Large Scale Bayesian Modelling of Complex Networks

Andreas Leon Aagaard Moth
Kristoffer Jon Albers

Kongens Lyngby 2013
M.Sc.-2013-92

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

M.Sc.-2013-92

Preface

This master thesis was written at DTU compute, Department of Applied Mathematics and Computer Science, Technical University of Denmark. The project has been completed in the period between March and September, 2013.

The thesis has been prepared by Andreas Leon Aagaard Moth and Kristoffer Jon Albers, rated at 30 ECTS points for each author. It is part of the requirements for obtaining the M.Sc. degree in Computer Science and Engineering at DTU.

The project has been supervised by associate professors Morten Mørup and Mikkel N. Schmidt.

Lyngby, September 2013

Abstract

Bayesian stochastic blockmodelling has proven a valuable tool for discovering community structure in complex networks. The Gibbs sampler is one of the most commonly used strategies for solving the inference problem of identifying the structure. Though it is a widely used strategy, the performance of the sampler has not been examined sufficiently for large scale modelling on real world complex networks. The Infinite Relational Model is a prominent non-parametric extension to the Bayesian stochastic blockmodel, which has previously been scaled to model large bipartite networks.

In this thesis we examine the performance of the Gibbs sampler and the more sophisticated Split-Merge sampler in the Infinite Relational Model. We push the limit for network modelling, as we implement a high performance sampler capable of performing large scale modelling on complex unipartite networks with millions of nodes and billions of links. We find that it is computationally possible to scale the sampling procedures to handle these huge networks.

By evaluating the performance of the samplers on different sized networks, we find that the mixing ability of both samplers decreases rapidly with the network size. Though we find that Split-Merge can increase the performance of the Gibbs sampler, these procedures are unable to properly mix over the posterior distribution already for networks with about 1000 nodes. These findings clearly indicates the need for better sampling strategies in order to expedite the studies of real world complex networks.

Resumé

Bayesiansk stokastisk blokmodellering har vist sig at være et værdifuldt værktøj til at undersøge gruppestruktur i komplekse netværk. Gibbs sampleren er en af de oftest brugte strategier til at løse inferensproblemet til identificering af gruppestrukturen. Selvom det er en ofte brugt strategi, er dens ydeevne ikke tilfredsstillende undersøgt for storskala modellering af komplekse netværk fra den virkelige verden. IRM-modellen er en prominent ikke parametrisk udvidelse af den stokastiske blokmodel, der tillader et uendeligt antal grupper. IRM-modellen har tidligere været skaleret til at modellere store todelte netværk.

I dette speciale undersøger vi hvordan Gibbs sampleren og den mere sofistikerede Split-Merge sampler yder i IRM-modellen. Vi bryder tidligere grænser for netværks-modellering ved at implementere en højtydende sampler, der er i stand til at udføre storskala modellering på komplekse udelte netværk med millioner af knuder og milliarder af kanter. Vi finder at det er beregningsmæssigt muligt at skalere sampler-procedurerne til at håndtere så store netværk.

Ved at evaluere samplernes ydeevne finder vi at deres evne til at mikse svinder kraftigt som netværk størrelsen stiger. Selvom vi finder at Split-Merge kan forbedre ydeevnen af Gibbs sampleren, er procedurerne ikke i stand til at mikse over posteriori fordelingen allerede for netværk med omkring 1000 knuder. Disse fund indikerer at det er nødvendigt at udvikle bedre sampling-strategier for at understøtte forskning af store komplekse netværk fra den virkelige verden.

Acknowledgements

We would like to thank Morten Mørup and Mikkel N. Schmidt for supervising the project and helping us throughout the project with their continous guidance and dedication to the project. We also thank them for participating in writing an article based on some of the findings in this project, that is to be presented at the Machine Learning for Signal Processing 2013 workshop.

We would also like to thank Kristoffer Hougaard Madsen for support and aid with the SPM8 framework.

Contents

Chapter 1

Introduction

All around us we encounter situations that can be described as complex networks. We participate in social relations and form large social networks with our family, friends and co-workers. We live in cities that are sustained by complex power and water supply lines, where we navigate in the complex traffic system and consume goods transported through logistically demanding supply lines. Furthermore, various computer and communication systems form complex and sophisticated networks that allow rapid and high capacity transmission of information. All these examples are networks that have emerged as a result of human behaviour. In the nature we also observe various environmental and biological networks. Even various aspects of our own bodies can be described as complex networks. This includes the network of protein-protein interactions in individual cells and the communication between billions of neurons in the brain.

Due to the huge importance these systems have on vast areas of our everyday lives, network science has become an important and emerging science, combining knowledge from many different research fields [3] that allows us to develop general network models by which we can investigate the underlying structures of these important systems. One way to investigate complex networks is to perform cluster analysis, where the network is divided into communities, based on the internal structure of the network. To detect community structures, cluster analysis is often based on Bayesian statistical modelling, such as the stochastic block-model. The stochastic block-model partitions networks into smaller

blocks, such that these blocks capture the underlying clustering structure of the network [12]. In this thesis we consider a non-parametric extension of the stochastic blockmodel called the Infinite Relational Model (IRM) [9][27], which allows for an unlimited number of blocks. Bayesian blockmodelling often uses Gibbs sampling to solve the combinatorial inference problem of identifying blocks. This has often been the case for IRM [20] [6], though more sophisticated sampling strategies as the Split-Merge procedure [8] has also been considered [20].

Statistical analysis of large scale networks is an active research area, where IRM has become a prominent network model. In order to model in large scale, efficient implementations are essential. In [28] stochastic relational modelling on large scale was performed on bipartite networks with half a million nodes and 100 million links. In [6] a highly optimized IRM implementation utilizing GPU resources performed bipartite co-clustering on graphs with about 8 million nodes and 500 million links. Due to the coupled nature of nodes, inference computations can easily be parallelized for bipartite graphs. Many interesting real world networks can not be modelled as bipartite graphs, and it is hence important to be able to model these complex unipartite networks. The understanding and robustness of the Gibbs sampler is well established for modelling of smaller unipartite networks, but it has not been thoroughly examined whether Gibbs sampling is good enough to infer clustering of large complex networks made from real world data.

Contribution

The motivation for this thesis is to provide insight to the scalability and limitations of Gibbs and Split-Merge sampling in the Infinite Relational Model. In this thesis we go beyond the network size and complexity limitations of previous studies, as we test the samplers performance and ability to mix over the posterior distribution on complex unipartite networks with millions of nodes.

Our contributions with this thesis are:

- A high performance Gibbs and Split-Merge sampler for complex unipartite network, capable of analysing:
  - Network with millions of nodes and billions of links.
  - Multiple graphs simultaneously by utilizing GPU resources.
- Evaluate mixing ability for large scale Gibbs and Split-Merge sampling:
  - When performed for millions of sampling iterations.
  - As the network size increases to millions of nodes.

- Determine the network size limit the Gibbs and Split-Merge samplers can converge for.
- Analyse how IRM performs on large scale real world data.
- Illustrate the clustering ability of the samplers on real world data.

In order to illustrate the clustering ability of the sampler we perform IRM analysis on data for resting state f-MRI scannings of neuronal activity for 172 subjects, with a spatial resolution of 1000 regions. The clustering found by the analysis is then mapped back into the brain and visualized to see how IRM has structured the brain into regions.

This thesis

This thesis starts with an introduction of the fundamental theory and technical terms used in the report. We then take a look at the design and requirements of the application, followed by the implementation and optimization of the Gibbs and Split-Merge sampler algorithms. The data we use to evaluate the samplers are then introduced and used in the following results section, in which we also discuss the implication of our findings. Finally we conclude on the findings and what the implications of these are.

Chapter 2

Theory

In this chapter we describe the theory behind the Infinite Relational Model, derive the two sampler algorithms and explain the theory behind computational optimizations of the algorithms in relation to implementing a high performance computer program.

2.1 Complex networks as graphs

Complex networks can be seen as graphs, $G(V, E)$ where the set of vertices/nodes V represents the objects in the system while the set of edges/links E represents the interactions between these objects.

Figure 2.1 illustrates three of the most common types of graphs; directed, undirected and bipartite graphs. In a directed graph the edges are oriented, such that they do not simply link two nodes, but link one node to another in only one direction. In an undirected graph the edges have no orientation. If there exists an edge between two nodes i and j then there is both a connection from i to j and from j to i. In figure 2.1 both (A) and (B) are examples of unipartite graphs, while (C) is a bipartite graph. This is a special type of graph, in which all the nodes can be divided into two disjoint sets N and M, such that no nodes

in N nor M are interconnected; all nodes in N are only connected to nodes in M and vice versa. to a node in M.

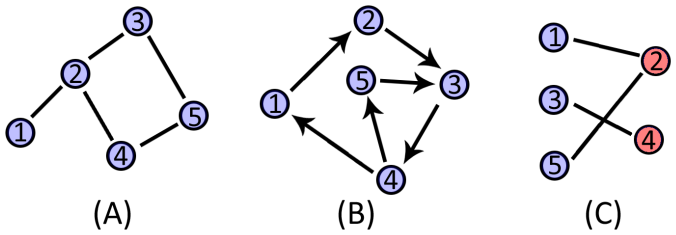


Figure 2.1: Exapmles of common types of graphs: (A) undirected unipartite, (B) directed unipartite, (C) bipartite.

A graph can be either weighted or unweighed. In a weighted graph, a value is associated with every link, known as the weight. Depending on the network the weight can represent various properties of the link. If the network for instance describes a system of cities connected by railroads, the values might represent the distance between the cities, the ticket cost of travelling between the cities or how many trains traverses the tracks every day. Though many other realworld situations might beneficially be described as weighted graphs, we will only consider unweighed graphs in this thesis as we are more interested in analysing the performance of the sampler than using it on specific networks. In the case of an unweighted graph the links can be represented by a binary adjacency matrix A, such that $A_{ij} = 1$ specifies a link between node i and j. The networks we examine do not contain self-loops and hence the diagonal of A only consist of zero-elements. The adjacency matrices for the graphs in figure 2.1 are:

$$A^{(A)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}, A^{(B)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, A^{(C)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

When performing IRM, this properties of a bipartite graph makes it naturally parallelizable, which has previously been utilized to perform IRM on large scale bipartite networks in [6].

Many real world situations such as neural networks, trade routes and social networks cannot be represented as bipartite graphs, and it is therefore important to be able to model and understand the structure of unipartite networks.

In this thesis we consider the more challenging undirected unipartite graphs. For undirected graphs the adjacency matrix is symmetric, such that for all i

and j, $A_{ij} = A_{ji}$. Hence we only need to consider the upper triangular part of the adjacency matrix.

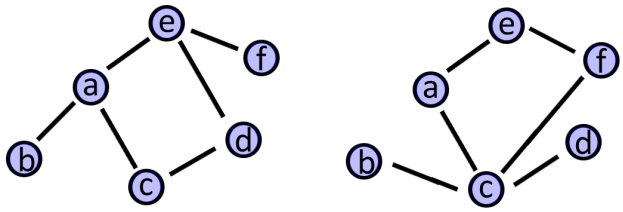


Figure 2.2: Two graphs linking the same nodes differently.

Many real world situations can be described as multiple graphs where the set of vertices V are the same in all graphs but the vertices are linked differently in each graph. This is illustrated in figure 2.2 where two graphs contain the exact same nodes, but linked differently. An example can be graphs of different airline companies sharing the same airports but not operating the same routes between the airports.

In this thesis we encounter multiple graphs as we perform IRM on graphs made from functional brain scans of multiple subjects. Here we assume that all scans contains the same regions, but due to the measured differences in neuronal activity these regions are linked differently for each subject.

2.2 Statistical cluster analysis

The main drive for this thesis is to investigate how Bayesian modelling can be applied in order to model systems that can be described as undirected unweighted unipartite networks.

The purpose of cluster analysis is to determine the underlying structure of the clustered data, when no other information than the data itself is available. Using cluster analysis we model the underlying structure of the networks, by grouping objects that act similar.

Many clustering methods are based on statistical models, where the data is assumed to have been generated by some statistical processes, these are called generative models. For these models the clustering problem becomes to first find a statistical model that is suitable to fit the data and then describe the distribution and associated parameters for this model.

2.2.1 Mixture model

Mixture models are a common statistical approach, which is a simple yet powerful procedure for modeling structure in complex networks [20]. It is the foundation for the Bayesian stochastic blockmodel, which the infinite relational model is based on. In a mixture model the data is assumed to have originated from a mixture of several distributions. If we consider a structure with n data points and K distributions, then the process that generated the data can be considered as n times choosing one of the K distributions and generating a data point from it. The probability of choosing each distribution is weighted, such that the i'th distribution is chosen with the weight w_i, upholding that each weight $w \in [0; 1]$ and $\sum_{i=1}^{K} w_i = 1$ [16]. In a parametric setting, the probability that a data point x originated from the i'th distribution is $p(x|h_i)$ where h_i is the set of parameters for the i'th distribution. The probability of a given data point x is then given by:

$$p(x|H) = \sum_{i=0}^{K} w_i p_i(x|h_i), \tag{2.1}$$

where H is the set of all parameters $H = \{h_1, ..., h_K\}$. Considering that the data points are generated indep

$$p(X|H) = \prod_{j=0}^{n} p(x_j|H). \tag{2.2}$$

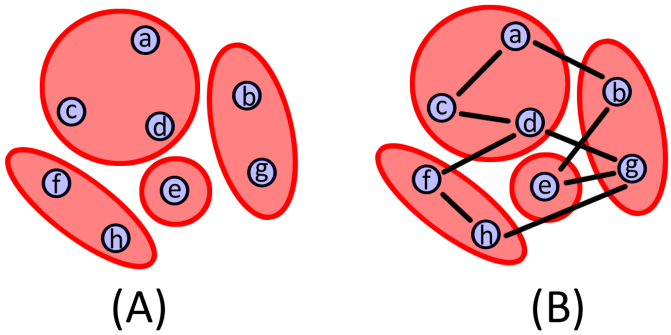


Figure 2.3: Example of a mixture model, without links (A) and with links (B).

An example of such a mixture model is shown in figure 2.3 (A).

When we look at cluster analysis for network data, each distribution describes a single cluster [16]. Each node in the network belongs to a single cluster, while each edge depends on the two clusters of the nodes linked by the edge. This is

illustrated in figure 2.3 (B). The probability of an edge between two nodes are given by a link probability between the two clusters linked by the edge. If z_i describes which cluster the i'th node is part of, the likelihood of the network is given by:

$$p(\mathbf{A}|\mathbf{Z},\eta) = \prod_{i\ j} p(A_{ij}|z_i, z_j, \eta), \tag{2.3}$$

where η denotes a matrix, representing the link probabilities between the individual blocks and $\mathbf{Z}$ specifies which block each of the nodes are assigned to.

2.2.2 Bayesian Stochastic blockmodel

A stochastic blockmodel is a generative model, used to model the structure in a graph by splitting it into blocks. Following the simple mixture model, the blockmodel only allows the graph to be split into a predefined number of blocks K.

Bayes theorem states the following relation between conditional probabilities:

$$p(X|Y) = \frac{p(Y|X)p(X)}{p(Y)} \tag{2.4}$$

where $p(X|Y)$ is the posterior (the degree of belief having accounted for Y), $p(X)$ is the prior (initial degree of belief in X) while $p(Y|X)$ is the likelihood. The fraction $\frac{P(Y|X)}{P(Y)}$ represents the support Y provides for X. We can apply Bayesian principles to the mixture model, leading to the so called Bayesian stochastic blockmodel [12].

Using Bayes theorem the posterior of the blockmodel is given as:

$$p(Z,\eta|A) = \frac{p(A|Z,\eta)p(Z)p(\eta)}{p(A)} \tag{2.5}$$

where $p(A)$ is a constant, as the links in the network are known. Each term in the likelihood in formula 2.3 defines the probability of a link between two nodes. This can be based on the Bernoulli distribution:

$$p(A_{ij}|z_i, z_j, \eta) = Bernoulli(\eta_{z_i z_j}) \tag{2.6}$$

where $\eta_{z_i z_j}$ is the link probability, giving the probability of a link between a node in block z_i and a node in block z_j. The prior for the link probabilities can be based on independent Beta distributions:

$$p(\eta_{lm}) = Beta(\beta^+, \beta^-) \tag{2.7}$$

The probability of assigning individual nodes to one of the K blocks is given by Z. The prior for Z is chosen to be generated from a Dirichlet distribution, which allows for different sizes of the blocks [mmpaper]:

$$p(\mu) \sim Dirichlet(\alpha) \tag{2.8}$$
$$p(\mathbf{Z}) \sim categorical(\mu) \tag{2.9}$$

In this thesis we focus on the Infinite Relational Model (IRM) as proposed by Kemp *et al.* [9] and Xu *et al.* [27]. This is an extension to the Bayesian stochastic blockmodel by theoretically allowing an unlimited number of clusters. This is achieved by basing the prior of the clustering on a Chinese Restaurant Process (CRP). As a generative model, IRM is based on the following three processes:

$$\mathbf{Z} \sim \text{CRP}(\alpha), \qquad \text{groups} \tag{2.10}$$
$$\eta_{lm} \sim \text{Beta}(\beta^+, \beta^-), \qquad \text{interactions} \tag{2.11}$$
$$A_{ij} \sim \text{Bernoulli}(\eta_{z_i, z_j}), \qquad \text{links.} \tag{2.12}$$

The model relies on the three hyperparameters $h = \{\alpha, \beta^+, \beta^-\}$, that specify how the model behaves. α is used by CRP in order to specify the probability that there are many or few clusters. A high α-value makes it more likely to have many clusters, while it is most likely to have few clusters for a low value. The β^+ and β^- are used in the beta probability to specify whether it is most likely to have links, non-links or some combination thereof.

These distributions are described further in section 2.3, where we also derive an algorithm for sampling the posterior distribution using the Gibbs and Split-Merge procedure. Inferring the posterior distribution for IRM is a computationally hard problem, and hence not feasible to do. Instead we relax the problem, such that the cluster assignments are estimated using Markov Chain Monte Carlo (MCMC) inference. Gibbs sampling has previously been proposed as Bayesian estimator for Stochastic Blockmodels in [12] and described for IRM in [20, 9], while the Split-Merge procedure has been presented in [8], and which has previously been utilized for IRM [18, 1, 15]. The idea behind MCMC is to iteratively change the model, ensuring that the entire posterior distribution will be traversed in the limit of infinitely many changes.

In Gibbs sampling each parameter is changed one at a time by drawing new values from the parameter's posterior marginal distribution. In IRM this corresponds to calculating the posterior marginal distribution of assigning each node to each of the existing clusters as well as to a new empty cluster. The Split-Merge procedure is capable of reassigning several nodes at once. It does this by either splitting a cluster into two or merging two clusters into one. The merging

of two clusters is deterministic as all nodes simply end up in the same cluster, but in the case that a cluster is split, Split-Merge utilizes restricted Gibbs sampling on these two clusters, in order to determine where each node should be placed.

2.3 The Infinite Relational Model

In this section we explain how the Bernoulli and Beta distributions work and deduce the Chinese Restaurant Process from the Dirichlet distribution. We relate these to the Infinite Relational Model and finally use them in order to derive a formula for sampling the posterior distribution using Gibbs and Split-Merge sampling.

2.3.1 Bernoulli distribution

Bernoulli is one of the simplest probability distributions. It can be considered a discrete probability distribution with two possible values; 0 and 1. The distribution of 0's and 1's are based on a weight parameter $\pi \in [0, 1]$. For instance a weight of 0.6 means that 60% of the distribution consists of ones and 40% consists of zeros. Using this knowledge it is possible to calculate the probability of having a random value $x \in \{0, 1\}$ given a weight π:

$$P(x|\pi) = \pi^x(1-\pi)^{1-x} = \begin{cases} \pi, & \text{if } x = 1 \\ 1-\pi, & \text{if } x = 0 \end{cases} \tag{2.13}$$

Likewise it is possible to calculate the probability that a specific sequence of independent variables D are drawn from this distribution:

$$P(D|\pi) = \pi^{N_1}(1-\pi)^{N_0}, \tag{2.14}$$

where N_1 and N_0 specifies the number of ones and zeros in the sequence D.

In the Infinite Relational Model there are not just one weight, but a weight between any two clusters l and m, written as η_{lm}. As we assume all links and

weights are independent, the probability yields:

$$P(\mathbf{A}|\mathbf{Z},\eta) = \prod_{l \leq m} \eta_{lm}^{N_{lm}^{+}} (1-\eta_{lm})^{N_{lm}^{-}}, \tag{2.15}$$

where N_{lm}^{+} and N_{lm}^{-} specifies the number of links and non-links between cluster l and m. As η is symmetric, we only consider the product of the upper triangle.

2.3.2 The Beta-distribution

The second distribution the Infinite Relational Model relies upon is the Beta-distribution. The beta function takes two positive parameters, β^{+} and β^{-}, which specifies the distribution of values. The probability that a value follows the Beta distribution is given by the probability density function:

$$Beta(\theta|\beta^{+},\beta^{-}) = \frac{\Gamma(\beta^{+}+\beta^{-})}{\Gamma(\beta^{+})\Gamma(\beta^{-})} \theta^{\beta^{+}-1}(1-\theta)^{\beta^{-}-1} \tag{2.16}$$

Where $\Gamma(x)$ is the gamma function:

$$\Gamma(x) = (x-1)! \tag{2.17}$$

There are two major reasons for using the beta distribution. First, depending upon the parameters, it can represent various distributions, such as uniform, normal and arcsine distributions, as shown in figure 2.4. Second, the Bernoulli and Beta distributions are conjugate distributions, which means they originate from the same family of distributions and are hence easy to combine. When we derive the algorithm for estimating the clustering, we are going to use the fact that these distributions are conjugate and hence can be combined in order to form an even simpler expression.

In the Infinite Relational Model we have a beta distribution for every two clusters l and m, given as η_{lm}, which describes the probability that there are connections between these (or internally in a cluster when $l = m$). As for Bernoulli, we assume that all links are independent, while β^{+} and β^{-} are constants. In this

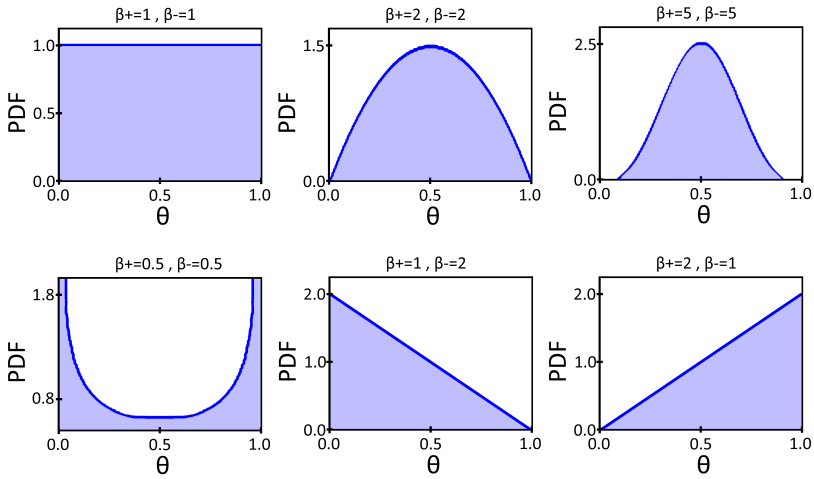


Figure 2.4: Probability density function (PDF) for the beta distribution

case we can create a single expression for the probability of all η:

$$P(\eta|\beta^+,\beta^-) = \prod_{l\leq m} \frac{\Gamma(\beta^+ + \beta^-)}{\Gamma(\beta^+)\Gamma(\beta^-)} \eta_{lm}^{\beta^+-1}(1-\eta_{lm})^{\beta^--1} \tag{2.18}$$

In formula 2.16 we can integrate over θ in order to get an expression that is denoted as the *Beta*-function, which we will be using later (the beta distribution integrates to 1):

$$B(\beta^+,\beta^-) = \frac{\Gamma(\beta^+)\Gamma(\beta^-)}{\Gamma(\beta^+ + \beta^-)} = \int_0^1 \theta^{\beta^+-1}(1-\theta)^{\beta^--1}d\theta \tag{2.19}$$

2.3.3 Chinese Restaurant Process

The final part of the IRM model is the Chinese Restaurant Process (CRP). This is a stochastic process that partitions D objects into a set of clusters, such that each cluster contains $0, ..., D$ objects, illustrated in figure 2.5.

To illustrate CRP we consider the situation where blocks represent tables in a restaurant. The objects then represent customers entering the restaurant one at a time. The customers are placed at the tables, such that:

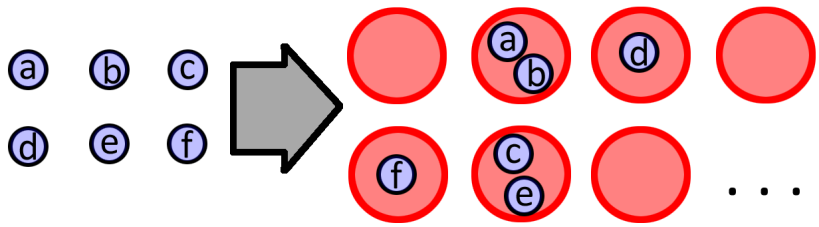


Figure 2.5: Example of letters a-f clustered by CRP.

1. The first customer is placed at a table.
2. The n'th customer is randomly placed at an empty or non-empty table dependent on how other customers are placed.

In order to express these placement probabilities, we first consider the finite Dirichlet distribution. The distribution specifies the probability of each value $\mu_1, ..., \mu_D$ given a vector α of D input parameters:

$$(\mu_1, \mu_2, ..., \mu_D) \sim Dirchlet(\alpha_1, \alpha_2, ..., \alpha_D)$$

For the restaurant example, the μ-vector specifies the probability of placing a customer at each of the D tables, while the α parameter changes the likelihood of placing customers at each table, the higher the α value, the more likely it is.

In cluster analysis nodes are partitioned into clusters rather than customers to tables. Here the μ-values represents one clusters each. For a given node i we are interested in finding the probability of placing it according to its cluster assignment z_i. Given a finite set of clusters D, we can calculate this probability as:

$$P(\mu|\alpha) = \frac{\Gamma(\sum_d \alpha_d)}{\prod_d \Gamma(\alpha_d)} \prod_{d=1}^{D} \mu_d^{\alpha_d - 1} \tag{2.20}$$

$$P(z_i|\mu) = \prod_{d=1}^{D} \mu_d^{z_{di}} \tag{2.21}$$

Where formula 2.20 is the dirichlet distribution and formula 2.21 simply returns the categorical probability that the node is placed in the cluster given by z_i.

Instead of only calculating the probability of a single node assignment, it is possible to generalize this to find the likelihood of assigning all nodes according to their respective clusters. As we assume that all assignments are independent, this becomes:

$$P(\mathbf{Z}|\mu) = \prod_{d=1}^{D} \prod_{i} \mu_d^{z_{di}} = \prod_{d=1}^{D} \mu_d^{\sum_i z_{di}} \tag{2.22}$$

This formula does however require that the optimal number of clusters D is known. In cluster analysis, this is rarely the case and to get rid of this assumption we let D go towards infinity, allowing there to be as many clusters as necessary. To do this we first redefine the Dirichlet prior vector α to a single value. This value has to go towards zero as D goes towards infinity, otherwise it is very unlikely that two nodes will be placed in the same cluster, as there are infinitely many empty clusters with the same constant probability. This means that the probability of placing a node in an empty cluster goes towards 1 as D goes towards infinity. We redefine α to α/D, such that the value goes towards zero as D goes towards infinity. Hence the probability of placing a node in a new cluster does not go towards 1 and as a result, nodes are more likely to be clustered together.

Furthermore we integrate out μ, in order to get rid of this unnecessary intermediate parameter:

$$\begin{aligned}
P(\mathbf{Z}|\alpha/D) &= \int P(\mathbf{Z}|\mu)P(\mu|\alpha/D)d\mu && (2.23)\\
&= \int \prod_{d=1}^{D} \left(\mu_d^{\sum_i z_{di}}\right) \frac{\Gamma(\sum_d \alpha/D)}{\prod_d \Gamma(\alpha/D)} \prod_{d=1}^{D} \mu_d^{\alpha/D-1} d\mu && (2.24)\\
&= \frac{\Gamma(\sum_d \alpha/D)}{\prod_k \Gamma(\alpha/D)} \int \prod_{d=1}^{D} \left(\mu_d^{\sum_i z_{di}}\right) \prod_{d=1}^{D} \mu_d^{\alpha/D-1} d\mu && (2.25)\\
&= \frac{\Gamma(\sum_d \alpha/D)}{\prod_d \Gamma(\alpha/D)} \int \prod_{d=1}^{D} \mu_d^{\sum_i z_{di}+\alpha/D-1} d\mu && (2.26)
\end{aligned}$$

This distribution looks somewhat similar to the dirichlet distribution in formula 2.20, except for the integral. It turns out if we integrate the dirichlet distribution, we get a formula that can be used to reduce this equation even further:

$$\int P(\mu|\alpha)d\mu = \int \frac{\Gamma(\sum_d \alpha_d)}{\prod_d \Gamma(\alpha_d)} \prod_{d=1}^{D} \mu_d^{\alpha_d - 1} d\mu \Leftrightarrow \tag{2.27}$$

$$1 = \frac{\Gamma(\sum_d \alpha_d)}{\prod_d \Gamma(\alpha_d)} \int \prod_{d=1}^{D} \mu_d^{\alpha_d - 1} d\mu \Leftrightarrow \tag{2.28}$$

$$\frac{\prod_d \Gamma(\alpha_d)}{\Gamma(\sum_d \alpha_d)} = \int \prod_{d=1}^{D} \mu_d^{\alpha_d - 1} d\mu \tag{2.29}$$

The integral in this formula is equal to the integral in formula 2.26 if we simply substitute α_d with $\sum_i z_{di} + \alpha/D$. By using this formula, we can now continue reducing formula 2.26:

$$P(Z|\alpha/D) = \frac{\Gamma(\sum_d \alpha/D) \prod_d \Gamma(\sum_i z_{di} + \alpha/D)}{\prod_d \Gamma(\alpha/D) \Gamma(\sum_d (\sum_i z_{di} + \alpha/D))} \tag{2.30}$$

$$= \frac{\Gamma(\alpha) \prod_d \Gamma(\sum_i z_{di} + \alpha/D)}{\prod_d \Gamma(\alpha/D) \Gamma(J + \alpha)}, \tag{2.31}$$

where $\sum_{d,i} z_{di}$ has been substituted with J (the number of nodes), as each node belong to exactly one cluster and hence the sum of all node assignments is equal to the number of nodes.

Using Bayes theorem we can now find the probability that a certain node j belongs to cluster d, conditioned on all other nodes remaining in their clusters:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \frac{P(Z_{\setminus j}, z_{dj} = 1 | \alpha/D)}{\sum_l P(z_{lj} = 1, Z_{\setminus j} | \alpha/D)} \tag{2.32}$$

From this we can immediately see that the constants $\Gamma(\alpha)$ and $\prod_d \Gamma(\alpha/D)\Gamma(J + \alpha)$ in formula 2.31 can be reduced and we get:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \frac{\Gamma(\sum_{i \neq j}(z_{di}) + 1 + \alpha/D) \prod_{m \neq d} \Gamma(\sum_{i \neq j}(z_{mi}) + \alpha/D)}{\sum_l \left(\Gamma \left(\sum_{i \neq j}(z_{li}) + 1 + \alpha/D \right) \prod_{m \neq l} \Gamma \left(\sum_{i \neq j}(z_{mi}) + \alpha/D \right) \right)}$$

This formula can be further reduced by dividing with $\prod_m \Gamma(\sum_{i \neq j}(z_{mi} + \alpha/D)$. In both the numerator and denominator the last part can be reduced without any problems. For the first part we will however need to use the definition of the gamma function, shown in formula 2.17. By expanding each of these gammas to several multiplications, we see that one multiplication remains in both the numerator and denominator, due to the +1:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \frac{\sum_{i \neq j}(z_{di}) + 1 + \alpha/D - 1}{\sum_l \left(\sum_{i \neq j}(z_{li}) + 1 + \alpha/D - 1 \right)} \tag{2.33}$$

$$= \frac{\sum_{i \neq j}(z_{di}) + \alpha/D}{\sum_l \left(\sum_{i \neq j} z_{li} \right) + \alpha} \tag{2.34}$$

By introducing a new variable n_d, denoting the number of nodes in cluster d, excluding node j we can get an even cleaner expression for this probability. Additionally the sum over the entire cluster assignment $\mathbf{Z}$, excluding node j is equal to $J - 1$. By applying this we can now reduce the formula to:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \frac{n_d + \alpha/D}{J - 1 + \alpha} \tag{2.35}$$

When the number of clusters goes towards infinity, so will the number of empty clusters in $\mathbf{Z}$. As the cluster order does not matter, we can however concatenate the probabilities of placing the node in any of the empty clusters. In order to do this we first introduce a new variable K, denoting the total number of non-empty clusters (ignoring node j). Hence the probability of placing a node in a cluster now becomes:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \begin{cases} \frac{n_d + \alpha/D}{J-1+\alpha} & \text{if } n_d > 0 \\ \frac{\alpha(D-K)/D}{J-1+\alpha} & \text{otherwise} \end{cases} \tag{2.36}$$

We now let D go towards infinity, in which case we find:

$$P(z_{dj} = 1 | Z_{\setminus j}, \alpha/D) = \begin{cases} \frac{n_d}{J-1+\alpha} & \text{if } n_d > 0 \\ \frac{\alpha}{J-1+\alpha} & \text{otherwise} \end{cases} \tag{2.37}$$

With this formula we can calculate the probability of placing a single node in a cluster. We are however interested in the probability of placing all nodes

according to their cluster assignment $\mathbf{Z}$. In the cluster assignment there are K non-empty clusters and hence K nodes will have been placed with the latter probability, which means we get α^K (ignoring the denominator for now). Each cluster will have been assigned n_k nodes, which means that each non-empty cluster has been assigned a node $n_k - 1$ times, the nominator in the first case gives $\prod_k (n_k - 1)!$. Finally J nodes were placed in total, resulting in a denominator of $\prod_{j=1}^{J}(j + \alpha - 1)$ which is equal to $\frac{(J+\alpha-1)!}{(\alpha-1)!}$. Adding all this up, we end up with the following formula:

$$P(\mathbf{Z}|\alpha) = \frac{\alpha^K (\alpha - 1)! \prod_k (n_k - 1)!}{(J + \alpha - 1)!} \tag{2.38}$$

Using the gamma function, this can be reduced to:

$$P(\mathbf{Z}|\alpha) = \frac{\alpha^K \Gamma(\alpha) \prod_k \Gamma(n_k)}{\Gamma(J + \alpha)} \tag{2.39}$$

With this equation we can calculate the probability of a cluster assignment $\mathbf{Z}$ with respect to the Chinese Restaurant Process.

2.3.4 Derivation Of The Infinite Relational Model

The idea behind the Infinite Relational Model (IRM) is to cluster a number of nodes based on their connection to one another, such that nodes that behaves similarly are clustered together. This is done by making some assumptions about how these nodes behaves in order to find an optimal model that clusters these nodes together. These assumptions have been mentioned earlier but to summerize, we assume that the nodes are assigned to clusters $\mathbf{Z}$ according to a Chinese Resourant Process and the probability of links between these clusters η can be described by the `Beta` distribution, while the probability that a link exist between two nodes A is given by a weighted Bernoulli distribution, based on the probability of links between the clusters these two nodes belong to.

$$\begin{aligned} \mathbf{Z} &\sim CRP(\alpha) \\ \eta_{lm} &\sim Beta(\beta^+, \beta^-) \\ \mathbf{A}_{ij} &\sim Bernoulli(\eta_{z_i, z_j}) \end{aligned}$$

The goal of IRM is to estimate the clustering $\mathbf{Z}$ given the connection between nodes A and the three hyper-parameters $h = \{\alpha, \beta^+, \beta^-\}$, in other words we want to sample the posterior distribution $P(\mathbf{Z}|A, h)$. This distribution is not given directly but we do know the probability of $\mathbf{Z}$, η and A, as the groups, interactions and links are assumed independent:

$$
\begin{aligned}
P(\mathbf{A}, \mathbf{Z}, \eta|\alpha, \beta^+, \beta^-) &= CRP(\mathbf{Z}|\alpha) Bernoulli(\mathbf{A}|\eta) Beta(\eta|\beta^+, \beta^-) \\
&= CRP(\mathbf{Z}|\alpha) \prod_{l \leq m} \eta_{lm}^{N_{lm}^+} (1-\eta_{lm})^{N_{lm}^-} \prod_{l \leq m} \frac{\Gamma(\beta^+ + \beta^-)}{\Gamma(\beta^+)\Gamma(\beta^-)} \eta_{lm}^{\beta^+ - 1} (1-\eta_{lm})^{\beta^- - 1} \\
&= CRP(\mathbf{Z}|\alpha) \prod_{l \leq m} \left(\frac{\Gamma(\beta^+ + \beta^-)}{\Gamma(\beta^+)\Gamma(\beta^-)} \eta_{lm}^{N_{lm}^+ + \beta^+ - 1} (1-\eta_{lm})^{N_{lm}^- + \beta^- - 1} \right)
\end{aligned}
$$

Where N_{lm}^+ and N_{lm}^- are the number of links and missing links between cluster l and m respectively.

We can obtain $P(\mathbf{A}, \mathbf{Z}|h)$ by integrating out η:

$$
\begin{aligned}
P(\mathbf{A}, \mathbf{Z}|h) &= \int_0^1 P(\mathbf{A}, \mathbf{Z}, \eta|h)\, d\eta \\
&= \int_0^1 CRP(\mathbf{Z}|\alpha) \prod_{l \leq m} \left(\frac{\Gamma(\beta^+ + \beta^-)}{\Gamma(\beta^+)\Gamma(\beta^-)} \eta_{lm}^{N_{lm}^+ + \beta^+ - 1} (1-\eta_{lm})^{N_{lm}^- + \beta^- - 1} \right) d\eta \\
&= CRP(\mathbf{Z}|\alpha) \prod_{l \leq m} \left(\frac{\Gamma(\beta^+ + \beta^-)}{\Gamma(\beta^+)\Gamma(\beta^-)} \int_0^1 \eta_{lm}^{N_{lm}^+ + \beta^+ - 1} (1-\eta_{lm})^{N_{lm}^- + \beta^- - 1}\, d\eta \right)
\end{aligned}
$$

In formula 2.19 we determined beta function as:

$$
B(a, b) = \int_0^1 \eta^{a-1} (1-\eta)^{b-1} d\eta = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{2.40}
$$

Using this definition of the beta function and inserting the formula for the Chinese Restourant Process from equation 2.39 yields:

$$
P(\mathbf{A}, \mathbf{Z}|h) = \frac{\alpha^K \Gamma(\alpha) \prod_k \Gamma(n_k)}{\Gamma(J+\alpha)} \prod_{l \leq m} \frac{B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-)}{B(\beta^+, \beta^-)} \tag{2.41}
$$

This is the likelihood of the Infinite Relational Model, which we use to evaluate the performance of the samplers.

Using Bayes theorem we can now find the posterior clustering probability $P(\mathbf{Z}|A,h)$ using this formula:

$$P(\mathbf{Z}|\mathbf{A},h) = \frac{P(\mathbf{A}|\mathbf{Z},h)P(\mathbf{Z}|h)}{\sum_{\mathbf{Z}} P(\mathbf{A},\mathbf{Z}|h)} \quad (2.42)$$

$$= \frac{P(\mathbf{A},\mathbf{Z}|h)}{\sum_{\mathbf{Z}} P(\mathbf{A},\mathbf{Z}|h)} \quad (2.43)$$

It is possible to use this formula to calculate the clustering $\mathbf{Z}$. Doing this is however infeasible, as the number of combinations for clustering the nodes increases exponentially with the number of nodes, which means that we need another way to estimate the clustering. Instead we relax the problem with Markov Chain Monte Carlo, to estimating the clustering using Gibbs sampling, which we later extend with Split-Merge sampling.

2.4 Markov Chain Monte Carlo

Bayesian modelling is often a hard computational problem, as obtaining the posterior distribution can demand integration of difficult and high-dimensional functions [25]. Instead of calculation an exact distribution, a common technique is to use a simulation to generate independent draws from the posterior distribution and use these draws to estimate the distribution.

Markov Chain Monte Carlo (MCMC) denotes a class of algorithms capable of generating chains of such simulated draws. A Markov Chain relies on a set of states S and some associated transition probabilities. For every pair of states s and s' the transition probability $T(s'|s)$ gives the probability of transitioning from state s to state s'. From some initial state s_0 a Markov chain of length n can be created by n times going to a new state, determined by the transition probabilities:

$$\{s^0, s^1, s^2, ..., s^n\}$$

Such a chain upholds the Markov property, that every state s^i in the chain only depends on the previous state s^{i-1}. A state in the Markov Chain contains an assignment for all the variables in the model. In the case of cluster analysis this refers to the cluster assignments of all nodes.

A Markov Chain is called *regular* if it upholds the following two properties:

- For all pair of probable states there exists a path where all transitions in the path have non-zero probabilities.
- For all probable states s, $T(s'|s)$ is non-zero.

A Markov Chain is *detailed balanced* if there exists a unique distribution p, such that for all pair of states s and s':

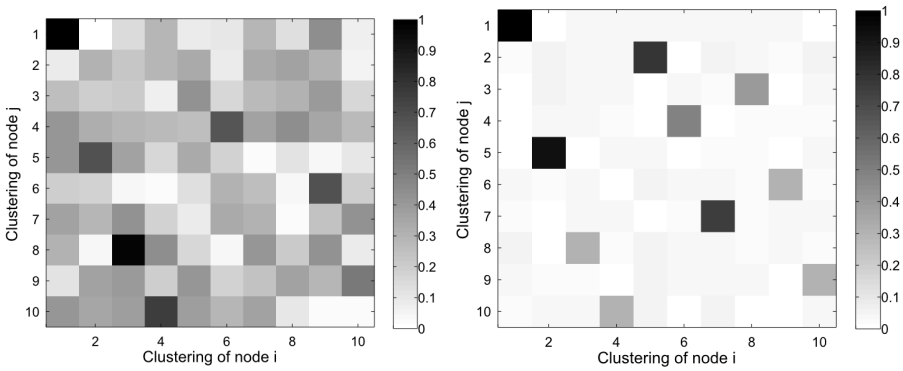
$$p(s)T(s'|s) = p(s')T(s|s')$$

A very important property of regular Markov Chains with detailed balance is that convergence to a unique stationary distribution will be reached eventually [14]. The advantage of using MCMC is therefore that the chain eventually will converge such that the draws closely approximate draws from the real posterior distribution. However it is unfortunately not possible to determine the time it takes until converges is certain. The procedure needs some *burn-in* time to allow it to approximate the posterior distribution. The most used MCMC algorithms include Metropolis sampling [11], Metropolis-Hastings [7] and Gibbs sampling [4, 23] The Gibbs sampler only changes one variable at a time. It might take a very long time before it reaches convergence and it might be very difficult to reach states that can only be reached by going through states with low probabilities. On the other hand, the Split-Merge sampler can reassign multiple nodes at a time, which can shorten burn-in time.

We will use both Gibbs and Split-Merge sampling in order to estimate the clustering $\mathbf{Z}$. These are random walk algorithms, that moves around the state space. In the Gibbs sampler, each parameter is changed one at a time while the Split-Merge sampler proposes an entire split or merge, which is then accepted or rejected according to the Metropolis-Hastings acceptance probability.

If there are many local minima in the state space, that are much more likely than their surrounding states, then MCMC is likely to get stuck. As an example, figure 2.6 shows the model likelihood of two state spaces (a) and (b) for two nodes. In (a) we see that by only changing one parameter at a time (along one axis), it is easy to jump from a high probability distribution to another, making the Gibbs sampler optimal for these types of problems. However in (b) we see that two parameters has to be changed simultaneously in order to jump from a high probability model to another. This means that for the Gibbs sampler to jump between high probability models, it first have to move to a very unlikely model, which may take towards infinity many attempts, depending on how unlikely these distributions are.

In order to estimate the clustering model $\mathbf{Z}$ and when it is found, the MCMC algorithm is run multiple times with different initializations, each creating a



(a) Easy Gibbs sampling state space. (b) Difficult Gibbs sampling state space.

Figure 2.6: State spaces for clustering of two nodes.

chain of accepted models. We then say that an acceptable model is found when several of these chains converge to the same model, although we cannot be absolutely certain that it is the real posterior distribution.

2.5 Gibbs sampling

Gibbs sampling works by starting in a random state. It then iteratively reassigns all nodes one at a time, according to the probability that the node belongs to each of the clusters, denoted as a Gibbs sweep. The probability of assigning a single node is given in relation to IRM, using Bayes theorem:

$$P(z_{ni} = 1|\mathbf{A}, \mathbf{Z}_{\setminus i}, h) = \frac{P(\mathbf{A}, \mathbf{Z}_{\setminus i}, z_{ni} = 1|h)}{\sum_{o \cup new} P(\mathbf{A}, \mathbf{Z}_{\setminus i}, z_{oi} = 1|h)} \tag{2.44}$$

where $\mathbf{Z}_{\setminus i}$ denotes the clustering of all nodes except node i and $z_{ni} = 1$ denotes that node i is assigned to cluster n. $o \cup new$ means that o represents all existing clusters as well as a new empty cluster. Using this formula, we only need to calculate the probability of placing the nodes in each of the existing cluster and in a new cluster, to find the probability that node i belongs to cluster n.

As the probability of $P(\mathbf{A}, \mathbf{Z}|h)$ is given in formula 2.41, we are now able to perform Gibbs sampling. After inserting the formula into 2.44, the Gibbs prob-

ability can however be simplified much more.

We immediately see that the constants $\Gamma(\alpha)$, $\Gamma(J+\alpha)$ and $B(\beta^+, \beta^-)$ in formula 2.41 can be reduced. When we expand the denominator, this equation becomes very large, so in order to simplify the equation we have defined a variable G as:

$$G = \prod_{l \leq m} \left(B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-) \right) \tag{2.45}$$

Furthermore we define $G_{i,o}$ as the value of G, where node i is inserted into cluster o. If $o = new$, then the node is inserted into an empty cluster. In other words, N_{lm}^+ and N_{lm}^- assumes values according to the cluster assignment of node i.

Inserting equation 2.41 into equation 2.44 yields two different cases as i is either assigned to an non-empty or empty cluster. In the first case we get:

$$\begin{aligned} & P(z_{ni} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) \\ &= \frac{G_{i,n}\alpha^K \cdot \Gamma(n_n + 1) \prod_{k \setminus n} \Gamma(n_k)}{\sum_o (G_{i,o}\alpha^K \Gamma(n_o + 1) \prod_{k \setminus o} \Gamma(n_k)) + G_{i,new}\alpha^{K+1}\Gamma(1) \prod_k \Gamma(n_k)} \\ &= \frac{G_{i,n}\alpha^K \cdot n_n \prod_k \Gamma(n_k)}{\sum_o (G_{i,o}\alpha^K n_o \prod_k \Gamma(n_k)) + G_{i,new}\alpha^{K+1} \prod_k \Gamma(n_k)} \end{aligned}$$

In the numerator of this equation cluster n contains one additional node, as node i is placed into it. The denominator has been split into two parts; first we sum over all the non-empty clusters o and then add the probability of belonging to an empty cluster. In the case of the empty cluster, K increases by one.

From this we see, that α^K and $\prod_k \Gamma(n_k)$ can be reduced:

$$P(z_{ni} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) = \frac{G_{i,n} n_n}{\sum_o (G_{i,o} n_o) + G_{i,new}\alpha} \tag{2.46}$$

Similarly we can find the probability for the second case where node i is placed in a new cluster. We again reduce the formula in the exact same way as above, but in this case we end up with an additional α in the numerator, as the number of cluster has increased and $n_{new} = 1$:

$$P(z_{new,i} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) = \frac{G_{i,new}\alpha}{\sum_o (G_{i,o} n_o) + G_{i,new}\alpha} \tag{2.47}$$

The denominator for both cases are exactly the same and the set of numerators for all cluster assignments has a 1 to 1 correspondence with each term in the denominator, as it should be since we calculate and normalize over all possible combinations.

Until now, we have only mentioned how $G_{i,o}$ works, but not described the math behind it. $G_{i,o}$ is equal to G except for the case where $l = o$, by defining r_{im} as the number of links node i has to cluster m, we can calculate $G_{i,o}$ for these changed values as:

$$G_{i,l=o} = \prod_m \left(B(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-) \right) \tag{2.48}$$

Furthermore $G_{i,o}$ can be defined by G and the change that occurs when node i is inserted into cluster o:

$$G_{i,o} = G \cdot G_{change_{i,o}} \tag{2.49}$$

If we insert this expression for $G_{i,o}$ into formula 2.46 and 2.47, G can be reduced:

$$P(z_{n,i} = 1 | \mathbf{A}, \mathbf{Z}_{\backslash i}, h) = \frac{G_{change_{i,n}} n_n}{\sum_o (G_{change_{i,o}} n_o) + G_{change_{i,new}} \alpha} \tag{2.50}$$

$$P(z_{new,i} = 1 | \mathbf{A}, \mathbf{Z}_{\backslash i}, h) = \frac{G_{change_{i,new}} \alpha}{\sum_o (G_{change_{i,o}} n_o) + G_{change_{i,new}} \alpha} \tag{2.51}$$

Using the definition of G and $G_{i,o}$ we can calculate $G_{change_{i,o}}$ using the relationship given in formula 2.49. Since G equals $G_{i,o}$ for all $l \neq o$ in definition 2.45, we reduce these terms:

$$G_{change_{i,o}} = \frac{G_{i,o}}{G} \tag{2.52}$$

$$G_{change_{i,o}} = \frac{\prod_m B(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-)}{\prod_{l=o} \prod_m B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-)} \tag{2.53}$$

$$= \prod_m \left(\frac{B(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-)}{B(N_{om}^+ + \beta^+, N_{om}^- + \beta^-)} \right) \tag{2.54}$$

This formula for $G_{change_{i,o}}$ is used for the cases where node i is placed into an existing cluster, it does however also apply to the case where the node i is placed into a new cluster. As a new cluster contains no nodes, it has neither any links nor any missing links. Thus all N^+ and N^- are equal to zero. Furthermore $r_{new,o}$ is equal to the number of links N^+ after inserting the node and $n_m - r_{new,o}$ is equal to the number of missing links N^- after inserting the node. $G_{i,new}$ can hence also be replaced $G_{change_{i,new}}$.

2.5.1 Ensuring numeric stability

In the previous section we looked at the mathematical simplification of the Gibbs sampling algorithm, in this section we will look at algorithmic optimizations in relation to a computer program.

First off in formula 2.50 and 2.51, the denominator are the same for placing a node i in any of the clusters. Secondly, the numerators has a one to one correspondance with the denominator elements. Hence we define a vector Q_i as all the probability numerators for placing node i in each of the clusters:

$$Q_i = \begin{bmatrix} G_{change_{i,1}} n_1 \\ G_{change_{i,2}} n_2 \\ ... \\ G_{change_{i,K}} n_K \\ G_{change_{i,new}} \alpha \end{bmatrix} \tag{2.55}$$

The first K elements represents the numerator probability of placing node i in cluster 1 through K respectively, while the last element represents placing the node in a new cluster.

Using this vector we can find the probability of placing the node in any of the existing or a new cluster by calculating each of these values and normalizing with their sum:

$$P_i = \begin{bmatrix} P(z_{1,i} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) \\ P(z_{2,i} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) \\ ... \\ P(z_{K,i} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) \\ P(z_{new,i} = 1 | \mathbf{A}, \mathbf{Z}_{\setminus i}, h) \end{bmatrix} = \frac{Q_i}{\mathsf{sum}(Q_i)} \tag{2.56}$$

The calculation of $G_{change_{i,o}}$ contains evaluations of the Beta function, which returns large values for even small input values, this means the division of these

quickly becomes numerically unstable. In order to ensure numeric stability for machine precision, these posterior probabilities are calculated in the log domain and each value is divided with the largest value.

First, we can divide with the largest value of Q_i without changing the probabilities, since P_i is calculated by normalizing each of these with the sum of Q_i. In order to do this we first introduce a new variable Q'_i such that:

$$\begin{aligned} Q'_i &= \frac{Q_i}{\max\limits_i(Q_i)} \\ P_i &= \frac{Q'_i}{\sum\limits_i(Q'_i)} \end{aligned} \tag{2.57}$$

We can now take the logarithm and exponent of Q'_i, to help stabilize the beta calculations:

$$\begin{aligned} Q'_i &= e^{\ln(Q'_i)} \\ &= e^{\ln(Q_i / \max\limits_i(Q_i))} \\ &= e^{\ln(Q_i) - \max\limits_i(\ln(Q_i))} \end{aligned} \tag{2.58}$$

$$ln(Q_i) = \begin{bmatrix} ln(G_{change_{i,1}}) + ln(n_1) \\ ln(G_{change_{i,2}}) + ln(n_2) \\ \dots \\ ln(G_{change_{i,K}}) + ln(n_K) \\ ln(G_{change_{i,new}}) + ln(\alpha) \end{bmatrix} \tag{2.59}$$

The logarithm of $G_{change_{i,o}}$ can be pushed into the calculations, in order to change beta to betaln, adding numerical stability to the algorithm:

$$\begin{aligned} ln(G_{change_{i,o}}) =& ln\left(\prod_m \left(\frac{B(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-)}{B(N_{om}^+ + \beta^+, N_{om}^- + \beta^-)}\right)\right) \\ =& \sum_m ln\left(\frac{B(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-)}{B(N_{om}^+ + \beta^+, N_{om}^- + \beta^-)}\right) \\ =& \sum_m \Big(Betaln(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-) \\ & - Betaln(N_{om}^+ + \beta^+, N_{om}^- + \beta^-)\Big) \end{aligned} \tag{2.60}$$

```
1 for each sweep
2     for each node i
3       calculate N+ and N−, n ignoring i
4       calculate r for the node
5       calculate the probability change of the model
            when assigning i to each cluster
6       choose a cluster assignment based on these
            probabilities
```

Figure 2.7: Naive Gibbs Pseudo-code.

Using these optimizations, we are now able to easily calculate the likelihood of assigning node i to each of the clusters with numerical stability. As described in section 2.4 MCMC can then use these probabilities to iteratively place each node in a cluster. Thus, the key operation necessary to evaluate the posterior is the calculation of the logarithm of the beta function.

2.5.2 Naive Gibbs algorithm

We now have all the necessary formulas to implement a MCMC procedure using Gibbs sampling with numeric stability. Figure 2.7 shows the pseudo-code for a naive implementation of a Gibbs sampler.

The calculations of N^+, N^-, n and r are straight forward, as they only rely on the current cluster assignments. The calculations of the change probabilities in P_i are a bit more complex. They can be calculated by the following procedure:

1. Calculate the logarithm of the beta-function for all cluster assignments for the current node i into all existing clusters o and a new cluster $o = new$; formula 2.60.
2. For all the resulting values we add the logarithm of the number of nodes for non-empty clusters and $ln(\alpha)$ for the empty cluster; formula 2.59.
3. Subtract each value with the maximum value and exponentiate them; formula 2.58.
4. Divide all values with their sum; formula 2.57.

Based on these probabilities a new cluster assignment can then be randomly selected for node i.

In the case where multiple graphs are used in the analysis, the nodes are placed in the same clusters for all graphs, but N^+ and N^- differs between the graphs. The probability of a node belonging to a cluster is independent between graphs. Hence the probability of assigning the node to a cluster can be calculated as the product of the probabilities of assigning the node to the cluster, for each graph.

2.6 Split-Merge sampling

The Split-Merge procedure with restricted Gibbs sampling is presented by Jain and Neal in [8]. The idea behind Split-Merge is that we split or merge entire clusters rather than moving one node at a time, as illustrated in figure 2.8. This is done by selecting two distinct nodes at random. If they are in the same cluster we propose to split the cluster, if they are in two different clusters we propose to merge the two clusters. These proposals are evaluated using Metropolis-Hastings acceptance probability to determine whether they are accepted or rejected [8].

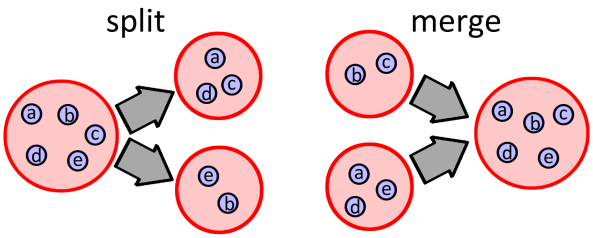


Figure 2.8: Splitting and merging clusters.

In the case that the two selected nodes i and j are assigned to the same cluster, we perform the following split procedure, which is illustrated in figure 2.9:

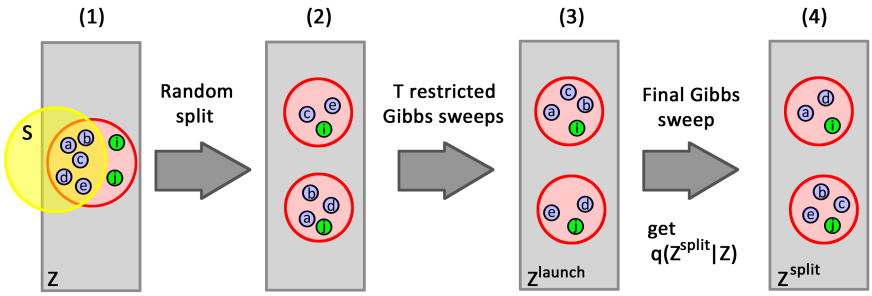


Figure 2.9: Split procedure.

1. Let S denote the nodes clustered with i and j, excluding i and j.
2. Place i and j in two empty clusters and assign each node in S randomly between these.
3. Perform T restricted Gibbs sweeps on the nodes in S and let $\mathbf{Z}^{launch}$ denote the resulting clustering.
4. Perform **one** final restricted Gibbs sweep over the nodes in S.
   (a) Let $\mathbf{Z}^{split}$ denote the resulting clustering.
   (b) As each node in S is assigned to $\mathbf{Z}^{split}$, calculate the product of these transitions, in order to find the proposal probability $q(\mathbf{Z}^{split}|\mathbf{Z})$.
   (c) If the proposal is accepted, then $\mathbf{Z}^{split}$ becomes the next state, otherwise the current state $\mathbf{Z}$ remains.

In the case that the two selected nodes i and j are assigned to different clusters, we perform the merge procedure. This procedure is very simple as the nodes are simply placed in the same cluster denoted $\mathbf{Z}^{merge}$. However in order to calculate the transition probability $q(\mathbf{Z}|\mathbf{Z}^{merge})$, we follow a procedure similar to that of splitting a cluster, in order to ensure detailed balance. The merge procedure follows the first 3 steps of the split procedure exactly, obtaining the cluster configuration $\mathbf{Z}^{launch}$. Using this cluster configuration, the merge procedure continues as follows and illustrated in figure 2.10.

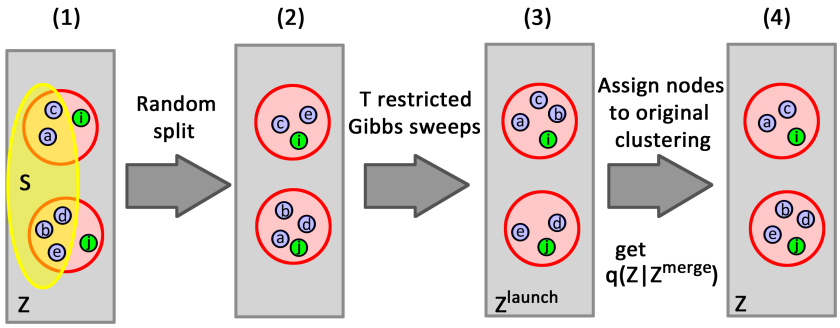


Figure 2.10: Merge procedure.

4. Assign each node in S to their original clustering $\mathbf{Z}$.
   (a) As each node in S is assigned according to $\mathbf{Z}$, calculate the product of these transitions, in order to find the proposal probability $q(\mathbf{Z}|\mathbf{Z}^{merge})$.

(b) If the proposal is accepted, then $\mathbf{Z}^{merge}$ becomes the next state, otherwise the current state $\mathbf{Z}$ remains.

To evaluate whether a proposal $\mathbf{Z}^{split}$ or $\mathbf{Z}^{merge}$ is accepted or rejected, the Metropolis-Hastings acceptance probability is used:

$$\alpha(\mathbf{Z}^*, \mathbf{Z}) = min\left[1, \frac{q(\mathbf{Z}|\mathbf{Z}^*)}{q(\mathbf{Z}^*|\mathbf{Z})}\frac{P(\mathbf{Z}^*)}{P(\mathbf{Z})}\frac{L(\mathbf{Z}^*|\beta^+, \beta^-)}{L(\mathbf{Z}|\beta^+, \beta^-)}\right] \tag{2.61}$$

where $\mathbf{Z}^*$ is either $\mathbf{Z}^{split}$ or $\mathbf{Z}^{merge}$. The calculation of $q(\mathbf{Z}^{split}|\mathbf{Z})$ and $q(\mathbf{Z}|\mathbf{Z}^{merge})$ are given in the procedures above, while their opposite $q(\mathbf{Z}|\mathbf{Z}^{split})$ and $q(\mathbf{Z}^{merge}|\mathbf{Z})$ is always 1; there is only one way to place all nodes in one cluster. The prior $P(\mathbf{Z})$ is given by the Chinese Restourant Process, while the likelihood $L(\mathbf{Z}|\beta^+, \beta^-)$ is given by the beta distribution. We now define i' and j' to specify the clusters z_i^* and z_j^* while i and j specifies z_i and z_j.

This means that we can simplify the expression even more, for the prior of splitting a cluster we get:

$$\frac{P(\mathbf{Z}^{split})}{P(\mathbf{Z})} = \frac{\alpha^{K+1}\Gamma(n_{i'}^{split})\Gamma(n_{j'}^{split})\prod_{k\setminus(i,j)}\Gamma(n_k)}{\alpha^K\prod_k\Gamma(n_k)} \tag{2.62}$$

$$= \frac{\alpha\Gamma(n_{i'}^{split})\Gamma(n_{j'}^{split})}{\Gamma(n_i)} \tag{2.63}$$

where $n_{i'}^{split}$ and $n_{j'}^{split}$ are the number of nodes in the two clusters created by the split procedure. Similarly for the merge case the prior can be simplified to:

$$\frac{P(\mathbf{Z}^{merge})}{P(\mathbf{Z})} = \frac{\alpha^{K-1}\Gamma(n_{i'}^{merge})\prod_{k\setminus(i,j)}\Gamma(n_k)}{\alpha^K\prod_k\Gamma(n_k)} \tag{2.64}$$

$$= \frac{\Gamma(n_{i'}^{merge})}{\alpha\Gamma(n_i)\Gamma(n_j)} \tag{2.65}$$

The likelihood for a clustering can be calculated using the reduced expression we found for the probability of links and interactions between clusters, in the

latter part of equation 2.41, which states:

$$P(\mathbf{Z}|\beta^+,\beta^-) = \prod_{l\leq m} \frac{B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-)}{B(\beta^+,\beta^-)} \tag{2.66}$$

For the likelihood fraction of split, this formula only changes for $l = z_i$ and hence we can reduce all other multiplications.

$$\begin{aligned}
\frac{L(\mathbf{Z}^{split}|\beta^+,\beta^-)}{L(\mathbf{Z}|\beta^+,\beta^-)} &= \frac{\prod_{l\in(i',j')}\prod_{m\in\mathbf{Z}^{split}} \frac{B(N_{lm}^{+,split}+\beta^+, N_{lm}^{-,split}+\beta^-)}{B(\beta^+,\beta^-)}}{\prod_{m=1}^{K} \frac{B(N_{im}^+ + \beta^+, N_{im}^- + \beta^-)}{B(\beta^+,\beta^-)}} \\
&= \frac{\prod_{l\in(i,j)}\prod_{m\in\mathbf{Z}^{split}} B(N_{lm}^{+,split}+\beta^+, N_{lm}^{-,split}+\beta^-)}{B(\beta^+,\beta^-)^{K+2}\prod_{m=1}^{K} B(N_{im}^+ + \beta^+, N_{im}^- + \beta^-)}
\end{aligned}$$

where $N_{lm}^{+,split}$ and $N_{lm}^{-,split}$ denotes the number of links and non-links between cluster l and m after the split. If we instead look at the likelihood fraction of merge, we get that two clusters changes in the orininal clustering, $l = i$ and $l = j$ and hence by reducing the other multiplications we get:

$$\begin{aligned}
\frac{L(\mathbf{Z}^{merge}|\beta^+,\beta^-)}{L(\mathbf{Z}|\beta^+,\beta^-)} &= \frac{\prod_{m\in\mathbf{Z}^{merge}} \frac{B(N_{i'm}^{+,merge}+\beta^+, N_{i'm}^{-,merge}+\beta^-)}{B(\beta^+,\beta^-)}}{\prod_{l\in(i,j)}\prod_{m=1}^{K} \frac{B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-)}{B(\beta^+,\beta^-)}} \\
&= \frac{B(\beta^+,\beta^-)^{K+2}\prod_{m\in\mathbf{Z}^{merge}} B(N_{i'm}^{+,merge}+\beta^+, N_{i'm}^{-,merge}+\beta^-)}{\prod_{l\in(i,j)}\prod_{m=1}^{K} B(N_{lm}^+ + \beta^+, N_{lm}^- + \beta^-)}
\end{aligned}$$

Using these formulas we can now calculate the probability of accepting a split

or merge. However to ensure numeric stability we calculate in the log-domain, exactly as explained in chapter 2.5.1 Ensuring numeric stability.

For multiple graphs the prior $P(\mathbf{Z})$ does not change, as it only depends on cluster sizes. The likelihood changes as we now need to iterate over all graphs, calculating the product of the independent likelihoods for each graph.

2.7 Model evaluation

In order to estimate how well the IRM model and samplers work, we need some means to compare the similarity between clusterings and some means to evaluate the mixing properties of the two samplers.

There are several ways to evaluate the performance of samplers. If the ground truth clustering is known in advance, then a similarity measure can be used between this and the best clustering found by the sampler. If on the other hand the ground truth is not known in advance, then the similarity between samples/states in several distinct runs can be used instead. When a sampler mixes, then these chains will become more similar as they progress, additionally the average similarity within a chain should be similar to the average similarity between two chains. This indicates that the sampler is able to freely move around the state space and hence not get stuck at some local mode.

There exists several similarity measures; conditional entropy, joint entropy, conditional mutual information and mutual information, among many others. All of these have both strengths and weaknesses, depending on what is being measured and how it should be similar. We have chosen to use normalized mutual information, as it is commonly used measure of dependence in information theory. Using several graphs for the same network it would also be possible to apply different strategies, such as cross-validation.

2.7.1 Ground truth

The Infinite Relational Model is a generative model, which means that we can use it to generate a synthetic network. By creating these network, we know the ground truth, since the clustering is determined as part of the generative process. We can then perform the IRM analysis on the synthetic data, and by evaluating the results against the ground truth, we can evaluate the performance of the model. In a real world network, we cannot be sure that the desired ground truth

is actually determinable by analysing the data. For a synthetic network we are certain that the structure is in the data, as it was generated to contain this structure. This also means that samplers may work well on generated models but not on real world samples. For this reason we have chosen to analyse real world samples rather than synthetic data.

2.7.2 Informational entropy

In order to understand how normalized mutual information works, we first introduce entropy, as the entropy of each of the two clusterings is used in order to normalize the mutual information.

The term entropy originated as a classical thermodynamic quantity related to the heating of a system by physical processes. In statistical thermodynamic, the concept of entropy can be considered a measure of the "disorder" or "uncertainty" of a given thermodynamic system. The concept of *information entropy* was first introduced by Claude Shannon in the paper "A Mathematical Theory of Communication" [21] as an aid to statistically quantify information loss in communication lines [Estimation of Mutual Information: A Survey]. For a discrete random variable X, the entropy can be expressed as:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i)$$

The general concept of entropy is well suited to describe the uncertainty or unpredictability of a single random variable, but is not usable for more than a single variable [26].

2.7.3 Normalized Mutual Information

To compare two clusterings we use the concept of normalized mutual information. This is based on the concept of mutual information, which was also first introduced by Shannon.

The mutual information $I(X, Y)$ of two discrete random variables X and Y measures how much the uncertainty is reduced in X when gaining the knowledge of Y. The mutual information is given by:

$$I(X, Y) = \sum_{x \in X, y \in Y} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right)$$

where $p(x)$ and $p(y)$ are marginal probability distribution functions of X and Y while $p(x, y)$ is the joint probability distribution.

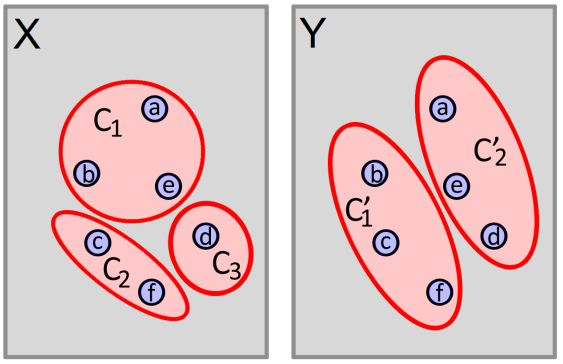


Figure 2.11: Two different clusterings of the same nodes.

An example of mutual information between clusterings can be seen from figure 2.11. The marginal distributions of the two clusterings X and Y gives the probability that a node belongs to a given cluster:

$$\text{for X: } p(x) = \begin{cases} 3/6, & x = c_1 \\ 2/6, & x = c_2 \\ 1/6, & x = c_3 \end{cases} \qquad \text{for Y: } p(y) = \begin{cases} 3/6, & y = c'_1 \\ 3/6, & y = c'_2 \end{cases}$$

The joint distribution gives the probability that a node belongs to a given cluster in X and a given cluster in Y:

$$p(x, y) = \begin{array}{c|cc} & c'_1 & c'_2 \\ \hline & 1/6 & 2/6 \\ & 2/6 & 0 \\ & 0 & 1/6 \end{array} \begin{array}{l} c_1 \\ c_2 \\ c_3 \end{array}$$

From this the mutual information can be calculated to $I(X, Y) = 0.3749$. The upper bound for the mutual information is dependent on the complexity of the variables X and Y. As we want a value in the range $[0; 1]$, we instead use the following normalized mutual information (NMI):

$$\text{NMI}(X, Y) = \frac{2 \cdot I(X, Y)}{H(X) + H(Y)}$$

where $H(X)$ and $H(Y)$ is the entropy of cluster configuration X and Y.

An NMI of zero indicates that the cluster configurations do not share any information, while an NMI of one indicates that cluster configurations shares as much information as they can, in which case clusterings are identical, although they may be permuted.

2.7.4 Mixing ability of the Sampler

We can calculate how the likelihood of how the model changes with the number of iterations using formula 2.41. By plotting this likelihood for multiple runs we can estimate how well the sampler mixes. If the sampler mixes, then no matter where we start a chain, the likelihood for each chain should converge with the likelihood of the other chains. If this is not the case, it indicates that the sampler may be stuck in a local minimum.

2.8 Large scale computation

In this section we describe the hardware and software aspects of performing large scale computations. We first introduce how the CPU and GPU works and then compare them.

2.8.1 CPU

The central processing unit (CPU) is the central part of a computer, capable of rapidly performing a wide range of arithmetic operations in sequence. It executes programs by processing the instructions provided by the program code. The program data and instructions are read from the memory, while stored data is written back to the memory. The CPU can process the data much faster than the transfer rate of data from the random access memory (RAM) to the CPU. This means the CPU has to wait for the data to catch up, resulting in reduced CPU performance. To overcome these issues, modern CPU's rely on different levels of memory caches and prefetching. Figure 2.12 shows a simplified version of the memory structure associated with the CPU.

Though the capacity of the memory caches are much smaller than for the RAM, their response time is much smaller and are hence used as buffers to allow fast read and write operations by the CPU.

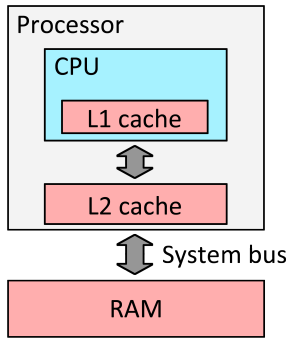


Figure 2.12: CPU memory architecture.

The L2 cache is larger than the L1 cache but is placed a bit off the processor chip and has a bit larger latency compared to the L1 cache. Accessing data from the L2 cache is however still a lot faster than accessing the RAM over the system bus. When the CPU needs an instruction or variable it will look for it in the caches in a fall through model, where it first checks the L1, L2 and then the RAM. Multi-core processors tends to have the L2 cache replaced by an L3 cache, which is shared by all the cores on the processor, while others have both an L2 and L3 cache.

The CPU uses these caches to prefetch data such that it is available before it is needed. This is implemented such that when a cache miss occurs (some accessed data is not in the cache), then an entire cache line of typically 1 kilo bytes are fetched from the RAM and stored in the cache, even though the current instruction only needed the first 32 bits. This means that if the next 32 bits are accessed in the next instruction, this will happen extremely fast.

In order to implement efficient CPU programs it is essential to be aware of this cache structure in order to avoid idle time for the processor.

2.8.2 GPU

A *graphical processing unit* (GPU) is a piece of dedicated hardware invented to manipulate, store and render image-data on a very large scale. The architecture of a GPU is therefore designed to facilitate graphic operations as efficiently as possible. This includes operations to perform geometric transformations, illumination and rasterization, which all require heavy computations. The GPU is designed such that the same operations can be performed on massive amounts

of data in parallel. This is known as Single Instruction Multiple Data (SIMD) and in order to provide this, the GPU utilizes hundreds of physical processing units (stream processors), that perform the same instructions in parallel. Many modern GPUs are now designed to not only perform graphic operations, but with a much more flexible architecture in mind that allows the huge computational capacity of the shader pipeline and memory structure to be utilized in a more generic way. This means that it is possible to work on data that is not just image data. Such hardware is often referred to as *general purpose graphics processing units* (GPGPU).

Recent development in improved compilers, new development tools, toolkits and language frameworks such as OpenCL and CUDA, provides a much higher level of abstraction from the hardware, making it possible for normal programmers to take advantage of the massive parallel computing power the GPGPU offers. Thereby allowing computer-intensive parts of an application to be run on a GPU, offloading the CPU and improving the performance of the entire application.

Code that is run on the GPU is called a kernel. To utilize the parallel resources a kernel needs parallelism, such that it can be computed by multiple threads simultaneously. On a GPU these threads are called work-items and are grouped into blocks called work-groups that all execute the same kernel in parallel. Individual groups of the physical stream processors on the GPU are called compute units (CU's), which have some associated shared memory. All items in a work-group are executed simultaneously on the same CU's, and can communicate with each other through the shared memory. Due to the GPU design, it is only possible to synchronize execution inside a kernel, making the GPGPU inefficient for code with interloop dependencies.

On a GPGPU a work-group can typically consist of 256 or 1024 threads, even though there are much fewer compute units. The GPGPU handles this by splitting the work-group into wavefronts, in which there are as many work-items as compute units. When one wavefront waits to receive data from the memory, all compute units switch to the next wavefront and executes it without any delay.

Figure 2.13 shows the memory structure of GPU compute units (ALUs). The largest memory block is the global GPU memory, which can be accessed by the host-program to store data from the RAM. The size of this memory is in the order of a few gigabytes, and can be seen by all work-items. Accessing this memory is however rather slow, as it is placed off the chip. The latency can however be hidden by using multiple wavefronts and hence avoid performance decrease in memory intensive applications[17].

[**?**]

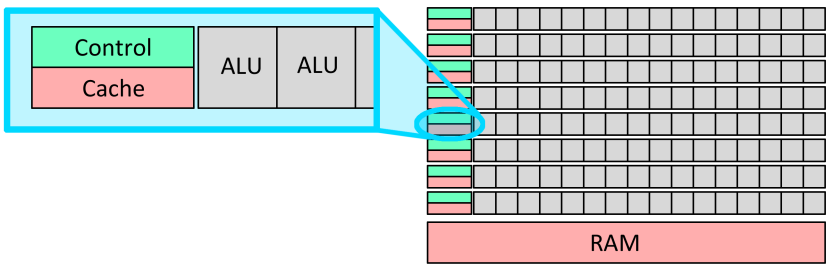


Figure 2.13: Gpu memory architecture.

Each work-group has a shared memory which allows the work-items to collaborate on retrieving data that they all need or share data that needs to be synchronized between work-items. Each work-item has a number of registers along with local memory. In this hierarchy the registers are fastest, then local memory, shared memory and finally the global memory, which is very slow.

Though GPU design is focused on providing high bandwidth, rather than high speed and low latency memory access, the stream paradigm can utilize the fact that multiple work items can collaboratively fetch data that is used by all of them into the shared memory, which can hide the latency of memory fetching.

2.8.3 Comparison of CPU and GPU

While a CPU is designed around a sequential paradigm, where single instructions are executed extremely fast, a GPU resembles the stream processing paradigm, which executes single instructions on multiple data. The difference in the architecture influences what types of problem each paradigm is most suitable for as well as how to utilize the resources efficiently.

The focus of the CPU design is speed, in order to provide very fast calculations. It can provide very high performance for applications with sequential data access patterns. The memory structure are larger than those on the GPU and allows fast random access for very small data structures, that can be stored in the cache. This makes the CPU very good for calculations where the same data is reused as the latency of fetching from the RAM is minimal.

The focus of the GPU design is to provide high bandwidth, where the same calculations are performed on a lot of data in parallel. Using multiple work-items the latency of random memory access can be hidden and hence allow high performance to highly parallel code. The GPU can handle some code dependencies

within a work-group, even without too much overhead. It is however impossible to synchronize execution across work-groups. This can only be achieved by waiting for the entire kernel to finish executing before starting the next kernel, that depended on the previous. This means that if kernels has to synchronize often, it is infeasible to use the GPU.

Ideal programs to be computed on a GPU, must hence contain large data set with a high degree of dataparallelism and few dependencies between data elements, otherwise the CPU is superior.

2.8.4 Program optimization

For large scale computations it is essential to utilize the available resources efficiently. Even though a program is written such that it theoretically is as fast as possible, it can typically still be optimized. This is because the theoretical speed does not take the specific hardware characteristics and limitations into account. Hence it is often possible to increase the performance of the program by applying various optimization techniques, such as changing data structures, calculations and code structure without having to change the algorithm of the program. Furthermore the program can often be parallelized on a CPU, GPU or across multiple computers.

An optimization may require that more data is accessed, more operations are performed or performed in a different order, but resulting in a faster runtime due to the hardware capabilities such as cache misses, parallelization and advanced machine instructions.

In many cases program optimization is a trade-off between memory consumption, computing speed and code readability. Our implementation is not intended for the typical end-user who runs multiple other programs simultaneously, instead we assume that all computer resources are dedicated to running this program. Hence we are less concerned with memory consumption than speed, in fact we always favour fast code at the cost of memory, when possible.

In the following sections we present performance issues that are important to address to work efficiently with large scale computations. We focus mainly on three important optimization techniques:

- Caching results to avoid unnecessary re-calculations.
- Avoid cache misses to avoid overhead of memory fetching.

- Use parallelization to utilize the resources of multiple processing units.

2.8.5 Caching results

In many applications the same calculation might be computed multiple times with the exact same parameters. This is a waste of resources. Instead of re-calculating the exact same value, the value can be stored, such that instead of computing the function it becomes a single lookup to get the value, the next time the function is called for the same parameters. This is known as caching results.

Caching results does not always lead to a higher performance of the program. Fetching the data from memory takes time. If the calculation can be performed on the CPU within that time then caching the results will actually decrease the performance of the program. Caching results may also require large amounts of memory. It therefore becomes a tradeoff between the memory usage and the runtime saved by caching.

2.8.6 Cache misses

Most programs that are not optimized specifically for cache misses, tends to spend a lot of their time waiting for data to be retrieved from the RAM, rather than actually utilizing the power of the CPU. These waiting periods are caused by the program trying to access some variable which are not currently available in the CPU's cache, but has to be fetched from the global RAM, resulting in cache misses. In some cases the data might not even be stored in the RAM, as it may have been moved to the harddisk, which results in extremely slow fetches. While the data is being fetched the CPU cannot continue executing the program, as it must perform each instruction in sequence, even if the next instruction does not have anything to do with the data the current instruction uses. This is one of the limitations of the CPU architecture.

To avoid cache misses, data elements can be stored and accessed sequentially in an array, such that all elements are stored next to each other in the memory. When the first cache miss occurs, both the current and next couple of elements are fetched from the RAM and stored in the cache. This means that when the next element is requested, it is already available in the cache. However since the pre-fetched array uses memory in the cache, it is important to use very little memory until the next element is used, i.e. computations using few variables/arrays. Otherwise the pre-fetched array may be overwritten in the

cache, in order to make room for other variables. This means that in order to avoid cache misses, we both have to place elements in a static list of some kind and balance the number of variables used in each calculation. A very good depiction of this problem was made by Bjarne Stroustrup, presenting a list as both a vector and a linked list to illustrate the huge performance benefits of static arrays in comparison to true object oriented lists [24].

2.8.7 Parallelization

In a non-parallel programming scheme it is only possible to reach a certain level of performance, as the maximal processing speed of CPUs is limited by fundamental physical quantities. For a CPU with a clock frequency of 3 Ghz, light is able to travel about 10 cm in the time it takes for the CPU to perform one instruction, which requires that the signal flow has reached from one end of the CPU to the other. This means that in order to further speed up a program, aside from performing the tricks previously described, we need to parallelize the program; use multiple threads that cooperates in order to perform tasks faster. However, only independent code can be parallelized, which means that if the strictly serial code takes 10 seconds to compute, then no matter how many CPUs cooperate on the task, it will never be faster than 10 seconds. The performance boost of using multiple CPUs can be calculated using Amdahl's law, which states:

$$T(N) = T(1)(B + (1 - B)/N) \tag{2.67}$$

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{B + \dfrac{1 - B}{N}} \tag{2.68}$$

where $T(N)$ is the time it takes for a program to finish using N threads, while $S(N)$ is the maximal speedup using N threads and B is the percentage of code that is strictly serial. An example of this can be seen in figure 2.14, which shows the speedup compared to the number of threads. From this we can see that even if only 5% of the code is strictly serial, it is not possible to get more than a 20 times speedup, no matter how many threads are used.

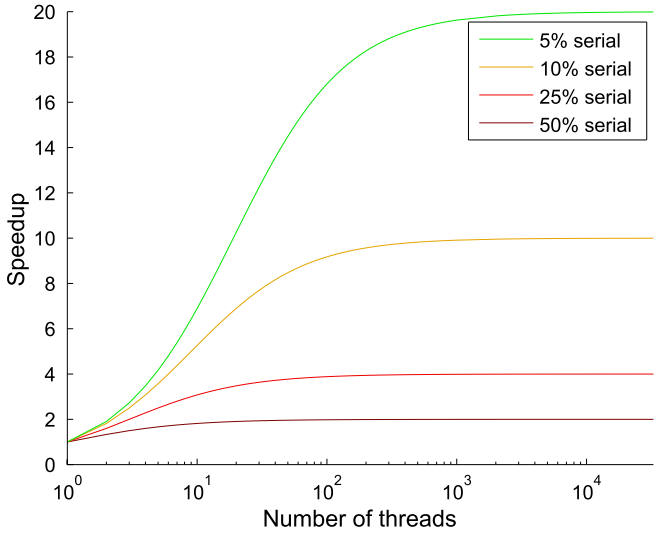


Figure 2.14: Amdahl's law.

Chapter 3

Application considerations

In this chapter we present both the internal as well as the external requirements of the application. For the external requirements we look at the input and output of the program and which libraries it uses. For the internal requirements we explore the different possibilities of programming language and parallelism. We explain what we need and how they affected the choises we made. Finally we also shortly describe the testing and development of the program.

An overview of the external requirements can be seen in figure 3.1. This figure shows that as input we take an optional state file, one or more networks and some parameters. The state file specifies the initial placement of the nodes, the networks specifies the connections between the nodes, while the parameters are used to tell the program how to sample and specify output settings. In order to perform the sampling, an external random number generator library is used. Finally during the sampling process, an output file is created along with several state files. The rate that these files are updated/created can be given as input parameters and otherwise depends on the network size.

The state file is a file containing a single number on each line, specifying which cluster the first-, second-, third-, ..., J'th-node belongs to, in the range [0-C]. The network is given as a set of connections, specified by two comma seperated node-numbers, specifying that those nodes has a connection. These connections are always seen as undirected, so if the connection is given in either direction,

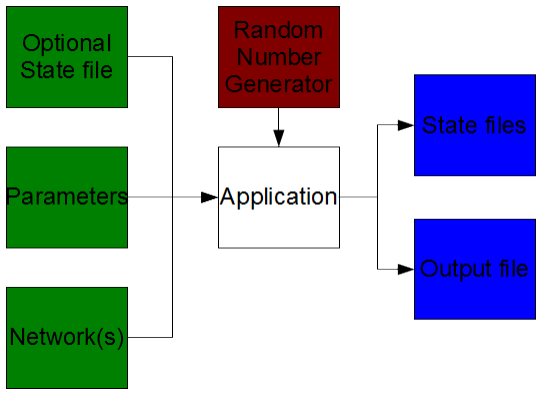


Figure 3.1: Application input/output

it is added to both nodes. The parameters specify the number of nodes in the network, sweeps that needs to be performed and the network file(s). These parameters can also be used to initially split the nodes into a given number of clusters, either linearly, randomly or as given in a state file. The last important option these parameters can specify is whether to use split-merge and how many restricted Gibbs sweep that should be performed in each split-merge. The random number generator is explained later in this chapter, but basically this random number generator is used to ensure that good random numbers are used, such that we can explore the entire state space, rather than only states with a certain probability threshold. The output file is continously updated with the iteration number, delta time since beginning, the number of clusters found and the log-likelihood of how well the current model explains the network(s).

3.1 Programming language

In order to work efficiently with large amounts of data it is important to choose the right programming language. We need a programming language which can perform computations and store/access data with a very low overhead. Additionally the language should be low level enough to be able to take advantage of hardware optimizations and parallelize efficiently. Matlab is among the most commonly used languages for statistical modelling, however the language is an interpreted language, which means that the code has to be compiled on the fly, making them run slower by wasting more CPU time on this. This is especially the case if complex functions/expressions are used. These interpreted languages usually also lack the option to specify optimizations flags for the compiler, to help it understand how the code can further be optimized, which can

have a huge impact on performance. Furthermore these interpreted languages usually has exception handling and has checks to verify that every command is valid, even though the programmer may know that they will always be valid in the given context, resulting in a much slower execution than is possible. Continously checking whether an error occured adds yet another large overhead. Furthermore high-level languages such as Matlab allows many different kinds of operations to be performed on any type of data structure, which means that in order to allow these, additional overhead is added in order to for instance get the name of the data structure, get the type, get functions of the class etc.

Due to these and other reasons, we have decided not to use Matlab, nor the common programming languages such as R, Java and C#. Instead we have chosen to go with C++, as it has low level access, very low overhead, extremely fast computations and allows complete control over the memory management, granting us the ability to store the data such that it can be accessed very fast. Good memory management is one of the keystones for creating applications, which works on gigabytes of data at the same time. The overhead for each data entry can be the difference between being able to allocate room for the data or not. Furthermore many speed optimizations depends on the exact memory mapping of the data structures, which we therefore need to be able to manipulate to a great extent. C++ has almost no checks to verify that the given command is valid, in fact it is possible to access memory in another application if the pointer is incorrect. This means that these commands can be executed extremely fast but also gives rise to potential bugs which may never be caught, as the program does not necessarily crash/cast an exception.

3.2 Random number generator

In order to perform proper MCMC, we need good random numbers, which is not a given for the default random number generators. For instance the windows implementation of the C++ rand function only returns values between 1 and 32767, which means that we cannot choose clusters with less than 0.003% probability, that is a problem! So in order to ensure that we always have good high precision random numbers instead of relying on the current implementation of the rand function for the specific operating system, we have chosen to use an external high performance random number generator.

We have chosen a random number generator based on Mersenne Twister, created by Agner Fog (http://www.agner.org/random/). The Mersenne twister were created by Makoto Matsumoto and Takuji Nishimura as a uniform pseudo-random number generator[10], which is specifically optimized for Monte Carlo

simulations[22] and hence an optimal choise for us. We settled on this library because it is sufficiently random for the purpose of performing MCMC, with a relative error less than 2^{-32} and it is extremely fast, in a simple test it generated a billion random numbers in just under 10 seconds. The numbers it generates are not random enough to be used for cryptography, but for MCMC they are fine. Looking at other papers we see that some of the state-of-art random number generators uses between 25 and 165 seconds to just generate a million random numbers, these are on the other hand way more precise random number generators[5], that can be used for encryption of data.

3.3 Parallelization

There are three common approaches for parallizing an application, across several computers using MPI, using the graphics card with OpenCL/CUDA, and using multi a CPU solution with shared memory, such as POSIX threads and OpenMP. MPI is an interface which makes it simple to write code that executes on processors with any type of memory architecture (shared and NUMA architecture), by using this interface to exchange memory and synchronize executions. OpenCL/CUDA are frameworks for writing code that can be executed in parallel on massive amounts of data, by using the GPU. If the parallel sections of the code does not have any dependencies, then these two frameworks are able to greatly increase performance. POSIX threads and OpenMP are both APIs for creating multiple threads which can work in synergy with each other in order to perform tasks faster, by dividing it between them. Using POSIX threads simply means that multiple threads are started, which may run the same code with possible different parameters. These threads can then communicate with each other to synchronize data and execution of the program, in order to delegate the workload. Doing all this does however require a lot of function calls in order to start threads, synchronize, create barriers, joining threads, closing threads etc. In order to ease the amount of code that needs to be written, OpenMP was created. It is a high-level threading API, capable of parallelize loops and sections of code using code comments, which will automatically create, join and destroy the threads. Simple code comments in OpenMP also allows these threads to synchronize data and execution.

The algorithm for Gibbs and Split-Merge sampling has an inter-loop dependency, as the threads needs to know the probability of placing the node in each cluster, in order to know which cluster the node should be placed in. Hence the probability for the next node cannot be calculated until the first node has been placed, as it will change these probabilities. Furthermore these probabilities are very inexpensive for single graphs and hence the overhead of OpenCL/CUDA

results in less performance. The same goes for MPI, as the program running on different processors has to synchronize data very often, over a slow network connection. However POSIX threads and OpenMP is optimal for this task. Since they work on the same shared memory, they are able to synchronize their data extremely fast (it simply has to be flushed from the processors caches to the RAM). We use OpenMP as it is very easy to implement while still granting high performance.

In the case of multiple graphs however, the amount of time used to calculate the probability increases to the point where it is faster to run a kernel on the GPU than calculating everything on a CPU with a very limited number of cores, which means that OpenCL/CUDA can be used to speed up the calculations. MPI is however still rather slow, as it is very slow to send a message over the network. POSIX threads and OpenMP can still be used for multiple graphs, but according to some tests we performed, the GPU is able to make these calculations faster, even with inter-loop dependencies, when only 10 graphs are given. For this reason we decided to go with a GPU solution for the multi-sample implementation of the algorithm. In the choice of OpenCL vs CUDA, we decided to go with OpenCL. Although CUDA is more polished than OpenCL, it is only possible to run CUDA code on Nvidia GPUs, while OpenCL can run on a wide variety of GPUs, including those capable of running CUDA, which means that more people will be able to use our program.

3.4 Test and development

During the development of this program we have continously verified that everything works using both unit and functional tests. We implemented a testing scheme able to perform unit tests on-the-fly, which means that while running the optimized code, we simultaneously recalculate everything using an old, stable version of the program, constantly verifying that all data structures are completely equal. This means that not only do we verify that the end result is correct, but with this we also catch minor bugs that may creep in but does not change the end result for the given unit tests.

Our intention with this thesis is to create a state-of-the-art product, that can be used to perform IRM analysis on a broad range of large scale networks. As such we have focused on ease-of-use, such that the program can be used without in-depth knowledge of the implementation. Hence we strive to make sure that we verify the structure of any input and data provided by the user and presents clear informative error messages, should anything fail while the program is running. This includes everything from invalid range of input parameters to lack of disk

space. As part of the ease-of-use, we also even allow the user to specify the links of the undirected network in any way they like, in one direction, both directions or some combination thereof.

Chapter 4

Implementation

In the previous chapters, we have laid a theoretical foundation for the Infinite Relational Model and presented how to perform Gibbs and Split-Merge sampling. In this chapter we describe how we have implemented and optimized these procedures to create high performance samplers, capable of performing fast modelling on networks with millions of nodes. We first present the naive pseudo-code for IRM, using Gibbs sampling to cluster a single network. From the pseudo-code we can identify parallel parts of the code and describe the relevant data structures. We then extend the implementation to include the Split-Merge procedure and analyze which factors influence the running time of the samplers. Finally we extend the implementation to allow multiple graphs.

4.1 Naive Gibbs sampling

The naive pseudo-code for IRM with Gibbs sampling is shown in figure 4.1. The code contains two nested loops, as we for every Gibbs sweep must iterate over all nodes i. For every node i we first calculate the number of nodes in each cluster n and number of links N^+ and non-links N^- between all clusters. In these calculations we assume that node i is not assigned to any cluster. We then calculate the vector r, containing the number of links node i has to nodes in

```
1 for each sweep
2     for each node i
3       calculate N+ and N−, n ignoring i
4       calculate r for the node
5       calculate the probabilities of assigning i to
            each cluster
6       choose a cluster based on these probabilities
```

Figure 4.1: Naive Gibbs sampling pseudo-code.

each cluster. With these data structures we can compute the probability of node i belonging to each of the C clusters, and hence assign the node accordingly.

4.2 Gibbs optimizations

In this section we describe how we have optimized the naive implementation, using the optimization strategies mentioned in the theory. One of the most important aspects of optimization is to create the entire program from scratch with parallization in mind. The required operations in each iteration is to calculate the probability of placing the current node in each cluster and assign the node. To do this we need N^+, N^-, n and r. These can easily be parrallelized as the probabilities of assigning a node to each of the clusters are independent, we can also calculate these probabilities in parallel.

In every iteration of the Gibbs sweep at most two values are actually changed in n, when the node is reassigned from one cluster to another, while only $2 \cdot C$ values are changed in N^+ and N^-. Instead of recalculating these structures every iteration, we can cache them globally and only update them within the iteration. The pseudo-code for caching these structures globally is shown in figure 4.2

All r-vectors for all the nodes could be stored in a global structure R, but as we will show later, the collective size of R becomes too large to be stored in memory for huge networks.

Using n and r we can update the elements in both N^+ and N^- in parallel. Furthermore nodes tends to remain in their cluster, when a good clustering is found and hence in order to avoid unnecessary updates we can calculate the probabilities without removing the current node as explained later in this chapter. Hence we only update the data structures if the node is assigned to

```
1 initialize N+, N−, n
2 for each sweep
3     for each node i
4       calculate r for i
5       exclude node i from N+, N− and n
6       calculate the probabilities of assigning i to
            each cluster
7       choose a cluster based on these probabilities
8       include node i to N+, N− and n
```

Figure 4.2: Pseudo-code for Gibbs sampling with static data structures.

```
1 initialize N+, N−, n
2 for each sweep
3     for each node i
4       calculate r for i
5       calculate the probabilities of assigning i to
            each cluster, ignoring i in N+, N− and n
6       choose a cluster based on these probabilities
7       if i changes cluster then update N+, N− and n
```

Figure 4.3: Optimized Gibbs sampling pseudo-code.

another cluster as shown in figure 4.3.

The choice of data structure depends on the problem, what type of data it contains, how often it is used and in what way it is accessed/modified. To perform IRM fast on large networks we need data structures where the overhead of fetching and storing data is small while the structures are compact enough to store large amount of data.

The network itself is represented by nodes and links. We only need to iterate over links when computing r for a given node i, and hence we need to know which nodes i is linked to. It is infeasible to store the entire adjacency matrix. As an r-vector is only computed for a singe node in each iteration of a Gibbs sweep, we can store links for the nodes in individual arrays. For every node we hence store what cluster it is currently assigned to and an array of integers indicating which nodes it is linked to, as illustrated in figure 4.4.

For n, N^+ and N^- the amount of data that need to be cached depends on the number of clusters, which can vary throughout the IRM analysis. Many data structures such as linked lists and vectors allow for a dynamic resizing. Linked lists are however very slow to iterate through while reallocating data in vectors

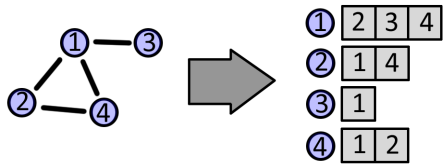


Figure 4.4: Links are stored as a one-dimensional array for each node.

might result in copying of data to new memory blocks, when the capacity of the vector is exceeded and this reallocation also requires synchronization of all threads, leading to even more overhead and complexity. In order to reduce the overhead of fetching data and the time it takes to allocate/deallocate memory as well as avoiding these thread synchronization issues, we have decided to store all data in static arrays with a constant size. By analysing the Gibbs and Split-Merge sampler we found that the samplers rarely find more than 200 clusters independent of the network size. Hence by allocating memory for a thousand clusters we can safely assume that there is room for all the clusters we need. Using static arrays we can utilize some optimizations when storing N^+ and N^-. If more clusters are necessary, this number can be changed through a parameter given to the program.

The data structures for N^+ and N^- can both be stored as one and two dimensional arrays, as shown in figure 4.5. Using two-dimensional arrays means that when a cluster is removed, we can move the content of another cluster to this position by only swapping two pointers. However a two dimensional array means that the first dimension is described by an array of pointers to other arrays, which means that following pointers from the first dimension will always result in cache misses. To avoid this, the two dimensional array can be turned into a one-dimensional array as we know the length of the second dimension. For a maximum cluster size of 1000, the size of the array would be roughly 8 megabytes, if values are stored as 64-bit integers, which is very little compared to the available memory on current computers. This shows that it is possible to allow for more clusters if necessary. Using a one dimensional array does however also mean that moving a cluster becomes more expensive, as all the data from that cluster has to be moved to the position of the cluster that is being removed, rather than changing two pointer references. This data is however placed in sequence and hence it is extremely fast to copy the data. Furthermore we later want to perform IRM analysis in OpenCL on a GPU, which uses one-dimensional arrays.

We must be able to handle both insertions and deletion of clusters in these static

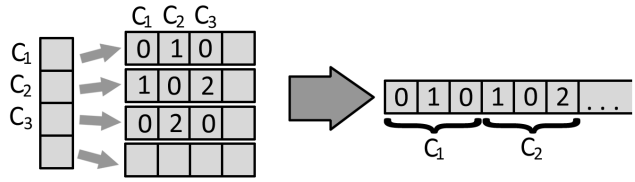


Figure 4.5: Data structures for links and non-links between clusters stored in a two and one-dimensional array.

arrays, which happens whenever a node is assigned to a new empty cluster or when the last node is removed from a cluster. When a cluster is removed it is no longer used in the analysis, but there is still allocated room for it in the data structures. As shown in figure 4.6 there are two simple ways to handle this situation; it can either be ignored in the structure or the last cluster can be moved up to fill the position, as cluster id does not matter in the IRM model.

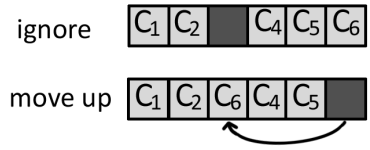


Figure 4.6: When a cluster is removed, we can either ignore its place or move the last cluster to its position.

If the cluster is ignored, an ignore list has to be created and updated to avoid that these ghost-clusters are included in the analysis. This means that the data structures now can contain 'holes', making it more likely to generate more cache misses as unused data now also uses up the cache.

Instead of having holes in the data structure we choose to reposition the last cluster. This does however also mean that nodes and values belonging to this cluster are affected and must be updated accordingly.

Figure 4.7 shows an example of the procedure to remove a cluster from the one-dimensional data structures N^+ and N^-. The example contains four clusters c_1, c_2, c_3 and c_4. When c_2 becomes empty, all values associated with c_2 are overwritten with the values belonging to the last cluster c_4. In step (3) the individual values representing c_2 for all clusters is replaced by values for c_4. In step (4) the sequential part representing c_4 is copied to the position where c_2 was located. One of the strengths of C++ is that the language is very efficient

at reading and writing sequential data, and hence this last operation can be performed extremely fast.

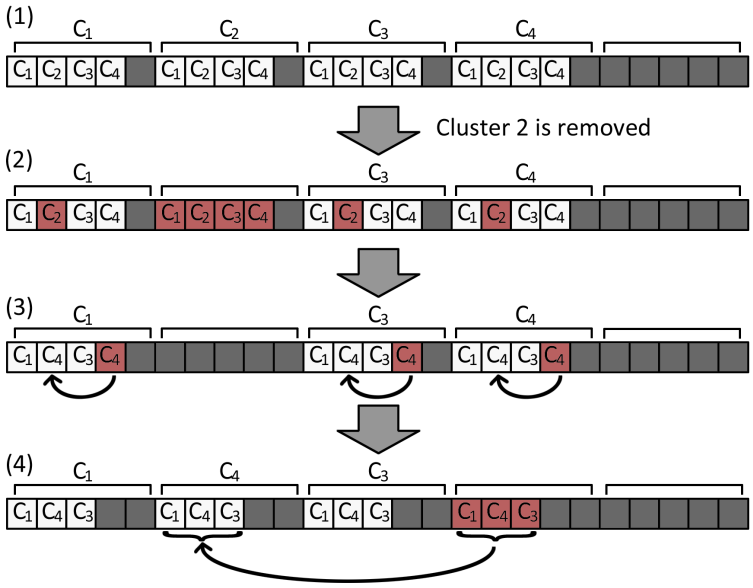


Figure 4.7: Procedure to remove a cluster from N^+ and N^- data structures. In the figure cluster c_2 is removed and c_4 is repositioned.

Every node have an associated value, stating the id of the cluster they are assigned to. If this value only states what position the cluster has in the data structures, then all these values must be updated when a cluster is repositioned. This can be avoided by using some assisting structures as illustrated in figure 4.8.

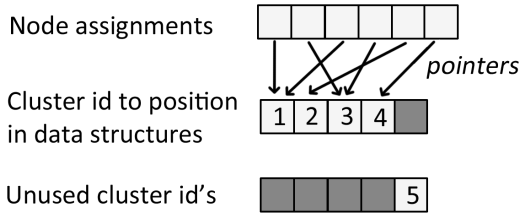


Figure 4.8: Eksample for assisting data structures to ease relocation of cluster data.

Instead of storing the cluster position for each node, each node points to a position in an array. The position of each field in this array represents a cluster

id, while the value represents the cluster position in the other data structures. When a cluster is moved to another position, only a single field in this array has to be changed and the position will automatically be updated for all nodes assigned to the cluster. When a cluster is removed, its cluster id is no longer in use. We therefore need an additional structure to keep track of unused cluster ids. When a new cluster is created it is associated with the first unused id.

The procedure of removing a cluster from the situation in figure 4.8 is shown in figure 4.9, while reinserting the cluster is shown in figure 4.10. First the cluster with id 2 removed from the cluster id structure and added to the unused id's, to indicate that this cluster id is available. Additionally the last cluster is now relocated to the second position in the data structures, exactly as shown in figure 4.8. When the cluster is reinserted, it gets the id 2, but is now located at position 4 in the data structures, as this is the first empty space.

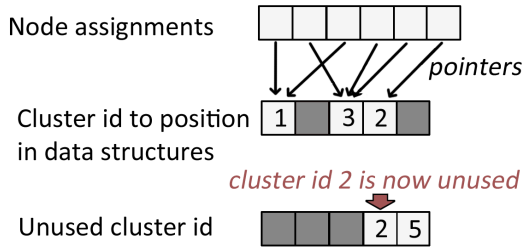


Figure 4.9: The cluster with id 2 is removed.

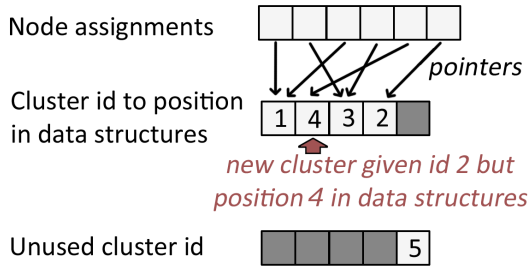


Figure 4.10: A cluster is created.

4.2.1 Avoid unnecessary updates

Since the nature of IRM is to cluster similar nodes together, it is more likely that a node will remain in a cluster with other similar nodes than being repositioned into another cluster. Hence nodes are often reassigned to the same cluster. In these cases it is a waste of resources to first remove and then add these nodes to N^+, N^- and n at the beginning and end of each iteration. To avoid these unnecessary updates, we do not remove the node from the data structures, but change the calculations of the probabilities to ignore the node.

In formula 2.60 the logarithm of the change in likelihood when assigning a node i to a cluster o was given as:

$$ln(G_{change_{i,o}}) = \sum_m \Big(Betaln(N_{om}^{+\backslash i} + r_{im} + \beta^+, N_{om}^{-\backslash i} + n_m - r_{im} + \beta^-) - Betaln(N_{om}^+ + \beta^+, N_{om}^- + \beta^-) \Big)$$

When we do not remove the node from its current cluster, this probability changes for all terms involving this cluster c:

$$betaln(N_{oc}^+ + \beta^+, N_{oc}^- + \beta^-) - betaln(N_{oc}^+ - r_{ic} + \beta^+, N_{oc}^- - (n_c - r_{ic}) + \beta^-)$$

At the same time n_c is subtracted one whenever it is used. This calculation uses the exact same variables but in a different order, hence this introduces no computational overhead.

4.2.2 Computing the beta function

The key operation for calculating the probabilities of assigning a node to each cluster is the logarithm of the beta function $betaln(a, b)$. This function is commonly expressed using the logarithm to the gamma function $ln(\Gamma(x))$ as:

$$betaln(a, b) = ln(\Gamma(a)) + ln(\Gamma(b)) - ln(\Gamma(a + b))$$

Calculating $ln(\Gamma(x))$ can be a somewhat expensive operation. As we only allows β^+ and β^- to take on integer values and links and non-links are whole numbers,

we know $(a, b) \in \mathbb{N}$ whenever we compute $betaln(a, b)$. We use a look-up table in which we pre-calculate and cache the first 245 million values of the $ln(\Gamma(x))$-function, and use this to speed up the computations of $betaln$-function. These values becomes so large that 64-bit double values are needed to represent them, resulting in an array size of $\sim 1.8Gb$. For $x > 245$ million we estimate $ln(\Gamma(x))$ by using Sterlings approximation, which states that $x \cdot ln(x) - x$ goes towards $ln(x!)$ as x increases. We cannot only rely on Sterling's approximation as it is not a very precise approximation for lower x. At the same time we find that it takes almost twice the time to use Sterling's approximation than using the look-up table, but for input values larger than 245 million, the error is negligible and hence we use this for values exceeding the lookup table.

4.3 Split-Merge

The pseudo-code for our implementation of Split-Merge is shown in figure 4.12, following the procedure as described in section 2.6. c_i and c_j denotes all the data structures associated with two new clusters.

First the data structures for cluster z_i is copied into c_i. If node i and j belongs to different clusters, then the cluster node j belongs to is similarly copied into cluster c_j else j is assigned to a new cluster and removed from c_i. This use of original data is shown in figure 4.11, which shows how the data is copied from N^+ into data structures n_i^+ and n_j^+ associated with c_i and c_j. Next we randomly split all nodes clustered with i and j, excluding i and j, into the two new clusters. We then perform T restricted Gibbs sweeps on these nodes. Depending on whether we split or merge, one final restricted Gibbs sweep is performed where the nodes are either assigned normally or forced into specific clusters. Finally the acceptance probability is calculated and if the model is accepted, the original cluster data is updated accordingly. This means we do not update N^+ and N^- for all the other clusters until the model is accepted and thus avoid cache misses performing these updates.

Copying the data may seem a bit expensive, but as it is all sequentially stored in the memory, it can be done extremely fast. Otherwise we would either have to store the changed values somewhere else and still access the old values from the original arrays or simply use the original arrays. Storing the changed values somewhere else can result in many cache misses, reducing the performance of the program, while using the original arrays will result in a lot of overhead, if the proposed model is not accepted.

We have decided to let the split-merge procedure be computed within a single

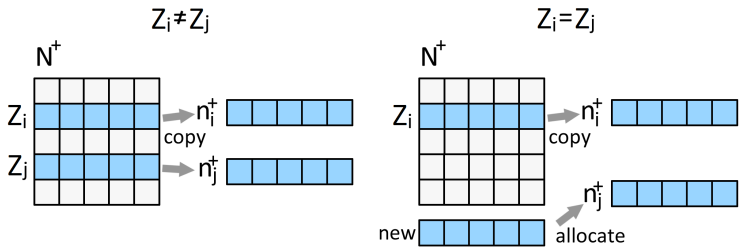


Figure 4.11: Copying data from original N^+ into new temporary arrays

thread, as it only iterates over two clusters and the nodes assigned to these. This means that the overhead of synchronizing multiple threads is very large compared to the computation time.

4.4 Runtime analysis

By analyzing the code it is possible to evaluate the performance of both the naive and less naive implementation without actually running the code. This will give a good indication of how the implementation scales with the number of nodes, clusters and connectivity level. We analyse the running time in O-notation, which specifies the order of which the algorithm scales. We first briefly evaluate the performance of the naive Gibbs algorithm, followed by a thourough analysis of the optimized Gibbs algorithm, in order to determine how these optimizations have affected the speed. We then estimate the runtime of the optimized Split-Merge algorithm.

4.4.1 Runtime of Gibbs sampling

The naive Gibbs algorithm shown in figure 4.1 is very expensive, as the links and non-links are re-calculated every iteration. Computing these requires iterating through all links, which is $O(links)$. The computation of the node assignment probabilities takes $O(C * C)$, where C is the number of clusters. Normalizing these probabilities, selecting a cluster and assigning the node to the cluster takes $O(C)$. Hence by adding the two loops, the computation time of this naive algorithm is:

$$O(S \cdot J \cdot \mathsf{max}(links, C^2)),$$

where S is the number of Gibbs sweeps and J is the number of nodes.

```
1 Pick two nodes i and j at random
2 create set S containing all nodes clustered with
     node i and j
3 if nodes are in different clusters
4    copy their respective clusters data structures
        into temporary structures denoted c_i and c_j
5 else
6    copy cluster data structures into temporary
        structures, denoted c_i.
7    allocate data structures for new array c_j
8    remove node j from c_i and insert it into c_j
9 for each node in S
10   pick a random cluster to place it in
11   if it changes cluster then update c_i and c_j
        accordingly
12 perform T restricted Gibbs sweep on c_i and c_j over
     the nodes in S
13 if i and j are in the same cluster
14   perform a restricted Gibbs sweep on c_i and c_j over
        the nodes in S, adding up the log
        probabilities of each node assignment
15   calculate the change in the priors
16   calculate the acceptance probabilities
17   if the model is accepted
18      overwrite the original data structures for z_i
           with c_i
19      add c_j to the original structures
20 else
21   perform a restricted Gibbs sweep on c_i and c_j over
        the nodes in S, where the nodes are forced
        into their original clusters, adding up the
        log probabilities of these forced assignments
22   calculate the change in the priors
23   calculate the acceptance probabilities
24   if the model is accepted
25      move all nodes in c_j into c_i, including node j
26      override the original data structures for z_i
           with c_i
27      remove cluster z_j in the original data
           structures
```

Figure 4.12: Split-Merge pseudo-code.

In the optimized Gibbs algorithm from figure 4.3 the data structures for links and non-links are first initialized, which means we have to loop over all links $O(links)$. This has not been parallelized, as it is only performed once and does not take very long.

Inside each Gibbs sweep we first calculate the links from the current node to all other nodes, which takes $O(l/N)$, where N is the number of threads and l is the number of links for the current node, such that $J \cdot l = 2 \cdot links$. In order to calculate the probability of placing the current node in each of the clusters we then parallelize over all clusters and for each of these, we calculate the change that would occur in the total probability if the node is placed here. This means that for each of these clusters we have to loop over all other clusters, which gives a runtime of $O(C \cdot C/N)$. We then normalize and select the cluster which this node will be assigned to. Since this is strictly serial code, this takes $O(C)$. Finally in the worst case where the node is assigned to a new cluster, it first has to be removed from the old cluster and then added to the new cluster. Adding and removing the node both costs $O(C)$ data changes to the links and non-links structures, while changes to the data structure n only requires constant time updates. In the case where a cluster becomes empty, the last cluster is repositioned, which also takes $O(C)$ and hence performing all these updates only takes $O(C)$. Adding these O-notations we get a speed of $O(\mathsf{max}(l/N, C \cdot C/N))$ for the inner loop and accounting for the iterations and Gibbs sweeps we get:

$$O(S \cdot J \cdot \mathsf{max}(l/N, C \cdot C/N)) = O(S \cdot \mathsf{max}(links/N, J \cdot C^2/N)) \tag{4.1}$$

This means that in O-notation this implementation still decreases exponentially with the number of clusters, however if there are few clusters the algorithm is only slowed by the number of links resulting in a speedup factor of J. Furthermore compared to the naive implementation multiple threads can be used to speed up the calculations, also all the optimizations we performed has given this implementation a much better theoretical performance, making it capable of handling much larger networks, within the same time.

4.4.2 Runtime of Split-Merge sampling

We have performed the same optimizations to Split-Merge as for Gibbs sampling, except we do not utilize multiple threads. The only time consuming part of the Split-Merge procedure are the restricted Gibbs sweeps and performing one additional sweep where the nodes are either placed randomly according to the probabilities or according to their initial clustering. Hence the speed is the same as for performing a Gibbs sweep, except that only a limited number of nodes are moved and we only have to calculate the probability of placing each node in

two clusters. We denote the number of nodes involved in the restricted Gibbs sweep M and the total number connections these nodes has $links_M$. Hence in O-notation the speed becomes:

$$O(T \cdot \mathsf{max}(links_M, M \cdot C)),$$

where T is the number of restricted Gibbs sweeps. This means that Split-Merge only scales linearly with the number of clusters and not exponentially as Gibbs does. Additionally it only scales on the number of nodes in two clusters, not the total number of nodes J. Though it also scales linearly with the number of restricted Gibbs sweeps.

4.5 Gibbs for Multiple graphs

We also want our application to be able to sample over multiple graphs. In order to do this, we could simply have increased the size of the data structures and added loops around the procedures for updating data structures and calculate assignment probabilities, P_i. This will however result in a linear decrease in performance for each graph, as nearly all operations has to be performed individually for each graph. Instead we utilize the GPU to parallelize the code over both graphs and clusters in order to build an efficient implementation for large numbers of graphs.

On the GPU there are no pre-fetching while if-statements and similar takes a long time, as all work-items in a work-group has to enter both the if and else block if they are both entered by some work-item. This means that if we use the same optimizations as for a single network, it will result in a major slowdown. Instead we have only applied optimizations that does not involve pre-fetching. This means pre-calculating the gamma lookup table, N^+, N^- and n. These data structures can be updated in parallel, along with the calculation of P_i. Only the normalization and cluster selection remains strictly serial.

Furthermore, on the GPU there are restrictions on the amount of memory that can be allocated for each array. On the graphics card we used, each array can contain 512 mb data. This means that we can accommodate networks with $512 \cdot 1024 \cdot 1024 \approx 130M$ links and assuming a link percentage of 2.5%, we find that it is possible to accommodate $\sqrt{130M/0.025} \approx 72000$ nodes for a single graph. However on the graphics card we are interested in analysing multiple graphs and assuming there are a thousand graphs we find that we can in fact only accommodate $\sqrt{130M/0.025/1000} \approx 2200$ nodes in each graph. This is not a lot, but as we will later see, the Gibbs with Split-Merge sampler is unable to mix in these large networks. This in turn also means that we are also able to

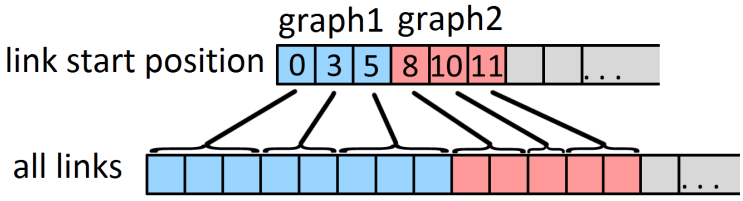


Figure 4.13: Data structures for storing node links on the GPU.

also pre-calculate the R-matrix and keep it updated rather than recalculating it every iteration, as we did for the single graph implementation.

The GPU uses one dimensional arrays rather than two-dimensional, which means that we have to flatten all our data structures. This is very simple for n, N^+ and N^- as these are already stored in one dimension. The R-matrix can also very easily be converted to a one dimensional array since we know the maximum number of clusters. The node link structure is on the other hand rather difficult to convert into a single dimension. We both need to store the links for all nodes in a single structure and also be able to know where the links for each node starts and how many links it has. In order to solve this problem we use two data structures, one containing all links for all nodes, structured such that it first contains the links for all nodes in the first graph, then all links for all nodes in the next graph etc., with no spaces between these as shown in figure 4.13. The second structure is then an array which specifies at what position the links of each node starts. This second array has the size $J \cdot samples$, which means that we can in real-time look up where each nodes links starts. Furthermore by looking at the element right after, we know where the links ends and hence how many links there are. Using these data structures we can hence also flatten the node connections.

Blocks of code that is executed on the GPU is called kernels. A number of work-items can execute the same kernel in parallel, but are given different id-values, shown in figure 4.14. In order to execute the Gibbs sampler with synchronization, these synchronization points define natural kernel boundaries, where a kernel stops and another starts, as work-items executing the same kernel cannot synchronize between work-groups. The overall structure of the code is shown in figure 4.15, where "kern:" denotes that the given operation is a kernel call.

Before calling any kernel, the CPU by pre-calculates n, N^+, N^-, gamma lookup table and the R-matrix, which are then transferred to the GPU memory. As these kernels execute in parallel, race-conditions may occur when n and the value for the number of clusters $numClusters$ is read and written to simulta-

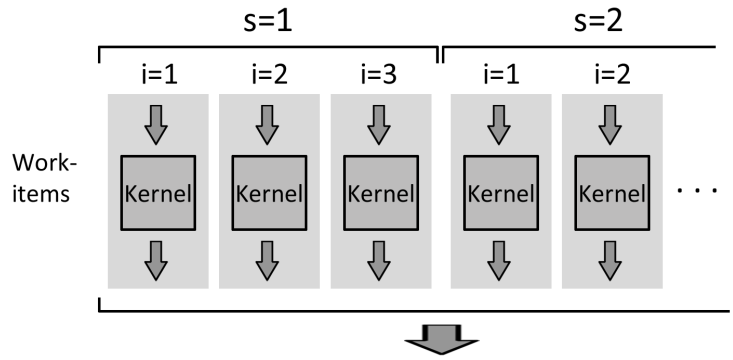


Figure 4.14: Execution of a kernel on multiple work-items.

```
1 calculate N+ and N−, n and r
2 transfer all datastructures to GPU
3 for each sweep
4    for each node i
5       set current node parameter for kernels
6       kern:removeNode
7       kern:lnQkernel
8       kern:lnQSumKernel
9       kern:maxKernel
10      kern:calcQnodeKernel
11      kern:findClusterKernel
12      kern:assignNode
```

Figure 4.15: Overall structure for multi-networks with kernel calls

neously. In order to avoid this problem, we have first doubled the size of the n array and turned $numClusters$ into an array of two elements, denoted $n1$, $n2$, $numClusters1$ and $numClusters2$. In the kernels we then change the code such that reads are performed from one part of the array and written to the other. This means that even if one thread changes the number of clusters, other threads are still able to retrieve how many clusters that existed when the kernel was initially called. We describe the precise race-conditions as we present the kernels they occur in.

The first kernel call is `removeNode`. It removes the assignment of a given node (by parameter) from all structures. First off we need to make sure that the node is only subtracted from n once and furthermore we need to know how many nodes were in that cluster before it was removed. Since everything is performed in parallel, that means that if we only use one array for n, then we do not know whether the node has been removed from n yet or not. This is the reason why we use two arrays. With two arrays we can then read the old value of n from the first array $n2$, subtract one and place it in the other array $n1$. No matter how this execution is performed, all threads will know what the old value of n is and are hence able to perform the operation. This does however also mean that all the other values of n needs to be copied over to the second array, or it will be a mess figuring out which array has the updated value of which cluster. Since the kernel has already been parallelized over the number of clusters, this simply translates to a single command. The exact same problem exists for $numClusters$, which is handled the same way.

The `removeNode` kernel is shown in figure 4.16. It is run with a workgroup size of $samples \times J$. Here we see that multiple threads helps each other with updating N^+ and N^- along with copying from $n2$ to $n1$, by only working on their cluster, c. The simple book-keeping tasks are however performed by all threads, however these do not scale in any way, in fact there are only two for-loops in this kernel, which is to update R.

This pseudo-code also illustrates why we need two arrays for n and $numClusters$. In the case that work-item 1 has finished and work-item 5 begins, where the node is being moved to cluster 5. In this case work-item 1 may have updated $n[5]$ in line 19, while work-item 5 now reads that value on line 4 and then increases the value one more at line 19. The same race-condition problem exists for $numClusters$ on line 4 and 8 if it were stored in a single value. There also appears to be a problem with r as it is updated on line 13 and used on line 10 to update the N^+ and N^- data structures. However it is only the connected nodes that are updated and hence there are no race-conditions.

The kernel `lnQkernel` calculates the probability of placing the current node in each of the clusters. The kernel is parallelized by the number of samples and

```
__kernel void removeNode(...)
s = id(0)
c = id(1)
nodeClust = cluster of the current node
clusters = numClusters2
if c < clusters
   update position  (c,nodeClust) and (nodeClust,c)
      in N+ and N- for sample s using R and n2
   if c == 0
      update R for nodes connected to the current
         node in realation to nodeCluster for
         sample s
if c != nodeClust
   n1[c] = n2[c]
curN = n2[nodeClust]-1

if curN != 0
   n1[nodeClust] = curN
   numClusters1 = clusters
else
   clusters--
   numClusters1 = clusters
   n1[nodeCluster] = n2[clusters]
   n1[clusters] = alpha
   change all nodeAssignments for nodes in the last
      cluster to nodeCluster
   update position (c,clusters) and (clusters,c) in
      N+ and N- for sample s using R and n2
   if c == 0
      update R for nodes connected to the current
         node in realation to nodeCluster for
         sample s
```

Figure 4.16: Kernel function to remove node from cluster

```
1 __kernel void lnQkernel(...)
2 s = id(0)
3 c = id(1)
4 clusters = numClusters1
5 if c >=clusters then return
6 calculate the probability of placing the current
      node in cluster c in relation to sample s
7 store this probability in global memory
```

Figure 4.17: Kernel function to calculate log-likelihood of placing the current node in each cluster

maximum number of clusters, such that each thread can sum up the probability change for all other clusters and place the result in a temporary array, shown in the pseudo-code in figure 4.17. Afterwards another kernel `lnQSumKernel` is then called to sum up the probability for all samples in order to arrive at a vector of log-probabilities of assigning the node to each cluster. Assuming there are enough available threads on the graphics card, this reduces the computation time from scaling exponentially to linearly with the number of clusters.

The kernels `maxKernel` and `calcQnodeKernel` is used to normalize the probabilities, utilizing a single thread. Finding the maximum value can be done in parallel, but we decided not to do so, as this kernel uses less than 0.1% of the total time. The cluster assignment is then picked by `findClusterKernel`, which does this in real-time by parallelizing over the number of clusters and setting the selected cluster to cluster whose probability is greater than or equal to the random value and where the probability of placing the node in the previous cluster is less than this value. The random value is supplied as a parameter to the function and is also generated from the Mersenne random number generator, to ensure good random values.

The `assignNode` kernel works pretty much the same way as the `removeNode` kernel, except that connections are added instead of removed and a new cluster may be created instead of removing a cluster. The pseudo-code for this kernel is shown in figure 4.18. It is run with a workgroup size of $samples \times maxClusters$, where $maxClusters$ are set to 128. In the pseudo-code $globalMoveToCluster$ is used, which is a reference to the global memory where `findClusterKernel` earlier in the Gibbs sweep stored the id of the cluster the current node is assigned to. The kernel is mainly split into two parts, depending on whether the node is placed into a new cluster or not. This means that all kernels enter the same region and hence only one block of the if-sentence is computed.

Some of the kernels could have been further enhanced by using shared memory,

```
1 __kernel void assignNode(...)
2 s = id(0)
3 c = id(1)
4 clusters = numClusters1
5 moveToCluster = *globalMoveToCluster
6 if c >=clusters then return
7 n2[c] = n1[c]
8 if moveToCluster == clusters
9    clusters++
10   n2[moveToCluster] = 1
11   update position  (c,moveToCluster) and (
        moveToCluster,c) in N+ and N- for sample s
        using R and n1
12   initialize position (moveToCluster,moveToCluster)
        in N+ and N- to β+ and β-
13   if c == 0
14      update R for nodes connected to the current
           node in realation to moveToCluster for
           sample s
15 else
16   update position  (c,moveToCluster) and (
        moveToCluster,c) in N+ and N- for sample s
        using R and n1
17   if c == 0
18      update R for nodes connected to the current
           node in realation to moveToCluster for
           sample s
19
20   n2[moveToCluster] = n1[moveToCluster] + 1
21 numClusters2 = clusters
22 update node cluster of current node to moveToCluster
```

Figure 4.18: Kernel function to assign node to cluster

but our main focus with this thesis has been the CPU version and furthermore we found that simply using empty kernels would only result in a speedup of around 10 times due to the interloop dependency, which requires that new kernels are continously started and stopped. Hence we have not implemented this.

4.6 Split-Merge for multiple graphs

In order to perform Split-Merge on multiple networks, we fetch the data structures for n, N^+, N^-, number of clusters and node assignments from the GPU memory. We then perform Split-Merge on the CPU and only if the model is changed, are these data transferred back to the GPU along with the updated R-matrix. The Split-Merge we use for multi networks is completely identical to that of a single network, except that we now also iterate over the number of samples. The reason why we perform Gibbs sampling on the CPU and not on the GPU is that we only look at two clusters. The CPU code can then be enhanced so much that it outperforms the GPU due to the interloop dependency in the Gibbs sampling procedure, even when accounting for time it takes to transfer the data for large number of graphs. Transferring data to and from the Graphics card has some overhead due to the response time, but the bandwidth to the graphics card is very large, which means that even for large networks with many samples, this transfer time does not increase significantly.

Chapter 5

Data

In order to make an in-depth analysis of the Gibbs and Split-Merge samplers, we use networks of various sizes and multiple samples. We need an average size network in order to investigate how the samplers perform after millions of sweeps. We also want to see how the performance of the sampler scales as we progress from small to large networks. Furthermore we also study the performance of the sampler in the case of multiple graphs. Finally we look at what structure these samplers are able to capture in multiple graphs, by looking at resting state fMRI scans for multiple subjects.

5.1 Single network datasets

There exists a digital library containing many large datasets called the Stanford Network Analysis Platform (SNAP), which is collected and used for research at Stanford University. The largest of these datasets contains just over 65.6 million nodes with 1.8 billion links, representing the users and their relation to each other on the online social network Friendster[1]. We have chosen this network in order to evaluate how IRM performs on huge networks as it is one of the largest publicly accessible network.

[1]The network is avalable from http://snap.stanford.edu/data/comFriendster.html

It is very difficult to analyze how the sampler performs as network size increases, as it usually takes a long time for real-world networks to naturally increase/decrease in size, while maintaining the same underlying clustering in the network. In order to test the sampler on a range of varying size networks, we can hence either subsample a large network to form smaller networks or use different networks can be used. In the first case some of the underlying structure is kept, but the amount is unknown. In the second case the networks can have very different underlying structure, but represent true real world samples. Even though the latter solution represents true real world samples, we have chosen to go with the prior solution, as we do not want the performance measure to be influenced by some samples being easier to cluster than others. By maintaining some of the same underlying structure, we encourage that all networks will be similarly easy/hard for IRM to cluster. We have chosen to subsample from the large network of social relations on Friendster, such that we get networks with respectively 10, 100, 1.000, 10.000, 100.000, 1.000.000 and 10.000.000 nodes, where each network contains the exact same nodes and links as the larger ones.

In order to analyze how the sampler performs after millions of sweep, we are interested in looking at real networks and not these grown networks, as we cannot be sure they still reflect real world samples. We hence consider the five networks of structural brain connectivity across 998 brain regions, which are presented by Hagmann *et al.* in **??**. These networks were derived by tractography on diffusion spectrum imaging. The first scan contain non-zero elements which we consider averaged and symmetrized over all five subjects. We end up with a single 'averaged' network of all five subjects with 998 nodes.

5.2 fCON1000 data set

An important research area within neuroscience is to analyze and describe the interactions of different regions in the brain. The 1000 Functional Connectomes Project (fCON1000) maintains a large dataset of resting-state fMRI scans for many hundred subjects[2], collected from various sites around the world. The project is presented in [2] and the data is made freely available in order to facilitate research of brain functionality. A resting state network can be considered a separation of the brain into regions while the subject is at rest. In order to detect activity in the brain, the technique of blood-oxygen-level dependant contrast imaging can be used in association with functional magnetic resonance imaging (fMRI). The technique relies on the fact that energy and oxygen reserves in the brain tissue are very limited. The neurons must hence rely on the

[2]These networks are avalable from http://fcon_1000.projects.nitrc.org

blood flow to deliver these important nutrients. This is done through the hemodynamic response process, by which more nutrients are released from the blood flow to active firing neurons than to inactive neurons. This creates a difference in oxygenated blood levels within active and inactive regions of the brain. As oxygenated and deoxygenated blood have slightly different magnetic properties, these differences can be detected using MRI [13], which then provide an indirect measure of the activity in different regions in the brain.

We have chosen to use a dataset created by Scheibe and Wind [19] for the Beijing_Zang data set from fCON1000. This set contains fMRI scans for 198 subjects with missing data. As our implementation cannot handle missing data, we discard those subjects where some of the brain is missing in the scans, leaving us with 172 subjects.

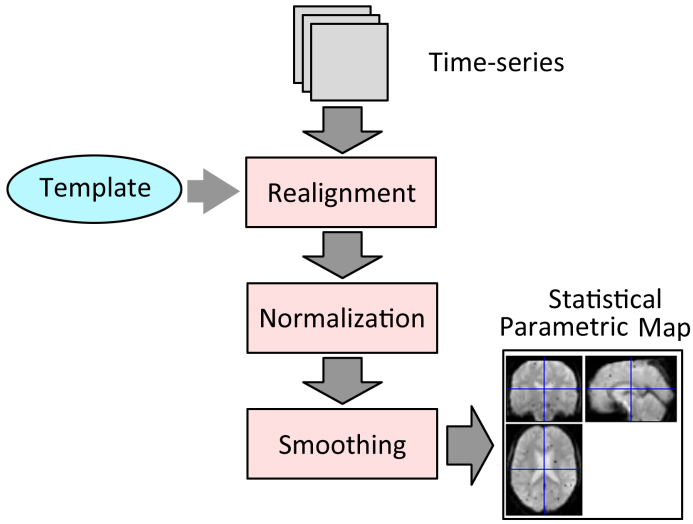


Figure 5.1: SPM transformation from time-series to statistical parametric map.

The dataset has been transformed into statistical parametric maps using the Matlab software package "Statistical Parametric Mapping 8" (SPM8), which is then converted into networks. SPM8 is specifically designed to aid in analysis of data containing sequences of brain images. In our case it is time-series of fMRI scannings for each of the single subjects. The linear process is illustrated in figure 5.1. Since all the brain scans are not evenly positioned, realignment is necessary in order to ensure that each voxel covers the same area of the brain. The brain scans must therefore be correctly rotated and translated in all three dimentions. Besides this, as the size and shape of the brain varies

between individual subjects, the scans must be normalized to ensure that we can correctly compare scans for different subjects in order to use them to perform multi-subject analysis.

Networks are generated by assigning each voxel to a region as specified by an automated anatomical labelling (AAL). The AAL used was created with the K-Means algorithm in order to merge voxels into 1000 clusters. We hence end up with 172 networks with the same number of nodes, representing the same regions of the brain. These networks can be used to investigate how IRM with Gibbs and Split-Merge sampling performs on multiple networks made from real world data. By considering the correlation-matrix the 2.5% strongest connections between regions is interpreted as links between nodes in each of the generated networks.

Chapter 6

Results and discussion

In this chapter we evaluate the performance and mixing ability of Gibbs and Split-Merge sampling in the Infinite Relational Model. We use our implementation to perform large scale modelling on unipartite complex networks, both in terms of network size and sampling iterations. To present these results the chapter is split into three sections:

- Runtime evaluation
- Sampler evaluation
- fCON1000 data analysis

We first examine the runtime of our application, to explore the performance of our implementation on large networks. This shows us to what extend it is computationally possible to handle large networks and how well our application actually does this. Using our application we then examine how the Infinite Relational Model behaves on large scale when using the Gibbs and Split-Merge sampling procedures. We investigate how the model scales with the size of the network and how it performs when run for millions of sweeps. In order to further evaluate the sampling procedures, we examine how the performance is influenced by the number of restricted Gibbs sweeps in Split-Merge and whether it helps to initially split the nodes into a given number of clusters.

We then present the results of performing IRM on the selected fCON1000-dataset. From this we examine how the model handles multiple graphs and we evaluate whether the model is actually able to capture structure in this real world data.

For all the network analysis performed, the hyper-parameters are set to $\beta^+ = 1$, $\beta^- = 1$ and $\alpha = \lfloor ln(J) \rfloor$, where J is the number of nodes in the network.

6.1 Runtime evaluation

The runtime of the Gibbs sampling algorithm as the network size increases is shown in figure 6.1. The figure also shows the number of Gibbs sweeps these runs were able to complete within 5 hours to at most 1 million Gibbs sweeps, when executed on the same computer.

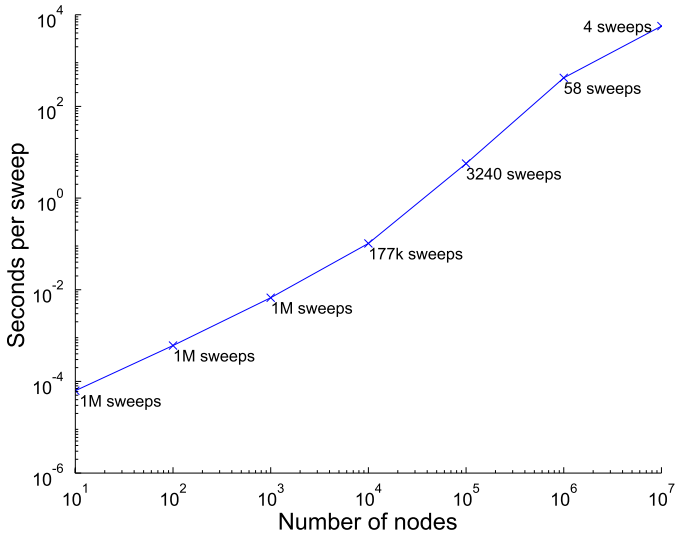


Figure 6.1: Average time to perform Gibbs sweeps on the different sized networks. For each run, the number of Gibbs sweeps performed is stated in the figure.

For smaller networks we see that the runtime is linearly bounded by the number of nodes. As the size of the network increases beyond 10.000 nodes the runtime appears to scale exponentially.

In the runtime analysis we estimated an order for the Gibbs sampler in expression 4.1:

$$O(sweeps \cdot \max(links/N, J \cdot C^2/N))$$

This expression states that the runtime scales linearly with the number of nodes. However it also scales linearly with the number of links, which usually scales exponentially with the number of nodes. This is most likely the reason why the runtime appears to scale exponentially for larger networks. This effect is negligible for smaller networks and hence not visible.

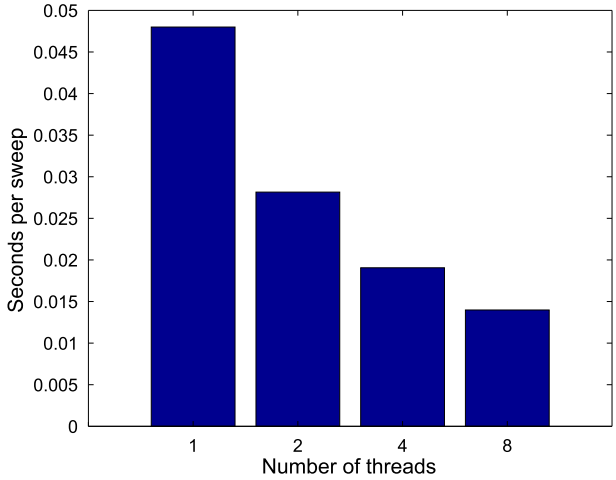


Figure 6.2: Average time to perform Gibbs sweeps when using different number of physical processors. Tests performed on the network of brain connectivity with 998 nodes.

The impact of parallelizing the code using OpenMP is shown in figure 6.2. From this we see that the non-parallelizable parts of the application becomes more and more influencial for the runtime, as more processors are used. We can use Amdahl's law to calculate the amount of non-parallelized code B. Using equation 2.67, we can calculate B as:

$$B = (\frac{T(N) \cdot N}{T(1)} - 1)/(N - 1)$$

We know from the figure that the runtime for 1 and 8 threads are $T(1) = 0.048$ and $T(8) = 0.014$, and hence we find that $B \approx 0.19$. This means that approximately 19% of the code is not parallelized. In this case the maximum

possible speedup, calculated using formula 2.68, is:

$$S(\infty) = \frac{1}{0.19 + \frac{1-0.19}{\infty}} = \frac{1}{0.19} = 5.26$$

No matter how many processors we use, the size of the serial part of the code results in a maximal speed up of 5.26 times for our implementation on this particular network. For other networks the maximal speed up will most likely be different. The more nodes there are in a network, the more time is spend calculating the likelihood of placing a node in each cluster. As these calculations are part of the highly parallelized code it results in a better speedup.

The runtime of our application for different sized networks and number of processors utilized is shown in figure 6.3. We see that for small networks more threads actually slow down the computation. This is likely due to the constant thread overhead and synchronization. For larger networks a smaller percentage of time is spend on overhead and synchronization, while more time is spend in the parallelized part of the code, resulting in a more noticeable utilization of threads.

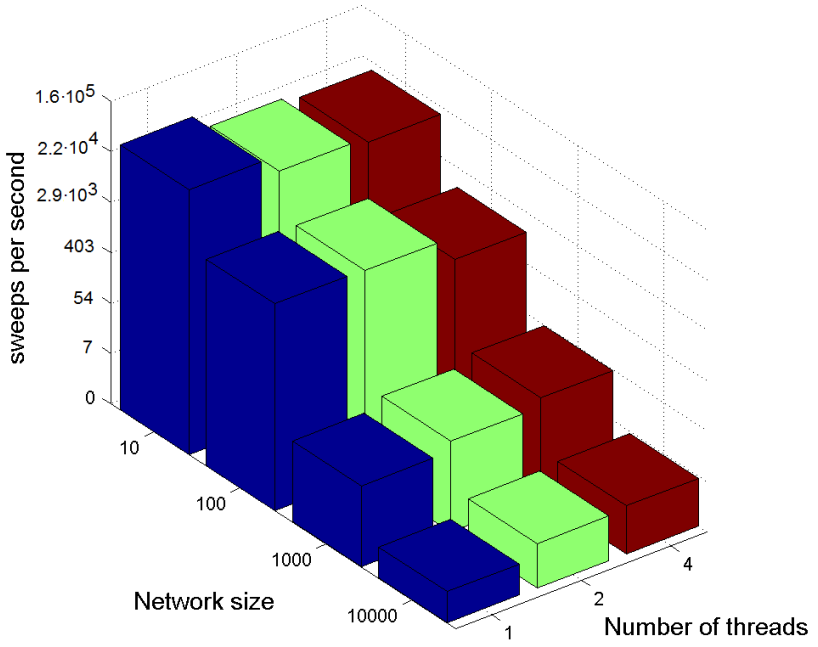


Figure 6.3: Runtime for the different sized networks.

From these evaluations we see that our implementation is capable of handling large networks, where the runtime depends on the size of the network as expected in the runtime analysis, section 4.4. The evaluation further shows that parallelization through OpenMP increases the performance of the application for network sizes of 1000 or more nodes, but only to a certain extend.

6.2 Sampler evaluation

In this section we investigate the performance of the Gibbs and Gibbs with Split-Merge sampling procedures. We compare the performance of the two sampling procedures in terms of their mixing ability. We then examine how the number of restricted Gibbs sweeps and initial clustering of the network influence the sampling.

6.2.1 Scaling sampling iteration

First we test how IRM performs when run for a long time. This test was performed on the average network of structural brain connectivity scans with 998 nodes.

The network has been clustered using both Gibbs and Gibbs with Split-Merge sampling. In the pure Gibbs sampling scenario 10 different runs were performed for 10 million Gibbs sweeps. Gibbs with Split-Merge were performed with 5 runs for 5 million iterations, where one iteration consists of a Gibbs sweep followed by a Split-Merge operation with 5 restricted Gibbs sweeps.

By comparing the log-likelihood for the different runs, we investigate the mixing ability of the samplers. This is shown for pure Gibbs sampling in figure 6.4 and for Gibbs with Split-Merge sampling in figure 6.5. The figures indicate that even after millions of sweeps the individual runs do not mix for either sampler. However the figures clearly indicate that the Gibbs with Split-Merge sampler performs better, as all runs reach a higher log-likelihood value faster and level out closer together than for the pure Gibbs sampler. We know that eventually the samplers will converge such that they sample from the real posterior distribution, but from these figures we cannot tell how long this convergence time is for either of the samplers. It does however seem like the convergence time is a lot shorter when using Split-Merge, as the chains reach a better likelihood significantly faster.

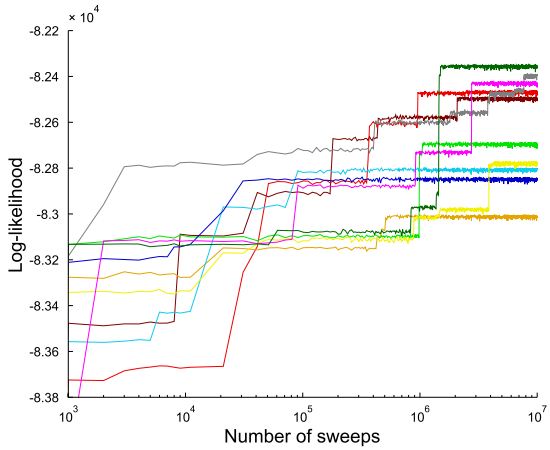


Figure 6.4: Pure Gibbs sampling for individual runs, indicated by different colors.

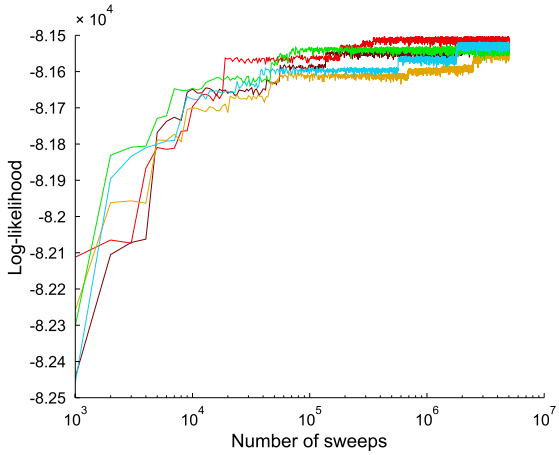


Figure 6.5: Gibbs sampling with Split-Merge. Each Gibbs sweep is followed by a single Split-Merge operation with 5 restricted Gibbs sweeps. Different colors indicate different runs.

To further investigate the performance of the samplers, we examine the normalized mutual information (NMI), both *between* and *within* runs, for the last half of the iterations. The NMI between runs shows how similar the runs are to each other while NMI within runs indicates whether the sampler mixes. These values are shown as box plots for evenly distributed states in all the runs. At each of

these states the NMI is calculated between every single pair of runs, making up the box plot for NMI between. For each of these states we also plot the NMI within. These states and their 9 next successive states are then used to compute the NMI within for each individual run. This NMI within is computed by comparing each state with the state half the number of iterations ago, illustrated in figure 6.6. These NMI values are then used to compute the box plot, such that each box contains ten data points for each run.

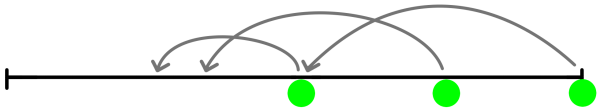


Figure 6.6: Principle of NMI-within datapoints for a single run. The point in iteration i is compared with the state at iteration $i/2$.

Intuitively, an NMI within close to 1 means that the chains do not move very much, indicating that the chains are not mixing and will hence not converge. In the case where the chains has reached convergence we would expect the NMI within and NMI between to be rather constant and about the same level, such that the chains continiously are just as similar with themself as with each other.

The NMI between and within runs are shown in figure 6.7 for the Gibbs sampler and in figure 6.8 for the Gibbs sampler with Split-Merge. The NMI between runs in the Gibbs sampler seems to remain in a rather stationary position, indicating that the runs do not approach each other. For Gibbs with Split-Merge we see that the NMI between runs does not remain stationary but curves, which does not show that the sampler improves over time. When we compare the NMI between runs for both samplers we see that it is consistently higher for the Gibbs with Split-Merge sampler than for pure Gibbs, indicating that the Split-Merge procedure does aid the Gibbs sampler.

The similarity within runs is very high and stationary for the pure Gibbs sampler, indicating that it does not sample very well. This indicates that it is unable to draw different samples that are accepted. In the case of Gibbs with Split-Merge, the NMI within runs has a large range, almost intersecting with the NMI between runs, indicating that it sometimes is able to mix. However as more iterations are performed, the NMI within becomes worse, indicating that even with the Split-Merge extension, the sampler still lacks mixing ability and will probably not converge within reasonable time.

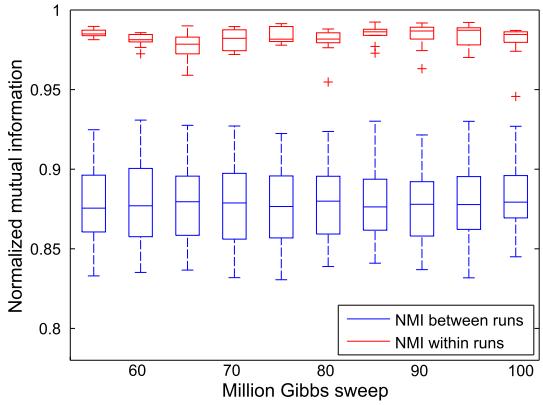


Figure 6.7: Normalized mutual information between and within runs, using Gibbs sampling.

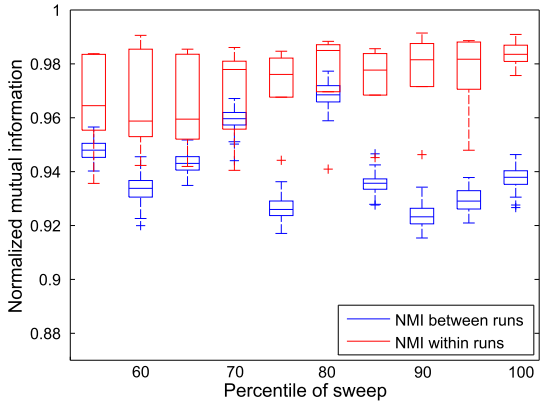


Figure 6.8: Normalized mutual information between and within runs, using Gibbs sampling with Split-Merge.

6.2.2 Scaling the network size

We have already seen that both the Gibbs and Gibbs with Split-Merge samplers perform poor mixing on a network of $\sim$ 1.000 nodes, even when allowed to perform millions of sweeps. In this section we investigate how the performance of the samplers scales with the size of the network.

To evaluate this we conduct tests on the different sized networks, sub-sampled from the large network of social interactions in the `Friendster`-network. We evaluate both Gibbs and Gibbs with Split-Merge (using 5 restricted Gibbs sweeps) on these networks. In order to estimate the tendency of the sampler, we only look at the results from which this can be evaluated.

We first examine how the mixing ability of both samplers are influenced by the number of nodes in the network. To do this we look at the log-likelihood against the number of sweeps. This is shown in figure 6.9 for Gibbs sampling and figure 6.10 for Gibs with Split-Merge sampling.

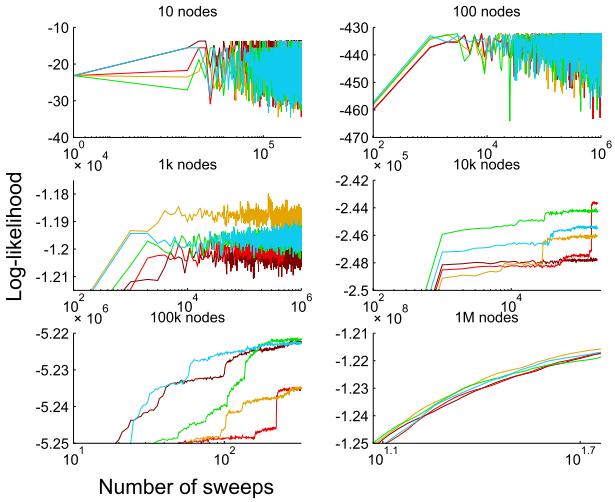


Figure 6.9: Log-likelihood for multiple runs on networks of different size using pure Gibbs sampling.

From figure 6.9 we see that the Gibbs sampler performs very well for the networks with less than 1.000 nodes. For both 10 and 100 nodes we see that the runs mix after around 10,000 Gibbs sweeps. For 1.000 nodes the chains do not mix and it does not appear they will converge anytime soon. For the larger networks the mixing ability of the sampler swindles rapidly. These figures clearly

indicates that pure Gibbs sampling is rather ineffective for networks of only 1.000 nodes.

By looking at the performance of the Gibbs with Split-Merge sampler shown in figure 6.10, we see that the Split-Merge procedure helps a lot. For the network with 1000 nodes, the figure clearly indicates that the runs now mix, although the larger networks are still no way near convergence.

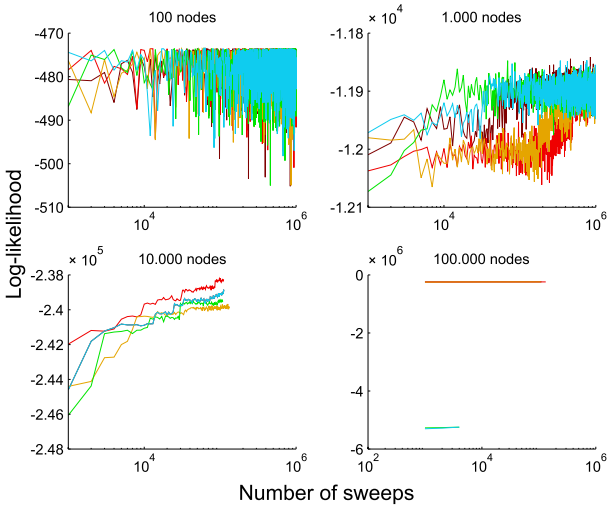


Figure 6.10: Log-likelihood for multiple runs on networks of different size using Gibbs with Split-Merge sampling.

To further investigate whether the samplers in fact mixes for these networks, we look at the NMI within and between the runs for the various sized networks. This is shown in figure 6.11 for pure Gibbs sampling and in figure 6.12 for Gibbs sampling with Split-Merge. These figures indicates the same as the log-likelihood figures. For pure Gibbs sampling the NMI between and NMI within are approximately in the same range for 10 and 100 nodes, which indicates that the runs mix. The larger networks are more similar to themselves than to each other, indicating a bad mixing ability. When using Split-Merge sampling it is also clearly indicated that the runs mix for 10, 100 and 1.000 nodes, while it does not mix for 10.000 nodes.

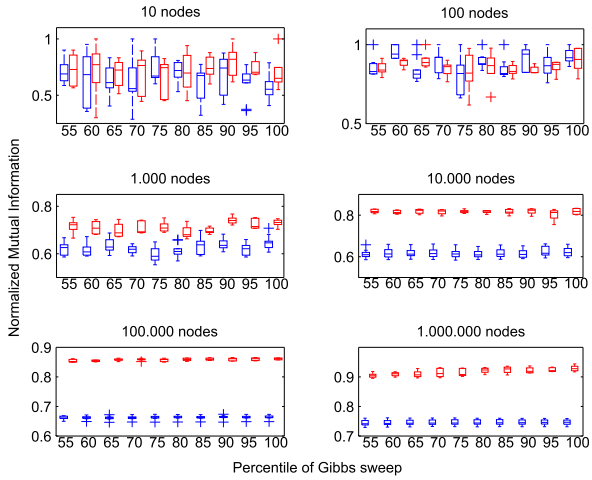


Figure 6.11: Normalized Mutual Information for different sized networks, when performing Gibbs sampling. (blue) between runs, (red) within runs.

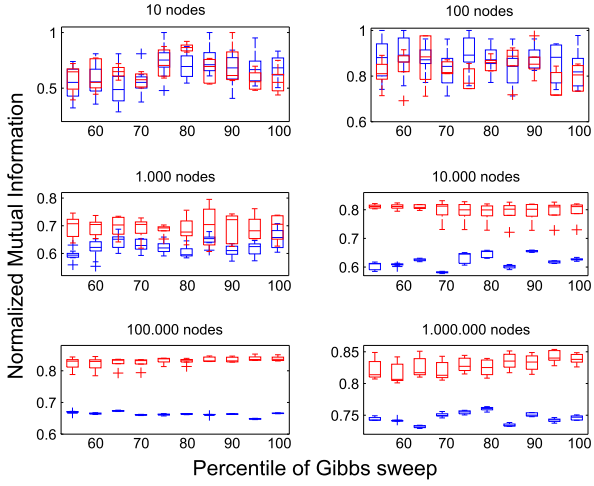


Figure 6.12: Normalized Mutual Information for different sized networks, when performing Gibbs sampling and Split-Merge. (blue) between runs, (red) within runs.

6.2.3 Parameter analysis

We have established that the Gibbs sampler with Split-Merge outperforms the pure Gibbs sampler. We now want to investigate how the performance of the Gibbs with Split-Merge sampler is influenced by the number of restricted Gibbs sweep and whether it helps to initially split the graph into multiple clusters before IRM is performed.

To investigate the influence of the restricted number of Gibbs sweeps in Split-Merge we have computed 5 runs for each of the cases with respectively 10, 50, 100, 200 and 500 restricted Gibbs. The log-likelihood of these tests are shown in figure 6.13 and 6.14. We notice that the figures indicate that no matter the amount of restricted Gibbs sweeps, the runs will still not mix.

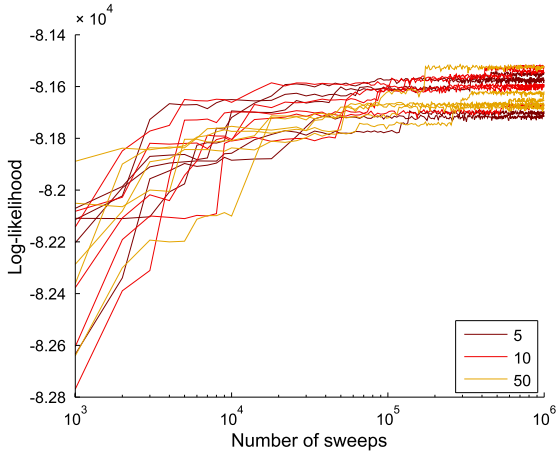


Figure 6.13: Gibbs with Split-Merge sampling for five runs for 5, 10 and 50 restricted Gibbs sweeps.

As these figures do not very well indicate what number of restricted Gibbs sweeps that performs best, but only indicates that the runs do not mix, we have also examined the NMI between and within runs. This is shown in figure 6.15 for the last half of the iterations. In the plots we see that the NMI within decreases for runs with 100 or more restricted Gibbs sweeps. This means that the chains in these cases look less like themself, indicating that the runs moves around more freely and are probably not as stuck in local modes. Though the sampler appears to perform better as it allows the runs to move more freely, this does not seem to allow chains to mix for any number of restricted sweeps. At the same time we notice that using 200 or 500 restricted sweeps do not seem to perform any better than using 100 restricted sweeps. In these tests it therefore

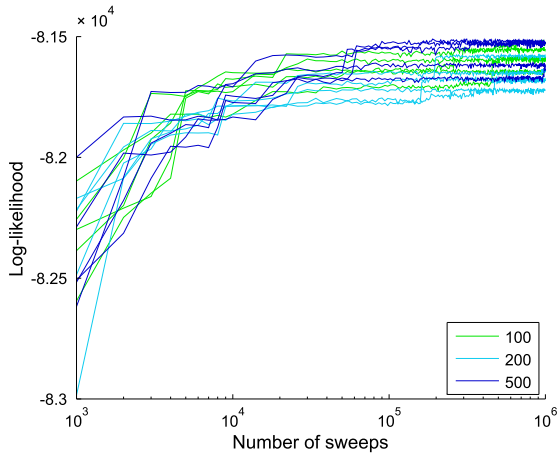


Figure 6.14: Gibbs with Split-Merge sampling for five runs for 100, 200 and 500 restricted Gibbs sweeps.

appears that the performance of the Split-Merge sampler is influenced by the number of restricted sweeps. Here it performs best when it is allowed to perform 100 restricted Gibbs sweeps, while more restricted sweeps does not seem to have no positive influence on the performance.

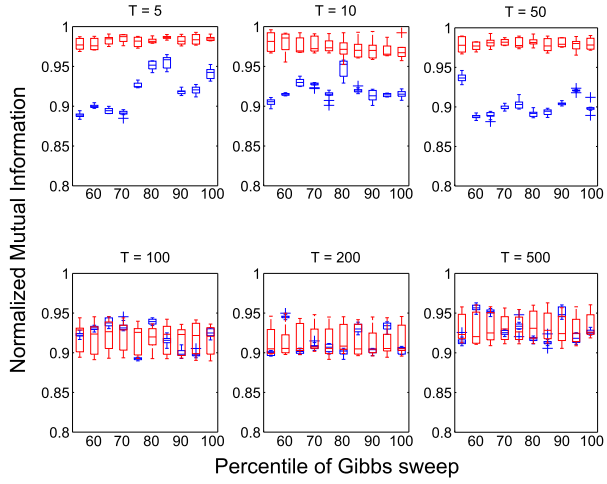


Figure 6.15: NMI within (red) and between (blue) for various numbers of restricted Gibbs sweeps (T).

We suspect that when the samplers are run on networks with many nodes it can sometimes be hard for them to create new clusters. So to help the sampler, the nodes can initially be split into a given number of clusters, to make it easier for the sampler to move the nodes around. To examine whether this helps, we compute 5 runs for each of the cases where the nodes have initially been split linearly into respectively 10, 50, 100, 200, 500 and 998 clusters. These ranges allows us to examine and compare initial configurations of a few clusters with many nodes up to the configuration with one cluster for each node. The likelihood for these tests are shown in figure 6.16 and 6.17. These figures looks very similar to those for using different numbers of restricted Gibbs sweeps. They do not indicate that the runs mix for any initial clustering.

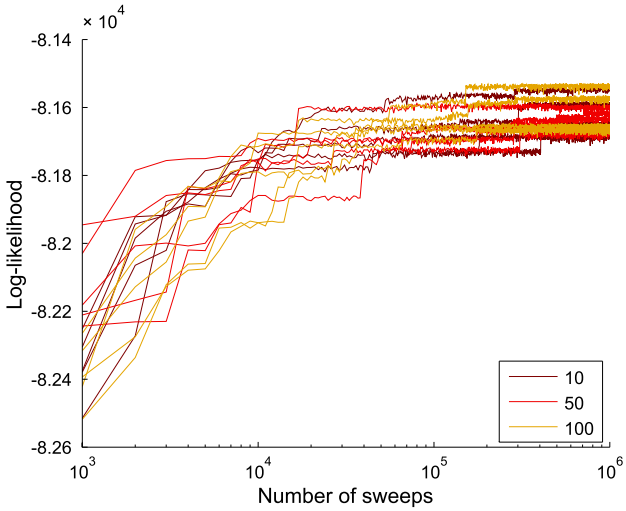


Figure 6.16: Gibbs with Split-Merge sampling for five runs where nodes are initially partitioned into 10, 50 and 100 clusters.

Looking at the mutual information, shown in figure 6.18, we again see that the runs do not seem to mix for any of the initial cluster configurations. The NMI within seems to fluctuate a little when all nodes are placed in seperate clusters, but nothing serious and the NMI between has an average value of around 0.92 in all the graphs. This means that initially clustering the nodes does not appear to have any significant influence at all.

From these findings we can conclude that the number of restricted Gibbs sweeps appears to clearly influence the performance of the sampler, while it does not matter whether the nodes are initially partitioned into multiple clusters or not.

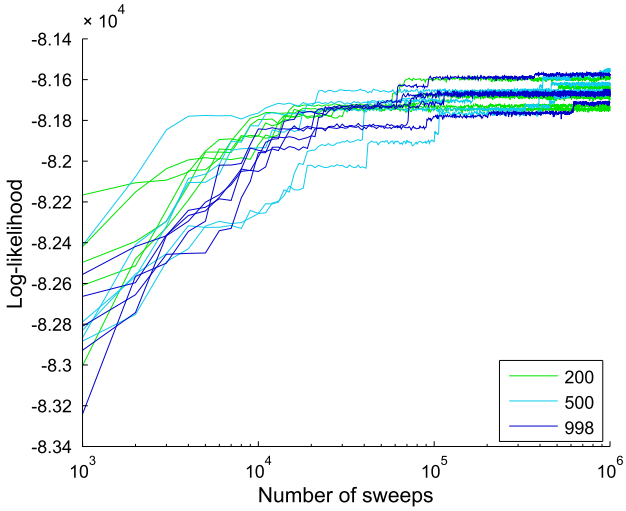


Figure 6.17: Gibbs with Split-Merge sampling for five runs where nodes are initially partitioned into 200, 500 and 998 clusters.

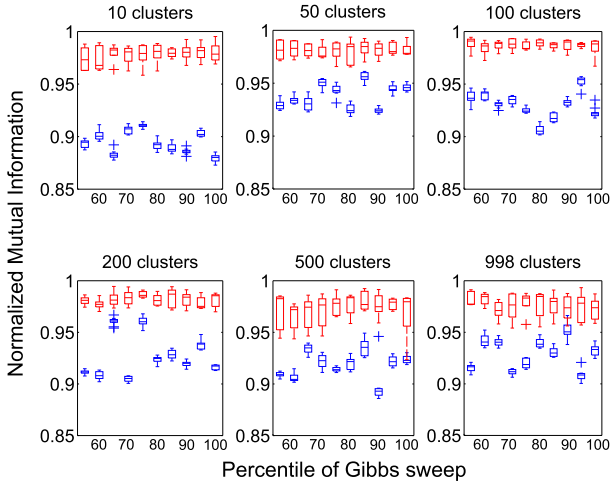


Figure 6.18: logP vs initial number of clusters, for Gibbs sampling with Split-Merge.

6.3 fCON1000 data evaluation

In this section we try to cluster the multi-graph brain activity data from the fCON1000 project, explained in chapter 5. Each graph contains 1.000 nodes representing individual regions of the brain for 172 subjects, where the top 2.5% most active signals in the brain are interpreted as links. We analyze these data using our GPU accelerated inplementation. In the previous section we found that Gibbs sampling with Split-Merge performed significantly better than pure Gibbs sampling. Hence we use the Gibbs with Split-Merge sampler on the fCON1000 data. Furthermore we set the number of restricted Gibbs sweeps to 100 as we found this to perform best on the analyzed real-world network. We initially consider all nodes in a single cluster, as initial partition into multiple clusters did not appear to have any performance benefits.

If we look at the log-likelihood and NMI between and within runs shown in figure 6.19 and 6.20 we clearly see that the chains do not mix and will not converge anytime soon. In fact the NMI within indicates that the chains are almost completely stuck in local minima.

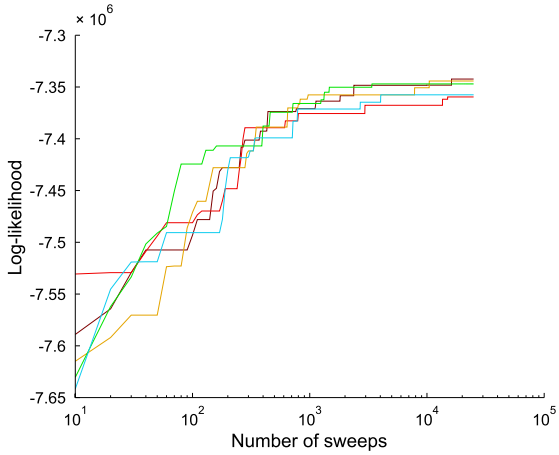


Figure 6.19: Likelihood for 5 runs on 172 subjects.

Even though the sampler is unable to mix for these networks, it is still able to model some clustering in the data, that convincingly seems to represent real underlying structure in the brain, shown in figure 6.21[1]. This figure shows the clusters that the sampler found in the data. The lines represents the known links between these clusters. The darker these lines are, the more connected

[1]The figure was created from the best clustering found by the five chains.

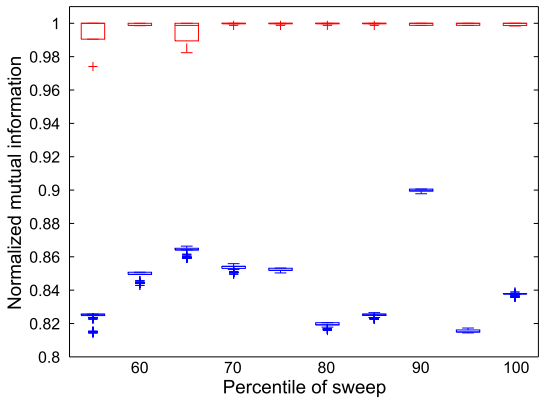


Figure 6.20: NMI between (blue) and within (red) for 5 runs on 172 subjects.

the clusters are. Similarly the links within clusters is indicated with darker backgrounds for each cluster. We can see that most of the clusters with a few exceptions are coherent and symmetric as we expect the brain to be; the clusters do not consist of single or random dots nor contains abnormal large amount of the brain. We have however not verified that the boundaries of these clusters matches any known anatomic regions in the brain.

We have also performed IRM on a single brain scan with the Gibbs with Split-Merge sampler for 100.000 iterations, shown in figure 6.22. The sampler performs much better on this single network than with multiple graphs, however from the figure we see that even though the sampler may perform better, the resulting clusters are not as expressive as it was the case for 172 subjects. In this case there are only found a few clusters that appears to be good, while most of the other clusters consists of small "random" parts of the brain, which visually does not appear to have anything to do with each other.

From this we can see that even though the samplers does not converge for multiple graphs, it may be very beneficial to use multiple graphs rather than a single graph. In multiple graphs the distinct local minima of the individual graphs are less significant, and hence the sampler is less likely to get stuck in minima that are not shared between multiple graphs, but more likely to get stuck in those that are. The sampler is still not able to converge to the posterior distribution, but it gets stuck in minima that better explain the network as they reflect structure that is present in multiple of the networks as indicated in this analysis.

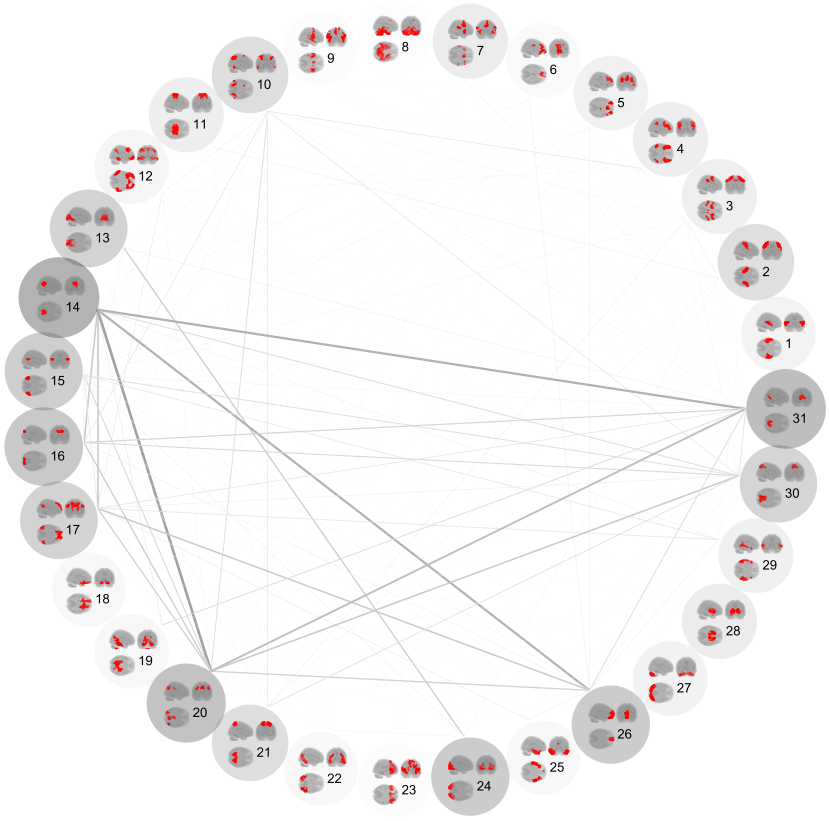


Figure 6.21: Clustering found by IRM on 172 subjects using Gibbs with Split-Merge sampling.

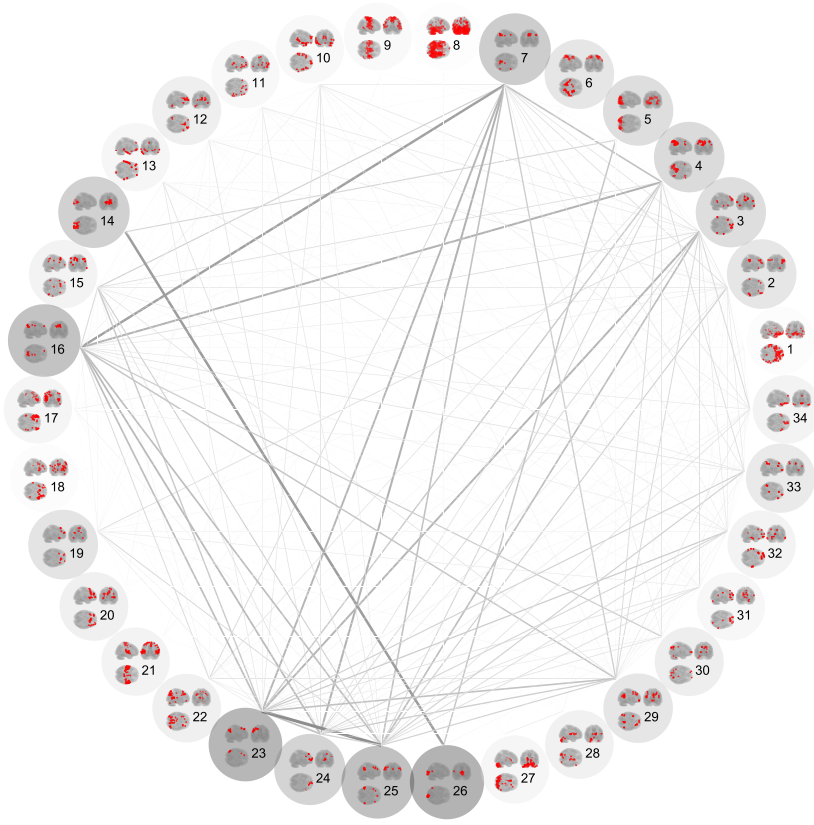


Figure 6.22: Clustering found by IRM on a single subject using Gibbs with Split-Merge sampling.

Chapter 7

Conclusion

In this thesis we have succesfully implemented a highly optimized Gibbs and Split-Merge sampler for the Infinite Relational Model (IRM) on complex unipartite networks. We showed that IRM with Gibbs and Split-Merge sampling can computationally be scaled to handle these large networks, as it is possible to compute Gibbs sweeps on millions of nodes within minutes and on smaller networks within milliseconds. This allowed us to investigate how these samplers perform as the network size is scaled up.

We both implemented a single network sampler, parallelized over multiple cores on the CPU and a multi network sampler utilizing the massive parallel computing power of the GPU.

By looking at different sized networks we found that Gibbs sampling converged for network sizes of up to 100 nodes, while it was not able to properly mix already for networks with 1000 nodes. For larger networks the performance kept decreasing rapidly. This clearly shows the limitations of the Gibbs sampler and highlights the need for better samplers.

By extending the Gibbs sampler with the more sophisticated Split-Merge procedure, the sampler were able to mix well for networks of 1000 nodes. Hence Split-Merge is a valuable aid to the Gibbs sampler, however even better samplers are essential to properly sample over the posterior distribution for larger

networks. This claim was supported by the fact, that even when performing million of sampling-iterations on a real world network with 1000 nodes, the Gibbs sampler with Split-Merge were unable to converge, even after millions of sampling-iterations. We however found that some interesting structure could still be extracted from real world data, as we analysed brain activity scans for 172 subjects.

Bibliography

[1] K.W. Andersen, M. Mørup, H. Siebner, Madsen K.H, and L.K. Hansen. Identifying modular relations in complex brain networks. *MLSP*, 2012.

[2] Bharat B. Biswal, Maarten Mennes, Xi-Nian N. Zuo, Suril Gohel, Clare Kelly, Steve M. Smith, Christian F. Beckmann, Jonathan S. Adelstein, Randy L. Buckner, Stan Colcombe, Anne-Marie M. Dogonowski, Monique Ernst, Damien Fair, Michelle Hampson, Matthew J. Hoptman, James S. Hyde, Vesa J. Kiviniemi, Rolf Kötter, Shi-Jiang J. Li, Ching-Po P. Lin, Mark J. Lowe, Clare Mackay, David J. Madden, Kristoffer H. Madsen, Daniel S. Margulies, Helen S. Mayberg, Katie McMahon, Christopher S. Monk, Stewart H. Mostofsky, Bonnie J. Nagel, James J. Pekar, Scott J. Peltier, Steven E. Petersen, Valentin Riedl, Serge A. Rombouts, Bart Rypma, Bradley L. Schlaggar, Sein Schmidt, Rachael D. Seidler, Greg J. Siegle, Christian Sorg, Gao-Jun J. Teng, Juha Veijola, Arno Villringer, Martin Walter, Lihong Wang, Xu-Chu C. Weng, Susan Whitfield-Gabrieli, Peter Williamson, Christian Windischberger, Yu-Feng F. Zang, Hong-Ying Y. Zhang, F. Xavier Castellanos, and Michael P. Milham. Toward discovery science of human brain function. *Proceedings of the National Academy of Sciences of the United States of America*, 107(10):4734–4739, 2010.

[3] K. Börner, S. Sanyal, and A. Vespignani. Network science. *Annual Review of Information Science & Technology*, 41:537–607, 2007.

[4] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

[5] William N. Graham. A comparison of four pseudo random number generators implemented in ada. *Newsletter: ACM SIGSIM Simulation Digest*, 22(2):3–18, 1992.

[6] T. J. Hansen, M. Mørup, and L. Kai Hansen. Non-parametric co-clustering of large scale sparse bipartite networks on the gpu. *2011 IEEE International Workshop on achine Learning for Signal Processing*, 2011.

[7] W.K Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[8] S. Jain and R.M. Neal. A split-merge markov chain monte carlo procedure for the dirichlet process mixture model. *Journal of Computational and Graphical Statistics*, 13(1):158–182, 2004.

[9] C. Kemp, J. B. Tenenbaum, T. L. Griffiths, T. Yamada, and N. Ueda. Learning systems of concepts with an infinite relational model. *Proceedings of the national conference on artificial intelligence*, 21(1):4–27, 2006.

[10] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation - Special issue on uniform random number generation*, 8(1):3–30, 1998.

[11] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[12] K. Nowicki and T. A. B. Snijders. Estimation and prediction for stochastic blockstructures. *The American Statistical Association*, 96(455):1077–1087, 2001.

[13] S. Ogawa, T. M. Lee, A. R. Kay, and D. W. Tank. Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *Proceedings of the National Academy of Sciences*, 87(24):9868–9872, 1990.

[14] David Page. Markov chain monte carlo (presentation 2009).

[15] K. Palla, D.A. Knowles, and Z. Ghahramani. An infinite latent attribute model for network data. *Proceedings of the 29th International Conference on Machine Learning*, pages 1607–1614, 2012.

[16] Franck Picard. An introduction to mixture models. 2007.

[17] Programming Guide: AMD Accelerated Parallel Processing. http://developer.amd.com/wordpress/media/2012/10/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf, 2012.

[18] M Mørup, K.H. Madsen, A.M. Dogonowski, H. Siebner, and L.K. Hansen. Infinite relational modeling of functional connectivity in resting state fmri. *Neural Information Processing Systems 23*, 2010.

[19] Johannes Scheibe and Kofoed Wind. Large-scale modelling of functionel connectivity in resting state fmri.

[20] M. N. Schmidt and M. Mørup. Non-parametric bayesian modeling of complex networks. *IEEE Signal Processing Magazine*, 30(3):110–128, 2013.

[21] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[22] N. Singla, M. Hall, B. Shands, and Chamberlain. Financial monte carlo simulation on architecturally diverse systems. *High Performance Computational Finance*, 2008.

[23] Alan E. Gelfand; Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, 1990.

[24] Bjarne Stroustrup. C++11 style. Presentation at: GoingNative 2012.

[25] B. Walsh. Markov chain monte carlo and gibbs sampling (lecture notes 2004).

[26] Janett Walters-Williams and Yan Li. Estimation of mutual information: A survey. *Lecture Notes in Computer Science*, 5589:389–396, 2009.

[27] Z. Xu, V. Tresp, K. Yu, and H.-P. Kriegel. Learning infinite hidden relational models. *IEEE Signal Processing Magazine*, 2006. in Proceedings of the 22nd International Conference on Uncertainty in Artificial Intelligence.

[28] S. Zhu, K. Yu, and Y. Gong. Stochastic relational models for large-scale dyadic data using mcmc. *Advances in Neural Information Processing Systems 21 - Proceedings of the 2008 Conference*, pages 1993–2000, 2009.