BSC Artificial Intelligence Thesis: Sampler convergence for inference of structural brain networks

Wieke Kanters
0610151
kantersw@gmail.com

Supervisors:
Max Hinne & Marcel van Gerven

November 22, 2013

Chapter 1

Introduction

Modern neuroscience often assumes that the mind is modular, meaning that our minds consist of a collection of semi-distinct modules each with its own function. Understanding how these modules are interconnected is essential to understanding how the brain works as a whole and could potentially yield valuable insight into psychological disorders and neurological diseases, for example clinical studies have already shown abnormalities in the brain networks of people with Alzheimer's or schizophrenia [2].

Both functional and structural brain networks are used to describe how the different areas within the brain are interconnected. Functional brain networks are based on the temporal correlation of brain activity, when a neuron is activated a signal is sent through its axon potentially spreading the activation to the neurons connected to it, so it is reasonable to assume that two areas in the brain are connected (possibly indirectly) if they show the same activation pattern. Structural brain networks, however, are based on the physical structure of the brain.

The basic structure of the brain is as follows: for the purpose of this paper there are two parts that are truly important, the parts that perform actual cognition [11] (consisting of *grey matter*) and the parts through which the gray matter is interconnected (consisting of *white matter*). Grey matter mainly consists of neurons (including dendrites and unmyelinated axons[1]), glia cells[2] and capillaries most of the grey matter can be found in the cortex (outer surface of the brain), the cerebellum (found at the lower back of the brain), thalamus and basal ganglia (both found at the center of the brain). White matter mainly consists of thick bundles of myelinated axons, called tracts, these provide the connections between the areas of grey matter, as a result most of the white matter resides in an area around the center of the brain (where the thalamus and basal ganglia are) and underneath the cortex.

At the moment one of the only ways we are able to, indirectly, observe white matter tracts in living subjects is using weighted diffusion MRI [6]. The data provided by a weighted diffusion MRI, or dMRI, consists of a three dimensional

[1]Some axons are myelinated so the signals travels faster and further. Myelin is primarily composed of lipids, which are white, hence the name white matter.

[2]The different kinds of glia cells provide nutrients, structural support, immune defence and electrical insulation to the neurons in the brain.

image where each voxel[3] contains information on the rate and direction of diffusion. Due to their internal structure white matter tracts inhibit diffusion in certain directions depending on the orientation of the tract, water will diffuse quicker along the direction of the white matter tract, which is why dMRI data can be used to infer the orientation and presence of these tracts.

The next step would be to determine what underlying structural brain network gave rise to the dMRI results. Conventional methods would, after preprocessing the dMRI data, arrive at a result by applying a simple threshold function (regions are connected if sufficient evidence for a connection can be found), yielding a single structural brain network. The actual method used in this thesis is described in a framework proposed by Hinne et al in [9], the framework proposes deriving structural brain networks from streamline data using Bayesian inference. It provides a model that describes the relationship between the dMRI data and the underlying structural brain network and yields a probability space of structural brain networks rather than just a single network.

In essence Bayesian inference is used to update the parameters of a model (in this case the parameter is the structural brain network) based on new information (in this case the dMRI data) and describes a probability space of updated parameters. In order to get actual results one will have to sample the described probability space using a class of algorithms called Markov Chain Monte Carlo Algorithms (or MCMC Algorithm), as defined by David Barber in *Bayesian Reasoning And Machine Learning* [1], for calculating it in its entirety would be utterly intractable. However depending on the number of subjects, the resolution of the data and other factors it can still take quite a while for the algorithm to yield results, assuming you even know the algorithm has finished.

The MCMC algorithms are all part of the broader class of *Monte Carlo methods*, a class of non-deterministic algorithms intended to solve problems whose domain of possible inputs is just too large to consider in its entirety and instead approximates the true result by considering only a limited random subset of inputs. The problem with this class of algorithms is knowing how large the subset of inputs you have to consider, in order to generate a worthwhile approximation, is. The general solution to this problem is to independently run multiple instances of the algorithm in parallel and monitor the difference between the results they yield, under the assumption that once the subset is large enough changing the exact input should not significantly change the approximation, this is called *Convergence Monitoring*.

Which brings us to the actual problem of this thesis: *can the time a Markov Chain Monte Carlo sampler requires to converge on a solution, when used for Bayesian inference of structural brain networks, be reduced?* The framework used for this thesis uses a *Metropolis Hastings* MCMC algorithm so this algorithm will be used as a baseline and be compared to a couple of variant algorithms, the *Small World* MCMC, the *Shotgun Stochastic Search* and *Simulated Annealing*. Because these variant algorithms are intended to yield results significantly different from the Metropolis-Hastings algorithm, the secondary problem also addressed in this thesis is: *how do the results of these different algorithms compare to one another?*

[3]A volumetric pixel, or voxel, is the 3d analogy of a pixel.

Chapter 2

Background

I will start by discussing the background of this project, Bayesian inference the method used by the framework of Hinne et al to infer a structural brain network, The model of the framework itself, how it relates to Bayesian inference and Markov Chains a mathematical concept used in the Markov Chain Monte Carlo algorithms.

2.1 Bayesian Inference

Bayes Theorem, named after Thomas Bayes (1701 - 1761) who was the first to suggest it, is used in probability theory to determine probability of the parameters of a model in light of new information (often interpreted as confidence instead of probability) [1]. In Bayesian theory this, the probability of the parameters of a model given certain new information, is called the *posterior*, in the specific case of this project the posterior is the probability of a specific brain network given the dMRI data. The probability of parameters of the model without taking new information into account is called the *prior*.

Say one has a model with parameters θ which can be used to determine the probability of an event x, the posterior would be $p(\theta|x)$ and the prior $p(\theta)$, Bayes theorem defines the relationship between these two as shown in equation (2.1). The two two other elements used in Bayes theorem are $p(x)$ the total probability of event x called the *marginal likelihood* and $p(x|\theta)$ the probability of event x given parameters θ called the *likelihood*.

$$p(\theta|x) = \frac{p(x|\theta)}{p(x)} p(\theta) \tag{2.1}$$

Bayes theorem can be derived using the definition of *conditional probability*[1] as follows:

$$\begin{array}{lll}
p(\theta|x) & = \frac{p(\theta,x)}{p(x)} & \text{if } p(x) \neq 0 \\
p(x|\theta) & = \frac{p(\theta,x)}{p(\theta)} & \text{if } p(\theta) \neq 0 \\
p(\theta,x) & = p(x|\theta)p(\theta) & \text{if } p(\theta) \neq 0 \\
p(\theta|x) & = \frac{p(\theta,x)}{p(x)} = \frac{p(x|\theta)p(\theta)}{p(x)} & \text{if } p(\theta) \neq 0
\end{array}$$

[1] $p(A|B) = \frac{p(A,B)}{p(B)}$

The issue with Bayes theorem is the marginal likelihood $p(x)$ because if the value of $p(x)$ is not already known it is going to have to be calculated which is likely to be computationally expensive or downright impossible. In a worst case scenario the marginal likelihood $p(x)$ would be determined using the law of total probability[2], but this would require integration over the entire parameter space from which θ is drawn, see (2.2), if this space is too large this integral may be difficult to obtain.

$$p(\theta|x) = \frac{p(x|\theta)}{\int p(x|\theta)p(\theta)d\theta} p(\theta) \tag{2.2}$$

However one does not necessarily need to divide $p(x|\theta)p(\theta)$ by $p(x)$ in order for the result to be useful. After all the value of $p(x)$ is independent from θ, even if we do not divide it by $p(x)$ $p(x|\theta)p(\theta)$ will still be in proportion to the posterior and therefore still useful as a comparative measure. Alternatively, when considering the distribution of $p(\theta|x)$, one could interpret $p(x)^{-1}$ as an normalization constant and $p(x|\theta)p(\theta)$ as a posterior distribution that has not been normalized.

Bayesian inference is used to infer (or update) the parameters of a model based on observed evidence. This can be achieved by calculating the entire *posterior distribution* and then use it to determine the optimal parameters. However calculating the entire posterior distribution is usually computationally prohibitive, as the parameter space is likely to be rather large. In order to circumvent this issue one can employ a sampling algorithm to get an approximation of the distribution and use that to determine the optimal parameters instead. The class of sampling algorithms most often used for Bayesian inference are the *Markov Chain Monte Carlo* algorithms, often shortened to MCMC algorithms, these algorithms are capable of sampling non normalized probability distributions [1, p. 598].

2.2 The Model

In order to use Bayesian inference to infer a structural brain network from streamline data one will need a definition for the likelihood and prior. In this case the model for the likelihood and prior has been provided by the framework proposed by Hinne et al [9].

The posterior, or the updated prior, is defined as (2.3), where the parameter N is the streamline data, A is the structural brain network defined as an adjacency matrix and a^+, a^- and p are the model's hyper parameters[3], whose exact definitions will be provided later. Note how (2.3) is equivalent to Bayes theorem without the normalization constant, as previously mentioned in 2.1 a probability distribution can be sampled by a MCMC algorithm even if the distribution is not normalized.

$$P(A|N, a^+, a^-, p) \propto P(N|A, a^+, a^-)P(A|p) \tag{2.3}$$

The prior $P(A|p)$, is based on the Erdös-Rényi model of random graphs described in [5]. It is a simple prior that defines the probability of the adjacency

[2] $p(y) = \int p(y|x)p(x)dx$

[3] Hyper parameters are parameters of the distribution of parameters.

matrix as the product of the probability of each edge (2.4), a present edge has probability p, an absent edge has probability $(1 - p)$.

$$P(A|p) = \prod_{i<j} p^{a_{ij}}(1-p)^{1-a_{ij}} \tag{2.4}$$

The likelihood $P(N|A, a^+, a^-)$ is a bit more complex, it is the probability of a particular distribution of streamlines N given adjacency matrix A and a couple of hyper parameters and requires the combination of two different probability distributions (a multinomial and Dirichlet distribution, as discussed below).

The drawing of streamlines, of a single voxel n_{ij}, is modelled by (2.5) a multinomial distribution. Like the multinomial distribution, the process of drawing voxels, consists of multiple trials (multiple streamlines are drawn), has results that belong to a fixed number of categories (the non-starting point voxels) and each category has a fixed probability (based on dMRI data that does not change while streamlines are drawn). The number of streamlines drawn between two voxels i and j is n_{ij} and the probability of streamlines drawn between i and j is x_{ij}

$$P(n_i|x_i) \propto \prod_{j=1}^{K} x_{ij}^{n_{ij}} \tag{2.5}$$

However the multinomial distribution requires a probability vector x_i, a vector that corresponds with the probability that a streamline will be drawn between two particular voxels. This vector is based on the structural brain network, so the probabilities reflect the structure of the brain. In the framework this is achieved by using (2.6), a *Dirichlet* distribution.

$$P(x_i|a_i, a^+, a^-) \propto \prod_{j=1}^{K} x_{ij}^{a_{ij}a^+ + (1-a_{ij})a^- - 1} \tag{2.6}$$

The shape of the Dirichlet distribution is determined by its parameters, in this case the hyper parameters a^+ and a^-. These parameters reflect the probability of a streamline being drawn between two voxels if a corresponding edge is present (a^+) or absent (a^-) in the adjacency matrix (remember this adjacency matrix represents a structural brain network). Due to the values of a^+ and a^- that were used ($a^+ = 1$ and $a^- = 0.5$) the Dirichlet distribution favours probability vectors which assign a low probability to edges that do not actually exist in the structural brain network, the probability of edges that do exist are irrelevant for in those cases the exponent in equation (2.6) will always equal 0.

Both the multinomial distribution (2.5) and the Dirichlet distribution (2.6) have a normalization constant which is being left out because the MCMC algorithm does not require normalized distributions. Both are also just defined for connectivity of a single area, or row/column in the adjacency or streamline matrices. However the combined probabilities are easily calculated, let $P(N|X)$ and $P(X|A, a^+, a^-)$ be the combined probabilities.

Finally the likelihood $P(N|A, a^+, a^-)$ is defined using the law of total probability[4] by integrating the product of the multinomial (2.5) and Dirichlet (2.6)

[4] $p(y) = \int p(y|x)p(x)dx$

distributions over X. The result of this integral is called a *Dirichlet compound multinomial distribution*, a probability distribution that is used often in Bayesian inference.

$$P(N|A, a^+, a^-) = \int P(N|X)P(X|A, a^+, a^-)dX \tag{2.7}$$

In short the structural brain network A is combined with a Dirichlet distribution (2.6) to define a space of hidden probabilities X which is combined with a multinomial distribution (2.5) to determine the probability of the streamline data N. The probability of the streamline data N given adjacency matrix A (2.7) is multiplied by a flat prior (2.4) the result of which is *in proportion* to the probability of the structural brain network A given the streamline data N (2.3), also known as the *posterior*.

2.3 Markov Chains

A Markov chain considers a system that undergoes random transitions between a finite number of states, represented as a sequence of states, and holds to the assumption of *conditional independence* [1, p. 489]. This means that each state of a Markov chain is only dependent on a fixed number of states preceding it. This number of preceding states is referred to as the *order L* of a Markov chain:

$$p(s_n|s_1, \ldots, s_{n-1}) = p(s_n|s_{n-L}, \ldots, s_{n-1}) \tag{2.8}$$

So a Markov chain of order 0 is completely independent of its past states, a Markov chain of order 1 is only dependent on its preceding state, and so forth.

The *marginal distribution* of a Markov chain is the distribution over states at a certain time. It can be interpreted as the relative frequency with which each state has occurred within the sequence, and is defined as follows with n being the time:

$$p(s_n = i) = \sum_j p(s_n = i|s_{n-1} = j)p(s_{n-1} = j) \tag{2.9}$$

For some Markov chains, as the time approaches infinity, the marginal distribution at that time will become independent of the initial distribution, this is referred to as the *equilibrium distribution*:

$$p(s_\infty) = p(s_\infty|s_1) \tag{2.10}$$

Chapter 3

Data and Methods

Next I will show the data used with this particular problem and the methods used in this thesis. The methods include four different algorithms and a performance measure called convergence monitoring, I will also show how the optimal parameters for each algorithm were determined.

The algorithms covered are the Metropolis-Hastings algorithm, which forms the baseline to which the others are compared, and the Small World, Shotgun Stochastic Search and Simulated Annealing algorithms, which are all variants of the former. In addition to the definition of each algorithm I will also show a number of visualisations based on a synthetic dataset in order to further clarify their behaviour

3.1 Streamline Data

The data used for this project was provide by Max Hinne, primary author of *"Bayesian inference of structural brain networks"* [9], and was obtained and preprocessed by the methods as described therein.

A streamline represents a possible connection between two regions of the brain via a white matter tract. Each streamline starts at a seed voxel, a voxel on the boundary between grey and white matter, and is drawn from voxel to voxel based on the diffusion direction, the length of a streamline is limited to a certain number of steps. A large number of streamlines is drawn for each seed voxel, each connecting two areas of the brain, which results in a large distribution of possible connections.

The streamline data consists of a matrix where each column and row represents an area of the brain and each cell shows the number of streamlines that connect the column and row areas. The ordering of areas 1 through 116 is arbitrary though the odd numbered areas correspond to the left hemisphere of the brain and the even numbers correspond to the right hemisphere, some areas such as the cerebellum (lower right in Figure 3.1) were grouped.

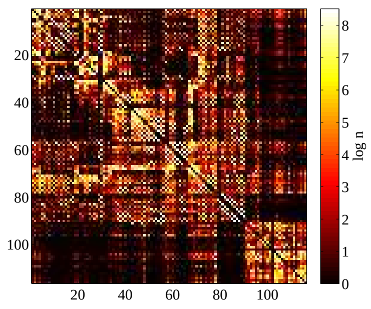

Figure 3.1: Image based on streamline data. The colour is based on the logarithm of the number of streamlines between two areas.

3.2 Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm algorithm, part of the class of algorithms known as *Markov Chain Monte Carlo* algorithms, is an algorithm for obtaining samples from a probability distribution. Like all MCMC algorithms it gathers samples by performing a *random walk* through the sample space. Such a random walk is described by a *Markov chain*, see section 2.3, where each sample corresponds with a state in the chain. The result of a MCMC algorithm is a set of samples.

MCMC algorithms are used when one needs to sample an probability distribution $p(x)$ for which normalization is considered intractable, the *Metropolis-Hastings acceptance function* describes a Markov chain which has the normalized distribution as its stationary distribution [1, p. 598]. Over time, as the algorithm progresses along the Markov chain, it will converge on the stationary distribution.

The Metropolis-Hastings algorithm requires a method to generate new samples in the Markov Chain, in our specific case it simply flips a random edge in the brain network, and a corresponding *proposal density* $\tilde{q}(x, x_p)$, which describes the transition probabilities between samples in the Markov chain. This important part is combined with the probability density function $p(x)$ of the probability distribution that needs to be sampled to form the *Metropolis-Hastings acceptance function* a , let x_p be the proposed sample and x the current sample then $a(x_p|x)$ is the probability with which proposed sample should be accepted [1, p. 599], as defined by:

$$a(x_p|x) = \min\left(1, \frac{\tilde{q}(x|x_p)p(x_p)}{\tilde{q}(x_p|x)p(x)}\right) \tag{3.1}$$

The main drawback of the Metropolis-Hastings algorithm is that it has no natural termination point. Over time it will converge to, and provide samples of, the normalized probability distribution but there is no inherent indicator of when this is occurring. Therefore it is up to the user to determine the number of iterations N the algorithm has to perform or implement a method of *convergence monitoring*, which is covered in section 3.3.

Algorithm 1 Metropolis-Hastings MCMC

```
1: Set the maximum number of iterations $I$.
2: Generate first sample $x_1$.
3: for $i = 2 \to I$ do
4:     Generate a proposal sample $x_c$.
5:     $a = \frac{\tilde{q}(x_{i-1}|x_p)p(x_p)}{\tilde{q}(x_p|x_{i-1})p(x_{i-1})}$
6:     if $a \geq 1$ then
7:         $x_i \leftarrow x_c$                                   ▷ Accept proposal.
8:     else
9:         Generate a uniform random number $r$ between 0 and 1.
10:        if $r < a$ then
11:            $x_i \leftarrow x_c$                               ▷ Accept proposal.
12:        else
13:            $x_i \leftarrow x_{i-1}$                           ▷ Reject proposal.
14:        end if
15:    end if
16: end for
```

3.3 Convergence Monitoring - Potential Scale Reduction Factor

As briefly touched on in the section 3.2 one of the issues with these algorithms is knowing when to stop.

Sampling is by its very nature a problem without a clear solution. Unless you are dealing with a normalized or highly structured probability distribution, in which case you should not be using a MCMC algorithm, there will be multiple solutions to the problem (different sets of samples that are all representative of the target distribution). And without comparing multiple sets of samples to each other (or a single set to the entire space, though that is likely to be intractable) you have no clear indication of how representative your samples are.

As discussed earlier regardless of the starting point within the sample space the algorithm will, over time, converge on the target distribution (2.10). If multiple instances of the algorithm are run in parallel, each with different starting points, over time the samples returned each step by the instances should be drawn from the same distribution. For while the random walks start at different points they will all, given enough iterations, converge on the target distribution. By performing a statistical analysis one can determine how probable it is that the sets of samples are being drawn from the same area in the distribution, this probability can be used as a measure of convergence. The process of determining whether or not the algorithm has converged upon the target distribution is called *convergence monitoring*.

The specific method of convergence monitoring used here is the *Potential Scale Reduction Factor* discussed by Brooks and Gelman [3, p. 436]. It is based on analysis of variance models in statistics (commonly referred to as ANOVA). It determines how similar the sets are by comparing the variance within each set with an estimation of the true variance.

The between variance B is defined as (3.2), a measure for the difference between the sets of samples.

$$B = \frac{n}{m-1}\sum_{j=1}^{m}(\bar{\psi}_{j\cdot} - \bar{\psi}_{\cdot\cdot})^2 \tag{3.2}$$

The within variance W is defined as (3.3), a measure for the difference within each set of samples.

$$W = \frac{1}{m(n-1)}\sum_{j=1}^{m}\sum_{t=1}^{n}(\psi_{jt} - \bar{\psi}_{j\cdot})^2 \tag{3.3}$$

Both between and within variance are calculated using ψ_{jt} a scalar summary of sample t of set j, a scalar summary represents the sample in such a way that it can be used to determine a mathematically sensible mean and variance, it however does not have to be the actual sample. The bar denotes an average so $\bar{\psi}_{j\cdot}$ is the average scalar summary of set j and $\bar{\psi}_{\cdot\cdot}$ is the average over all sets. The variables m and n represent the number of chains and the number of samples respectively.

The true variance $\hat{\sigma}^2_+$ as defined by (3.4), an overestimation of within the *entire* sample space. Note that if the starting samples of the algorithms are not sufficiently dispersed throughout the sample space $\hat{\sigma}^2_+$ will be too low and this can possibly lead to a false indication of convergence [3, p.437]. If the random walks start in the same place they will be similar from the start, rather than becoming more similar as the algorithm converges.

$$\hat{\sigma}^2_+ = \frac{n-1}{n}W + \frac{B}{n} \tag{3.4}$$

Two of these variances, the within and total variance, are used to calculate the Potential Scale Reduction Factor $\hat{R}$:

$$\hat{R} = \frac{m+1}{m}\frac{\hat{\sigma}^2_+}{W} - \frac{n-1}{mn} \tag{3.5}$$

As the algorithm converges, $\hat{R}$ will converge to 1, if the algorithm is not converged $\hat{R}$ will be greater than 1. An often used threshold is 1.1. Early in the random walk $\hat{R}$ is high because the samples of all the walks taken together will be fairly dispersed (leading to a high $\hat{\sigma}^2_+$) but the samples of each individual walk will be close together (leading to a low W). As the algorithm progresses, until it converges, each step will increase the W and decrease $\hat{\sigma}^2_+$ because individually the random walks are becoming more different (each step within the chain will be drawn from a different area of the distribution) but collectively they are becoming more alike (because all chains will be moving towards the target distribution).

3.4 Alternate Algorithms

3.4.1 Small World MCMC

The first modification of the MCMC algorithm I am going to cover is, the Small World MCMC algorithm as defined by Guan et al in their paper *Markov Chain Monte Carlo in small worlds* [7].

As we know the probability of the MCMC algorithm accepting a new sample is equal to its acceptance ratio, which is 1 if the new sample has a higher probability than the current sample and lower than 1 if it does not, this means that the algorithm has the tendency to move towards the peaks within the sampled probability distribution. This behaviour will not be a problem if the probability distribution is unimodal[1], with a smooth slope leading to the mode, however if the probability distribution is multimodal[2] the algorithm can get stuck in one of the peaks causing other peaks to be ignored, which would skew the distribution of samples. This problem is further exacerbated if the proposal distribution is rather flat tailed and only really considers samples that differ little from the current sample.

In order to resolve this issue Guan et al [7] propose to change the proposal distribution to a more heavy-tailed distribution. In essence they propose to turn the sample space into a *Small World network*, a network where one can travel between two random nodes in a small number of steps, changing the proposal distribution and method in such a way that the chance of moving from one random sample to another random non-adjacent sample increases dramatically. The proposal distribution is expanded to include *"jump"* proposals, which would connect two distant areas in the proposal distribution, the probability of proposing a jump should be small and the jump should move the algorithm far away from the current sample.

The basic concept of the Metropolis-Hastings algorithm is easily modified to include jumps, see Algorithm 2 lines 6 through 10. However the specific implementation can be considerably harder. In their paper Guan et al [7, p. 3] offer a method to determine what the jump probability and jump size should be, but the methods require one to know certain properties of the sample space, for example how large the areas around the mode are compared to the rest of the space and how far the modes lie apart. Given that my knowledge of the shape probability space is limited, and determining the shape is likely to be intractable, I can not use these methods. Instead, in section 3.6.2, a grid search will be used to determine the optimal parameters for the algorithm.

[1]A probability distribution with a single peak.

[2]A probability distribution with multiple peaks.

Algorithm 2 Metropolis-Hastings with Small World proposals

```
1: Set the maximum number of iterations $I$.
2: Set the jump chance $J$.
3: Generate first sample $x_1$.
4: for $i = 2 \to I$ do
5:     Generate a uniform random number $r$ between 0 and 1.
6:     if $r < J$ then
7:         Generate a jump proposal sample $x_c$.
8:     else
9:         Generate a normal proposal sample $x_c$.
10:    end if
11:    $a = \frac{\tilde{q}(x_{i-1}|x_p)p(x_p)}{\tilde{q}(x_p|x_{i-1})p(x_{i-1})}$
12:    if $a \geq 1$ then
13:        $x_i \leftarrow x_c$                          ▷ Accept proposal.
14:    else
15:        Generate a uniform random number $r$ between 0 and 1.
16:        if $r < a$ then
17:            $x_i \leftarrow x_c$                      ▷ Accept proposal.
18:        else
19:            $x_i \leftarrow x_{i-1}$                  ▷ Reject proposal.
20:        end if
21:    end if
22: end for
```

3.4.2 Shotgun Stochastic Search MCMC

The Metropolis-Hastings algorithm can, for certain problems, require a large amount of iterations to converge on a solution. Which, depending on processing speed, can result in processing times in the order of days or weeks. An often used strategy to optimise an algorithm is to try to redefine the algorithm so large parts of it run in parallel. But the nature of Markov Chains does not allow this since each step in the chain depends on the step before it you can not calculate steps out of order. At best one could parallelize the processes within each iteration, the parts within the for-loop in Algorithm 1 starting at line 3, which if properly implemented are not that computationally expensive to begin with.

The Shotgun Stochastic Search, which I will refer to as SSS, proposed by Hans et al [8] is based on the Metropolis-Hastings algorithm and allows for greater parallelization. At the core of this modification lies the *proposal neighbourhood*. Unlike Metropolis-Hastings, which only proposes a single sample each iteration, the SSS proposes multiple samples each iteration, this collection of samples is referred to as the proposal neighbourhood. First the Metropolis-Hastings acceptance ratio a is calculated for each sample in the proposal neighbourhood, after which the sample with the highest a in the neighbourhood is determined and accepted as the new proposal with a probability of a (just as the single sample the Metropolis-Hastings algorithm proposes is also accepted with a probability equal to its acceptance ratio). The result is, as intended by Hans et al, that, on average, the sample the SSS algorithm accepts each iteration will be of significantly higher probability than the sample the Metropolis-Hastings algorithms would accept each iteration, resulting in an algorithm that quickly moves to the nearest mode of the distribution and samples a tight area around it.

Because a neighbourhood of proposals is considered instead of a single proposals, the chance that a good proposal is considered increases significantly. In order to further capitalize on this effect one should try to ensure variability within the proposal neighbourhood. In their paper Hans et al [8], while mainly considering highly dimensional problems, suggest generating the neighbourhood by applying three different types of *moves* on the current sample. The *addition* move adds a single variable, the *deletion* move removes a single variable and the *replacement* move replaces a single variable with another variable (combining a deletion and addition move). In essence this is a sampling problem, where one samples the space of proposal samples. In the specific case covered in this thesis, the derivation of structural brain networks, the addition, deletion and replacement moves either add, remove or replace a single edge in the network.

However one should keep in mind that improved parallelisation is not the main goal of the SSS algorithm, this is merely an intentional side effect of the modification. The main goal of the SSS algorithm is to quickly provide high probability samples found in a small area around the mode of the probability distribution, which is significantly different from the goal of the Metropolis-Hastings algorithm which, is to sample the entire sample space. So when using the SSS algorithm with Bayesian inference you will not get an estimate of the posterior distribution.

Each iteration of the SSS is far more computationally complex that that of the Metropolis-Hastings algorithm. But the generation and evaluation of proposals within the neighbourhood can be performed independently from one another and can therefore be run in parallel. Ideally this would alleviate the increase in runtime per iteration to the point where the total runtime is actually less than that of the Metropolis-Hastings algorithm.

Algorithm 3 Shotgun Stochastic Search

```
1: Set the maximum number of iterations $I$.
2: Set the neighbourhood size $N$.
3: Generate first sample $x_1$.
4: for $i = 2 \to I$ do
5:     Generate $N$ proposals.
6:     $h \leftarrow 0$
7:     for each $N$ as $x_p$ do
8:         $a = \frac{\tilde{q}(x_{i-1}|x_p)p(x_p)}{\tilde{q}(x_p|x_{i-1})p(x_{i-1})}$
9:         if $a > h$ then
10:            $h \leftarrow a$
11:            $x_h \leftarrow x_p$
12:        end if
13:    end for
14:    if $h \geq 1$ then
15:        $x_i \leftarrow x_h$                                  ▷ Accept proposal.
16:    else
17:        Generate a uniform random number $r$ between 0 and 1.
18:        if $r < a$ then
19:            $x_i \leftarrow x_h$                              ▷ Accept proposal.
20:        else
21:            $x_i \leftarrow x_{i-1}$                          ▷ Reject proposal.
22:        end if
23:    end if
24: end for
```

3.4.3 Simulated Annealing

The Simulated Annealing variant is a method for finding an approximation of the global maximum, it was independently suggested by both Cerny [4] and Kirkpatrick et al [10], not to provide a neat sampling of the target distribution. The results of this algorithm is, unlike the other algorithms, the final sample it returns and not a set of samples.

The Simulated Annealing algorithm was inspired by a form of heat treatment used in metallurgy, called annealing, which is used to relieve internal stress of a metal making it more malleable. In metallurgy annealing basically means one has to heat the metal to a great temperature, typically until it glows, allowing the atoms in the metal to redistribute relieving internal stress and then cool it off slowly so the stress has time to dissipate completely. This concept is applied to the Metropolis-Hastings algorithm, resulting the Simulated Annealing algorithm, by introducing a temperature variable that regulates the mobility of the algorithm. When the temperature is high, mobility is high as well, increasing the probability of accepting samples with a low acceptance ratio (enabling the algorithm to move out of a local maximum). When the temperature is low, mobility is low as well, decreasing the probability of accepting samples with a low acceptance ratio (forcing the algorithm to move up the slope of the probability distribution towards the closest mode).

The actual modification is fairly simple, see Algorithm 4. The temperature T and cooling rate c are added, instead of checking if acceptance ratio a is smaller than a random value between 0 and 1 we check if $a^{\frac{1}{t}}$ is smaller and each iteration the value of t is multiplied by the cooling rate c.

Algorithm 4 Simulated Annealing

```
1:  Set the maximum number of iterations $I$.
2:  Set the initial temperature $T$.
3:  Set the cooling rate $c$.
4:  Generate first sample $x_1$.
5:  **for** $i = 2 \rightarrow I$ **do**
6:      Generate a proposal sample $x_c$.
7:      $a = \frac{\tilde{q}(x_{i-1}|x_p)p(x_p)}{\tilde{q}(x_p|x_{i-1})p(x_{i-1})}$
8:      **if** $\min(1, a^{\frac{1}{T}}) \geq 1$ **then**
9:          $x_i \leftarrow x_c$                                  ▷ Accept proposal.
10:     **else**
11:         Generate a uniform random number $r$ between 0 and 1.
12:         **if** $r < a$ **then**
13:             $x_i \leftarrow x_c$                              ▷ Accept proposal.
14:         **else**
15:             $x_i \leftarrow x_{i-1}$                          ▷ Reject proposal.
16:         **end if**
17:     **end if**
18:     $T \leftarrow c \cdot T$
19: **end for**
```

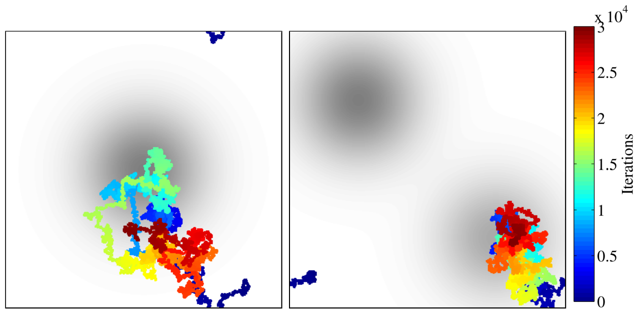

Figure 3.2: Random walk of a Metropolis-Hastings algorithm

3.5 Synthetic Dataset Simulations

The algorithms discussed above all behave quite differently from one another, and while the differences can be understood by the differences in their definition, it would be better demonstrated by the differences between the random walks they perform. However the problem of brain inference does not have a probability space that lends itself to easy visualisation, so in order to demonstrate the differences between the random walks I will be using a synthetic dataset based on a two-dimensional normal distribution.

The following simulations were performed using two different datasets. One *unimodal*, defined by:

$$u(x, y, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{x^2 - y^2}{2\sigma^2}} \tag{3.6}$$

and one *bimodal*, defined by:

$$\begin{aligned} m(x, y, \sigma) = {} & 1.5 \cdot u(x - 100, y - 100, \sigma) \\ & + u(x + 100, y + 100, \sigma) \end{aligned} \tag{3.7}$$

Both dimensions x and y are limited to integers from -200 to 200 (including bounds), with the space is shaped like the surface of a torus, so a step "up" from $y = 200$ brings you to $y = -200$. This was done so the proposal distribution is identical at each point in the space and it could be left out of the Metropolis-Hastings acceptance function. The proposal samples itself is generated using the current sample by randomly moving to an adjacent sample in the distribution (including the diagonally adjacent samples) with an equal chance for each possibility. The standard deviation σ had a value of 60 and the mean μ, while left out of the equations below, had a value of 0.

Each chain was started at a random point in the space and performed 30 000 steps.

Metropolis-Hastings

The results of the simulations of the Metropolis-Hastings algorithms are shown in Figure 3.2. While the Metropolis-Hastings algorithm is the standard to which the others are compared there is something one should take note of. Theoretically the Metropolis-Hastings algorithm should sample the entire space and

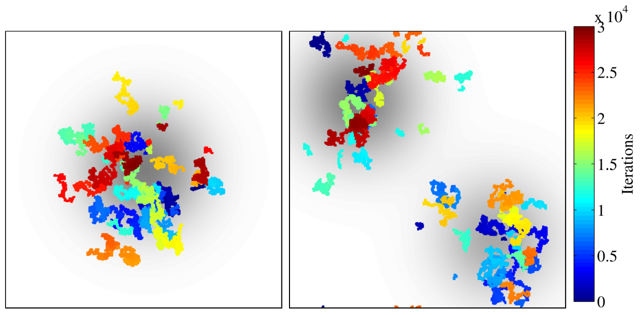

Figure 3.3: Random walk of a Small World algorithm

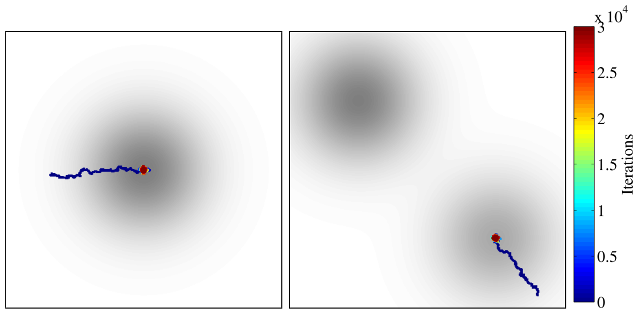

Figure 3.4: Random walk of a Shotgun Stochastic Search algorithm

should therefore be capable of moving out of an area of high probability, as shown in the unimodal simulation. however it could still get stuck in a local maximum due to the limited number of steps performed we are actually able to perform, as shown by the bimodal simulation.

Small-World

The simulations of the Small World algorithm were performed with a jump chance $p_j = 0.005$ and a jump size $s_j = 200$ the results are shown in Figure 3.3. The figures clearly show how the Small World differs from the Metropolis-Hastings algorithms. Unlike the latter the jumps enable it to cover a wider area of the probability space and is capable of moving from one mode to another, as shown by the bimodal simulation. When sampling multimodal probability distributions the Small World algorithm has a clear advantage over the Metropolis-Hastings algorithm.

Shotgun Stochastic Search

The simulations of the SSS algorithm are shown in Figure 3.4. As expected this random walk is quite different compared to the walks of the Metropolis-Hastings and Small World algorithms for unlike those two, which are intended to sample the entire space, the SSS is intended to only sample areas of the highest probability. As a result the walk of the SSS algorithm quickly moves up the slope of the distribution, getting stuck in a local maximum if the slope leads to one, and samples a small area around the mode of the distribution.

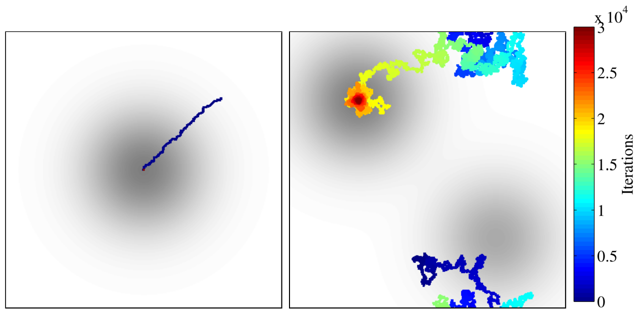

Figure 3.5: Random walk of a Simulated Annealing algorithm, using different parameters.

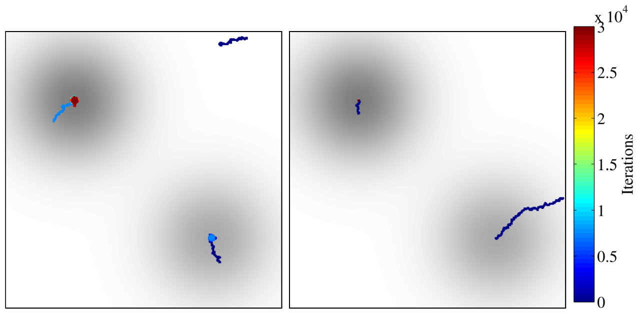

Figure 3.6: Random walk of a Shotgun Stochastic Search algorithm (left) and a Simulated Annealing algorithm (right), both using a Small World proposal distribution.

Simulated Annealing

The results of the unimodal simulation of the Simulated Annealing algorithm was performed using a coolrate $c = 0.5$ and initial temperature $T = 1$, while the bimodal simulation was performed using $c = 0.9995$ and $T = 1000$. As these simulations show, the choice of parameters greatly influences how this algorithm functions. The unimodal simulation basically skips the period of high mobility and immediately has a high preference for samples that are better than the current sample. As a result it quickly moves up the slope towards the mode of the distribution and just like the SSS algorithm it will move to whatever maximum lies at the top of the slope be it global or local. The bimodal simulation however starts with a long period of high mobility performing a random walk much like the Metropolis-Hastings's random walk, except that it has a greater tendency to accept moves towards a sample of lower probability. It should be pointed out however that the high mobility does not guarantee that the algorithm moves out of a local maximum and towards the global maximum, the reverse can also happen.

Figure 3.6 shows the result of a bimodal simulation using a SSS and Simulated Annealing algorithm both using a Small World proposal distribution with jump chance $p_j = 0.005$ and jump size $s_j = 200$. While both of them do not jump nearly as frequently as the normal Small World algorithm, what few jumps they do perform enable them to find the global mode.

3.6 Optimization

Before the performance of the different algorithms using the brain inference problem can be compared, the actual point of this entire thesis, the optimal parameters need to be determined, these parameters can have quite a large impact.

In the following the variable n_c will refer to the number of chains used, n_s will refer to the number of samples obtained, n_i will refer to the number of iterations the algorithm was allowed to run and d will refer to the density of the initial sample.

Please note that the error bars indicate the first quantile, mean and third quantile of the data.

3.6.1 Metropolis-Hastings Initial Sample

As described in section 3.2 the Metropolis-Hastings algorithm does not have any true parameters, no external variables that influence the behaviour of the algorithm. However it does require a random starting sample, as described by the framework in [9], which means we need to determine the optimal density of the initial random sample.

In order to determine the optimal density $\hat{R}$ was determined using $n_c = 12$, $n_s = 1000$, $n_i = 10^6$ and $d \in \{0, 0.1, \ldots, 1\}$, this entire process was repeated 10 times, the average psrf values were used to draw Figure 3.7.

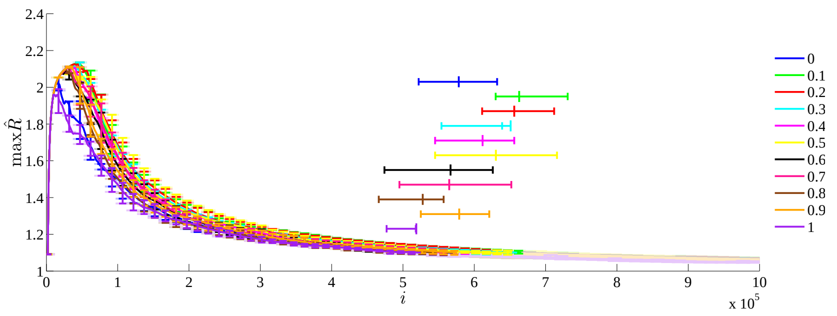

Figure 3.7: Maximum psrf value $\hat{R}$ at iteration i of the Metropolis-Hastings algorithm using different initial densities. The change in line colour and the horizontal errorbars indicate the moment of convergence.

During the first 100 000 iterations, see Figure 3.7, the chains with a density of 0 and 1 appear to have the best performance. Despite these being opposites this is not really surprising, remember that $\hat{R}$ is low when chains are sampling the same area (see section 3.3) and, unlike the other chains, these chains all start at exactly the same spot in the distribution. So this is not an actual indication of better performance but rather a result of the limitations of convergence monitoring.

A closer look at the data reveals that there is a difference of about 100 000 iterations between the number of iterations required for convergence given different starting densities, however the variability of the number of iterations required is so high that it seems improbable that there is a real effect. Which is supported by the theory behind the MCMC algorithm which states that, given sufficient iterations, the current sample being obtained should be independent of the initial sample, so the initial sample should not really matter.

So I will adopt the initial density used in the framework by Hinne et al [9], which is a density of $d = 0.5$.

3.6.2 Small World jump chance and size

As described in section 3.4.1 the Small World algorithm has two parameters jump chance and jump size. Theoretically one could determine the optimal value of these parameters mathematically (described by Guan et al in [7]) using the properties of the distribution however practically these properties are quite hard to determine, especially given the size of the sample space, so instead the optimal parameters were determined using a grid search.

In order to determine the optimal jump chance and size $\hat{R}$ was determined using $n_c = 12$, $n_s = 1000$, $n_i = 10000$ and $d = 0.5$ for each possible combination of jump chances $p_j \in \{0.05, 0.1, \ldots, 0.5\}$ and jump size $s_j \in \{0.1\%, 0.2\%, \ldots, 1\%\}$ of edges , this entire process was repeated 10 times and the average psrf values were used to draw figures 3.8, 3.10 and 3.11.

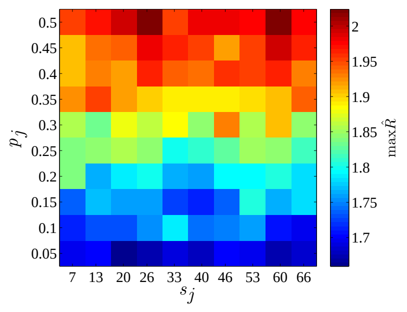

Figure 3.8: Maximum psrf value $\hat{R}$ of last iteration for jump size s_j and jump chance p_k.

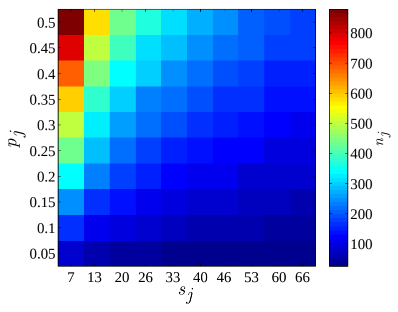

Figure 3.9: Number of accepted jumps n_j over 100 000 iterations for jump size s_j and jump chance p_k.

The results of the grid search, shown in Figure 3.8, are based on a limited number of iterations, about 50 times less than would be required for convergence, this was done because of practical considerations for a search that required actual convergence would probably have taken nearly a year to compute. However even this limited search seems to indicate an inverse linear relationship between jump chance and psrf value. This effect is shown more clearly in Figure 3.10 which is based on Figure 3.8 with the jump size dimension averaged out. Both Figure 3.8 and 3.11 do not show any relationship between jump size and psrf value.

The lack of effect of jump size seems to suggests that either jumps are accepted and somehow jump size does not matter or, a bit more plausible, no jump is accepted regardless of size. And by comparing figures 3.8 and 3.9 one can clearly see that the number of actually accepted jumps, not merely the amount of jumps proposed, has no relation to $\hat{R}$. Furthermore remember, as discussed in 3.4.1, each proposed jump comes at the cost of a normal step in the algorithm, so even if each proposed jump is rejected it will still have an adverse effect on the efficiency of the algorithm because a rejected jump still delays the algorithm by a single iteration. The inverse linear relationship between jump chance and psrf value, while no relation between size and psrf value is shown, is simply the result of the delays each proposed jump causes.

The results of the grid search seem to indicate that using $p_j = 0.05$ and any $s_j \in \{0.1\%, 0.2\%, \ldots, 1\%\}$ is optimal ($s_j = 40$, 0.6% of edges was used). When extrapolating from the results, especially Figure 3.10, one might conclude that $p_j = 0$ would be best, however jump proposals are what defines a Small World algorithm, using $p_j = 0$ would just turn it back into a Metropolis-Hastings MCMC algorithm.

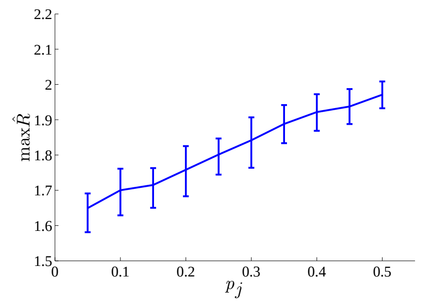

Figure 3.10: Maximum psrf $\hat{R}$ value of last iteration by jump chance p_j, based on Figure 3.8

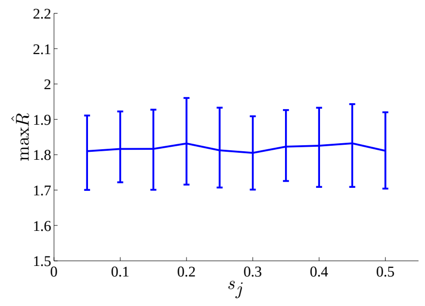

Figure 3.11: Maximum psrf $\hat{R}$ value by jump size s_j, based on Figure 3.8

3.6.3 Shotgun Stochastic Search neighbourhood distribution and size

As described in section 3.4.2 the SSS algorithm has one parameter, the composition of the neighbourhood. The most straightforward way of determining the optimal neighbourhood composition would be a 3d grid search, with number of replacement, addition and deletion moves each as a dimension, however even limiting the search to a small number of increments per dimension (say 10) would result in a impractically long search. Instead I opted to split the optimization process in two parts, first the ratio of moves and second the total size of the neighbourhood.

In order to determine the optimal neighbourhood ratio $n_c = 10$ chains of $n_i = 5 * 10^4$ iterations were generated and every 100 iterations the type of moves used during those iterations was stored as well as the density of the current sample, the neighbourhood consisted of 50 of each type of move, this was used to draw Figure 3.12. In order to determine the neighbourhood size $\hat{R}$ was determined using $n_c = 12$, $n_s = 100$, $n_i = 5 \cdot 10^5$, $d = 0.5$ for neighbourhood size $n_n \in \{2, 4, \ldots, 18\} \cup \{20, 30, \ldots, 200\}$, with equal parts deletion and addition moves (no replacement moves), this entire process was repeated 10 times the average psrf values were used to draw Figure 3.13.

The distribution of what type of move is accepted is shown in Figure 3.12, a clear pattern is established after about 3000 iterations (which is long before convergence occurs), deletion and addition moves are used in roughly equal parts, replacement moves are barely used at all while the density remains constant. The initial sample is random, with a density of 0.5 while the density of the

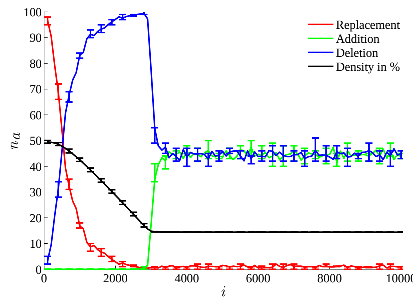

Figure 3.12: Total number of moves accepted n_a by type over the last 100 iterations and the density of the sample. Neighbourhood composed of 50 of each type.

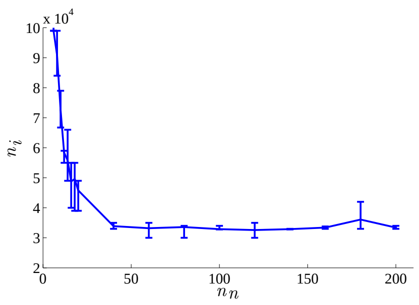

Figure 3.13: Number of iterations required for convergence n_c, $\max\hat{R} < 1.1$, by size of the neighbourhood n_n. Neighbourhood composed of deletion and addition moves only.

samples at the target distribution is about 0.16, so at first there are going to be a lot of edges that need to be removed (resulting in a lot of good potential deletion moves) and some edges that need to be added (resulting in a decent number of good addition moves). However this means that good replacements moves will also be common and the Metropolis-Hastings acceptance ratio of a good combination of two moves is far greater than either move on its own.

Figure 3.13 shows the number of iterations required for convergence using a certain neighbourhood size n_n, initially the number of iterations drop quickly as n_n increases but as n_n reaches 50 the number of iterations required for convergence levels off. The initial drop is expected for when n_n increases the neighbourhood will better represent the entire proposal distribution, so the odds of proposing one of the better moves found in the proposal distribution increases as well. However only a single good proposal is required, as only one move can be performed per iteration, once n_n is high enough that the neighbourhood reliably proposes at least one good move per iteration increasing it further is not likely to yield better results.

In order to shed some additional light on these results I ran an extra test where multiple neighbourhood proposals were generated using neighbourhood size $n_n \in \{2, 4, \ldots, 18\} \cup \{20, 30, \ldots, 200\}$, consisting of only a single move type based on either the initial random sample or the sample of the 4 000th iteration, the maximum acceptance ratio found in each neighbourhood was stored, this was repeated 1 000 times and used to draw Figure 3.14.

Regarding the distribution of move acceptance Figure 3.14 clearly shows that the moves that were proposed at the start of the chain have an acceptance ratio of an order of magnitude higher than the moves proposed after the equilibrium had been reached. Furthermore the relative quality of each move type also changes as expected, with replacement moves initially being an order of magnitude better than the rest, while they are far worse after the equilibrium has been reached.

And regarding neighbourhood size Figure 3.14 shows that while the best move in a neighbourhood does increase with neighbourhood size this trend does not last and it starts levelling off at a neighbourhood size of about 20.

The results found in Figure 3.12 and 3.13 seem to indicate that using a neighbourhood size $n_n = 50$ while only proposing addition and deletion moves is optimal, since proposed replacement moves get rarely chosen and the gains

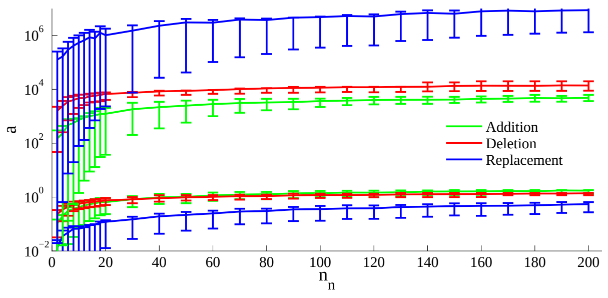

Figure 3.14: Highest Metropolis-Hastings acceptance ratio in neighbourhood of the proposal distribution of an initial sample (upper three) and a post equilibrium sample (lower three).

from increasing n_n level off at about 50.

3.6.4 Simulated Annealing cooling rate and initial temperature

As described in section 3.4.3 the Simulated Annealing has two different parameters, the *initial temperature* T and the *cooling rate* c. The initial temperature determines how high the mobility of the algorithm is at the start, while the cooling rate determines how fast this mobility drops as the algorithm progresses.

In order to determine the optimal initial temperature and cooling rate $\hat{R}$ was determined using $n_c = 12$, $n_s = 10^3$, $n_i = 10^5$ and $d = 0.5$ for each possible combination of initial temperature $T \in \{10^0, 10^1, \ldots, 10^9\}$ and cooling rate $c \in \{1 - 5 \cdot 10^{-1}, 1 - 5 \cdot 10^{-1.5}, \ldots, 1 - 5 \cdot 10^{-5}\}$, this entire process was repeated 10 times and the average psrf values were used to draw figures 3.15.

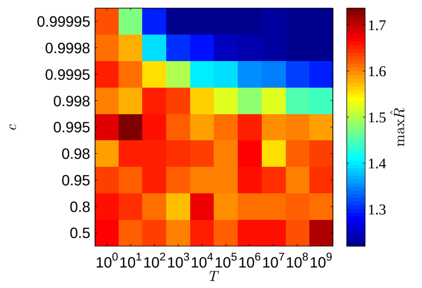

Figure 3.15: Maximum psrf value $\hat{R}$ of last iteration for initial temperature T and cooling rate c.

The results in Figure 3.15 shows an unexpected pattern, the chains with the longest period of high mobility (a high initial temperature combined with a cooling rate close to 1) seem to converge the fastest. Intuitively this is not what one would expect since high mobility should prevent the algorithm from staying in a single area and should therefore delay convergence. Just in case the grid search showed unexpected results I stored the last sample of one chain of each possible parameter combination in the grid search. The samples of most of the blue area appear to be random, while the samples corresponding to chains outside of the blue area show a pattern similar to the streamline data 3.1 and furthermore are all identical. And they had been identical for quite a number of iterations indicating that all these chains had already reached the peak of the global mode.

It is surprising however that the chains consisting of mostly random samples have such a low $\hat{R}$ value since one would expect these chains to have a higher variance than the non-random chains. However while the between and within variance of the random chains is much higher than that of the other chains the ratio $\frac{\text{between}}{\text{within}}$, which is what $\hat{R}$ actually depends on, is smaller leading to a smaller value of $\hat{R}$.

So it appears that the Simulated Annealing algorithm performs so well that it reaches the peak of the global mode long before $\hat{R}$ reaches 1.1 suggesting that this form of convergence monitoring does not suffice for this algorithm. Luckily an alternative presents it self as well. If 2 or more chains of Simulated Annealing algorithms have reached the same sample and have stopped moving, it clearly has converged.

The next test is a repeat of the grid search shown above except for the number of chains which is $n_c = 4$ and the convergence measure used. During this test convergence was monitored using the difference between samples, defined for samples a and b as (3.8), each iteration this differences was calculated for every possible combination of chains. The test was repeated 30 times, and the results were used to draw Figure 3.16.

$$\sum_{ij}(a_{ij} - b_{ij})^2 \tag{3.8}$$

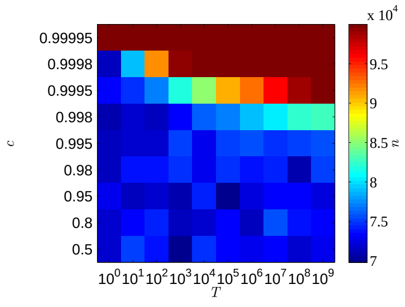

Figure 3.16: Number of iterations n required until the chains become identical for initial temperature T and cooling rate c.

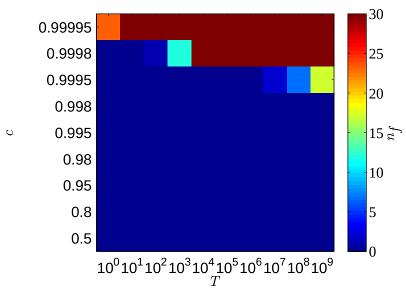

Figure 3.17: Number of times n_f, out of the 30 tests performed, that the chains failed to converge on the same sample, for initial temperature T and cooling rate c.

The results in Figure 3.16 show a similar pattern as the results in 3.15 except inverted. The chains with the cooling rates close to 1 and a high initial temperature converge slowly, or not at all within the 100 000 iterations performed during this test, while the rest of the chains seem to converge in about 75 000 iterations.

Seeing as nearly the entire first column of chains, the chains with an initial temperature of 1 which means they lack a period of high mobility, converges just as quickly on the solution as most of the other chains, it seems as if the period of mobility does not add anything to the performance of the algorithm. Furthermore, as Figure 3.17 shows, the number of times that a group of chains fails to converge n_f seems to depend only on the speed at which they converge, suggesting the probability distribution is unimodal rather than multimodal. For should the distribution have been multimodal one would expect the faster cooling groups of chains to fail occasionally by having some of its members get stuck in separate modes.

A Simulated Annealing algorithm with a sufficiently low temperature will only accept proposed moves that move it up the slope of the probability distribution, as an acceptance ratio of $a < 1$ will be drastically reduced by applying the temperature modification when the temperature is far below 1, while an acceptance ratio of $a = 1$ will remain the same regardless of the temperature. So if moving continuously up the slope of the distribution will inevitably lead one to the mode of the distribution, which seems to be the case according to Figure 3.17, it would be preferable to skip the period of high mobility and start the algorithm with a low temperature and fast cooling rate.

Given these results it seems that using a fast cooling algorithm with a low starting temperature is the optimal choice, the lowest combination of parameters tested is $T = 1$ and $c = 0.5$ so I will be using those.

Chapter 4

Results

Finally I am going to discuss the performance and results of the algorithms. The convergence speed, performance measured by the number of iterations required for a solution to be found, as indicated by the potential scaler reduction factor and the runtime required to compute the required number of iterations. Furthermore I will compare the actual results of the algorithms with one another, since these algorithms are supposed to yield different results it would be interesting to see if they each yield the kind of results they are intended to yield.

Please note that the error bars indicate the first quantile, mean and third quantile of the data.

4.1 Convergence

The convergence speed, the number of iterations required for $\hat{R}$ to reach 1.1, of the different algorithms was calculated using $n_c = 12$, $n_s = 1000$, $n_i = 10^6$, $d = 0.5$, a jump chance $p_j = 0.05$ and jump size $s_j = 40$ for the Small World algorithm, a neighbourhood size of $n_n = 50$ with only addition and deletion moves for the SSS algorithm and a initial temperature $T = 1$ and cooling rate $c = 0.5$ for the Simulated Annealing, this entire process was repeated 10 times. The average psrf values were used to draw Figure 3.13.

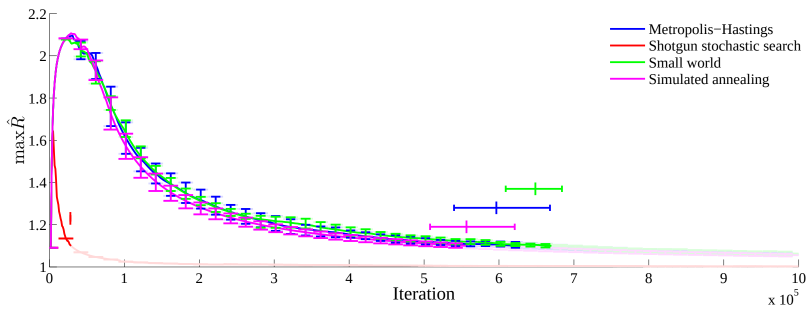

Figure 4.1: Maximum psrf value $\hat{R}$ at iteration i. The change in line colour and the horizontal errorbars indicate the moment of convergence.

Figure 4.1 shows clear results, the SSS is a lot quicker, the Small World is a bit slower than the Metropolis-Hastings algorithm and the Simulated Annealing a bit faster, the Metropolis-Hastings takes $n_{mh} = 556 \cdot 10^3$ iterations, the SSS $n_{sss} = 32 \cdot 10^3$ iterations (0.058 times that of Metropolis-Hastings), Smalls world $n_{sw} = 647 \cdot 10^3$ iterations (1.16 times that of Metropolis-Hastings) and Simulated Annealing $n_{sa} = 444 \cdot 10^3$ iterations (0.798 times that of the Metropolis-Hastings).

However as discussed in section 3.6.4 the psrf value may not be a good indication of whether or not the Simulated Annealing algorithm is finished. So as an extra measure the number of different edges between 12 Simulated Annealing chains (60 differences it total) over 100 000 iterations has been calculated and the results shown in Figure 4.2, it shows that the Simulated Annealing algorithm is finished after $n^*_{sa} = 72 \cdot 10^3$ iterations (0.127 times that of the Metropolis-Hastings).

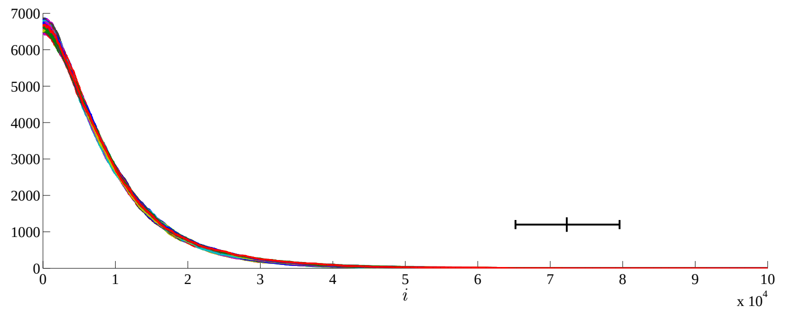

Figure 4.2: Difference between 12 chains of the Simulated Annealing algorithm. The horizontal errorbar indicates the moment the difference becomes 0.

Small World discrepancy

The difference between Small World and Metropolis-Hastings is interesting, we know that a Small World that does not accept any jumps should be similar in performance as the Metropolis-Hastings, allowing for wasted iterations due to proposed (but rejected) jumps, let n'_{sw} be an estimate of n_{sw} without jump proposals:

$$n'_{sw} = \frac{n_{sw}}{1 + p_j} = \frac{647 \cdot 10^3}{1 + 0.05} = 616 \cdot 10^3 \tag{4.1}$$

As we can see n'_{sw} is still larger than n_{mh} there probably have been a number of accepted jumps which deterred the algorithm sufficiently to explain the extra $60 \cdot 10^3$ iterations of lag, or they adversely affected the convergence monitoring.

A separate test was performed to determine the number of jumps a chain accepts, using the same s_j and p_j as defined above, the results of which were used to draw Figure 4.3. It shows that all the accepted jumps occur during the first 10^4 iterations and that there are relatively few of them.

The question that remains is if these 35 odd jumps have a significant impact on the characteristics of the chains of the Small World algorithm when compared to the Metropolis-Hastings algorithm. In order to shed some light on this the variance of the first 10 samples (taken over 10 000 iterations), the density and

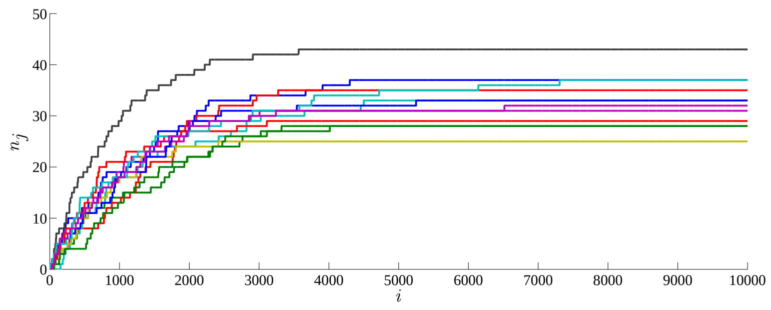

Figure 4.3: Number of jumps n_j performed by 12 different chains of the Small World algorithm over 10 000 iterations i. Iterations 10^4 to 10^6 were omitted because no jumps occurred during those.

	Variance	Density
Metropolis-Hastings	$7.46 \cdot 10^3 \pm 0.37 \cdot 10^3$	0.25 ± 0.005
Small World	$7.84 \cdot 10^3 \pm 0.39 \cdot 10^3$	0.26 ± 0.004

Table 4.1: Average and standard deviation of characteristics of 1000 chains.

acceptance ratio of the 10th sample were calculated for 1 000 different chains and the results consolidated into table 4.1.

The density is a rough indicator of the algorithm's progress, the samples of target distribution have an average density of 16% so as the algorithm progresses the density of the samples will also become 16%. As table 4.1 show the density at 10 000 iterations is practically the same for both algorithms just after the Small World chains have stopped jumping. So while the first 10 Small World samples have increased variance compared to the Metropolis-Hastings samples, it seems that that the jumps have not actually impeded the progress of the algorithm but that they throw off the convergence monitoring.

4.2 Runtime

While the number of iterations required for convergence can be quite interesting, the actual real-world time required for the computations to be completed is also of concern. The Small World and SSS algorithms are both significantly more complex than the baseline Metropolis-Hastings algorithm and this increase in complexity could completely undo the advantage the SSS has.

The runtime of a single iteration was determined by performing 10 000 iterations of each algorithm and measuring the time it took to compute each iteration. The runtime until convergence was determined by measuring how long it takes to perform the required number of iterations, as determined in section 4.1, this process was repeated 24 times. Both these tests were performed using the parameters as defined in section 4.1 and the results are shown in table 4.2. Note that the SSS algorithm was not using any parallelisation for this test.

	$\bar{t}_i$ (ms)	$\frac{\bar{n}}{\bar{n}_{mh}} \cdot \bar{t}_i$ (ms)	$\bar{t}_c$ (s)
Metropolis-Hastings	0.535±0.148	0.535±0.148	450±33.3
Small World	1.17±2.45	1.32±2.75	1083±12.9
Shotgun Stochastic Search	23.2±0.479	1.05±0.0216	1018±116.6
Simulated Annealing	0.547±0.14	0.49±0.12	350±22.5
difference measure		0.06±0.02	55.4±3.36

Table 4.2: Average and standard deviation of the runtime of a single iteration, a weighted iteration and until convergence.

Shotgun Stochastic Search parallelisation

As discussed in section 3.4.2 one of the major advantages of this algorithm is that the generation and evaluation of the neighbourhood could be run in parallel, potentially yielding a significant increase in performance.

Matlab offers parallel computing in the form of the *parfor* loop, while this loop is convenient to use it does come with significant overhead. Using this loop to run multiple chains in parallel works, because the overhead is relatively small compared to the runtime of a single chain, but using it to parallelize processes within a single iteration just is not feasible. It is still possible to determine the efficiency of a SSS algorithm run in parallel by comparing the performance of a parallel SSS with 12 threads available to it to the performance of a parallel SSS with only a single thread available to it, this way the overhead is present under both conditions.

The performance of a the parallel SSS algorithm was tested by performing 10 000 iterations with neighbourhood size $n = \{24, 36, \cdots, 120\}$ once using 12 threads and once using a single thread and measuring the time it took to compute each iteration, the result were used to draw Figure 4.4. Let t_{sss1} be the runtime when using a single thread, t_{sss12} when using 12 threads and e_p be the efficiency of using parallelisation defined as $e_p = \frac{t_{sss12}}{t_{sss1}}$.

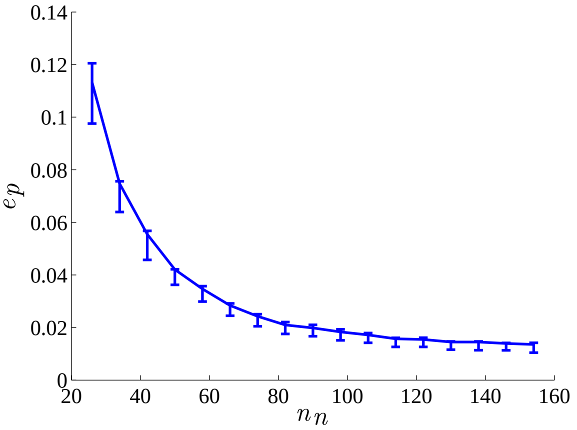

Figure 4.4: The reduction in runtime when using a parallel Shotgun Stochastic Search instead of a single threaded Shotgun Stochastic Search over different neighbourhood sizes. Only 12 threads available.

As Figure 4.4 clearly shows a parallelised SSS algorithm is considerably faster than one that is not, with efficiency increasing as neighbourhood size increases. However a single iteration, when using a neighbourhood size of 50, takes 65.7 ms to compute, still considerably longer than the non parallelised version. The results of this test can be used to determine an estimate of how fast t_c, from section 4.2, would be for the SSS algorithm had it been possible to paralellise it without a large amount of overhead, let t'_c be this estimate:

$$t'_c = t_c \cdot e_p(50) = 1018\text{sec} \cdot 0.0421 = 42.8\text{sec} \tag{4.2}$$

4.3 Comparison of samples

As discussed in sections 3.4.1 through 3.4.3 while the Small World, SSS and Simulated Annealing algorithms are all based on the Metropolis-Hastings algorithm they are all intended to yield different results. Consider figures 4.5, 4.6, 4.7 and 4.8 each shows the average connectivity matrix of 12 000 samples obtained once every 100 iterations after the algorithm in question had converged, p_e reflects the probability of an edge being present in a sample.

Figure 4.5 shows the result of the Metropolis-Hastings algorithm, this will be the baseline to which the other results will be compared.

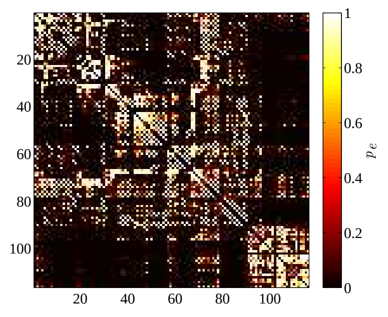

Figure 4.5: Metropolis-Hastings average of 300 post-convergence samples.

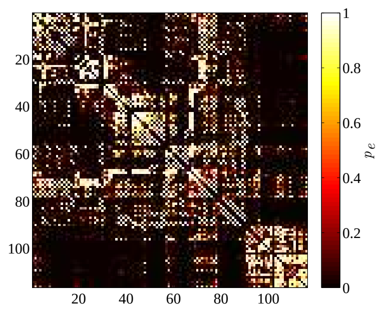

Figure 4.6: Small World average of 300 post-convergence samples..

Figure 4.6 shows the results of the Small World algorithm. As discussed in section 3.4.1 the Small World is intended to, during sampling, jump between the different modes of a multi-modal distribution. Yet with this particular problem the algorithm stops jumping after about 8 000 iterations, as shown in Figure 4.3, long before convergence is reached. So when the samples were obtained the Small World algorithm actually behaved as a Metropolis-Hastings algorithm (which was confirmed by monitoring the number of jumps performed, zero, during sampling), no wonder that that figures 4.5 and 4.6 are virtually indistinguishable.

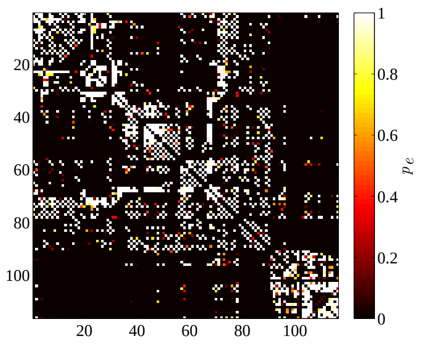

Figure 4.7: SSS average of 300 post-convergence samples.

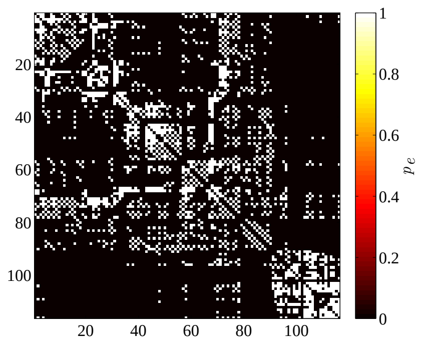

Figure 4.8: Simulated annealing average of 300 post-convergence samples.

The results of the SSS and Simulated annealing, shown in Figure 4.7 and 4.8 respectively, are quite different from the Metropolis-Hastings results. calculate The SSS algorithm is intended to aggressively pursues high probability samples, as discussed in section 3.4.2, and ends up sampling an smaller area much closer around the mode of the distribution than the Metropolis-Hastings algorithm. This is reflected in the results by the samples being less varied in Figure 4.7. Most edges have a probability p_e that is either 0 or 1, they are usually either present or absent in all samples, compared to the Metropolis-Hastings results.

As discussed in section 3.4.3 the Simulated Annealing algorithm is designed to quickly provide a single high probability sample. Where the Metropolis-Hastings samples a large area around the peak of the distribution, the Small World samples multiple areas around multiple peaks and the SSS samples a small area around the peak, the Simulated Annealing moves to the top of the peak and then stops. The 300 samples taken after the algorithm had converged were all identical as every edge in Figure 4.8 has a probability p_e of either 0 or 1, which is only possible if every sample was identical.

In order to put the difference between the results in clearer contrast the results found in figures 4.5 through 4.8 were consolidated into a single histogram found in Figure 4.9, note that the height of the first bin in the histogram was divided by 10 so a reasonable y scale could be used. It shows the effects found

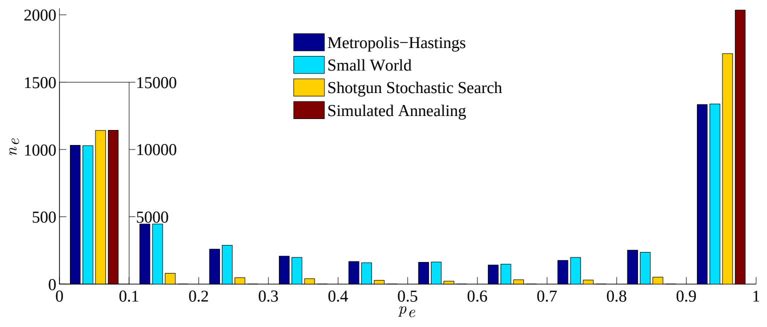

Figure 4.9: Comparison of edge probability in figures 4.5, 4.6, 4.7 and 4.8. **Warning** the n_e of the first bin, 0 to 0.1, was divided by 10 in order to improve readability.

in the other figures a bit more clearly. The distribution of edge probabilities of the SSS results clearly favour $e_p \in \{0, 1\}$, compared to the Metropolis-Hastings and Small World, while the probabilities of the Simulated Annealing results are exclusively either one or zero.

Chapter 5

Conclusion

There is no clear winner, which algorithm is best depends on your needs and how much effort one is prepared to put in the actual implementation.

The Small World algorithm is designed to be used to sample a multi-modal distribution, it is supposed to jump from mode to mode providing samples of each, something the other algorithms are unlikely, practically unable, to do. Yet while the simulations of section 3.5 do show that given the right probability distribution the Small World algorithm will behave in this fashion, the results of the optimization tests in section 3.6.2 show the algorithm behaving as a normal unmodified Metropolis-Hastings algorithm (except for the first 10 000 iterations when it will jump about 40 times). In addition the Simulated Annealing test, and to a lesser extend the others, suggest strongly that the probability distribution of this brain inference problem is a unimodal distribution. So given that it worsens performance, see section 4.1, while yielding the same results I would advise against using the Small World algorithm for this particular problem.

The Simulated Annealing algorithm is the fastest of the algorithms evaluated, assuming you are using the alternate method of convergence monitoring, see section 3.6.4, otherwise it will be about as fast, both convergence and runtime wise, as the Metropolis-Hastings.

The SSS algorithm, for this problem, does require far less iterations to converge than the Metropolis-Hastings algorithm, while actually requiring more time per iteration, which in the end causes it to take about 2.5 times longer to converge than the Metropolis-Hastings algorithm, see table 4.2. However the algorithm does allow for parallelisation, so given the right implementation it could actually end up being significantly quicker than the Metropolis-Hastings. Theoretically it could even be faster than the Simulated Annealing, assuming a near perfect parallelisation, since it requires less iterations to converge than even a Simulated Annealing algorithm using the alternate convergence method.

However there is still the problem of convergence monitoring, which in most cases requires running multiple chains of the algorithm, by doing so one already has an opportunity to run parts of the calculation in parallel which could negate the advantage of the SSS completely. The SSS algorithm could be faster than a Metropolis-Hastings algorithm by taking advantage of any unused parallel processing power, however if multiple chains are going to be calculated there may be no parallel processing power left unused. Having said that, in theory the number of iterations required for convergence should be relatively constant within a single type of problem, so in theory you would not be required to use convergence monitoring all the time.

In the end though these algorithms are all designed to yield different results, which one you should use depends on your needs. If you only need a single model (in this case a structural brain network) with a high, possibly highest, posterior you should definitely use the Simulated Annealing. If you need more than one model but they all need to have a high posterior use the Shotgun Stochastic Search. And if you need a general, non specific, sampling of the posterior distribution use the normal Metropolis-Hastings. And if these algorithms perform unreliably, if their convergence speed is inconsistent even though the parameters are, the distribution may be multi-modal and chains may be getting stuck in different modes, in which case you should probably use the Small World algorithm (or use a Small World proposal distribution with any of the other algorithms).

So in conclusion for the purpose of Bayesian inference of whole brain networks using a Small World algorithm would not be better than the Metropolis-Hastings and using a SSS or Simulated Annealing algorithm can be beneficial, depending on your exact purpose, implementation and hardware.

Bibliography

[1] David Barber. *Bayesian Reasoning And Machine Learning.* Cambridge University Press, 2012.

[2] Danielle S. Bassett and Edward T. Bullmore. Human brain networks in health and disease. *Current opinion in neurology*, page 340–347, 2009.

[3] Stephen P. Brooks and Andrew Gelman. General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*, 7:434 – 455, 1998.

[4] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[5] P. Erdős and A Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Acadamy of Sciences*, pages 17–61, 1960.

[6] Aaron G. Filler. The history, development and impact of computed imaging in neurological diagnosis and neurosurgery: Ct, mri, and dti. *The Internet Journal of Neurosurgery*, 7(1), 2009.

[7] Yongtoa Guan, Roland Fleissner, and Paul Joyce. Markov chain monte carlo in small worlds. *Statistics and Computing*, 16:193 – 202, 2006.

[8] Chris Hans, Adrian Dobra, and Mike West. Shotgun stochastic search for "large p" regression. *Journal of the American Statistical Association*, 102:507 – 516, 2007.

[9] Max Hinne, Tom Heskes, Christian F. Beckmann, and Marcel A.J. van Gerven. Bayesian inference of structural brain networks. *NeuroImage*, 66(0):543 – 552, 2013.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[11] Eric R. Kandel And J.H. Schwartz. *Principles of Neural Science.* McGraw-Hill Medical, 2000.