcal{B}_*]$ as usual represents the weights.

4.2.3 Computation of G_*

We start out by defining the set $\tilde{\mathcal{B}}_* = \mathcal{B}_{-r} \bigcup \mathcal{P}(N_r)$. Next we establish the DAG $\tilde{G}_* = \mathcal{G}(\tilde{\mathcal{B}}_*, \beta[\tilde{\mathcal{B}}_*])$, where the weights can be determined from

$$\beta_*(\Lambda, u) = \begin{cases} \beta(\Lambda, u) + \Delta\beta(\Lambda, u) & \text{if} \quad (\Lambda, u) \in \mathcal{B}_{-r} \bigcap \mathcal{P}(N_r) \\ \beta(\Lambda, u) & \text{if} \quad (\Lambda, u) \in \mathcal{B}_{-r} \setminus \mathcal{P}(N_r) \\ \Delta\beta(\Lambda, u) & \text{if} \quad (\Lambda, u) \in \mathcal{P}(N_r) \setminus \mathcal{B}_{-r} \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

The weighted DAG $\tilde{G}_*$ is a valid representation of the distribution $\pi(z_{-r})$. However, this DAG may be containing nodes that according to (16) should not be there. So, to complete our decomposition we need to remove these nodes. The results is the DAG G_*, where $\mathcal{B}_* \subseteq \tilde{\mathcal{B}}_*$.

So far in this section we have decomposed $\pi(z)$ according to (18). The algorithm for doing this is shown in Figure 9. This procedure can, as already explained, be iterated for $\pi(z_{-r})$. That is, in the next iteration we need to determine $\pi(z_s|z_{-\{r,s\}})$ and $\pi(z_{-\{r,s\}})$, where s is the next variable we choose to decompose from the system, $t(2) = s$ in (14). When all the variables are decomposed according to (14), we are finished with the forward part of our algorithm.

In Section 1 we mentioned that (14) is a partially ordered Markov model, and in this section we have seen how the conditional distribution $\pi(z_{t(k)}|z_{t(l)}, l = k + 1, ..., n)$ for $k = 1, ..., n - 1$ is obtained from the forward part of our algorithm. However, because of the Markov property (1) of the original distribution $\pi(z)$, the conditional distribution is only dependent of a subset of $z_{t(l)}$ for $l = k + 1, ..., n$. This subset will, in terms of Cressie & Davidson (1998), be called an adjacent lower neighbourhood, and it will be denoted by $L_{t(k)}\ \forall\ k \in S$, where $L_{t(1)} = N_{t(1)}$ and $L_{t(n)} = \emptyset$ by definition. When all these sets are determined, we write (14) as

$$\pi(z) = \left[\prod_{k=1}^{n-1} \pi(z_{t(k)}|z_{L_{t(k)}})\right] \pi(z_{t(n)}), \tag{22}$$

where $\pi(z_{t(n)}|z_{L_{t(n)}}) = \pi(z_{t(n)}|\emptyset) = \pi(z_{t(n)})$. Note that in the decomposition of node $t(k)$, for $k = 2, ..., n$, we need to determine the set $\mathcal{P}(L_{t(k)})$ when performing step 2 in Figure 9 to obtain $\pi(z_{-\{t(1),...,t(k)\}})$. This is because we generally only have $N_{t(k)} = L_{t(k)}$ for $k = 1$.

4.3 The backward part

In this section we present two alternative backward parts to our forward-backward algorithm. First of all we are able to simulate from the decomposed system. This is done by first simulating $z_{t(n)}$, then $z_{t(n-1)}$, and so on. Second of all we present a

1. Split the DAG $G = \mathcal{G}(\mathcal{B}, \beta[\mathcal{B}])$ into the two DAGs $G_r = \mathcal{G}(\mathcal{B}_r, \beta[\mathcal{B}_r])$ and $G_{-r} = \mathcal{G}(\mathcal{B}_{-r}, \beta[\mathcal{B}_{-r}])$. The DAG G_r is a representation of $\pi(z_r|z_{-r})$.
2. Find the set of neighbours N_r from G_r by $N_r = \{k \in S_{-r} | \{k, r\} \in \mathcal{B}_r\}$, and establish the set $\mathcal{P}(N_r)$.
3. Calculate the interaction parameters $\Delta\beta(\Lambda, u) \ \forall \ (\Lambda, u) \in \mathcal{P}(N_r)$, and establish the DAG $\mathcal{G}(\mathcal{P}(N_r), \Delta\beta[\mathcal{P}(N_r)])$.
4. Add $\mathcal{G}(\mathcal{P}(N_r), \Delta\beta[\mathcal{P}(N_r)])$ to G_{-r} to find the DAG $\tilde{G}_* = \mathcal{G}(\tilde{\mathcal{B}}_*, \beta_*[\tilde{\mathcal{B}}_*])$, where $\tilde{\mathcal{B}}_* = \mathcal{B}_{-r} \bigcup \mathcal{P}(N_r)$.
5. Delete all nodes $(\Lambda, u) \in \tilde{\mathcal{B}}_*$ in the DAG $\tilde{G}_*$ where the parameters $\beta_*(A, v) = 0 \ \forall \ (A, v) \in \{(A, v) \in \tilde{\mathcal{B}}_* | \Lambda \subseteq A, (\chi(A, v))_\Lambda = u\}$. This gives the DAG G_*, which is a valid representation of $\pi(z_{-r})$.

Figure 9: Algorithm for decomposing $\pi(z)$ into $\pi(z_r|z_{-r})$ and $\pi(z_{-r})$.

recursive formula for calculating the marginal distribution for all $k \in S$. That is, we show how to establish the distributions $\pi(z_k) \ \forall \ k \in S$.

4.3.1 Simulation

After performing the full decomposition of the distribution $\pi(z)$, we are able to simulate from $\pi(z)$ in the following way. First we simulate $z_{t(n)} \sim \pi(z_{t(n)})$, where the normalizing constant for this distribution can be obtained by

$$c = \left[\sum_{i=0}^{K-1} \pi(z_{t(n)} = i) \right]^{-1}. \tag{23}$$

Note that this is actually the same normalizing constant as in (6). Next we simulate $z_{t(n-1)} \sim \pi(z_{t(n-1)}|z_{t(n)}) = \pi(z_{t(n-1)}|z_{L_{t(n-1)}})$, and so on until we finally simulate $z_{t(1)} \sim \pi(z_{t(1)}|z_{t(2)}, ..., z_{t(n)}) = \pi(z_{t(1)}|z_{L_{t(1)}})$. The result of this backward part of our algorithm is a vector $z = (z_1, z_2, ..., z_n)$, which is the wanted realisation from $\pi(z)$. Remember that $t(1), t(2), ..., t(n)$ is a permutation of the elements in S. This completes our exact recursive forward-backward algorithm for simulation of discrete Markov random fields.

4.3.2 Calculating the marginals

In this section we focus on calculating the marginal distributions for all $k \in S$. That is, as an alternative to the backward part of our algorithm that simulates from $\pi(z)$, we show how to find the distributions $\pi(z_k)\ \forall\ k \in S$. In this section we assume the decomposition to be done according to $t(k) = k$ in (14). Calculation of the marginals is interesting in many ways. For instance, we know that the marginals for a Potts model defined on a lattice with $\Omega = \{0, 1, ..., K-1\}$ are

$$\pi(z_k) = \frac{1}{K}\ \forall\ k \in S,$$

and we can use this to check the correctness of our implemented algorithm. For a generalized Potts model the marginals are not analytically known, but we will be able to calculate them by the formulas presented in this section. Calculating the marginals can also be used to find the marginal posterior mode estimate of an underlying picture in a Bayesian setting. We will take a look at this in Section 4.4.

The marginal distribution $\pi(z_n)$ is given from the forward part of our algorithm. We are also given the conditional distributions $\pi(z_k|z_{L_k})\ \forall\ k = n-1, ..., 1$, and we will in the following use these distributions to calculate the rest of the desired marginals. The recursive formulas that we give look complicated, but we will carefully explain them, and see that they are in fact quite intuitive. We start out by defining $A_n = \{n\}$, and write $\pi(z_n) = \pi(z_{A_n})$. With help from the union of all the adjacent lower neighbourhoods up to a node k, that is $U_k = L_1 \bigcup L_2 \bigcup ... \bigcup L_k$, we can recursively calculate all the marginal distributions by

$$\pi(z_{A_k}) = \sum_{z_{A_{k+1}\setminus U_k}} \pi(z_k|z_{L_k})\pi(z_{A_{k+1}}),\ \text{where } A_k = \{k\} \bigcup (A_{k+1} \bigcap U_k), \tag{24}$$

$$\pi(z_k) = \sum_{z_{A_k \setminus \{k\}}} \pi(z_{A_k}),\ \text{for } k = n-1, ..., 1. \tag{25}$$

To explain these formulas we start out by looking at the sets A_k for $k = 1, ..., n$. First of all we note that $k \in A_k$ by definition. The rest of the set contains all nodes where we have already calculated a marginal distribution, and where these nodes still are in at least one of the adjacent lower neighbourhoods too come. This means that a variable z_l for $l \in A_{k+1}$ is to be summed out in (24) if, and only if, node l is not present in any of the sets $L_1, ..., L_k$. By this we also know that $L_k \subseteq A_{k+1}$, which allows us to write $\pi(z_k|z_{A_{k+1}}) = \pi(z_k|z_{L_k})$ in (24), by the Markov property (1). When the probability distribution $\pi(z_{A_k})$ for $k \in S$ is obtained, we are able to calculate the marginal distribution $\pi(z_k)$ by the forward decomposition explained in Section 4.2. We note that A_k not is the only useful set when using these formulas for all $k \in S$. In fact we can use any set $\tilde{A}_k \subseteq \{k, ..., n\}$ as long as $A_k \subseteq \tilde{A}_k$. However, the sets A_k for all $k \in S$ are optimal when using these recursive formulas.

Let us illustrate this recursion by an example. Assume we have completed the decomposition for the nodes in Figure 1, and that we have calculated the marginals $\pi(z_k)$ for $k = 20, ..., 9$. It can be shown that we have $A_9 = \{9, 10, 11, 12, 13, 14\}$ and $U_8 = \{1, 2, ..., 13\}$ at this point. Using (24), we easily obtain the set $A_8 = \{8, 9, 10, 11, 12, 13\}$, and find that the sum in this equation needs to be over the variables in $A_9 \setminus U_8 = \{14\}$ to obtain $\pi(z_{A_8})$. Next we use this distribution to find the marginal distribution $\pi(z_8)$ by (25), which completes the iteration. In fact, this example illustrates a case where the optimal A_k is easy to find for all $k \in S$. Let us generalize the example by assuming a $m' \times m$ lattice with a first order neighbourhood system, for instance a Potts model. We label the nodes $S = \{1, 2, ..., m'm\}$ in lexicographical order. When calculating the marginals in this system, we find that the optimal $A_k \ \forall\ k \in S$ is $A_k = \{k, ..., m'm\}$ for $k = m'm, ..., m'm - m$ and $A_k = \{k, ..., k + m\}$ for $k = m'm - m - 1, ..., 1$. Because of this dependency, Friel & Rue (2007) term this is a lag m model.

4.4 Using the algorithm for Bayesian calculations

In this section we explain how our algorithm can be used to estimate an underlying picture in a Bayesian setting. To read more about this problem, see for instance Besag (1986) or Geman & Geman (1984). Assume that an unobserved picture z is a realisation from a Markov random field with probability distribution $p(z)$. This distribution is called the prior distribution in this Bayesian setting. Next we assume that we have observed a stochastic degenerated version y of z. That is, we assume a likelihood model such that $y \sim l(y|z)$. Using Bayes theorem we find the posterior distribution by

$$\pi(z|y) \propto l(y|z)p(z). \tag{26}$$

The goal is now to estimate the underlying picture z. By writing the posterior distribution with an energy function as in (15), running our recursive algorithm, and calculating the marginal distributions, $\pi(z_k|y)$, we can estimate z by

$$\hat{z}_k = \arg\max_{i \in \Omega} \pi(z_k = i|y) \ \forall\ k \in S, \tag{27}$$

such that $\hat{z} = (\hat{z}_1, \hat{z}_2, ..., \hat{z}_n)$ becomes our final marginal posterior mode estimate. Instead of calculating the marginal distributions, as shown in the previous section, we can also estimate them by

$$\hat{\pi}(z_k = i|y) = \frac{1}{M}\sum_{j=1}^{M} I(z_k^{(j)} = i) \ \forall\ k \in S,\ i \in \Omega, \tag{28}$$

where $z^{(j)}$ for $j = 1, ..., M$ are M simulated realisations. Estimating the marginals according to this formula is, for complex problems, computationally cheaper than calculating them. Let us give an example of such a Bayesian problem, and show how to write the posterior distribution with an energy function as in (15).

4.4.1 Example on a Bayesian model

As an example of the concept presented in the previous section, we assume that our prior distribution $p(z)$ is a generalized Potts model. The underlying picture z is therefore assumed to be from this distribution. Secondly, let us assume that our observed picture y is obtained from a likelihood model $l(y|z)$ that provides independent Gaussian noise to each element in z. That is,

$$l(y|z) = \prod_{k=1}^{n} f(y_k|z_k) \propto \exp\left\{-\frac{1}{2\sigma^2}\sum_{k=1}^{n}(y_k - z_k)^2\right\}, \tag{29}$$

where σ^2 is the variance. As explained in Section 4.4, we need to determine the posterior distribution $\pi(z|y)$ and write it on canonical form. From Bayes theorem it is easy to see that the posterior distribution becomes

$$\pi(z|y) = l(y|z)p(z) \propto \exp\left\{-\frac{1}{2\sigma^2}\sum_{k=1}^{n}(y_k - z_k)^2 - \frac{1}{2}\sum_{k=1}^{n}\sum_{l\in N_k}\alpha(z_k, z_l)\right\},$$

and by following the procedure of Section 4.1.1, we calculate that

$$\begin{aligned}
\beta(\emptyset) &= \frac{1}{2\sigma^2}\sum_{i=1}^{n} y_i^2, \\
\beta(\{k\}, (z_k)) &= |N_k|\alpha(z_k, 0) + \frac{1}{2\sigma^2}(z_k^2 - 2z_k y_k)\ \forall\ k \in S,\ z_k \in \mathcal{Z},\ \text{and} \\
\beta(\{k, l\}), (z_k, z_l)) &= \alpha(z_k, z_l) - \alpha(0, z_l) - \alpha(z_k, 0)\ \forall\ k \in S, l \in N_k,\ z_k, z_l \in \mathcal{Z}.
\end{aligned}$$

By these expressions we have obtained all the interaction parameters for the posterior distribution in this Bayesian setting. Examples of using our algorithm in this setting are given in Section 7.6.

5 Approximations

Exact calculations of Markov random fields are very limited by the size of S, Ω, and the neighbourhoods. Friel & Rue (2007), for instance, manage to do exact calculations on a binary lattice of size $m \times m'$, where $m \leq m'$ and $m \leq 19$, and with a first order neighbourhood system. Tjelmeland & Austad (2010) do exact calculations on 15×15 lattices for the same situation in their examples. Because they both assume binary lattices, we expect our exact algorithm to be even more limited as a function of $|\Omega|$. We attack this problem by first observing why the exact algorithm so fast becomes inefficient, and even impossible to run. Next we introduce our approximations to the exact algorithm, and they will closely follow the ones suggested in Tjelmeland & Austad (2010).

5.1 Why approximations?

Assume a lattice as shown in Figure 1, and that $\Omega = \{0, 1, 2\}$ for this example. As already discussed, this is a first order neighbourhood system where the cliques are the empty set, sets containing one node, and set containing two nodes where these two nodes are neighbours. Let us assume that the variables are decomposed from the distribution $\pi(z)$ in the order $t(k) = k$ for $k = 1, ..., 20$. According to the algorithm in Figure 9, we need to determine $\mathcal{P}(N_1)$, and because $N_1 = \{2, 6\}$ we get $\mathcal{P}(N_1) = \mathcal{P}(L_1) = \{\emptyset, (\{2\}, (1)), ..., (\{2, 6\}, (2, 2))\}$. From this set we see that the nodes $(\{2, 6\}, (z_2, z_6))\ \forall\ z_2, z_6 \in \mathcal{Z}$ could be present in the DAG G_*. This means that in the next iteration we may have interactions between node 2 and 6. In the next iteration we get $\mathcal{P}(L_2) = \{\emptyset, (\{3\}, (1)), ..., (\{3, 6, 7\}, (2, 2, 2))\}$, which means that as a result of the decomposition so far, we could get a clique of size 3 and node 2 may interact with one node more that it initially did. As we go on with our algorithm it is easy to understand that for large lattices this effect leads to computational problems. In fact, when decomposing the nodes lexicographically in an $m' \times m$ lattice with a first order neighbourhood system, we obtain cliques and adjacent lower neighbourhoods of size m. This is of course computationally impossible when m is large. First of all the CPU time becomes too large, but we also get difficulties with saving all the different interaction parameters in the memory. This motivates the use of the approximations that will be presented and discussed in the rest of this section.

5.2 The approximations

As in Tjelmeland & Austad (2010), we define two types of approximations. First of all we approximate an interaction parameter $\Delta\beta(\Lambda, u)$ to 0 if $|\Delta\beta(\Lambda, u)| < \varepsilon$. In words, we choose a parameter $\varepsilon > 0$ that is a minimum allowed value for the absolute value of the interaction parameters. If we choose ε large enough, this solves the problem of saving all the interaction parameters in the memory. However, we still need to calculate all the interaction parameters to know which of them to approximate to 0. This problem is dealt with by assuming that higher order interactions are smaller than lower order interactions. The interaction parameter $\Delta\beta(\Lambda, u)$ for the node $(\Lambda, u) \in \mathcal{P}(L_k)$, for $k = 1, ..., n-1$, is approximated to 0 if the interaction parameters of all the children of the node are approximated to 0. More precisely, a parameter $\Delta\beta(\Lambda, u)$ is approximated to 0 if $\Delta\beta(A, v)$ is approximated to 0 for all $A \subset \Lambda$, $|\Lambda \setminus A| = 1$, and $(\chi(\Lambda, u))_A = v$. We observe that setting $\varepsilon = 0$ gives us our exact algorithm, and as we increase ε, the quality of our approximations decreases.

The approximations presented in this section are performed in each of the iterations of the algorithm. First the distribution $\pi(z)$ is decomposed into $\pi(z_r|z_{-r})$, and after the first approximations we get the approximate distribution $\tilde{\pi}(z_{-r})$. Secondly, $\tilde{\pi}(z_{-r})$ is decomposed into $\tilde{\pi}(z_s|z_{-\{r,s\}})$, and after new approximations we obtain $\tilde{\pi}(z_{-\{r,s\}})$. Continuing this procedure until all the variables are decomposed into conditional distribution, we get the approximate distribution

$$\tilde{\pi}(z) = \pi(z_{t(1)}|z_{-t(1)}) \left[\prod_{k=2}^{n-1} \tilde{\pi}(z_{t(k)}|z_{t(l)}, l = k+1, ..., n)\right] \tilde{\pi}(z_{t(n)}) \approx \pi(z), \tag{30}$$

where $t(1), ..., t(n)$ again is a chosen permutation of the nodes in S. These approximations make it computationally possible to decompose $\pi(z)$ into $\pi(z_r|z_{-r})$ and $\tilde{\pi}(z_{-r})$. In addition, the approximations may introduce conditional independencies to the resulting Markov random field z_{-r} that are not present in $\pi(z_{-r})$. This is again an advantage when the rest of decomposition is performed. As a consequence of the conditional independencies that the approximations introduce, the sets L_k for $k = 2, ..., n-1$ may become smaller than in the exact case.

The simulation part of our algorithm is carried out by simulating from the approximate distribution. That is, we first simulate $z_{t(n)} \sim \tilde{\pi}(z_{t(n)})$, then simulate $z_{t(n-1)} \sim \tilde{\pi}(z_{t(n-1)}|z_{t(n)}) = \tilde{\pi}(z_{t(n-1)}|z_{L_{t(n-1)}})$, and so on until $z_{t(1)} \sim \pi(z_{t(1)}|z_{t(2)}, ..., z_{t(n)}) = \pi(z_{t(1)}|z_{L_{t(1)}})$ is simulated. The vector $z = (z_1, z_2, ..., z_n)$ is an approximate realisation from the distribution $\pi(z)$.

When calculating the marginal distributions, we may need to make more approximations. Namely, when summing out the variables in (24) and (25). These sums are evaluated by performing forward steps of our algorithm. For instance in (25), the nodes $A_k \setminus \{k\}$ needs to be decomposed from $\pi(z_{A_k})$ to find the marginal distribution $\pi(z_k)$. In these decompositions we introduce the same approximations as presented above, with or without the same value of ε.

This section completes the presentation of our approximate recursive forward-backward algorithm for calculations of discrete Markov random fields defined on graphs. In the next section we explain how to evaluate the performance of this approximate algorithm.

5.3 Evaluation of the approximations

To evaluate our algorithm we look at several aspects. First of all we need to investigate the quality of the simulated realisations. In other words, we want to find out how good the distribution $\pi(z)$ is approximated by $\tilde{\pi}(z)$. We evaluate this quantity by estimating the acceptance rate when using $\tilde{\pi}(z)$ as the proposal distribution in a Metropolis-Hastings algorithm with $\pi(z)$ as the target distribution (Gamerman & Lopes 2006). We use importance sampling (Casella & Robert 1999) to find our estimate. From the general theory of importance sampling we know that to estimate

$$\mu = E_f(h(x)),$$

where f is the distribution of x and $h(x)$ is a function of x, we can simulate $x_i \sim g(x)$ for $i = 1, ..., M$, and estimate μ by

$$\hat{\mu} = \sum_{i=1}^{M} h(x_i) W_i, \tag{31}$$

where W_i is a weight calculated by

$$W_i = \frac{f(x_i)}{g(x_i)} \Bigg/ \sum_{j=1}^{M} \frac{f(x_j)}{g(x_j)} .$$

In our case we want to find an estimate for the acceptance rate *acc* when going from a state $z^{(i)} \sim \pi(z^{(i)})$ and proposing a new state $z^{(j)} \sim \tilde{\pi}(z^{(j)})$. This means that $x = (z^{(i)}, z^{(j)})$ and $f(z^{(i)}, z^{(j)}) = \pi(z^{(i)})\tilde{\pi}(z^{(j)})$ when using the notation from above. Because $z^{(i)}$ and $z^{(j)}$ are independent, we write $(z^{(i)}, z^{(j)}) \sim g(z^{(i)}, z^{(j)}) = \tilde{\pi}(z^{(i)})\tilde{\pi}(z^{(j)})$. Calculation of the weights in this case gives us

$$W_{i,j} = \frac{\pi(z^{(i)})\tilde{\pi}(z^{(j)})}{\tilde{\pi}(z^{(i)})\tilde{\pi}(z^{(j)})} \Bigg/ \sum_{k=1}^{M} \sum_{\substack{l=1 \\ l \neq k}}^{M} \frac{\pi(z^{(k)})\tilde{\pi}(z^{(l)})}{\tilde{\pi}(z^{(k)})\tilde{\pi}(z^{(l)})} = \frac{\pi(z^{(i)})}{\tilde{\pi}(z^{(i)})} \Bigg/ \sum_{k=1}^{M} (M-1) \frac{\pi(z^{(k)})}{\tilde{\pi}(z^{(k)})} ,$$

for M realisations from $\tilde{\pi}(z)$, and where the double sum gives us all possible pairs $(z^{(i)}, z^{(j)})$ for $i, j = 1, ..., M$ and $i \neq j$. Inserting these weights into (31), gives us our final estimate

$$\begin{aligned}\widehat{acc} &= \sum_{i=1}^{M}\left[\frac{\omega_i}{M-1}\sum_{\substack{j=1\\ i\neq j}}^{M}\min\left\{1,\frac{\pi(z^{(j)})\tilde{\pi}(z^{(i)})}{\pi(z^{(i)})\tilde{\pi}(z^{(j)})}\right\}\right], \text{ where} \qquad (32)\\ \omega_i &= \frac{\pi(z^{(i)})}{\tilde{\pi}(z^{(i)})}\Big/\sum_{j=1}^{M}\frac{\pi(z^{(j)})}{\tilde{\pi}(z^{(j)})}.\end{aligned}$$

If the approximations are good, we expect $\widehat{acc}$ to be close to 1, but if the approximations are poor, we expect an estimate close to 0. Some results using this estimate are given in Section 7.3.

Let us assume the decomposition to be done according to $t(k) = k \;\forall\; k \in S$. Because of the construction of the forward part of our algorithm, the conditional probabilities $\pi(z_k = i|z_{L_k}) \;\forall\; k \in S$ and $i \in \Omega$ are easily accessible. It is interesting to evaluate the error done in these probabilities because of the approximations. If we assume a Potts model, many of the conditional probabilities are equal to each other for a node $k \in S$. This is because there are no discrimination of the values in Ω for this model. For instance, the two probabilities $p_1 = \pi(z_k = 1|z_{k+1} = 2, z_{L_k\setminus\{k+1\}} = \mathbf{0})$ and $p_2 = \pi(z_k = 1|z_{k+1} = 0, z_{L_k\setminus\{k+1\}} = \mathbf{2})$ are by definition equal to each other. Here a bold number means a vector containing only that number, for instance $\mathbf{2} = (2, 2, ..., 2)$. However, because the value 0 is treated differently by the algorithm than the other values in Ω, this value is also treated differently by the approximations. It is therefore interesting to see how two conditional probabilities that are supposed to be equal, becomes unequal because of the approximations and the discrimination of the value 0. Some results concerning this are given in Section 7.4.1.

Another way of evaluating the behaviour of the approximations is by tracking the error done in one node $l \in S$, as a result of the approximations. This can be done by evaluating the errors in the probabilities $p_2 = \pi(z_k = 1|z_{k+1} = 0, z_{L_k\setminus\{k+1\}} = \mathbf{2}) \;\forall\; k \in S$ when performing approximations only for $k = l$. For the remaining nodes we use the exact version of our algorithm. This requires, of course, that we are looking at a problem where the exact algorithm can be applied. Some numerical examples of such evaluations are shown in Section 7.4.2.

The approximations that are performed by the forward part of our algorithm effect the size of the adjacent lower neighbourhoods. As mentioned in Section 5.1, exact lexicographically decomposition of an $m' \times m$ lattice, with a first order neighbourhood system, gives cliques and adjacent lower neighbourhoods of size m. The approximations introduce conditional independencies to the resulting distributions, potentially making the sets L_k for $k = 2, ..., n-1$ smaller. An investigation of the size of these sets, as a function of α and ε, is given in Section 7.5.

6 Implementation

To implement the algorithm we use the computer language C. The input to our program is a text file containing the interaction parameters for the distribution we wish do calculations on. This text file is read, and the initial DAG is built. In this section we discuss how this DAG is constructed, and how the rest of the algorithm is implemented. The implementation consists of mainly five parts. These are, splitting a DAG with respect to a node r, growing a DAG from the adjacent lower neighbourhood of r, finding the union of two DAGs, simulating from the decomposed system, and calculating the marginals. We will go through the implementation of these five parts, but first we introduced the data structure used to represent a DAG. The algorithm is implemented such that it decomposes the nodes in order $t(k) = k$, $k = 1, 2, ..., n$. This means that if we want to decompose the problem in a different order, we have to relabel the nodes before running the algorithm. In our implementation we always include all first order interactions regardless of their β- or $\Delta\beta$-values. This is done to ease implementation. In the following we explain an implementation of the approximate version of the algorithm. But, as we have already seen, setting $\varepsilon = 0$ gives us our exact algorithm.

6.1 Data structure

To represent a node in the DAG we define a *struct*. To this *struct* we declare several *field*s. First of all we declare an *int* to represent $|\Lambda|$, and a *double* to represent the β-value of the node. Next we define a pointer to a list of pointers. Each of the pointers in this list points to a child of the node. In the same way we define a pointer to a list of pointers which points to all the parents of the node. This way we can easily reach all the children and all the parents of a node. By these lists of pointers, the edges in the DAG are represented. We are able to reach all the nodes in the DAG by a pointer to the root node $\emptyset$. In addition to these lists we declare the length of them as two *int*s. These two *field*s represent the number of parents and children of a node in the DAG.

We also need to declare *field*s that somehow represent (Λ, u) in each node. The intuitive way of doing this is by making two lists of length $|\Lambda|$ to hold the values in Λ and u. However, this is not a good idea since it will cost a lot of memory. Fortunately, we can avoid this problem by carefully order the lists of children in a way that in the following will be explained. Assume we are looking at the node

$(\Lambda, u) = (\{1,2,3\}, (1,2,1))$ in Figure 4. Remembering that the elements in Λ always are sorted in increasing order, we choose to save only the last element in Λ, which is $\{3\}$. Likewise, we only save the last element in u, which is (1), since the elements in u are ordered according to Λ, see (7). Next we see that the node has three children, namely $(\{1,2\},(1,2))$, $(\{1,3\},(1,1))$ and $(\{2,3\},(2,1))$. We now define the first child in the nodes' list of children to be $(\{1,2\},(1,2))$. In the node $(\{1,2\},(1,2))$ we only save the values $\{2\}$ and (2), and define its first child to be $(\{1\},(1))$. Here we save the values $\{1\}$ and (1). Using this system for all the nodes in a DAG, we can always find the values in a (Λ, u)-pair by a recursive search to the first child of the node. So, instead of declaring two lists of *int*s, we only need to declare two single *int*s to uniquely represents each (Λ, u)-pair in a DAG. This way a lot of memory is saved, and Tjelmeland & Austad (2010) used this technique in their implementation of the binary case as well.

The last *field* in our *struct* is a pointer to a list of *doubles.* This list is holding the γ_g-values of a node when the recursion from Section 3.3.2 is performed. This could, in the same way as discussed above, cause memory problems. However, this *field* is only in use when a DAG is constructed from an adjacent lower neighbourhood, and we will later in this section see that these DAGs are deleted from the memory after each iteration of the algorithm.

6.2 Splitting

After the initial DAG is constructed, the algorithm starts to decompose node 1. This is just a question of deleting edges in the DAG so that G_1 is isolated from G_{-1}. Remember that G_1 is the DAG representing the set $\mathcal{B}_1$, and that G_{-1} is the DAG representing the set $\mathcal{B}_{-1}$. It exists an easy and efficient way to do this, and it starts by deleting all pointers between $\emptyset$ and the nodes $(\{1\}, u)$ $\forall$ $u \in \mathcal{Z}$. We declare pointers to these nodes so that they can be reached later. Next we recursively move upwards from these nodes and observe that all nodes we visit from now on must have $1 \in \Lambda$. For each new node (Λ, u) we visit, one edge must be deleted, namely the edge to the child $(\Lambda \setminus \{1\}, (\chi(\Lambda, u))_{\Lambda\setminus\{1\}})$. For instance, the node $(\{1,2\},(1,1))$ in Figure 5 have an edge to node $(\{2\},(1))$ that needs to be deleted when node 1 is decomposed from the system. After a full split is performed, no pointers between the DAG G_1 and the DAG G_{-1} exist. Note that the DAG G_1 consists of $|\mathcal{Z}|$ isolated parts, see Figure 6.

The spitting procedure can of course be done for any $r \in S$ to establish the DAGs G_r and G_{-r}. After a split is performed, we write G_r to a binary file and clear it from the memory. This way we can reach all the DAGs later, and because the DAGs are written to the hard disk, we also save a lot of memory.

6.3 Making a DAG from an adjacent lower neighbourhood

After the split in the previous section is performed, the adjacent lower neighbourhood of the decomposed node needs to be determined. Assume that the split is performed with respect to node $r \in S$, such that the DAG G_r can be reached. All the adjacent lower neighbours of node r can be found in this DAG, and we observe that we only need to search through the nodes $(\Lambda, u) \in \mathcal{B}_r$ where $|\Lambda| = 2$ to establish L_r. This is because the definition of $\mathcal{B}$, see (16), and therefore also the definition of $\mathcal{B}_r$, requires all elements of L_r to be present at this level.

When L_r is established we need to make the DAG representing the set $\mathcal{P}(L_r)$. This is done by first making a new root node $\emptyset$, followed by the creation of all the $(\Lambda, u) \in \mathcal{P}(L_r)$ where $|\Lambda| = 1$. Next we make all nodes where $|\Lambda| = 2$, and so on until $|\Lambda| = |L_r|$ or the process is stopped by the approximations discussed in Section 5.2. However, the implemented approximation process differs from the one in Section 5.2 on one point. If a $\Delta\beta$-value is calculated, then the node is always made and added to DAG regardless of its value. This could result in a DAG with more nodes than suggested in (16). The DAG is implemented this way to ease implementation, and because it in worst case scenario leads to an equally accurate algorithm. However, if all the children of a potential node (Λ, u) have interaction parameters close to 0, that is $|\Delta\beta(A, v)| < \varepsilon \ \forall \ A \subset \Lambda$, $|\Lambda \setminus A| = 1$ and $(\chi(\Lambda, u))_A = v$, then $\Delta\beta(\Lambda, u)$ is not calculated and the node is not made, as discussed in Section 5.2.

When adding a node to a DAG, we need to locate all its children and make pointers to them. This is an easy task if all the children already exist, as they did when we made the initial DAG for $\pi(z)$. However, when growing the DAG representing $\mathcal{P}(L_r)$, there is a possibility that a desired child does not exist in the DAG. In that case we calculate the correct $\Delta\beta$-value for this child and add it to the DAG.

6.4 Union of two DAGs

Our next task is to find the union of the DAG G_{-r} and the DAG representing $\mathcal{P}(L_r)$. We have chosen to do this by traversing all the nodes in the DAG representing $\mathcal{P}(L_r)$, starting with the nodes that have no parents. Several outcomes are now possible for such a node (Λ, u). If the node also exist in G_{-r}, we only need to sum the two interaction parameters and store the result in G_{-r}. If the node does not exist in this DAG, there are two possible cases to consider. If the $\Delta\beta$-value is too small, that is if $|\Delta\beta(\Lambda, u)| < \varepsilon$, we do nothing. However, if $|\Delta\beta(\Lambda, u)| \geq \varepsilon$ we make a new node with $\beta(\Lambda, u) = \Delta\beta(\Lambda, u)$ and add it to G_{-r}. In that case we have to locate all the children of this node, and again there is a possibility that one or more of these children do not exist. Such nodes must be created, and in this case their interaction parameters are set to 0. Alternatively we could have calculated the interaction parameters of such nodes in order of higher accuracy. Setting the interaction parameters to 0 in this case ease implementation a lot, but it is also in the interest of saving CPU-time.

After finding the union of these two DAGs, we have to prune the DAG representing $\tilde{\mathcal{B}}_*$ according to (16). This is done by a recursive search through the nodes without any parents, and such a node (Λ, u) is deleted if $|\beta(\Lambda, u)| < \varepsilon$ as usual. Here we carefully correct the list of the parents of the deleted nodes' children. Finally this gives us the DAG G_*. Note that by not performing this pruning process, we instantly save CPU time. However, when not performing this process, we get a larger DAG which again results in an increase in the CPU time later in the algorithm. We have learned through our implementation that the difference between pruning and not pruning is very small. We choose to perform the pruning, while Tjelmeland & Austad (2010) chose not to in their implementation of the binary case.

6.5 Simulation

Implementing the functions needed to simulate from the decomposed system is rather straight forward. All the conditional DAGs, including the DAG representing $\pi(z_{t(n)})$ are, as described earlier, saved in a binary file. From this file we have to find and read the needed DAGs to simulate from $\pi(z)$. First we read the DAG representing $\pi(z_{t(n)})$ and simulate from it. When this is done, the DAG is deleted from the memory. We go on by reading the binary file to simulate from the distributions $\pi(z_{t(k)}|z_{t(l)}, l = k+1, ...n)$ for $k = 1, ..., n-1$. Each of the DAGs representing these distributions is of course also deleted from the memory after they are used. To do this simulation we first have to compute the normalizing constant c. Generally this constant becomes very small, so instead we compute $\ln(c)$. Likewise, we always compute $\ln(\pi(z_{t(n)}))\ \forall\ z_{t(n)} \in \mathcal{Z}$ and $\ln(\pi(z_{t(k)}|z_{t(l)}, l = k+1, ...n))\ \forall\ z_{t(k)} \in \mathcal{Z}$ when we simulate. We do this to avoid numerical problems in the computer.

6.6 Calculating the marginals

Finally we need to implement the recursive formulas for calculation of the marginal distributions, see Section 4.3.2. This is done by reading the binary file in the same way as in the previous section. However, using the forward decomposition, the sums in (24) and (25) can not be directly evaluated for DAGs read from the binary file. This is because the property that allows us to represent a (Λ, u)-pair by just two *int*s and a sorted list of children, see Section 6.1, does not allow a node k_2 to be decomposed before a node k_1 if $k_1 < k_2$. When calculating the product of the two DAGs in (24), we therefore need to relabel the nodes such that node n becomes node 1, node $n-1$ becomes node 2, and so on. This way the sums can be evaluated by a forward decomposition.

When implementing the recursive formulas from Section 4.3.2, we make no attempt to calculate $U_k \; \forall \; k \in S$, see (24). Instead we calculate

$$r = \max_{k \in S} \left[\max_{l \in L_k}(l) - k \right],$$

from the forward part of our algorithm. Using this r, we can choose $A_k = \{k, ..., n\}$ for $k = n, ..., n-r$ and $A_k = \{k, ..., k+r\}$ for $k = n-r-1, ..., 1$ to carry out the calculation of the marginal distributions. The calculation of r is easily implemented, and we note that for a lattice where the nodes are decomposed from the system in lexicographical order, this procedure gives us the optimal set $A_k \; \forall \; k \in S$ as discussed in Section 4.3.2. Note, however, that for a Markov random field defined on a general graph this procedure may not give us the optimal sets. Also note that different labelling of the nodes in a graph may lead to different values of r, and calculations of the marginals are more efficient for lower values of r than for higher values of r.

7 Results

In this section we present some results concerning our implemented algorithm. In our investigations we always assume a Potts model or a generalized Potts model defined on a lattice with a first order neighbourhood system. Also, we always decompose the nodes from the lattice in lexicographical order. First of all we look at some realisations from our forward-backward algorithm for different values on the parameters in the two models. Secondly we assume a Potts model, and investigate the CPU time used by our algorithm for different values of α and for different values of the approximation parameter ε. Next we evaluate the approximations by the procedures presented in Section 5.3, and observe how the approximations effect the adjacent lower neighbourhoods. In Section 7.6 we show some results of using our algorithm to estimate an underlying picture in a Bayesian setting.

7.1 Some realisations

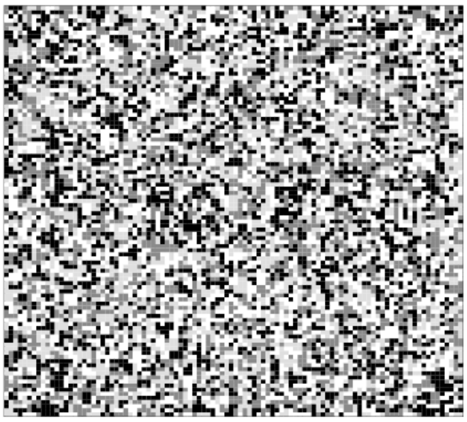
In this section we look at some realisations provided by our forward-backward algorithm. We do this to get an impression of the effect of different choices on the parameters in the Potts model and in the generalized Potts model. In the examples of this section we assume our Markov random field to be defined on a 100×100 lattice. First we assume a Potts model with $\Omega = \{0, 1, 2, 3\}$. Figure 10 shows realisations for four different values of α and with $\varepsilon = 0.001$. We clearly see how increasing α gives an increasing level of clustering of the values in Ω. Note, however, that the qualities of these realisations are unknown at this point. We will come back to the evaluation of this in Section 7.3.

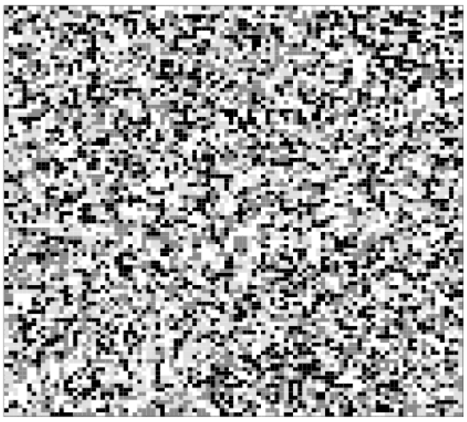
Let us assume a generalized Potts model with $\Omega = \{0, 1, 2\}$, $\alpha(0, 1) = \alpha(1, 2) = 0.8$, and $\alpha(0, 2) = 0.2$, defined on a 100×100 lattice. By choosing a high value on $\alpha(0, 1)$ and $\alpha(1, 2)$, compared to $\alpha(0, 2)$, we make it unlikely to find value 1 as a neighbour to the values 0 and 2. This is because such combinations result in a high contribution to the sums in (4). Since these three values are the only values in Ω, we expect value 1 to be rare in realisations from this distribution. In Figure 11a we see a realisation from this situation when running our algorithm with $\varepsilon = 0.0001$. We clearly see the effect discussed above, and if we approximate the marginals by (24) and (25), we find that the maximum value of $\pi(z_k = 1)$ for all $k \in S$ is 0.173. However, we still have some clustering of the value 1 in this figure. This is because it is equally unlikely to find the values 0 and 2 in the neighbourhood of the value 1, as it is finding value 1 in the neighbourhood of the values 0 and 2.

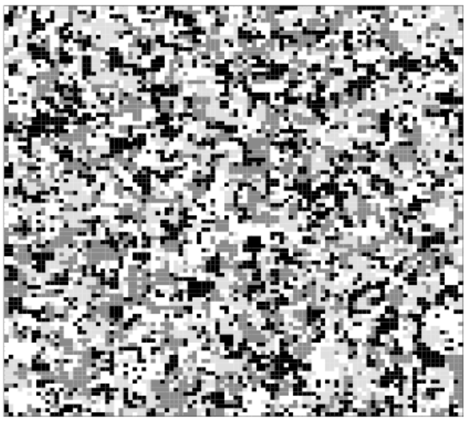
In our last example we assume that $\Omega = \{0, 1, 2\}$, $\alpha(0, 1) = \alpha(1, 2) = 0.8$, and $\alpha(0, 2) = 2$ for a generalized Potts model in the same situation as above. By this choice of parameters we get a high contribution to the sums in (4) when the values 0 and 2 are neighbours. This should result in realisations where the values 0 and 2 are unlikely to be close to each other. A realisation from this distribution is shown in Figure 11b. If we closely investigate this realisation, we see that it is dominated by transitions between the values 1 and 2 and the values 0 and 1, while transitions between the values 0 and 2 are rare. However, the clustering effect is still present.

7.2 CPU times

In this section we investigate the CPU time used by our approximate algorithm. We assume a Potts model with $\Omega = \{0, 1, 2, 3\}$ defined on a 100×100 lattice. From (3) we calculate that the critical temperature for this situation is $\alpha_c = 1.0986$. As discussed in Section 2.2.1, and observed in the previous section, the clustering of the values in Ω increases as α approaches and exceeds this critical temperature. In Table 1 we see the CPU time used by the forward part of our algorithm for five different values of α and five different value of ε. After the forward part is completed we make 1000 realisations by the backward part of our algorithm. The


(a) Potts model with $\alpha = 0.4$


(b) Potts model with $\alpha = 0.6$


(c) Potts model with $\alpha = 0.8$

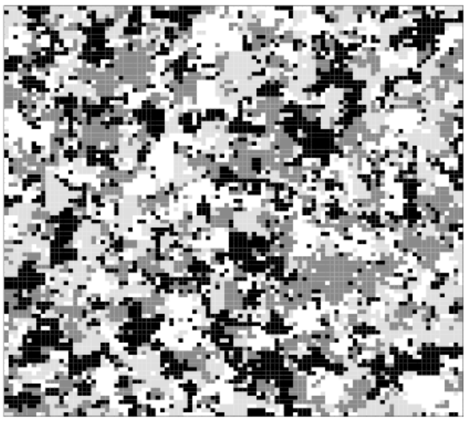
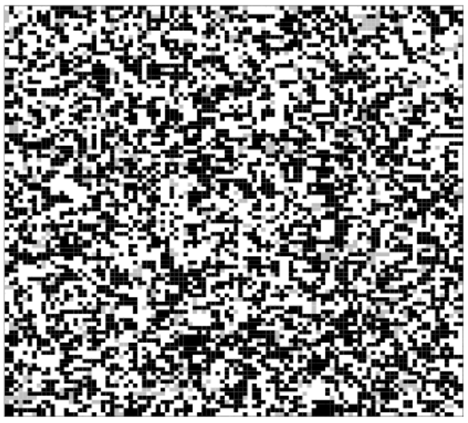

(d) Potts model with $\alpha = 1.0$

Figure 10: Realisations from four different Potts models with $\varepsilon = 0.001$ and $\Omega = \{0, 1, 2, 3\}$. White=0 and black=3.


(a) Generalized Potts model with low probability to draw the value 1.

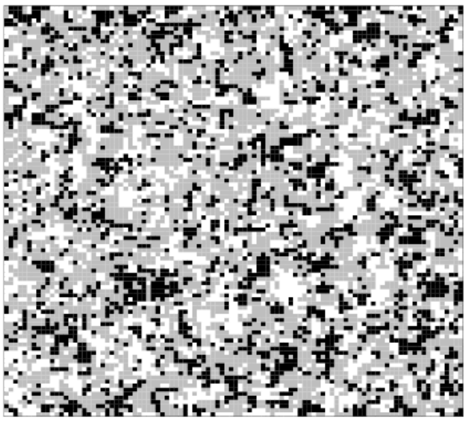

(b) Generalized Potts model with low probability to draw the values 0 and 2 close to each other.

Figure 11: Realisations from two generalized Potts models with $\varepsilon = 0.0001$ and $\Omega = \{0, 1, 2\}$. White=0 and black=2

Table 1: Time to complete decomposition for a Potts model defined on a 100×100 lattice with $\Omega = \{0, 1, 2, 3\}$.

$\varepsilon \setminus \alpha$	0.4	0.6	0.8	0.9	1.0
10^{-1}	180	187	193	195	197
10^{-2}	192	196	245	498	825
10^{-3}	234	466	2294	12518	333157
10^{-4}	476	1611	69636	926342	NA
10^{-5}	1674	21809	1223719	NA	NA

Table 2: Average time to make one realisation from a Potts model defined on a 100×100 lattice with $\Omega = \{0, 1, 2, 3\}$.

$\varepsilon \setminus \alpha$	0.4	0.6	0.8	0.9	1.0
10^{-1}	0.0129	0.0159	0.0170	0.0185	0.0174
10^{-2}	0.0176	0.0194	0.0328	0.0682	0.1019
10^{-3}	0.0284	0.0610	0.1976	0.7202	8.36018
10^{-4}	0.0597	0.1423	1.8891	13.084	NA
10^{-5}	0.1345	0.7539	16.510	NA	NA

average CPU time for making one realisation is shown in Table 2.

We clearly see from these two tables that the CPU time increase as a function of decreasing ε and increasing α. This is expected because as ε decreases we get closer to our exact algorithm, which means that the number of nodes in the different DAGs, and thereby also the number of calculations needed, increases. As discussed above, an increasing α means more clustering of the different values in Ω. In other words, we make the interactions between the nodes stronger, and less interaction parameters will be approximated to 0 for a given value of ε. For the three most extreme cases in Table 1 the decomposition was impossible to complete within reasonable time, and no realisations could be simulated because of this. In the next section we investigate the quality of the approximations done by our approximate algorithm.

7.3 Quality of the approximations

Let us again assume a Potts model defined on a 100×100 lattice with $\Omega = \{0, 1, 2, 3\}$. In the previous section we saw how the CPU time needed to run our approximate algorithm increased as a function of decreasing ε and increasing α. However, we are also interested in knowing the quality of the approximations done by the algorithm. This can be done as discussed in Section 5.3. We choose

$M = 1000$ in (32), and plot the estimated acceptance rate as a function of ε. If we do this for five different values of α, we get the result shown in Figure 12. This figure shows us how the estimated acceptance rates are increasing as ε is decreasing, that is, as we are approaching the exact algorithm. We also see that as α increases, the quality of the realisations are decreasing for a given value of ε. As already discussed, as α approaches the critical temperature, we need the value of ε to be closer and closer to 0 in order to maintain the same quality on the approximations. Combining this with the information in Table 1 and 2, we see that as α increases, it gets harder and harder to get good approximations. For instance we obtain $\widehat{acc} = 0.999$ in about thirty minutes when running our algorithm for $\alpha = 0.4$ with $\varepsilon = 0.00001$, but for the case where $\alpha = 1.0$ we only manage to run our algorithm with $\varepsilon = 0.001$ to get $\widehat{acc} = 0.022$, and this calculation took almost 4 days to complete.

7.3.1 Comparing realisations visually

If the estimated acceptance rate $\widehat{acc}$ is low, it means that the approximate distribution $\tilde{\pi}(z)$ is in fact very different from the desired distribution $\pi(z)$. Exactly how these two distributions are different from each other is an interesting question. This motivates us to visually compare realisations from a distribution $\tilde{\pi}(z)$, with low $\widehat{acc}$, and realisations from $\pi(z)$. We obtain the desired realisations from $\pi(z)$ using a single-site Gibbs algorithm (Gamerman & Lopes 2006). Investigating Potts models this way, we spotted a difference between the estimated ratio of the value 0 and the estimated ratios of the values in $\mathcal{Z}$. Remember that $\mathcal{Z} = \Omega \setminus \{0\}$, and that we expect all these estimated ratios to become equal to each other as M increases in the exact case, see Section 4.3.2. Table 3 shows the estimated ratios of the values in Ω for Potts models with $\Omega = \{0, 1, 2, 3\}$. In this table we clearly see that the ratio of the value 0 becomes different from the ratios of the values in $\mathcal{Z}$ when $\widehat{acc}$ is low. This is a result of treating the value 0 differently than the other values in Ω in the approximate algorithm. Because we are not discriminating between the values in $\mathcal{Z}$, we expect the estimated ratios of these values to become equal to each other, for any value of ε, as M increases. No other differences between realisations from $\tilde{\pi}(z)$, with low $\widehat{acc}$, and realisations from $\pi(z)$ could be spotted by us.

7.4 Conditional probabilities

As discussed in Section 5.3, the conditional probabilities $\pi(z_k = i | z_{L_k}) \ \forall \ k \in S, \ i \in \Omega$ are easily accessible. In this section we investigate the error that arises in these probabilities because of the approximations that are done in the forward part of our algorithm. First we look at how two probabilities that are supposed

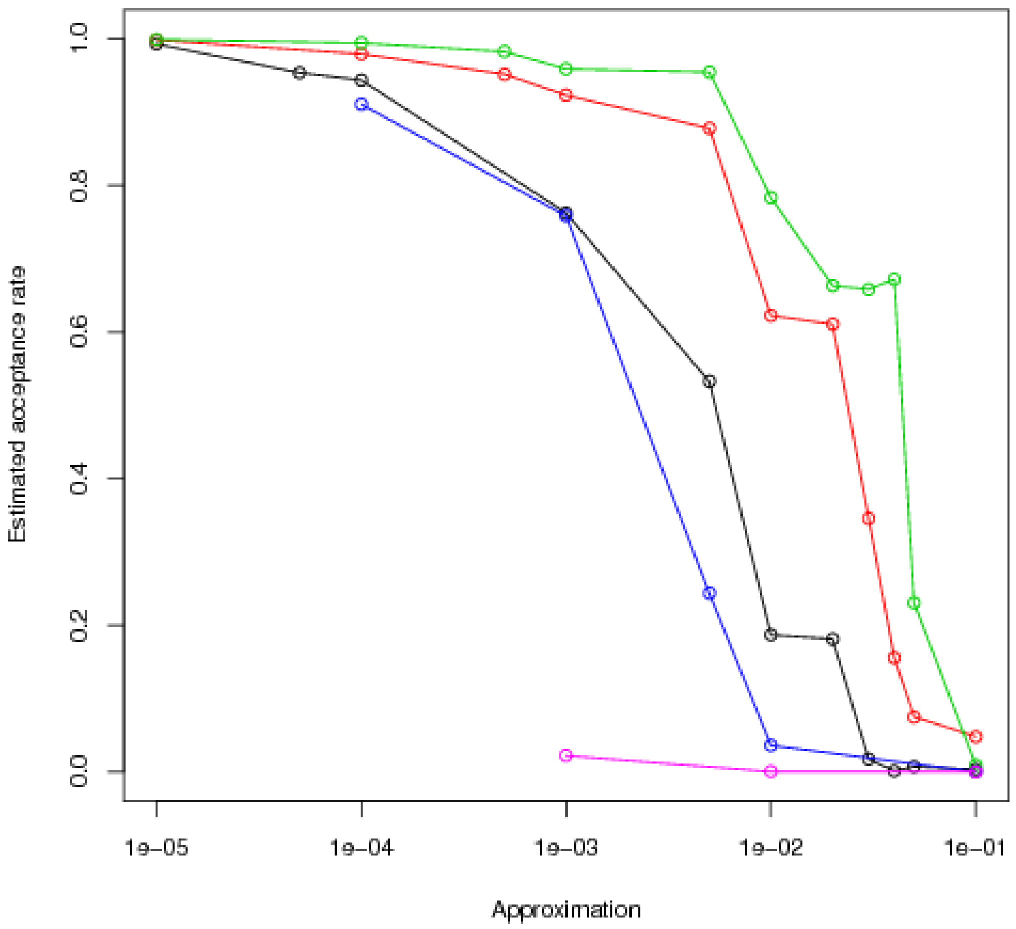


Figure 12: Estimated acceptance rates for Potts models defined on a 100×100 lattice with the following values of α: 1.0 (pink), 0.9 (blue), 0.8 (black), 0.6 (red), 0.4 (green)

Table 3: Estimated ratios of the different values in $\Omega = \{0, 1, 2, 3\}$ from $M = 1000$ realisations. All calculations are done for Potts models defined on a 100×100 lattice.

$\alpha(\hat{acc})$	0	1	2	3
1(2.2%)	0.2186	0.2604	0.2612	0.2597
0.9(3.6%)	0.2736	0.2420	0.2423	0.2421
0.9(75.8%)	0.2527	0.2495	0.2490	0.2490

to be equal becomes unequal because of the discrimination of the value 0 in the approximate algorithm. Secondly we use the conditional probabilities to observe how an error done in one node develops through the rest of the decomposition. These two investigations will provide us with a better understanding of the error that arises because of the approximations.

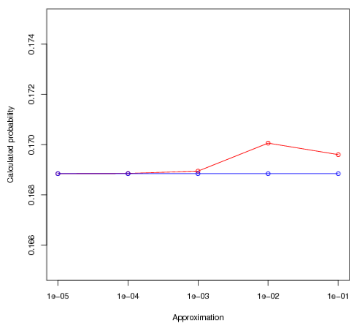
7.4.1 Error because of the approximations

In the following we assume a Potts model, and investigate the equal probabilities $p_1 = \pi(z_k = 1|z_{k+1} = 2, z_{L_k\setminus\{k+1\}} = \mathbf{0})$ and $p_2 = \pi(z_k = 1|z_{k+1} = 0, z_{L_k\setminus\{k+1\}} = \mathbf{2})$ as discussed in Section 5.3. We look at a 100×100 lattice with $\Omega = \{0, 1, 2\}$ and assume that $\alpha = 0.6$ for this situation. If we run our algorithm for 5 different values of ε and calculate the desired probabilities for $k = 50, 260, 4800, 4950, 7575, 9950$, we get the result shown in Figure 13. This figure shows that the two probabilities become unequal as a result of the approximations. Note that the calculations for the three interior nodes, $k = 260, 4950, 7575$, give essentially the same result. This stability in the error is a very important property because it shows us that the error does not increase as a function of k. We can conclude that the total error late in the decomposition not necessarily is larger than the total error earlier in the decomposition. Because of boundary effects, the error is different for the three boundary nodes.

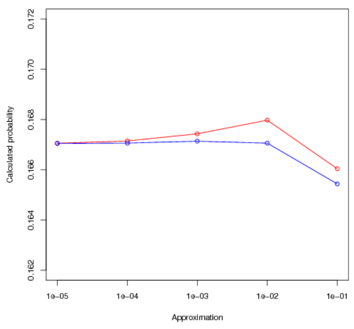
The calculation of p_2 involves the highest number of interaction parameters, compared to p_1. That is, more of the indicator functions are equal to 1 in the energy function for this probability. Because of this, p_2 is the probability that often seems to differ more from the true value. This is again a product of treating the value 0 differently than the rest of the values in Ω. Note that, for instance, the probability $p_3 = \pi(z_k = 2|z_{k+1} = 0, z_{L_k\setminus\{k+1\}} = \mathbf{1})$ will follow the curves of p_2 in Figure 13 exactly because there is no discrimination between the values 1 and 2 in the approximations.

7.4.2 Error development

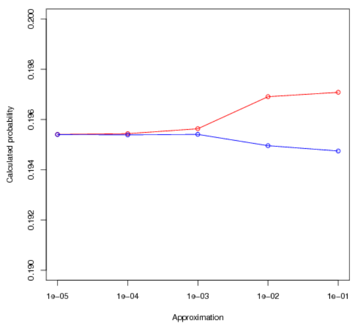
The stability in the error discovered in the previous section motivates an investigation of how an error done in one node develops through the rest of the decomposition, see Section 5.3. The focus in this section will be on the conditional probability $p_2 = \pi(z_k = 1|z_{k+1} = 0, z_{N_k\setminus\{k+1\}} = \mathbf{2})\ \forall\ k \in S$. To investigate the development of an error done in node $l \in S$ we only do approximations when l is decomposed from the system. In the rest of the nodes we use our exact algorithm. We look at two different Potts models in this section. In the first case



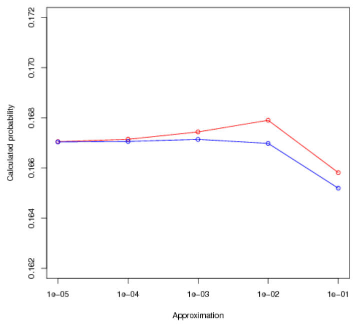
(a) Node $k = 50$.



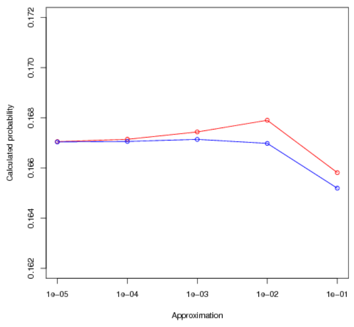
(b) Node $k = 260$.



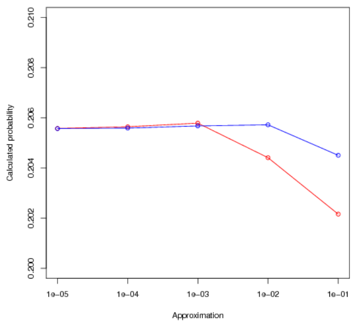
(c) Node $k = 4800$.



(d) Node $k = 4950$.

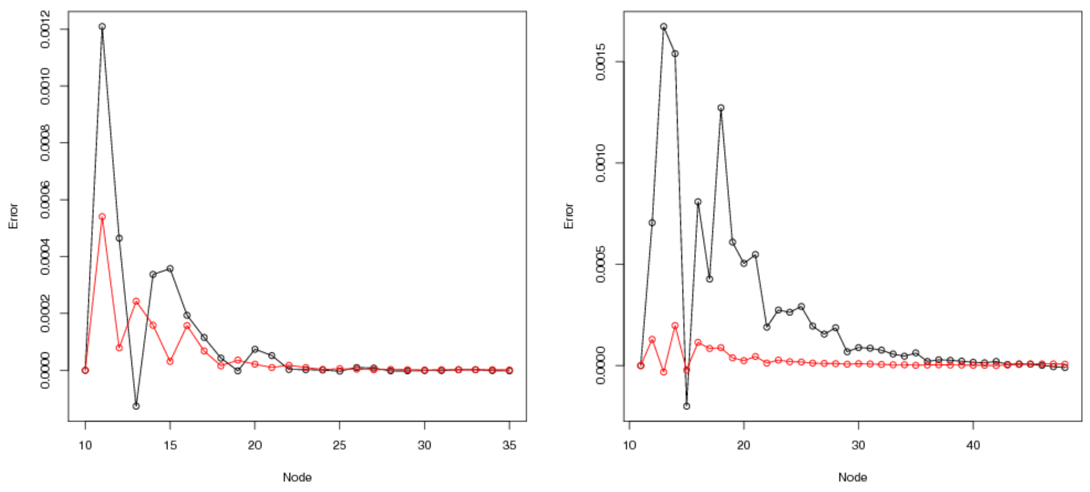


(e) Node $k = 7575$.



(f) Node $k = 9950$.

Figure 13: The approximate conditional probabilities p_1 (blue) and p_2 (red) for different values of $k \in S$ for a Potts model with $\alpha = 0.6$ and $\Omega = \{0, 1, 2\}$.



(a) A 6×6 lattice with approximations done in node $l = 10$ when $\alpha = 0.6$ and $\Omega = \{0, 1, 2\}$.

(b) 7×7 lattice with approximations done in node $l = 11$ when $\alpha = 1$ and $\Omega = \{0, 1, 2, 3\}$.

Figure 14: Error in the conditional probability p_2 with $\varepsilon = 0.001$ (red) and $\varepsilon = 0.01$ (black) for two different Potts models.

we have a 6×6 lattice where $\alpha = 0.6$ and $\Omega = \{0, 1, 2\}$, and in the second case we have a 7×7 lattice where $\alpha = 1.0$ and $\Omega = \{0, 1, 2, 3\}$. The approximations will be done in node 10 and 11, respectively. If we plot the error done in p_2 as a function of k, we get the result shown in Figure 14. These two plots clearly show how the approximations done in node l result in smaller and smaller error as the decomposition goes on. This reduction somewhat explains the stability discussed in the previous section. However, the reduction is not enough to imply stability alone, but if we, for instance, assume the errors to be additive, we can say that if the reduction of the errors is strong enough, then we obtain the stability. As discussed in the previous section this is a very important property. For instance, when estimating an underlying picture in a Bayesian setting, see Section 4.4, we want the estimate for a node late in the decomposition to be as accurate as the estimate for a node early in the decomposition.

The calculations in this section were done for two different values of ε, see Figure 14. In this figure we see how the most accurate decomposition, $\varepsilon = 0.001$, results in a smaller error than in the case with $\varepsilon = 0.01$.

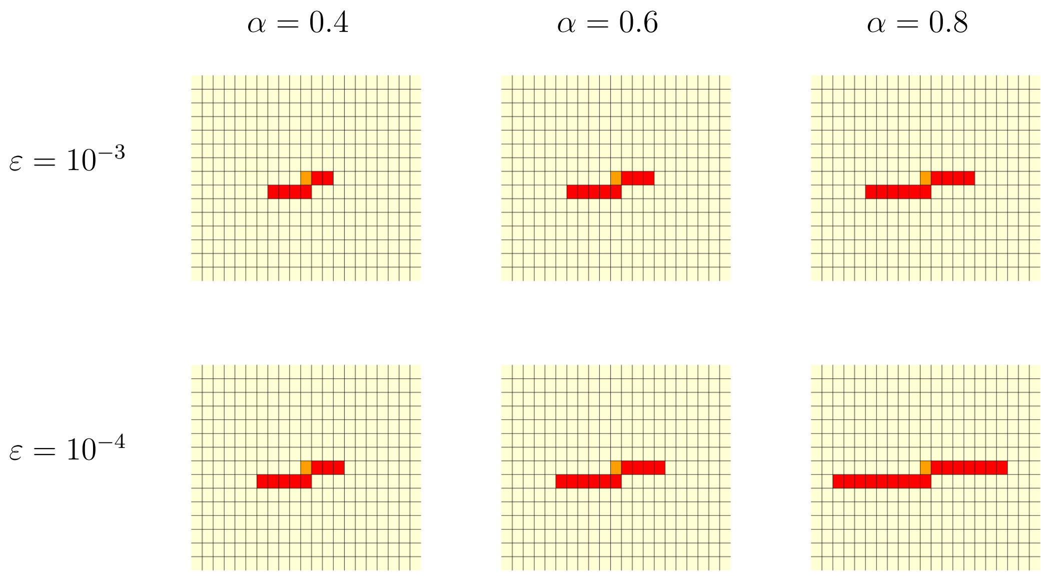


Figure 15: The adjacent lower neighbourhood of a node k that is not close to the boundary of the lattice for a Potts model. The node k is orange while its neighbours are red.

7.5 The adjacent lower neighbourhoods

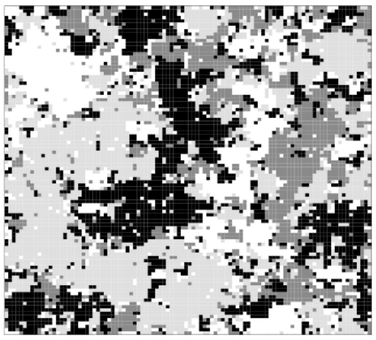
Let us assume a Potts model defined on a 100×100 lattice with $\Omega = \{0, 1, 2, 3\}$. Before the decomposition of the nodes we have a first order neighbourhood system, but as the decomposition is carried forward we obtain the adjacent lower neighbourhoods $L_k \; \forall \; k \in S$. See Section 5.1 for the discussion concerning this. In this section we investigate the size of L_k as a function α and ε. If we plot L_k for a node $k \in S$ that are not close to the boundary of the lattice, we get the result shown in Figure 15. In this figure we see how L_k becomes larger with increasing α, and smaller with increasing ε. This is also as expected because with increasing α, and decreasing ε, fewer interaction parameters are approximated to 0. As discussed in Section 5.1, an exact decomposition of the nodes in this situation would give cliques and adjacent lower neighbourhoods of size 100. However, when for instance running our approximate algorithm with $\varepsilon = 0.0001$ for a Potts model with $\alpha = 0.8$, we are able to reduce these extreme sets with 84 nodes, still getting an estimated acceptance rate $\widehat{acc} \approx 0.94$, see Figure 12.

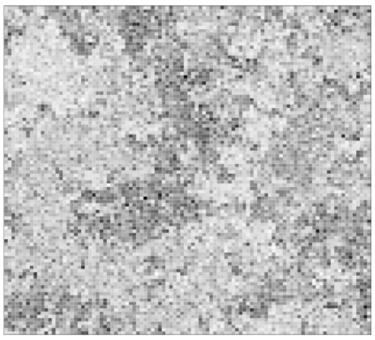
7.6 Bayesian calculations

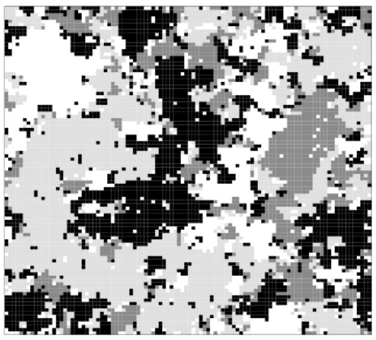
As discussed in Section 4.4, our algorithm can be used to estimate an underlying picture z from an observed degenerated version y of z. In this section we present some results of this procedure when using the Bayesian model given in Section 4.4.1. That is, with a Potts model or a generalized Potts model as a prior distribution, and with a likelihood model that provides independent Gaussian noise to the nodes in the lattice. All our examples will be on a 100×100 lattice with a first order neighbourhood system.

Assume Figure 16a to be an unobserved underlying picture z. This picture is generated from a Potts model with $\alpha = 1.1$ and $\Omega = \{0, 1, 2, 3\}$, using a single site Gibbs algorithm. Adding Gaussian noise with $\sigma = 1.0$ according to (29), we get the result shown in Figure 16b. This will be our observed picture y, and the goal is now to estimate z using this picture and the posterior distribution. For this example we choose a Potts model with $\alpha = 1.0$ and $\Omega = \{0, 1, 2, 3\}$ as our prior distribution, and we run the forward part of our algorithm with $\varepsilon = 0.001$. The marginals were first estimated using 1000 realisations from the approximate posterior distribution $\tilde{\pi}(z|y)$. Secondly the marginals were approximated using the same value on ε as in the forward part. The marginal posterior mode estimates for these two approaches are shown in Figure 16c and 16d, respectively. As we can see, the results are almost identical. When estimating the marginals, 75.5 % of the nodes were assigned the right value, compared to 75.4 % when the marginals were approximated. However, the CPU time used to approximate the marginals was a lot higher than the CPU time used to estimate them. Estimation is therefore the preferred technique in this situation.

As a second example, we use the hand drawn picture in Figure 17a as our underlying picture z. This picture is degenerate with $\sigma = 1.0$, see Figure 17b, and with $\sigma = 1.5$, see Figure 17c. If we closely inspect Figure 17a, we see that there are few transitions between the values 1 and 3 in this picture. Therefore we choose a generalized Potts model with $\Omega = \{0, 1, 2, 3\}$, and with $\alpha(0, 1) = \alpha(0, 2) = \alpha(0, 3) = \alpha(1, 2) = \alpha(2, 3) = 1.0$ and $\alpha(1, 3) = 1.3$ as our prior model in these two situations. With this choice of parameters we make it harder for the algorithm to assign the values 1 and 3 to sites close to each other, compared to the other possible transitions, see Section 7.1. We run our algorithm with $\varepsilon = 0.001$, and estimate the marginals from $M = 1000$ realisations. The results from finding the marginal posterior mode estimates for these two situations are shown in Figure 17d and 17e. For the case with $\sigma = 1.0$ the algorithm assigns correct value to 90.3 % of the nodes, while in the case with $\sigma = 1.5$ we get 81.5 % correct assigned values. As expected the result is better for the less degenerated example, and we


(a) Underlying picture: A realisation from a Potts model with $\alpha = 1.1$.


(b) Observed picture: Independent Gaussian noise with $\sigma = 1.0$ added to each of the nodes in the underlying picture.


(c) Restored picture when the marginals where estimated from 1000 realisations. Correct classified nodes: 75.5 %.

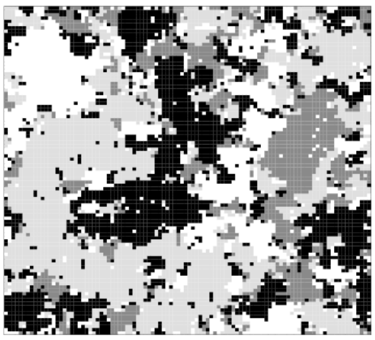

(d) Restored picture when the marginals where approximated with $\varepsilon = 0.001$. Correct classified nodes: 75.4 %.

Figure 16: Restoration of a picture simulated from a Potts model with $\alpha = 1.1$ and $\Omega = \{0, 1, 2, 3\}$. A Potts model with $\alpha = 1.0$ was used a prior model, and the decomposition was done with $\varepsilon = 0.001$. White=0 and black=3.

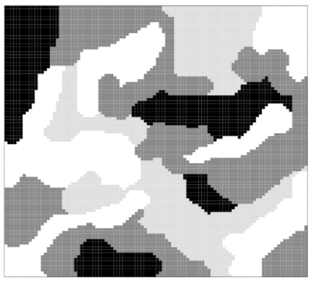
can clearly see this difference when comparing the two restored pictures visually. For instance, in the top left corner of Figure 17a we have a light gray area on a white background. In the case with $\sigma = 1.0$ this area is somewhat restored, but in the case with $\sigma = 1.5$ this area is almost completely gone. In these two restored pictures we clearly see the effect of choosing $\alpha(1,3) = 1.3$ as very few such transitions exist in the estimates. Even the few transitions that did exist between these two values in the original picture are separated by the other values in Ω. As discussed in Section 2.2.2 the generalized Potts model enables us to model such information into the prior distribution.

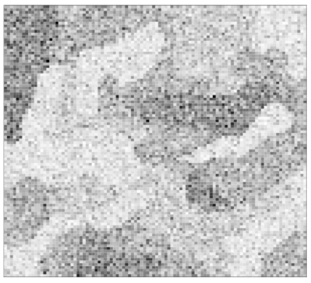
8 Closing remarks

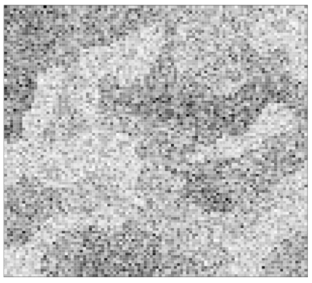
In this thesis we presented an approximate recursive forward-backward algorithm for calculations of discrete Markov random fields defined on graphs. This work is a generalization of Tjelmeland & Austad (2010), and it is a continuing of the work done in Arnesen (2009). Through the backward part of our algorithm we were able to simulate from the probability distribution of a discrete Markov random field. We also presented an alternative backward part that enabled us to approximate the marginal distributions for all the nodes in the graph.

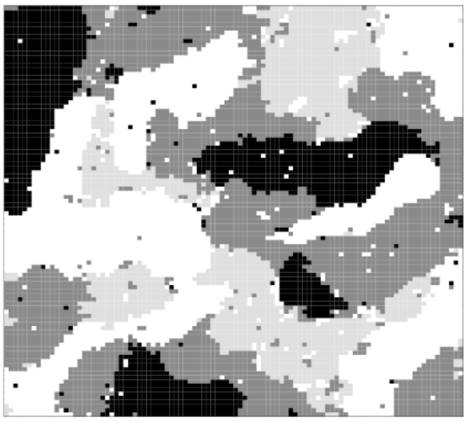
We started out by expressing the probability distribution of a Markov random field in terms of an energy function. This energy function was a sum of interaction parameters between different subsets of the nodes in the graph. Next we showed how to represent the probability distribution as a vertex-weighted DAG. The weights in this DAG representation were defined to be the interaction parameters. In the forward part of our forward-backward algorithm we managed to represent the probability distribution as a product of conditional distributions. We recursively calculated one DAG for each of these distributions. In this forward part we introduced approximations to the calculations. These approximations enabled us to run our algorithm for more complex problems than we could have evaluated with our exact version of the algorithm. The level of approximations was controlled by a parameter $\varepsilon > 0$, and setting $\varepsilon = 0$ gave us our exact algorithm.

We implemented our algorithm using the programming language C. The performance and accuracy of our implemented algorithm were evaluated by the CPU time, by estimating the acceptance rate in a Metropolis-Hasting algorithm, and by investigation of conditional probabilities. We learned that the CPU time increased, and that the quality of the approximations decreased, as a function of decreasing ε and stronger interactions between the nodes in the graph. Because of this, it became harder and harder to establish good approximations when the


(a) Underlying picture: A hand drawn picture.


(b) Observed picture: Independent Gaussian noise with $\sigma = 1.0$ added to each of the nodes in the underlying picture.


(c) Observed picture: Independent Gaussian noise with $\sigma = 1.5$ added to each of the nodes in the underlying picture.


(d) Restored picture in the situation with $\sigma = 1.0$. Correct classified nodes: 90.3 %.

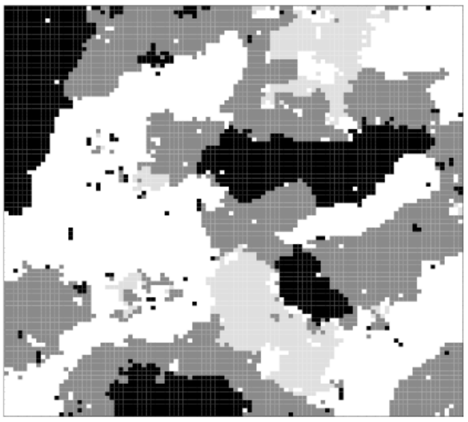

(e) Restored picture in the situation with $\sigma = 1.5$. Correct classified nodes: 81.5 %.

Figure 17: Restoration of a hand drawn picture. A generalized Potts model with $\alpha(0,1) = \alpha(0,2) = \alpha(0,3) = \alpha(1,2) = \alpha(2,3) = 1$ and $\alpha(1,3) = 1.3$ was used as a prior model, and the decomposition was done with $\varepsilon = 0.001$. White=0 and black=3.

interactions between the nodes increased. In the algorithm we treat the value 0 differently than the rest of the values in Ω, and therefore this value is also treated differently by the approximations. This property was investigated by evaluating the conditional probabilities for different values of ε. As a last evaluation of the algorithm, we saw how an error done in one node, as a result of the approximations, was developed through the rest of the decomposition. We observed how this error decreased as we continued to run our exact version of the algorithm. We concluded that this is a very important property of our approximate algorithm, because it means that the total error late in the decomposition not necessary is larger than the total error early in the decomposition.

We only investigated our algorithm for lattices with a first order neighbourhood system. As a continuing of this work, it is natural to look at other neighbourhood systems as well, for instance a second order neighbourhood system. Also, we only decomposed the nodes in our lattices in lexicographical order. An interesting exercise would be to find a better, or may be an optimal way, of decomposing the nodes. All our examples were done on lattices and not on more general graphs, even though the algorithm, and the implementation, was designed to handle them. An investigation of the performance and accuracy of the algorithm on more general graphs could also be interesting.

When we introduced our approximations to the exact algorithm we assumed that higher order interactions were smaller than lower order interactions. Tjelmeland & Austad (2010) checked this assumption for frequently used binary Markov random fields, and they concluded that is was good. However, we have in the discrete case done no such formal investigation, although it certainly could have been done by running our exact algorithm and comparing the values of the interaction parameters for different orders of interaction.

Because the marginal distributions are available through approximate calculations and estimation, we chose to use our algorithm to estimate an unknown underlying picture in a Bayesian setting, using the marginal posterior mode estimate. We gave three examples, all with independent Gaussian noise as a degeneration process. Here a lot more work can be done. For instance we can add blur to the degeneration process, use other priors than the generalized Potts with a first order neighbourhood system, and use other estimates for the underlying picture (Geman & Geman 1984). Also running our algorithm on more realistic data would have been interesting.

Another suggestion for further work is estimating the parameters, for instance the

α-parameter in a Potts model, from a given graph, using our approximate algorithm. That is, for a given graph z with probability distribution $\pi(z|\theta)$, where θ is the set of parameters, we wish to find an estimate $\hat{\theta}$ for θ, for instance the maximum likelihood estimator. If a degenerated picture y, from an assumed distribution $\pi(y|z,\theta)$, is observed instead of the underlying picture z, we can define a Bayesian problem. By adopting a prior $p(x|\theta)$ for x and a prior $p(\theta)$ for θ, we can try to estimate θ, and may be also x, from the posterior distribution using our approximate algorithm. These types of parameter estimation problems for Markov random fields are also widely discussed in the literature, see for instance Besag (1974) or Friel & Rue (2007).

Exact calculations of Markov random fields are very limited by an intractable normalizing constant. Our recursive forward-backward algorithm allows approximate calculations of discrete Markov random fields defined on graphs. By these approximations much more complex problems can be evaluate than in the exact case.

References

Arnesen, P. (2009). Recursive algorithm for exact and approximate simulation of discrete Markov random fields, *Project in norwegian*, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway.

Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems, *Journal of the Royal Statistical Society. Series B (Methodological)* **36**(2): 192–236.

Besag, J. (1986). On the statistic analysis of dirty pictures, *Journal of the Royal Statistical Society. Series B (Methodological)* **48**(3): 259–302.

Casella, G. & Robert, C. P. (1999). *Monte Carlo Statistical Methods*, first edn, Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA.

Clifford, P. (1990). Markov random fields in statistics, *in* G. Grimmett & D. J. Welsh (eds), *Disorder in Physical Systems, A Volume in Honour of John M.Hammersley*, Oxford University Press.

Cressie, N. & Davidson, J. L. (1998). Image analysis with partially ordered Markov models, *Computational Statistics and Data Analysis* **29**(1): 1–26.

Friel, N. & Rue, H. (2007). Recursive computing and simulation-free inference for general factorizable models, *Biometrika* **94**(3): 661–672.

Gamerman, D. & Lopes, H. F. (2006). *Markov Chain Monte Carlo*, second edn, Chapman & Hall/CRC, Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742.

Geman, S. & Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images, *IEEE Trans. PAMI* **6**: 721–741.

Hurn, M., Husby, O. & Rue, H. (2003). A tutorial on image analysis, *in* J. Møller (ed.), *Spatial statistics and computational methods*, Vol. 173 of *Lecture Notes in Statistics*, Springer Verlag, pp. 87–141.

Kindermann, R. & Snell, J. L. (1980). *Markov Random Fields and Their Applications*, American Mathematical Society, Providence, Rhode Island.

Reeves, R. & Pettitt, A. (2004). Efficient recursions for general factorisable models, *Biometrika* **91**(3): 751–757.

Scott, S. L. (2002). Bayesian methods for hidden Markov models; recursive computing in the 21st century, *Journal of the American Statistical Association* **97**: 337–351.

Tjelmeland, H. & Austad, H. M. (2010). Exact and approximate recursive calculations for binary Markov random fields defined on graphs, *Technical report, statistics 2/2010*, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway.

Wu, F. Y. (1982). The Potts model, *Reviews of Modern Physics* **54**(1): 235–268.