Spring 2014

Application of Message Passing and Sinkhorn Balancing Algorithms for Probabilistic Graphical Models

Lakshmi Ananthagopal
San Jose State University

Application of Message Passing and Sinkhorn Balancing Algorithms for Probabilistic Graphical Models

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Lakshmi Ananthagopal

May 2014

The Designated Project Committee Approves the Project Titled

Application of Message Passing and Sinkhorn Balancing Algorithms for Probabilistic Graphical Models

by

Lakshmi Ananthagopal

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2014

Dr. Sami Khuri Department of Computer Science

Dr. Mark Stamp Department of Computer Science

Dr. Chris Tseng Department of Computer Science

ABSTRACT

Application of Message Passing and Sinkhorn Balancing Algorithms for Probabilistic Graphical Models

by Lakshmi Ananthagopal

Probabilistic Graphical Models (PGMs) allow us to map real world scenarios to a declarative representation and use it as a basis for predictive analysis. It is a framework that allows us to express complex probability distributions in a simple way. PGMs can be applied to a variety of scenarios wherein a model is built to reflect the conditional dependencies between random variables and then used to simulate the interactions between them to draw conclusions. The framework further provides many algorithms to analyze these models and extract information.

One of the applications of PGMs is in solving mathematical puzzles such as Sudoku. Sudoku is a popular number puzzle that involves filling in empty cells in an 'N x N' grid in such a way that numbers 1 to N appear only once in each row, column and '$N^{1/2}$ x $N^{1/2}$' sub-grid. We can model this problem as a PGM and represent it in the form of a bipartite graph. The main concepts we employ to obtain an algorithm to solve Sudoku puzzles are factor graphs and message passing algorithms. In this project we attempt to modify the sum-product message passing algorithm to solve the puzzle. Additionally, we implement a solution using Sinkhorn balancing to overcome the impact of loopy propagation and compare its performance with the former.

ACKNOWLEDGEMENTS

I am extremely thankful to my project advisor, Dr. Sami Khuri, for his support, guidance, and encouragement throughout this project. I would also like to thank my committee members, Dr. Mark Stamp and Dr. Chris Tseng for their time and support.

My special thanks to Natalia Khuri and Professor Fernando Lobo for their invaluable insights and suggestions during the course of this project.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

There are many real world scenarios where a person or a system has to make use of available data to draw conclusions regarding the situation. There are numerous real world problems in fields such as Artificial Intelligence, Robotics, Computational Biology, Computer Vision etc., where the system or individual needs to extract useful information from highly complex yet structured data. In other words, we need to gain global real world insight from the limited local observations that we have.

To gain some perspective we can consider a few real world examples. If we were to visit a doctor, in order to diagnose the problem, he requires some additional information about the patient. This includes the patient's symptoms, test results, food habits, known allergies, personal characteristics such as height and weight, etc. These data help the doctor narrow down on the possible disease and suggest a suitable course of action. This is achieved by mapping symptoms to a predefined model that defines the possible causes for those symptoms.

This reference model that we build is usually ***declarative*** in nature. In this approach we construct a model of the real world scenario that we would like to study. The model encompasses our understanding and knowledge of the system in a machine readable form, so that this information can then be processed by various algorithms to answer questions. The main property of such a declarative representation is that it clearly separates the information from the reasoning. The representation is independent of the reasoning algorithms that will be applied on it and has its own clear semantics. Thus, models with declarative representations contain

knowledge in a format that may be manipulated, decomposed and analyzed by the various reasoning algorithms, independent of its content.

In order to understand this property better, let us go back to the example of the medical diagnosis we considered earlier. We could have a model where we represent our knowledge about different diseases and how they relate to various symptoms. The model could also contain various test result data. Now given this, we can use any reasoning algorithm which takes this model and the patient's symptoms as input and then attempts to answer questions regarding the patient's condition. Thus, the algorithms being developed can be generic and applied to any model within a broader class. Conversely, we can improve our model for a specific application domain without constantly modifying our reasoning algorithms [5].

Another major property of the systems that PGMs are applied to is ***uncertainty***. Uncertainty is a given condition when we are dealing with real-world applications due to many factors: partial observations, noisy data and difficulty in modeling the observations. For example, in the medical diagnosis example we see that quite often it is very tough to establish a concrete relationship between a disease and symptoms. There are many overlapping features that prevent us from defining universally true relationships between a disease and its symptoms or prognosis. Thus, uncertainty arises primarily due to our limited ability to observe the real world and model it accurately.

Given this ambivalence, in order to draw meaningful conclusions from these models, we employ probability theory which basically provides us with a framework to consider multiple possible outcomes and their probability. It allows us to consider options that are unlikely, but not

impossible, and sideline the multitude of exception scenarios which would otherwise increase our effort.

1.1 Probabilistic Models

A common characteristic of complex systems is the presence of many interrelated domains, which need to be considered during the reasoning phase. These domains are represented in terms of a set of ***random variables***, whose values define a property of the system. Identifying these random variables is an important step during the design phase and it depends on the questions that we wish to have answered. The primary aim then of the reasoning algorithms is to probabilistically guess the values of a variable when observations about others may be given. To achieve this we construct a ***joint distribution*** over all the possible values of a set of random variables.

Consider the following example [4]:

Example 1.1: Suppose we choose to represent a student's grades in a subject. The random variables that might be considered here are:

Difficulty: D_0 and D_1 (where D_0 represents that the subject is not difficult and D_1 represents that it is difficult)

Intelligence: I_0 and I_1 (where I_0 represents that the student is not intelligent and I_1 represents that the student is intelligent)

Grade: G_0 (A), G_1 (B) and G_3 (C) (where each represents the grade the student got in that subject)

Thus, in this example our probability space has 2x2x3=12 values corresponding to the values assigned to these 3 variables.

Table 1: Joint Probability Distribution

I	D	G	Probability
i^0	d^0	g^1	x^1
i^0	d^0	g^2	x^2
i^0	d^0	g^3	x^3
i^0	d^1	g^1	x^4
i^0	d^1	g^2	x^5
i^0	d^1	g^3	x^6
i^1	d^0	g^1	x^7
i^1	d^0	g^2	x^8
i^1	d^0	g^3	x^9
i^1	d^1	g^1	x^{10}
i^1	d^1	g^2	x^{11}
i^1	d^1	g^3	x^{12}

$$\sum_{1}^{12} \boldsymbol{x^i = 1}$$

Given such a joint distribution we can, for example, ask questions such as how likely is the student to get an A grade in the subject given that he is intelligent and the course is easy.

P (Grade=A| Intelligent=True, Difficulty=False)

1.2 Probabilistic Graphical Models

In Example 1.1, we have only 12 distinct possibilities. However, in the real world there might be hundreds of attributes to be considered. In such a scenario, we can use probabilistic graphical models to describe them compactly.

PGMs use a graph-based approach to compactly represent complex distributions with a large number of variables. In this form of representation, the nodes correspond to the random variables and the edges signify the interactions between them.

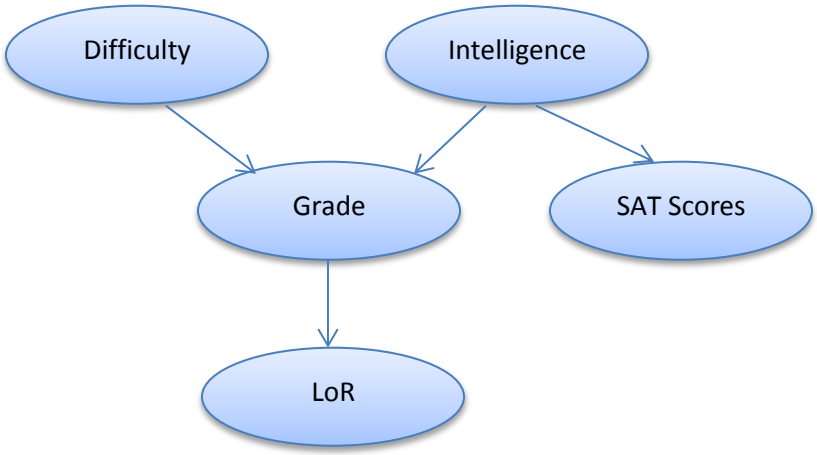


Figure 1: Bayesian Network (Directed Graph)

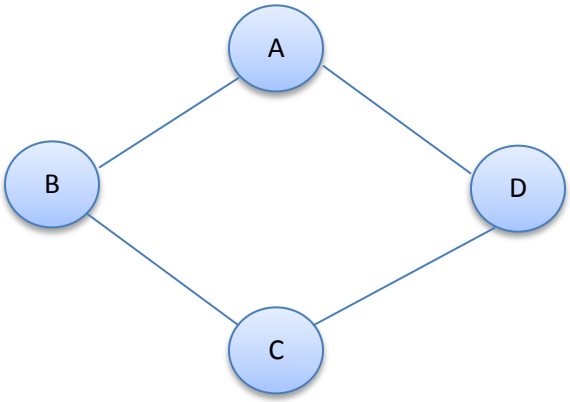


Figure 2: Markov Network (Undirected Graph)

The graphical representations primarily appear in two flavors – directed graphs called 'Bayesian Networks' and undirected graphs called 'Markov Networks'. Both however emphasize assertion of independence and factorization of the distribution.

From Figure 1 we can see that the student's intelligence is completely independent of the difficulty of the subject. The grade he obtains, however, is dependent on both the difficulty and the intelligence. The SAT score is only dependent on the intelligence and getting a letter of recommendation depends only on the grade. Additionally, the Joint Probability Distribution of the system can be obtained by factorizing the entire distribution for convenience. From the figure, we see that

$$\mathbf{P(D, I, G, S, L) = P(D)\,P(I)\,P(G|I, D)\,P(S|I)\,P(L|G)} \qquad \textbf{Eq. 1.1}$$

Similarly in Figure 2, for example, we see that 'A' is independent of 'C' given 'B' and 'D', and 'B' is independent of 'D' given 'A' and 'C'. This form of representation also supports factorization where we can represent the joint probability distribution in the following way:

$$\mathbf{P(A, B, C, D) = \frac{1}{Z}\,\phi_1(A, B)\,\phi_2(B, C)\,\phi_3(C, D)\,\phi_4(A, D)} \qquad \textbf{Eq. 1.2}$$

Thus, the graph effectively represents the independencies in the system while also serving as a skeletal framework to help factorize the distribution.

1.2.1 Representation, Inference and Learning

One of the main perks of employing a graphical framework is that it highlights the property that variables in a system usually only interact directly with few others. This allows us to greatly simplify joint distributions that are immensely large and ***represent*** them in a very

transparent way. We can also draw ***inferences*** effectively and efficiently by applying reasoning algorithms to these graph structures. Lastly, this framework supports ***learning*** from data models based on past observations.

Thus, PGMs employ a data driven approach to model a real world scenario and draw meaningful information from it. In this effort, the three components – namely representation, inference and learning – play a very important role in enhancing the intelligence of the resulting system. The declarative representation allows us to convincingly model the real world; we are then able to draw inferences to answer a range of questions and finally when put together with other accumulated data and knowledge, we learn new information about the problem.

Probabilistic Graphical Models as a whole encompasses a lot of concepts. In the next chapters we cover only those aspects of PGMs that are relevant to this work. Chapter 4 then introduces the Sudoku problem and depicts how it can be modelled as a PGM. Chapter 5 details the three algorithms that can be used to solve the puzzle, while Chapter 6 focuses on the analysis and implementation details. In Chapter 7 we present some experimental results and finally the conclusion and future work scope is covered in Chapter 8.

CHAPTER 2

Background Concepts

2.1 Factors

Factors are nothing but a function/table that takes all possible combinations of Random Variables (RVs) in the model and gives a result for each assignment of those RVs. These RVs or arguments are referred to as the ***scope*** of the factor. An example of a factor is any Joint Distribution or probability vector.

2.1.1 Properties of Factors

a) **Factor Product** – Given two factors $\phi_1(A, B)$ and $\phi_2(B, C)$ the factor product is given by

$$\phi_1 (A, B) * \phi_2 (B, C) = \phi_3 (A, B, C) \qquad \text{Eq. 2.1}$$

Example 2.1: If we have $\phi_1 (A, B)$ and $\phi_2 (B, C)$ as

A1	B1	X1
A1	B2	X2

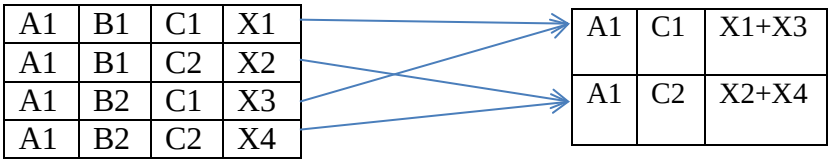
B1	C1	Y1
B1	C2	Y2
B2	C1	Y3
B2	C2	Y4

Then $\phi_3 (A, B, C)$ is given by

A1	B1	C1	X1*Y1
A1	B1	C2	X1*Y2
A1	B2	C1	X2*Y3
A1	B2	C2	X2*Y4

b) **Factor Marginalization** – Given factor ϕ_1 (A, B, C), to reduce scope to ϕ_2 (A, C) we marginalize 'B'.

Example 2.2:

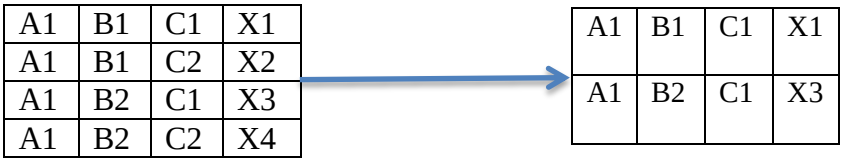


A1	B1	C1	X1
A1	B1	C2	X2
A1	B2	C1	X3
A1	B2	C2	X4

A1	C1	X1+X3
A1	C2	X2+X4

c) **Factor Reduction** - Given factor ϕ_1 (A, B, C), we reduce it to a Conditional Probability Distribution where C_1 is given.

Example 2.3:



A1	B1	C1	X1
A1	B1	C2	X2
A1	B2	C1	X3
A1	B2	C2	X4

A1	B1	C1	X1
A1	B2	C1	X3

2.2 Graphs

A graph is a data structure that contains a set of nodes and edges. A pair of nodes 'X_i' and 'X_j' is usually connected by an edge which can either be directed ($X_i \rightarrow X_j$) or undirected ($X_i - X_j$).

2.2.1 Basic graph concepts and definitions

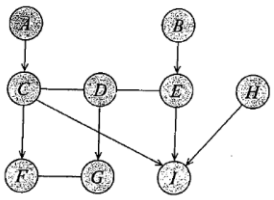


Figure 3: Sample Graph [5]

Based on *Figure 3* we can define the following terms.

a) **Child and Parent nodes** – Whenever we have '$X_i \rightarrow X_j$' we say that 'X_j' is the ***child*** of 'X_i' and 'X_i' is the ***parent*** of 'X_j'.

   Example 2.4: A is the parent of C and C is the child of A.

b) **Neighbor and Adjacent nodes** – When we have '$X_i - X_j$' then 'X_i' and 'X_j' are considered neighbors. However all nodes connected to 'X_i' by either a directed or an undirected edge are considered adjacent nodes.

   Example 2.5: The only neighbor of C is D but its adjacent nodes are A, F, D and I.

c) **Degree and Indegree** – The degree is the total number of edges from/to a node while the indegree is the number of directed edges to the node.

   Example 2.6: Degree of C is 4 and its indegree is 1.

2.2.2 Subgraphs

"A subgraph is a part of a graph that consists of only a subset of the nodes." [5]

a) Induced Subgraph – "A subgraph *H* of a graph *G* is said to be **induced** (or **full**) if, for any pair of vertices *x* and *y* of H, *xy* is an edge of *H* if and only if *xy* is an edge of G." [7]

Example 2.7: Figure 4 below shows an induced sub graph of Figure 3

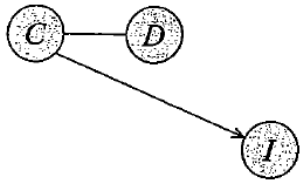


Figure 4: Induced Subgraph [5]

b) Cliques

"A clique is a maximal fully connected sub graph." [5]

In other words, a *Clique* is a complete graph; it is defined as a graph where every vertex is adjacent to every other vertex.

A ***maximal clique*** is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique.

A ***maximum clique*** is a clique of the largest possible size in a given graph. Maximum cliques are therefore maximal cliques (but not necessarily vice versa).

2.2.3 Cluster Graphs

The cluster graph is a data structure where each node is a cluster containing a subset of the variables. The nodes or clusters are connected by undirected edges when they both contain some intersecting variables. An example is shown in Figure 5.

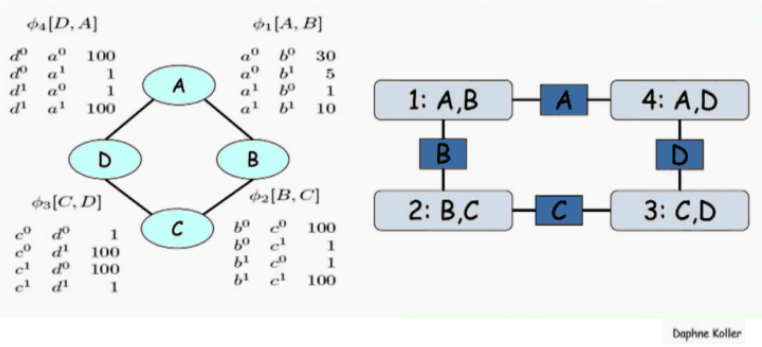


Figure 5: Example of Cluster Graph [4]

2.2.3.1 Properties of Cluster Graphs

Two main properties of Cluster Graphs are [4]:

1. **Family Preservation**

   "Given a set of factors ϕ, we assign each ϕ_k to a cluster $C_{\alpha(k)}$ such that Scope $[\phi_k]$ is a subset of $C_{\alpha(k)}$."

2. **Running Intersection**

   "For each pair of clusters C_i, C_j and variable $X \in C_i \cap C_j$ there exists a unique path between C_i and C_j for which all clusters and sepsets contain X."

2.2.4 Factor Graphs

Definition: "A factor graph is a bipartite graph representing the factorization of a function."[8]

Thus, a factor graph, in summary, contains both regular variable nodes and factor nodes, and represents the dependencies between them. If a variable appears in the scope of a factor then an edge is introduced between the factor node and variable node.

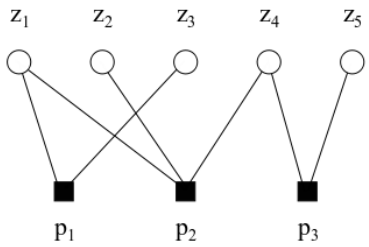


Figure 6: Example of a Factor Graph [8]

In the next chapter we introduce the Belief Propagation algorithms and see how they are applied to factor graphs.

CHAPTER 3

Algorithms and Concepts

3.1 Belief Propagation Algorithms

"Belief propagation algorithms are normally presented as message update equations on a factor graph, involving messages between variable nodes and their neighboring factor nodes and vice versa." [9]

Generalized message passing between nodes/clusters is carried out as follows:

1. We have an undirected Graph whose nodes are clusters and the edges between the nodes (termed as sepsets) contain the intersecting variables of the two clusters being connected.
2. We are also given a set of factors (ϕ) which are assigned to one of the clusters such that the scope of the factor is a subset of the cluster.
3. We then define the new factors associated with each cluster as the product of all the factors assigned to the cluster. This new factor is represented as $'\Psi i(Ci)'$.
4. Subsequently a message from cluster 'i' to cluster 'j' is defined by the product of the factor of cluster 'i' and the sum of all the incoming messages to cluster 'i', except the message from cluster 'j' to cluster 'i'.

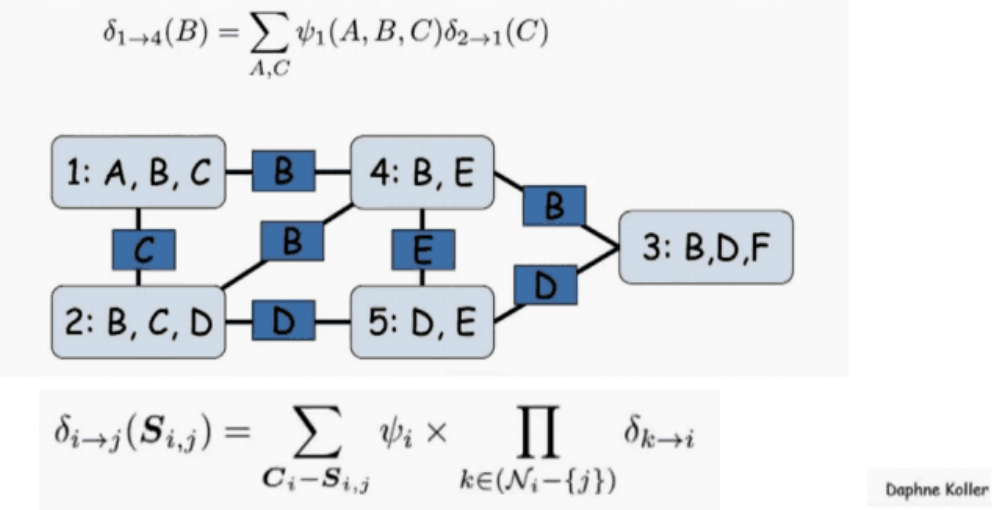


Figure 7: Example of Message Passing [4]

3.1.1 Properties of Belief Propagation Algorithm

1. **Convergence of a Belief Propagation Algorithm implies Calibration**

   First, let us define Calibration. Given that a Cluster belief is defined as

   $\beta_i(C_i) = \Psi i * \prod_{k\in Ni} \delta_{k\to i}$ **Eq. 3.1**

   A cluster graph is ***calibrated*** if every pair of adjacent clusters C_i, C_j agree on their sepset ($S_{i,j}$) (variables shared between the two clusters) i.e.

   $\sum_{Ci\to Si,j} \beta_i(C_i) = \sum_{Ci\to Si,j} \beta_j(C_j)$ **Eq. 3.2**

   The algorithm has converged if the message at the next time stamp is equal to the message at the current time stamp:

   $\delta_{i\to j}(S_{i,j}) = \delta'_{i\to j}(S_{i,j})$ **Eq. 3.3**

   Thus, from this we can prove that the graph is calibrated as per the following proof [4]:

$$\delta'_{i\to j}(S_{i,j}) = \sum_{Ci-Si,j}(\Psi i * \prod_{k\epsilon(Ni-\{j\})} \delta_{k\to i}) \quad \textbf{Eq. 3.4}$$

But based on Eq. 3.1, we can make the following substitution

$$\delta'_{i\to j}(S_{i,j}) = \sum_{Ci-Si,j}(\beta_i(C_i)) / \delta_{j\to i}(S_{i,j})$$

That implies, $\delta_{j\to i}(S_{i,j}) * \delta'_{i\to j}(S_{i,j}) = \sum_{Ci-Si,j}(\beta_i(C_i))$

Similarly, $\delta_{j\to i}(S_{i,j}) * \delta'_{i\to j}(S_{i,j}) = \sum_{Cj-Si,j}(\beta_j(C_j))$

Thus, $\boxed{\sum_{Ci-Si,j}(\beta_i(C_i)) = \sum_{Cj-Si,j}(\beta_j(C_j))}$ Eq. 3.5

Hence, Convergence of a Belief Propagation algorithm implies Calibration.

2. Reparameterization

This property basically implies that cluster graph beliefs are nothing but a different set of parameters that capture the original un-normalized measures that define our distribution. We see that the ratio of the Cluster beliefs to the Sepset beliefs is nothing but the original un-normalized measure and hence can be reassured that no information was lost due to the belief propagation algorithm.

Next, we move to the subject of 'Reparameterization' or no information loss due to Belief Propagation [4]:

We have beliefs of clusters defined in Eq. 3.1 and sepset belief is defined by the equation

$$\mu_{i,j}(S_{i,j}) = \delta_{j \to i}\, \delta_{i \to j} \qquad \text{Eq. 3.6}$$

If we divide the product of all beliefs by the products of all sepsets we get:

$$\left(\prod_i \beta_i\right) / \left(\prod_{i,j} \mu_{i,j}\right) = \left(\prod_i \left(\Psi_i * \prod_{k \epsilon N i} \delta_{k \to i}\right)\right) / \left(\prod_{i,j} \delta_{i \to j}\right)$$

$$= \prod_i \Psi_i \quad \text{(Since each message expression appears exactly twice)}$$

Thus, $\left(\prod_i \beta_i\right) / \left(\prod_{i,j} \mu_{i,j}\right)$ **= Unnormalized measure** Eq. 3.7

Hence, Belief Propagation does not lead to any information loss**.**

3.1.2 Loopy Belief Propagation

Loopy Belief Propagation occurs when we perform message passing in cyclic graphs. In such a scenario, due to the loop back property, information is counted twice leading to a bias in the belief.

Assume we have three nodes ‘A’, ‘B’ and ‘C’ which are all inter-connected. When node ‘A’ sends a message to node ‘B’, ‘B’ updates its belief and passes on the message to ‘C’. Similarly, ‘C’ updates its beliefs and forwards the message to ‘A’. When this message reaches node ‘A’, it is under the assumption that this is brand new information and uses it to update its

beliefs. However, in reality this message is diluted in value as it contains information originally passed by 'A'.

3.2 Sum-Product Algorithm

The Sum-Product message passing algorithm can perform efficient and exact inference provided that the factor graph has no loops i.e. it is a tree. In case the graph contains loops then there is a possibility that the algorithm will never converge.

The algorithm is defined as follows [4]:

Sum Product Belief Propagation Algorithm

Assign each factor $\phi_k \in \Phi$ to a cluster $C_{\alpha(k)}$

Construct initial potentials: $\Psi_i\ C_i = \prod_{k:\alpha(k)=i}\ \phi_k$ Eq. 3.8

Initialize all messages to be 1

Repeat

Select edge and pass message: $\boldsymbol{\delta_{i\to j}\ (S_{i,j})\ =\ \sum_{Ci-Si,j}(\Psi i * \prod_{k\epsilon(Ni-\{j\})}\ \delta_{k\to i})}$

End Repeat

Compute Belief: $\boldsymbol{\beta_i\ (C_i) = \Psi_i * \prod_{k\epsilon Ni}\ \delta_{k\to i}}$

3.3 Max-Product Algorithm

The Max-Product Algorithm is a variant of the Sum-Product Algorithm where we replace the 'summation' function by the 'max' function. The main advantage of max product over sum product is seen when it is applied to cyclic graphs where sum-product algorithm faces the loopy propagation issue.

The algorithm is defined as follows:

Max Product Belief Propagation Algorithm

Assign each factor $\phi_k \in \Phi$ to a cluster $C_{\alpha(k)}$

Construct initial potentials: $\Psi_i\ C_i = \prod_{k:\alpha(k)=i} \phi_k$

Initialize all messages to be 1

Repeat

Select edge and pass message: $\boldsymbol{\delta_{i\to j}\ (S_{i,j}) = \max\left(\Psi i * \prod_{k\epsilon(Ni-\{j\})} \delta_{k\to i}\right)}$

End Repeat

Compute Belief: $\boldsymbol{\beta_i\ (C_i) = \Psi_i * \prod_{k\epsilon Ni} \delta_{k\to i}}$

The next chapter introduces the Sudoku problem and shows how it can be modelled as a PGM. We get acquainted with the various notations and definitions that we will be using throughout this work and also understand how the generic Belief Propagation algorithms that we presented in this chapter can be adapted to solve the Sudoku puzzle.

CHAPTER 4

The Sudoku Problem

4.1 What is the Sudoku problem?

The traditional Sudoku problem is a logic based number puzzle. The main objective is to fill an ‘N x N’ grid (where N is the square of a number) with digits ranging from {1...N}, such that each digit appears only once in each row, each column and each of the N sub-grids which in turn are of size ‘sqrt (N) x sqrt (N)’.

In a standard Sudoku problem, we deal with a ‘9 x 9’ grid which is partially filled with numbers ranging from 1 to 9. However, the puzzle can be scaled to any ‘N x N’ representation. In any case, a combination of the number of filled cells and the arrangement of filled cells determines the difficulty level of a puzzle.

4.1.1 A brief introduction to the Mathematics of Sudoku.

Sudoku puzzles belong to a class of combinatorial mathematical problems. It deals with identifying a unique solution that satisfies all the set constraints. In a sense, the completed Sudoku puzzle can be viewed as a Latin Square, with the additional constraint of unique values in each of the N sub-grids.

The mathematical analysis of Sudoku primarily covers the possible solutions for different variants of the puzzle, the influence of the initial given values in the puzzle and the logic behind solving a puzzle. In this writing project the focus is on the third point.

The general problem of solving an ‘N x N’ puzzle is known to be NP-Complete [11] [12]. As such, an approach to solve these puzzles will involve employing some sort of approximation, randomization, parameterization or heuristic algorithm. A computer can solve a ‘9 x 9’ Sudoku within seconds using such methods but on a larger scale it becomes quite impossible to check the vast number of potential combinations in reasonable finite time and even then there is no guarantee of finding a solution.

In this project we map the Sudoku puzzle to a PGM and employ three algorithms that work on this model to arrive at a solution. All three algorithms can be classified as a type of probabilistic solver. This approach differs from the traditional solutions as it employs general rules of inference to solve the problem as opposed to specific human-like tricks. As such, this solution can be applied to a wide range of similar constraint satisfaction problems like the ‘Low Density Parity Check’ (LDPC) decoding problem. Interestingly, both the Sudoku and the LDPC problems take on the same form when modelled as a PGM; and hence the success of the BP approach in solving certain LDPC problems inspired a similar approach to solve the Sudoku puzzle [1].

4.2 Representing the Sudoku puzzle as a PGM.

4.2.1 Graph representation

The puzzle can be represented as a bipartite graph. A bipartite graph is defined as “a graph whose vertices can be divided into two disjoint sets S and C such that the graph’s edges connect vertices in S only to vertices in C and vice versa.” [1]

Thus, all the cells in an 'N x N' puzzle can be mapped to set 'S' and all the constraints that need to be satisfied can be mapped to set 'C'. All cells (S_n) are mapped to the set 'S' by assigning indexes from 1 to N^2 in a row-wise scan order. Similarly, the different row, column and sub-grid constraints (C_m) are mapped to the set 'C' in that order. Once the two sets are defined, edges are introduced based on the relationship between a cell and a constraint. An (undirected) edge is present between a cell node and a constraint node if the constraint is applicable to that cell node. Figure 8 defines the relationships for various value cells and constraints in a '9 x 9' puzzle.

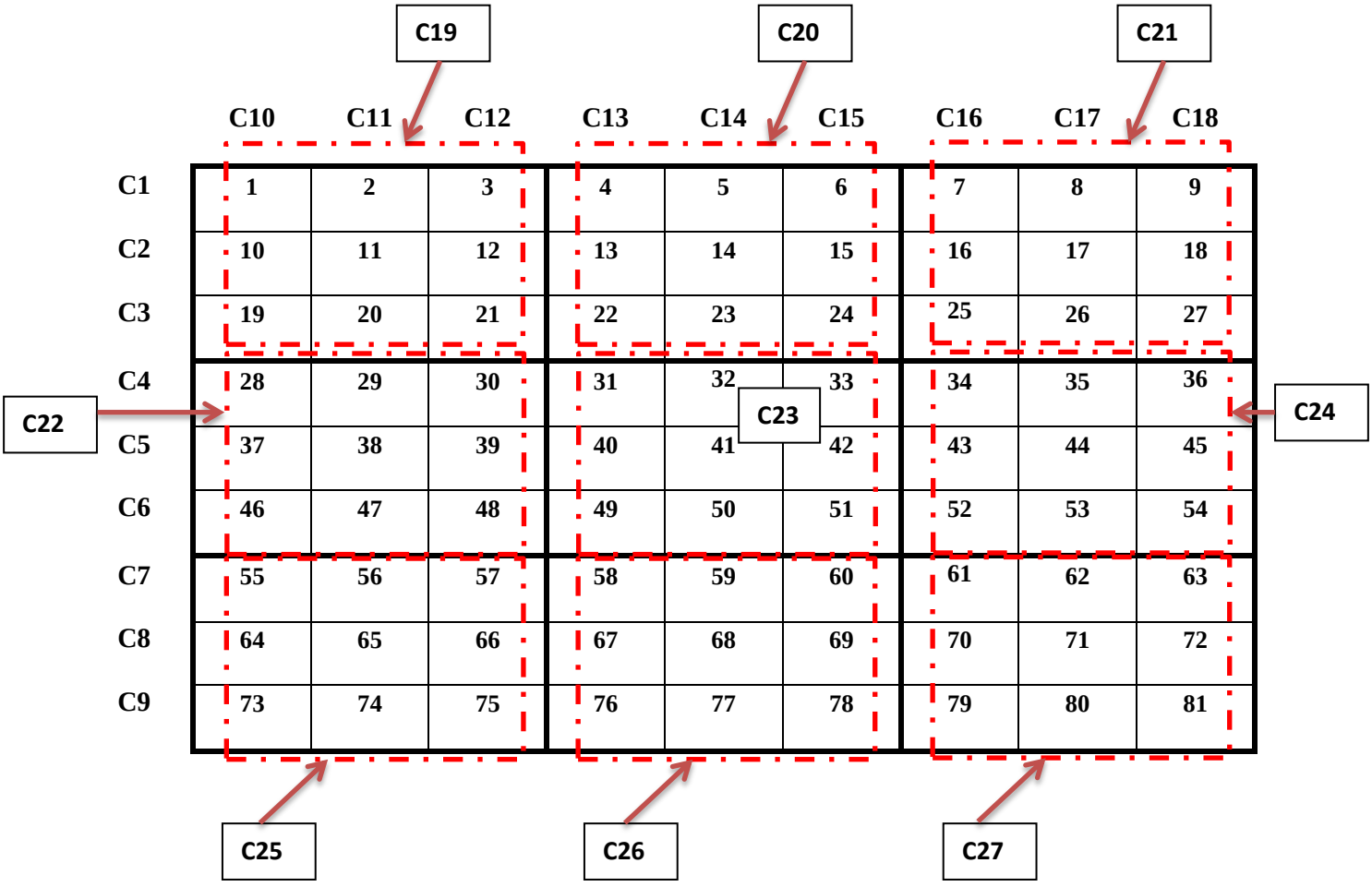


Figure 8: Constraints for a '9 x 9' Sudoku puzzle

The value/constraints matrix given in Figure 8 can be mapped to a bipartite graph as shown in Figure 9:

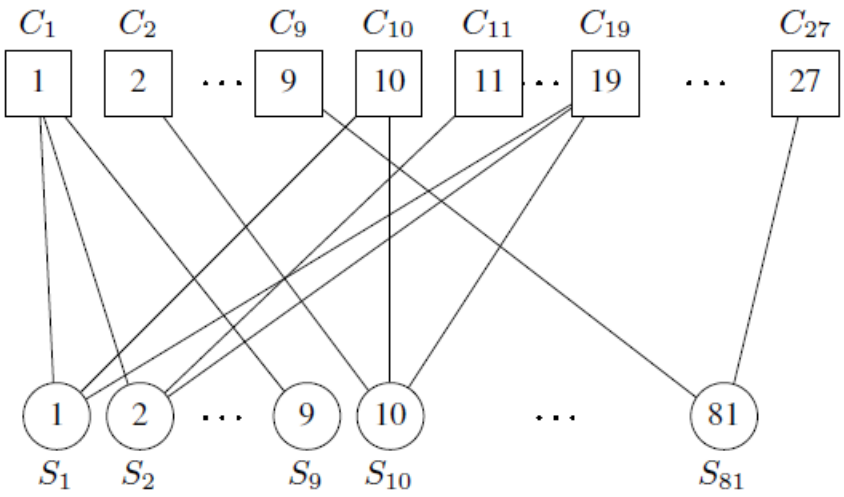


Figure 9: Bipartite graph associated with the '9 x 9' Sudoku puzzle [2]

As can be observed from Figure 8 and Figure 9, an edge is present between a cell node and a constraint node **if and only if** S_n is involved in C_m. Thus, every cell node is connected to exactly 3 constraint nodes (one each for row, column and sub-grid constraint) and each constraint node is connected to 9 (in the case of '9 x 9' puzzle) cell nodes since each row, column and sub-grid encompass 9 cells.

4.2.2 Introducing the probability factor

As the name 'Probabilistic Graphical Models' suggests, any problem set in this class ideally needs to satisfy two main characteristics: a) The ability to be represented graphically and b) Represent some form of uncertainty in the system. Having shown how the problem can be mapped to a bipartite graph, we now introduce the probability factor.

In an empty puzzle each cell in the grid can hold a value from 1 to 9 (again in the case of a '9 x 9' Sudoku puzzle) with equal probability. However, as mentioned earlier a typical Sudoku puzzle comes with some cells prefilled with values. In any case, each cell node stores a factor of the form [2].

$$p_n = [P(S_n=1)\ P(S_n=2) \ldots P(S_n=N)] \qquad \text{Eq. 4.1}$$

This factor is nothing but a probability vector associated with a cell node. If the cell already contains a value, say k (where $k \in \{1. . . N\}$), then the vector is initialized to have 1 in the k^{th} position and (N-1) zeroes in the other positions. If the cell is empty, then the probability is equally distributed among all contending values after eliminating the values that violate the three constraints that the cell is associated with. At any given point in time, the values in a cell's probability vector always sum to 1 and in the case of a solved puzzle, we end up with a set of factors with 1 in the k^{th} position depending on the final cell value.

4.3 Notations and Definitions

Some of the common terms and notations we need in subsequent sections are now defined [2]. All cases consider a standard '9 x 9' Sudoku puzzle.

1. As already mentioned, the values of each cell in the puzzle is denoted by S_n, where $S_n \in \{1, 2, . . . , 9\}$ for n = 1 to 81.
2. The probability vectors associated with each cell is given by Eq. 4.1 for n= 1 to 81 and N=9.
3. A constraint C_m is always associated with 9 cell nodes. At any stage of the puzzle a constraint can hold only binary values based on whether it is satisfied or not satisfied. A

constraint is said to be satisfied if all the 9 cell nodes associated with it contain distinct values.

$$C_m(s_1, s_2, \ldots, s_9) = \begin{cases} 1 & s_1, s_2, \ldots, s_9 \text{ are all distinct} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 10: Constraint Function definition [2]

4. The factors associated with a constraint node C_m are denoted by q_m. It represents the probability of a constraint being satisfied. A constraint is satisfied when all the cells associated with it hold a distinct value. So for example, if 5 out of the 9 cells in the constraint hold distinct values, then the probability associated with that constraint is given by: $q_m = (5/9) = 0.56$

   This probability value is calculated using the information about the probability of a cell's content from the cell nodes.

5. We denote the set of indices of the cell nodes that participate in a given constraint C_m by N_m. For example, N_1 contains the index values of all the cells in the grid that are a part of constraint 1 i.e. indices of all the cells in the first row of the grid.

   Thus, from Figure 8 we have $N_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

   Similarly $N_{11} = \{2, 11, 20, 29, 38, 47, 56, 65, 74\}$ and $N_{25} = \{55, 56, 57, 64, 65, 66, 73, 74, 75\}$

6. We denote the set of indices of the constraint nodes that are connected to a cell node S_n by M_n. For example, M_1 contains the index values of all the constraint nodes that are

associated with it i.e. the constraint associated with the first row, first column and first sub-grid.

Thus, from Figure 8 we have M_1 = {1, 10, 19}

Similarly M_{11} = {2, 11, 19} and M_{25} = {3, 16, 21}

7. We use a double subscript notation ($N_{m, n}$) to identify elements that have been removed from a set. This notation implies all the cells involved in constraint 'm' except cell 'n'. For example, N_{25} = {55, 56, 57, 64, 65, 66, 73, 74, 75} and if we were to remove cell 64 from this set it would be represented as $N_{25, 64}$ = {55, 56, 57, 65, 66, 73, 74, 75}.

4.3.1 Summary of terms and values

Based on the definitions previously seen we can summarize the values associated with each of the terms and arrive at a general pattern that can be adapted for any 'N x N' Sudoku puzzle.

Table 2: Summary of terms and values

Term	4x4 Puzzle	9x9 Puzzle	16x16 Puzzle	NxN Puzzle
Number of cell nodes	16	81	256	N^2
Number of constraint nodes	12	27	48	3N
Size of probability vector associated with each cell node	4	9	16	N
Dimensions of N_m matrix	12x4	27x9	48x16	3NxN
Dimensions of M_n matrix	16x3	81x3	256x3	N^2x3

4.4 Missing Nodes and Value Nodes

The initial distribution of values in a Sudoku puzzle greatly influences the outcome as well as the time it takes to arrive at a solution. In this context, when we are given a puzzle, the

cells that contain an initial value are called ‘Value Nodes’ and the empty cells are called ‘Missing Nodes’.

It has been shown that to arrive at a unique solution, the minimum number of required value nodes in a 9x9 puzzle is 17 [13]. Establishing this is of importance as the presence of multiple solutions hinders the convergence of the algorithms used. Thus, in our test data set we only use puzzles that have more than 17 value nodes and a unique solution.

In the next chapter, we describe in detail the three algorithms that we use to solve the Sudoku puzzles. We explain how each of the algorithms can be adapted specifically to the problem defined and also discuss potential shortcomings in each approach.

CHAPTER 5

Solving the Sudoku puzzle

Having represented the Sudoku problem as a probabilistic model, we employ different 'probability solver' algorithms to arrive at a solution. Unlike the conventional methods which employ specific tricks to solve the problem, here we use the concept of general inference. The first two algorithms are variants of the belief propagation algorithm and the third is a solution based on Sinkhorn balancing. All three however, address the constraint satisfaction problem and can be applied to any problem that falls within that general class.

We have implemented and compared the three algorithms to see how they weigh against each other in terms of effectiveness and performance. Each algorithm tries to address some shortcomings in the previous algorithm, with the ultimate aim being finding a solution for the maximum number of puzzles.

5.1 Solution using Belief Propagation

The belief propagation algorithms are based on the concept of sending probabilistic messages between connected nodes in a graph. In the case of the Sudoku puzzle, a constraint node sends to its adjacent cell nodes, the probability that it is satisfied. It computes this value based on the information from the cell nodes participating in that constraint about the probabilities of their contents. On the other hand, given information about all the connected constraints, a cell node sends a message that essentially contains the probability vector associated with that cell. This exchange of messages continues until all constraints are met or a maximum number of iterations are reached.

Before we get into the algorithm, we define four key terms that form the crux of the algorithm:

1. Constraint to cell node message - $\mathbf{r_{m,n}(x)}$

   The message that constraint C_m sends to cell S_n is defined as:

   $r_{m,n}(x) = P(C_{m\ is}\ \text{satisfied} \mid S_n = x)$

2. Cell node to constraint message - $\mathbf{q_{n,m}(x)}$

   The message that cell S_n sends to constraint C_m is defined as:

   $q_{n,m}(x) = P(S_n = x \mid \text{all constraints involving } S_n \text{ except } C_m \text{ are satisfied})$

3. A Priori probabilities - **P (n = x)**

   This is the probability vector associated with the cell nodes.

4. A Posteriori beliefs - $\mathbf{q_n(x)}$

   This is the calculated cell vector value based on the messages sent and received.

5.1.1 Solution using Sequential Message Passing Algorithm

Sequential Message Passing Algorithm

Initialization: Set the missing nodes to 0; Initialize S_n, N_m, M_n, C_m (See section 4.3) and define the initial probability vector distribution for S_n (called sVector) and C_m (called cVector) according to initial clues and uniformly in cells with no clues. Set count = 0 and maxIterations.

Repeat:

For each cell S_n, Perform a row wise scan and identify the first missing node.

Send constraint to cell message: $\mathbf{r_{m,n}(x) = \prod_{n' \in Nm,n} (1 - q_{n',m}(x))}$ Eq. 5.1 [1]

Calculate the a posteriori beliefs: $\mathbf{q_n(x) = P(n = x) \prod_{m \in Mn} r_{m,n}(x)}$ Eq. 5.2 [1]

Normalize the a posteriori belief

Update the cell probability vector value

Send cell to constraint message: $\mathbf{q_{n,m}(x) = P(n = x) \prod_{m' \in Mn,m} r_{m,n}(x)}$ Eq. 5.3 [1]

End of S_n

Update the cell values and check if a valid solution has been found. If yes, break with success.

Increment count = count + 1.

If count > maxIterations, break with failure.

End Repeat

5.1.1.1 Issues due to Loopy Belief Propagation

If there were no cycles in the graph, then based on the belief propagation theory, after a certain number of iterations the resulting a posteriori belief would contain sufficient information from the constraints and the prior probability to give an exact solution. However, in the case of the Sudoku problem there exist many short cycles in the graph which bias the results.

To understand the impact of loopy belief propagation let us consider the following simple example [1]:

Example 5.1: Consider three cell nodes S_1, S_2 and S_3 all of which are related under constraint C_1. Node S_1 will send a message to S_2 through C_1 about the probability of it taking on different values. Based on this information S_2 reevaluates its probability vector distribution and passes that

on to S_3 via C_1 again. Similarly S_3 will use the message from S_2 to send a new message to S_1 via C_1. Now, S_1 takes into consideration this new information and updates its probability vector. However, the message is biased as it contains information originally sent by S_1.

It is due to this reason that many puzzles are not solvable by the message passing technique. However, there is one minor advantage in the Sudoku problem. Since this puzzle is presented with a given set of value nodes, there will be no loopy belief propagation on the associated loops of these cells [1]. In the case of a value cell node with value k, the corresponding message vector or probability vector consists of a 1 in the k^{th} position and N-1 zeroes. This is called a Kronecker delta function [1]. And since we normalize the message vector at each step, this forces the outgoing messages from those cell nodes to also be Kronecker delta functions irrespective of the input value it receives. Thus, no node in a cycle can receive looped back information via this node. Thus, we can see that the initial distribution and arrangement of value nodes and the order of visiting missing nodes has a great influence on the loopy belief propagation. In the case of the sequential algorithm, since the missing nodes are always scanned in a row wise manner it always encounters the same loops and falls prey to loopy belief propagation.

5.1.2 Solution using Randomized Message Passing Algorithm

The randomized message passing algorithm seeks to overcome the loopy belief propagation problem in the sequential method by visiting missing nodes in a random fashion. Visiting and resolving values in certain missing nodes could lead to a more favorable distribution of value nodes in the next iteration and could potentially avoid loopy belief propagation.

Randomized Message Passing Algorithm

Initialization: Set the missing nodes to 0; Initialize S_n, N_m, M_n, C_m (See section 4.3); Define the initial probability vector distribution for S_n (called sVector) and C_m (called cVector) according to initial clues and uniformly in cells with no clues. Set count = 0 and maxIterations.

Repeat:

For each cell in S

Generate a random index from 1 to N^2 and check if it is a missing cell.

If yes, check against visited cells stack to see if it has already been visited.

If no, then push cell index into the visited cells stack and perform the following steps

Send constraint to cell message using Eq. 5.1

Calculate the a posteriori beliefs using Eq. 5.2

Normalize the a posteriori belief

Update the cell probability vector value

Send cell to constraint message using Eq. 5.3

End of S_n

Update the cell values and check if a valid solution has been found. If yes break with success.

Increment count = count + 1.

If count > maxIterations, break with failure.

End Repeat

However, while the randomized message passing algorithm increases the chances of arriving at a solution, it is still dependent on the arbitrary selection of missing nodes and is hence not foolproof.

5.2 Solution using Sinkhorn Balancing

The solution based on Sinkhorn balancing is aimed at overcoming the issues faced by the belief propagation algorithms. It is similar to the BP algorithm in the sense that the contents of the puzzle are still modelled probabilistically leading to a reduced search space for potential solutions. However, this solution is based on the concept of grouping belief vectors within a constraint and arriving at doubly stochastic matrix from this initial condition. Thus, from the approach defined, the algorithm is quite evidently not affected by the presence of loops in the graph.

5.2.1 What is Sinkhorn Balancing?

Sinkhorn Balancing, as shown in the example in Figure 11, is the procedure we follow to get a doubly stochastic matrix from an arbitrary matrix that has non-negative elements [3]. A doubly stochastic matrix is one whose rows and columns all sum to 1. Thus, in order to achieve this result, we take the given matrix and normalize all rows and columns successively till we reach some predefined convergence criteria as defined in the algorithm shown in Figure 12.

Figure 11: One iteration of Sinkhorn Balancing [1]

Sinkhorn Balancing (SB)

Input/Initialize: An $N \times N$ matrix Q; max number of iterations M; a tolerance ϵ. Set $k = 0$. Set $Q^{[0]} = Q$.

Repeat:

For $j = 1 : N$ (Vertical: sum and scale columns)

$\chi_j = \sum_l q_{lj}^{[k]}; \quad q_{ij}^{|} = q_{ij}^{[k]} / \chi_j, i = 1, 2, \ldots, N$

End For j

For $i = 1 : N$ (Horizontal: sum and scale rows)

$\rho_i = \sum_l q_{il}^{||}; \quad q_{ij}^{[k+1]} = q_{ij}^{|} / \rho_i, j = 1, 2, \ldots, N$

End For i

If $||Q^{[k+1]} - Q^{[k]}|| < \epsilon$, Set $Q = Q^{[k+1]}$, Break

$k \leftarrow k + 1$. If $k > M, Q = Q^{[k]}$ Break

end Repeat.

Return Q

Figure 12: Sinkhorn Balancing algorithm [3]

5.2.2 Applying 'Sinkhorn Balancing' to the Sudoku problem

The application of Sinkhorn Balancing to the Sudoku problem is, as we mentioned earlier, based on the concept of doubly stochastic matrices. In the case of a solved Sudoku puzzle, the distribution of cell node probability vectors will contain a 1 and 'N-1' zeroes. Now, if we were to group the probability vectors of cells associated with every constraint into a matrix

then in the ideal case, if the constraint is satisfied, what we get should be a doubly stochastic matrix. Thus, this becomes one of the necessary conditions to represent a valid solution.

Based on this idea, we use Sinkhorn balancing to convert the given constraint probability matrix to a doubly stochastic matrix. As shown in the algorithm defined in Figure 13, this step is repeated for each of the constraints in turn till either they are all satisfied or a maximum number of iterations is reached.

Sinkhorn Sudoku Solution (SSS)

Initialization: Set initial probability vectors $\mathbf{p}_1, \ldots, \mathbf{p}_{N^2}$ according to initial clues and uniformly in cells with no clues. Set $k = 0$

Repeat:

For each constraint $m \in \{1, 2, \ldots, 3N\}$:

Form the probability constraint matrix $Q_m^{[k]}$

Sinkhorn balance: $Q_m^{[k+1]} = \mathrm{SB}(Q_m^{[k]})$.

Extract the probabilities $\mathbf{p}_n$ from $Q_m^{[k+1]}$.

End for m

Determine most probable contents S_n from $\mathbf{p}_n$:

$$S_n = \arg\max_j p_{n,j}$$

If all constraints are satisfied, Break with success.

Increment iteration count: $k \leftarrow k + 1$.

If too many iterations, Break with failure

End Repeat

Figure 13: Sinkhorn Sudoku Solution Algorithm [3]

In this chapter, we introduced the three algorithms and studied their particularities. In the next chapter, we study their implementation details and try to analyze the performance of each. Additionally, we cover some of the heuristics that were considered to evaluate their relative performance.

CHAPTER 6

Algorithm Implementation Details

6.1 Data structures and notations

All three algorithm implementations are based on the notations defined in section 4.3 and Table 2. The problem instance and its factors are represented by an array, as follows, for an 'N x N' puzzle:

- matrix=new int[N][N]
- N_m= new int[3*N][N]
- M_n= new int [N*N][3]
- S_n = new int[N*N]
- p_n = new double[N*N][N]
- C_m = new int [3*N]
- qValue = new double [3*N]

6.2 Flowcharts

The flowchart for the Sudoku solver implemented based on the 'Message Passing' and 'Sinkhorn Balancing' algorithms is shown in Figure 14. It defines the flow of the main function. Subroutine calls are expanded in subsequent flowcharts. These flowcharts are defined in Figures 15 to 21.

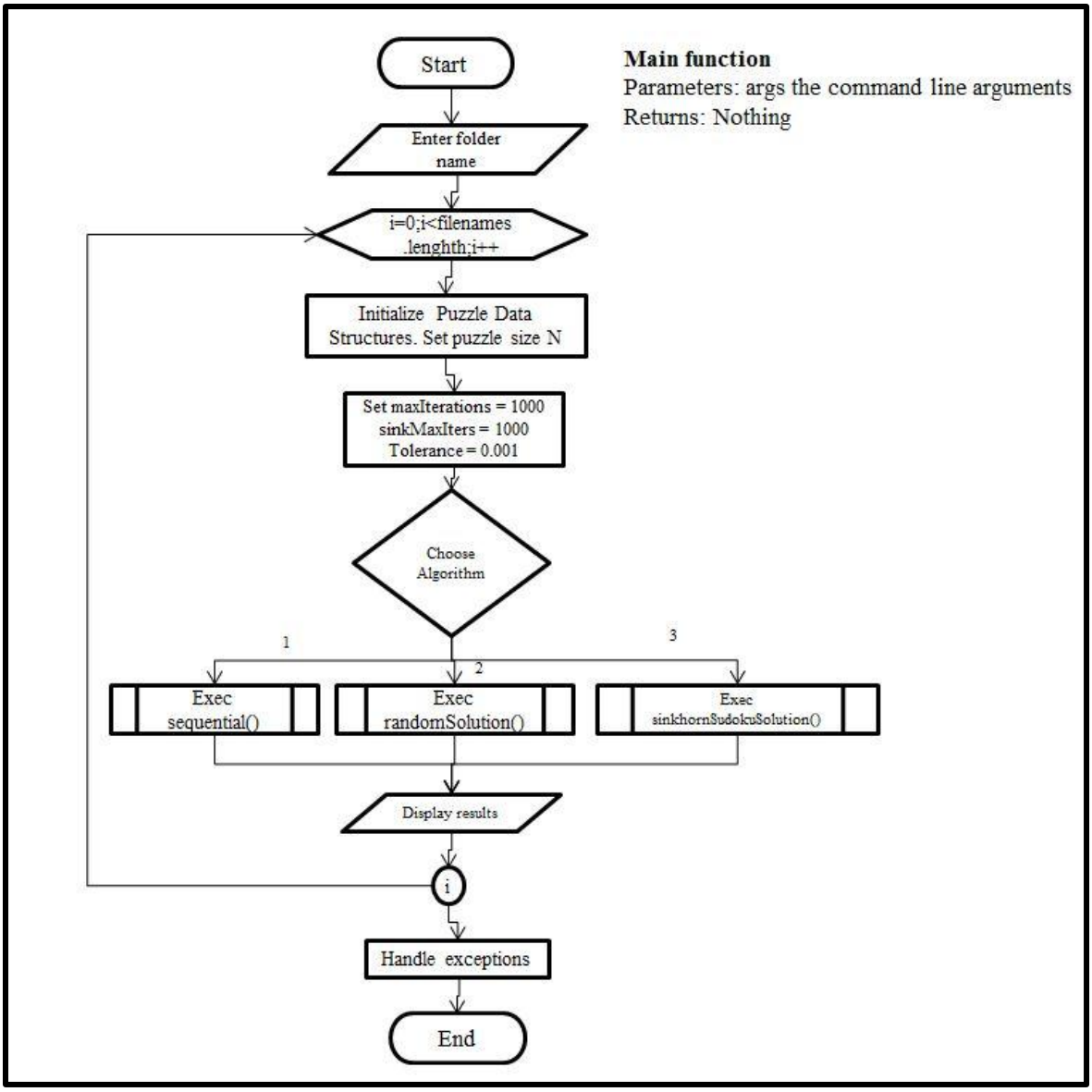


Figure 14: Flowchart of Main Function

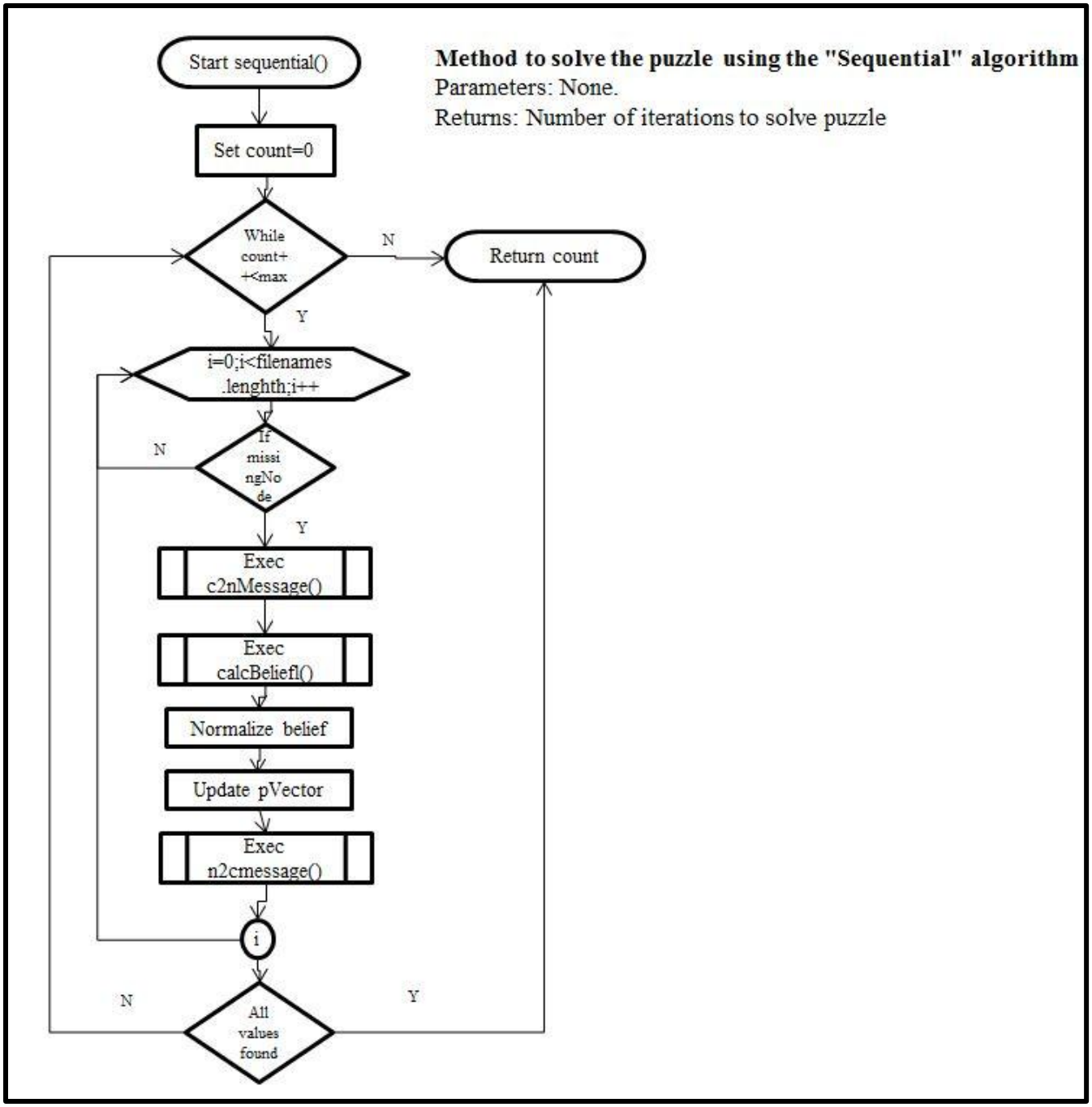


Figure 15: Sequential MP Flowchart

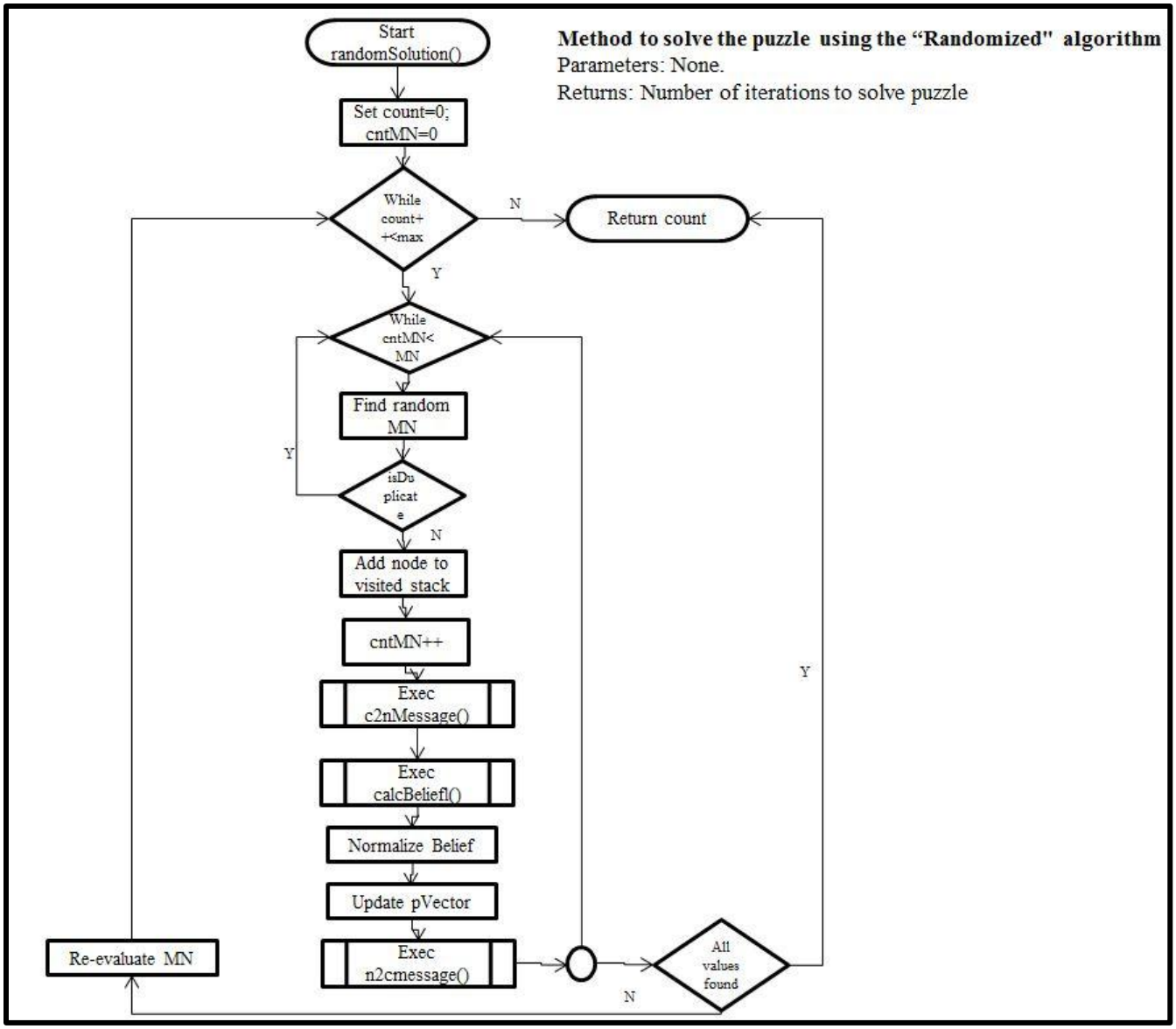


Figure 16: Randomized MP Flowchart

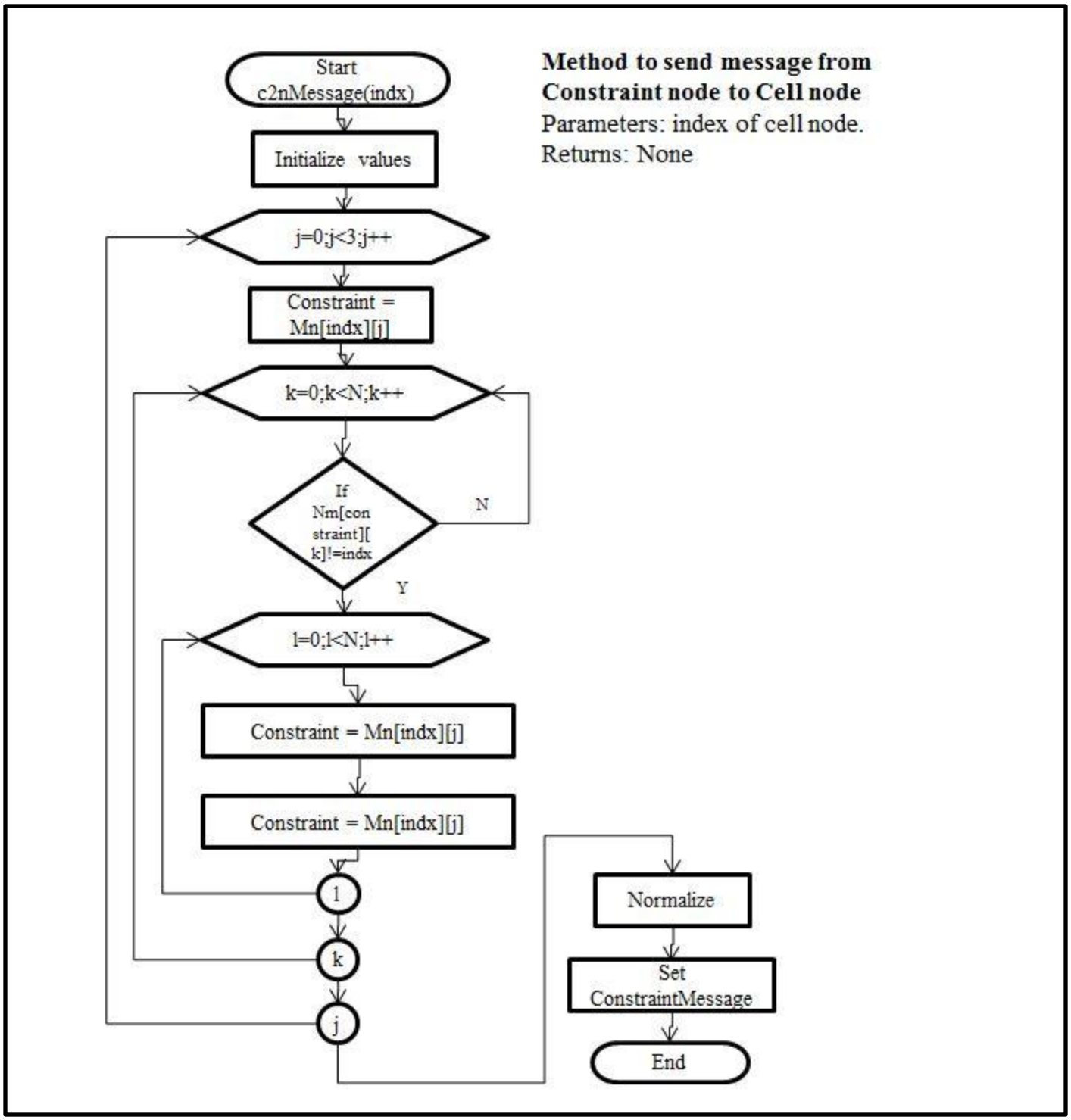


Figure 17: Flowchart of Constraint node to Cell node message

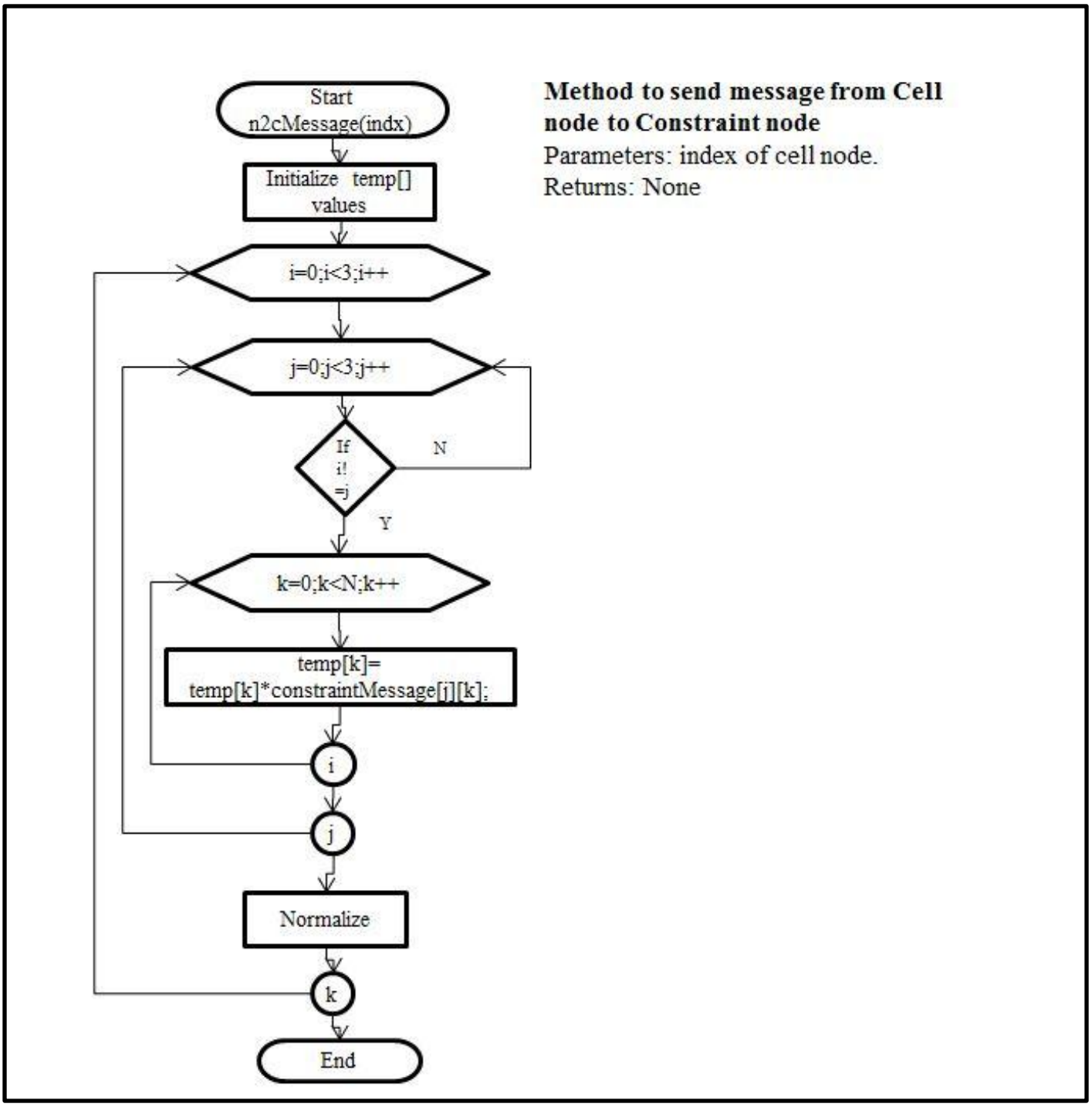


Figure 18: Flowchart of Node cell to Constraint cell message

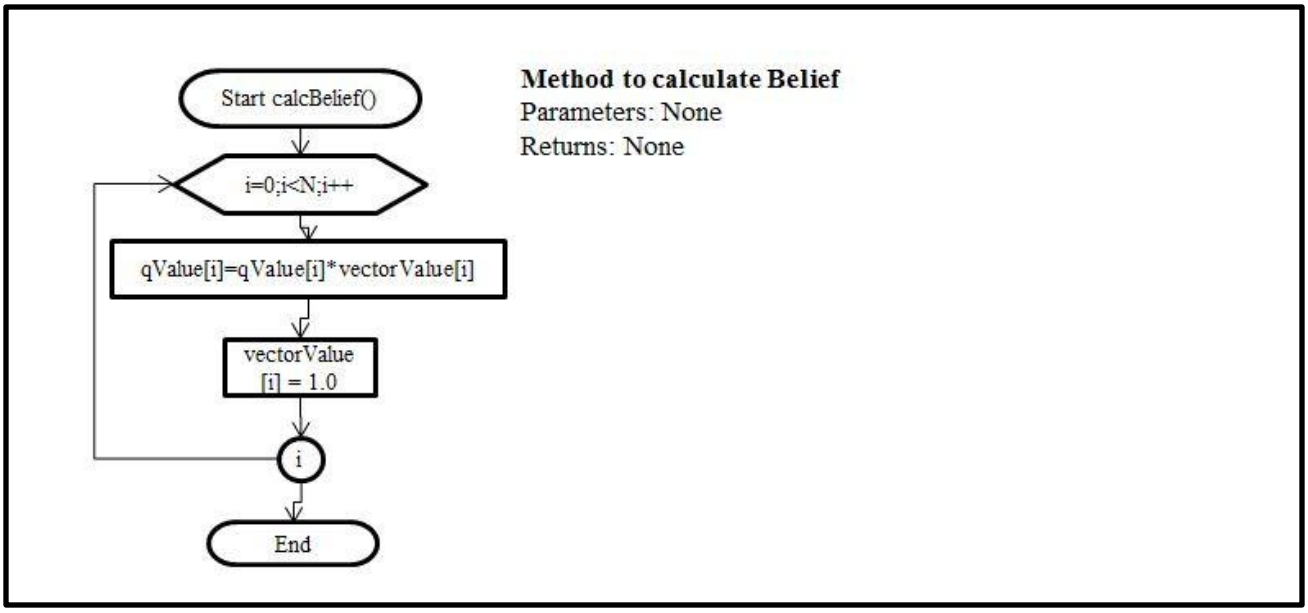


Figure 19: Flowchart of Belief calculation

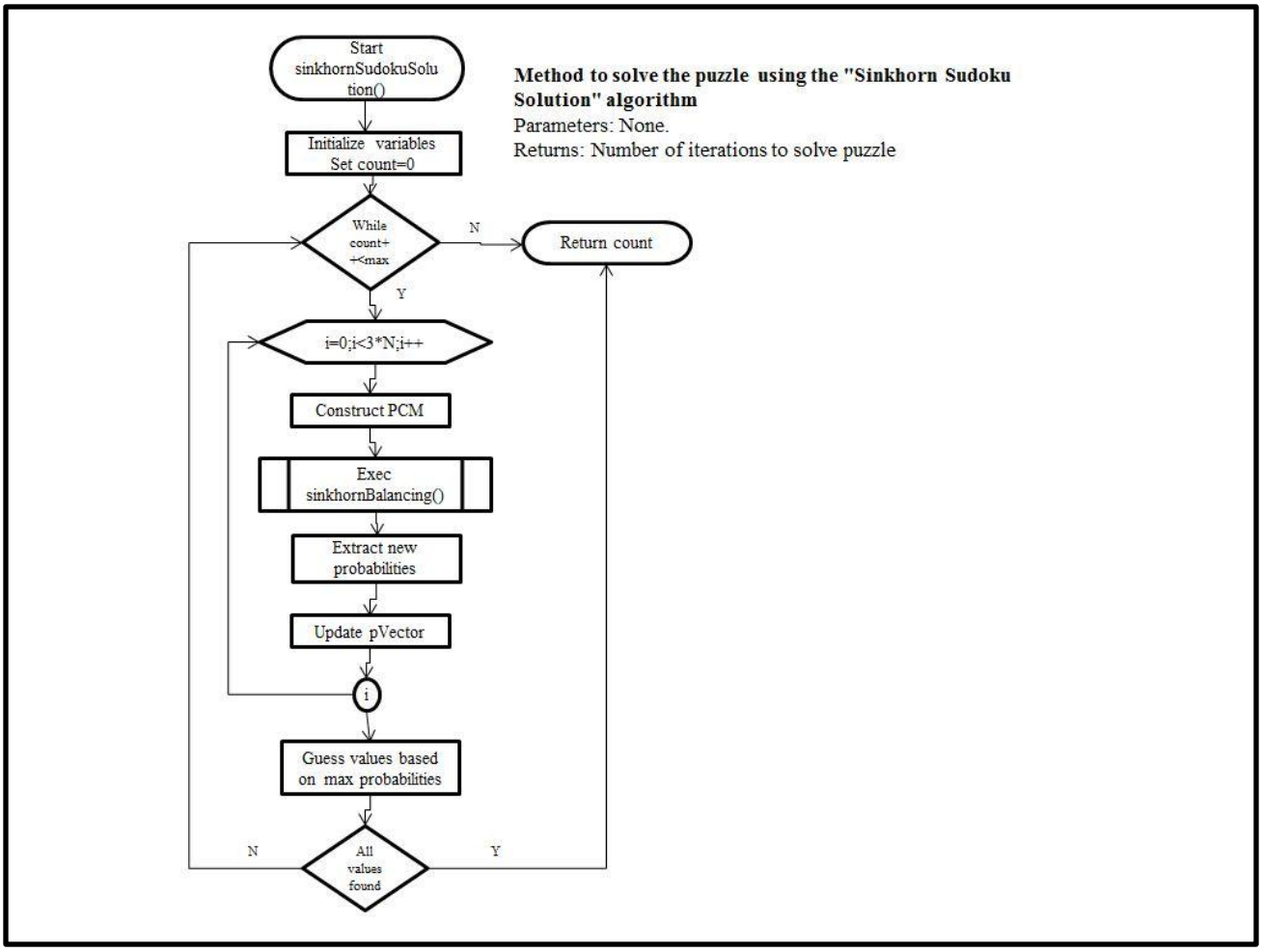


Figure 20: Sinkhorn Sudoku Solution Flowchart

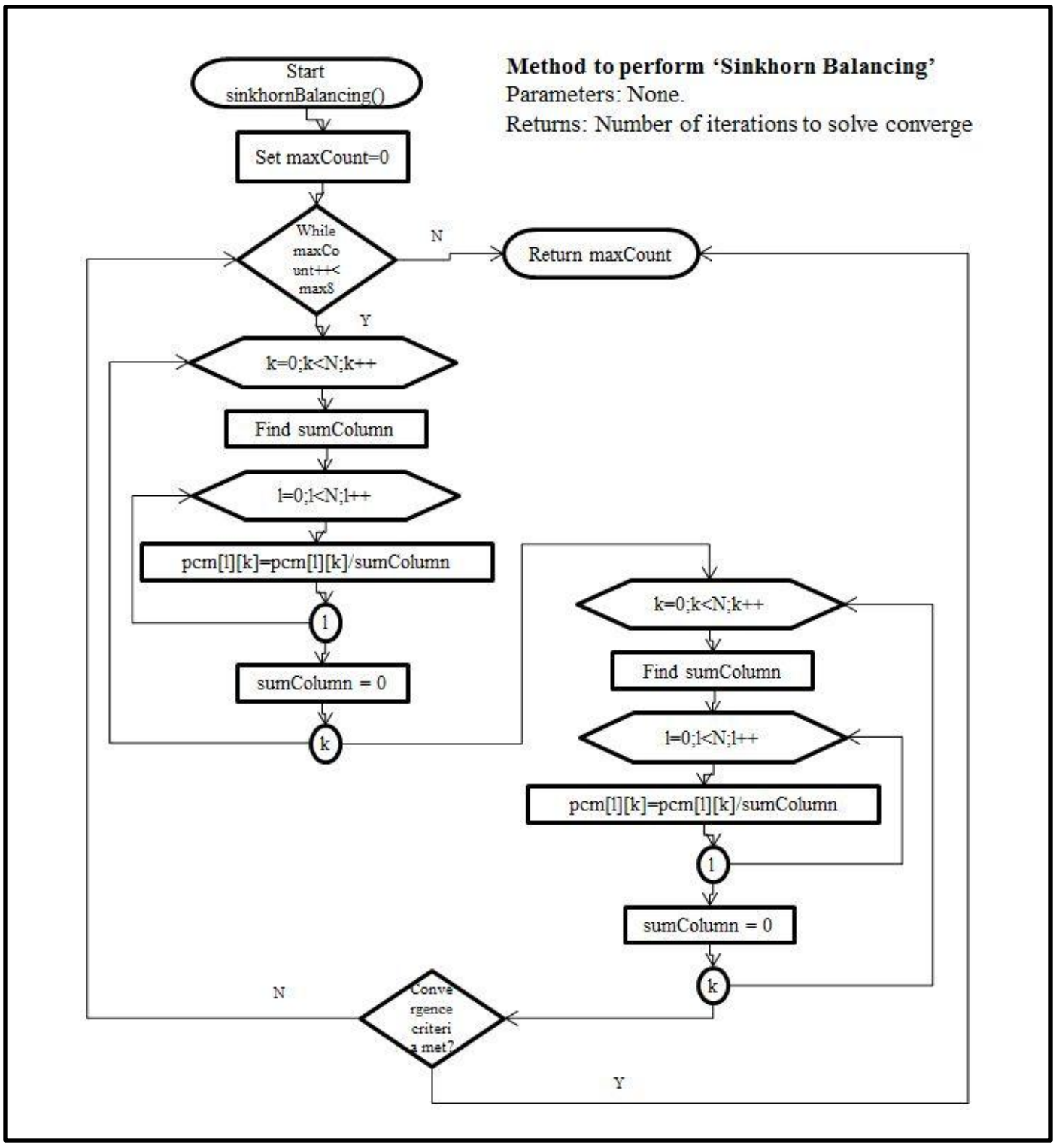


Figure 21: Sinkhorn balancing Flowchart

6.3 Sequential and Randomized Message Passing Algorithms

6.3.1 Running Time [1]

In the original sum product and max product message passing algorithms [2] the message from the constraint nodes to the cell nodes is defined by Eq. 6.1.

$$\mathbf{r_{m,\,n}(x)} = \sum_{\substack{\{n' \in Nmn\} \\ n=x,\ n'=xn' \text{ all unique}}} \prod_{l \in \mathbf{Nm,n}} \mathbf{q_{l,\,m}(x_l))} \qquad \text{Eq. 6.1}$$

Hence, for each message, there are N! assignment of values to the cell nodes in that constraint and computing the message takes O (N! N^2) time. And for the $3N^2$ messages, the time complexity is O (N! N^4).

However, based on Eq. 5.1 provided in [1], we get a reduced time complexity of O (N^4) to compute the message.

6.3.2 Implementation Details

Primary class: **SudokuPGM**

Primary Methods and their descriptions:

- **public int sequential () -** Method to solve the puzzle using the 'Sequential MP' algorithm. In this approach, the missing nodes are traversed using a sequential row wise scan and messages are passed to and from connected constraint nodes.
- **public int randomSolution () -** Method to solve the puzzle using the 'Randomized MP' algorithm. In this approach, the missing nodes are visited in a random order during each pass and messages are passed to and from connected constraint nodes to update the probability vector.

- **public void c2nMessage (int index)-** Method to pass message from connected constraint nodes to a cell node.
- **public void n2cMessage (int index) -** Method to pass message from a cell node to the constraint nodes connected to it.
- **public void calcBelief ()** - Method to calculate the cluster/node belief.

Some of the key points to note with respect to the implementation are:

1. The implementation of the message passing algorithm employs Eq. 5.1 for constraint to cell messages.
2. It was noted that in cases where the MP algorithms arrived at a solution, it took less than 50 iterations to do so. However, the Sinkhorn Sudoku Solution algorithm had a more diverse convergence rate. Thus, in order to maintain a consistent maximum iteration limit, we have chosen the value of 1000. This value is optimal as it does not adversely affect the running time of the MP algorithm in situations where convergence does not happen and is also able to accommodate most of the SSS test cases.
3. Initial probabilities for missing cell nodes are uniformly distributed over all 'N' possibilities.

6.4 Sinkhorn Sudoku Solution Algorithm

6.4.1 Running Time

The Sinkhorn balancing function performs N row normalizations and N column normalizations for each of the rows and columns in the 'N x N' probability constraint matrix. This function is then repeated for each of the 3N constraints in the given puzzle. Thus, the time complexity of the algorithm is O (N^2).

6.4.2 Implementation Details

Primary class: **SudokuPGM**

Primary Methods and their descriptions:

- **public int sinkhornSudokuSolution () -** Method to solve the puzzle using the 'Sinkhorn Sudoku Solution' algorithm
- **public int sinkhornBalancing () -** Method to perform Sinkhorn Balancing.

Some of the key points to note with respect to the implementation are:

1. Initial probabilities for missing cell nodes are uniformly distributed over all N possibilities.
2. The maximum number of iterations was set to 1000. This is to maintain consistency with the Message Passing algorithms; to allow a fair comparison and understand their relative performance
3. The tolerance value for convergence of Sinkhorn balancing was set to 0.001; this is an arbitrarily selected value between 0 and 1, based on trial and error. If the value is too high, then the Sinkhorn balancing algorithm of Figure 12 will converge too soon and if it is too low then it will never converge. Thus, we have chosen an optimal value based on our observations on how this value affects the running time and accuracy of the results.
4. The maximum number of iterations for Sinkhorn balancing was set to 1000 in order to complement the tolerance value. If the value is too low then the tolerance limit function will never be satisfied and if it is too high the running time of the program increases.

6.5 Understanding the algorithms using an example

6.5.1 Message Passing Algorithms

The two message passing algorithms differ only in the order of visiting missing cells during belief propagation. We demonstrate the performance of both algorithms using an example.

Example 6.1: Consider the '9 x 9' Sudoku puzzle given in Figure 22.

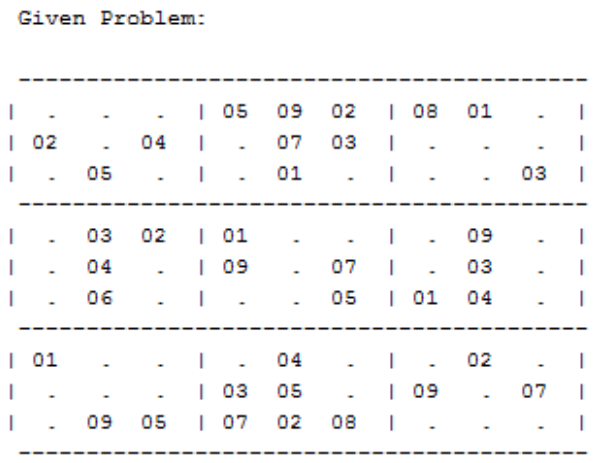


Figure 22: Problem instance for Example 6.1

Based on the problem in Figure 22, we generate the initial probability vector distribution for the cell nodes. As mentioned earlier, in the case of missing cell nodes, the probabilities are equally distributed over all nine values. The initial probability vector distribution for the first 27 cell nodes is shown in Figure 23.

Probability vector distribution:

0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 23: Initial probability vector distribution

In the first iteration, we visit each of the missing cell nodes and reevaluate their probability vectors. This is done based on the information known about the other cell nodes that they are associated with through a constraint. If we get strong evidence regarding the values of other cells in the constraint, then we can evaluate the value of a missing cell. As a result of this process, some of the missing cell nodes get populated and in turn contribute towards evaluating other missing cell values in successive iterations.

```
Matrix value after 1 iteration:
-----------------------------------------
|  .  07   .  | 05  09  02  | 08  01   .  |
| 02   .  04  |  .  07  03  |  .   .   .  |
|  .  05   .  |  .  01   .  |  .   .  03  |
-----------------------------------------
|  .  03  02  | 01   .   .  |  .  09   .  |
|  .  04   .  | 09   .  07  |  .  03   .  |
|  .  06   .  |  .   .  05  | 01  04   .  |
-----------------------------------------
| 01  08   .  | 06  04  09  |  .  02  05  |
|  .  02  06  | 03  05  01  | 09  08  07  |
|  .  09  05  | 07  02  08  |  .  06   .  |
-----------------------------------------

Probability vector distribution:
0.00    0.00    0.33    0.00    0.00    0.33    0.33    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
0.00    0.00    0.50    0.00    0.00    0.50    0.00    0.00    0.00
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.75    0.00    0.25    0.00    0.00    0.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.50    0.00    0.00    0.00    0.00    0.00    0.00    0.50    0.00
0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.67    0.00    0.33    0.00
0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.80    0.20    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.20    0.80    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.02    0.00    0.00    0.00    0.97
0.00    0.00    0.00    0.00    0.00    0.13    0.00    0.29    0.58
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.23    0.00    0.45    0.32
0.00    0.00    0.00    0.80    0.00    0.06    0.00    0.14    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.17    0.00    0.83    0.00    0.00    0.00
0.00    0.49    0.00    0.02    0.00    0.00    0.49    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.01    0.99    0.00    0.00
0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00
```

Figure 24: Updated values after first iteration

The new matrix and probability vectors of the first 27 cell nodes are shown in Figure 24. Even though there are no new values in these cells, you can see that the first iteration filled values into other cells which in turn have affected the probability vectors of these cells.

In this manner, we get additional information about the cells' values, until at one point we either arrive at a solution or the algorithm fails to converge.

```
Matrix value after 2 iteration:
 ----------------------------------------
|  .  07  03  | 05  09  02  | 08  01   .  |
| 02  01  04  | 08  07  03  |  .  05   .  |
|  .  05   .  | 04  01  06  |  .  07  03  |
 ----------------------------------------
|  .  03  02  | 01   .  04  |  .  09   .  |
|  .  04   .  | 09   .  07  |  .  03   .  |
|  .  06   .  | 02   .  05  | 01  04  08  |
 ----------------------------------------
| 01  08  07  | 06  04  09  | 03  02  05  |
| 04  02  06  | 03  05  01  | 09  08  07  |
| 03  09  05  | 07  02  08  | 04  06  01  |
 ----------------------------------------

Matrix value after 3 iteration:
 ----------------------------------------
| 06  07  03  | 05  09  02  | 08  01  04  |
| 02  01  04  | 08  07  03  | 06  05  09  |
|  .  05   .  | 04  01  06  | 02  07  03  |
 ----------------------------------------
|  .  03  02  | 01   .  04  |  .  09  06  |
|  .  04   .  | 09   .  07  | 05  03  02  |
|  .  06  09  | 02  03  05  | 01  04  08  |
 ----------------------------------------
| 01  08  07  | 06  04  09  | 03  02  05  |
| 04  02  06  | 03  05  01  | 09  08  07  |
| 03  09  05  | 07  02  08  | 04  06  01  |
 ----------------------------------------

Matrix value after 4 iteration:
 ----------------------------------------
| 06  07  03  | 05  09  02  | 08  01  04  |
| 02  01  04  | 08  07  03  | 06  05  09  |
|  .  05  08  | 04  01  06  | 02  07  03  |
 ----------------------------------------
|  .  03  02  | 01  08  04  | 07  09  06  |
| 08  04  01  | 09  06  07  | 05  03  02  |
| 07  06  09  | 02  03  05  | 01  04  08  |
 ----------------------------------------
| 01  08  07  | 06  04  09  | 03  02  05  |
| 04  02  06  | 03  05  01  | 09  08  07  |
| 03  09  05  | 07  02  08  | 04  06  01  |
 ----------------------------------------
```

Figure 25: Matrix contents after each intermediate iteration

As shown in Figure 25, after every loop we are able to fill additional cells in the puzzle. This is done based on the messages that the cells receive from the constraints and vice versa.

```
Solved in 5 iterations.

Final Solution:

 -----------------------------------------
| 06  07  03  | 05  09  02  | 08  01  04  |
| 02  01  04  | 08  07  03  | 06  05  09  |
| 09  05  08  | 04  01  06  | 02  07  03  |
 -----------------------------------------
| 05  03  02  | 01  08  04  | 07  09  06  |
| 08  04  01  | 09  06  07  | 05  03  02  |
| 07  06  09  | 02  03  05  | 01  04  08  |
 -----------------------------------------
| 01  08  07  | 06  04  09  | 03  02  05  |
| 04  02  06  | 03  05  01  | 09  08  07  |
| 03  09  05  | 07  02  08  | 04  06  01  |
 -----------------------------------------

 Probability vector distribution:
0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
```

Figure 26: Values after final iteration

This puzzle was solved in 5 iterations and from Figure 26 we see that the resulting probability vectors take the form of a Kronecker Delta function.

6.5.2 Sinkhorn Balancing algorithm

Recall that the Sinkhorn Sudoku Solution algorithm was given in Figure 13. We will demonstrate how the algorithm works using Example 6.2.

Example 6.2: Consider the ‘9 x 9’ Sudoku puzzle given in Figure 22.

The Sinkhorn Sudoku solution algorithm also generates an initial probability vector distribution for all the cell nodes based on the information given. In this distribution, the probabilities are distributed equally for the missing cells without considering the influence of

other value nodes. The initial probability vector distribution for the first row of the puzzle is shown in Figure 27.

```
Probability vector distribution:
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
```

Figure 27: Initial probability distribution

These vectors, when grouped together, form the probability constraint matrix for constraint ‘C1’ as shown in Figure 28.

```
Probability Constraint Matrix for C1 before Sinkhorn Balancing:
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11    0.11
```

Figure 28: PCM before Sinkhorn Balancing

Now we perform Sinkhorn Balancing on this matrix. This is essentially nothing but performing successive column and row normalizations till some convergence criteria is met. The resulting matrix is a doubly stochastic matrix as seen in Figure 29.

```
Probability Constraint Matrix for C1 after Sinkhorn Balancing:
0.00    0.00    0.25    0.25    0.00    0.25    0.25    0.00    0.00
0.00    0.00    0.25    0.25    0.00    0.25    0.25    0.00    0.00
0.00    0.00    0.25    0.25    0.00    0.25    0.25    0.00    0.00
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.25    0.25    0.00    0.25    0.25    0.00    0.00
```

Figure 29: PCM after Sinkhorn Balancing

We extract the new probability vectors from this matrix and then repeat the process for all the other constraints. At the end of the process, we guess the most probable value for each cell by taking the index of the largest probability value in its probability vector. Based on these assumptions we fill out the Sudoku grid and validate the result. If the solution is valid we terminate and update the probability vectors to take the form of Kronecker Delta functions, else we repeat the process for all the constraints.

```
Solved in 3 iterations.

Final Solution:

 -----------------------------------------
| 06  07  03  | 05  09  02  | 08  01  04  |
| 02  01  04  | 08  07  03  | 06  05  09  |
| 09  05  08  | 04  01  06  | 02  07  03  |
 -----------------------------------------
| 05  03  02  | 01  08  04  | 07  09  06  |
| 08  04  01  | 09  06  07  | 05  03  02  |
| 07  06  09  | 02  03  05  | 01  04  08  |
 -----------------------------------------
| 01  08  07  | 06  04  09  | 03  02  05  |
| 04  02  06  | 03  05  01  | 09  08  07  |
| 03  09  05  | 07  02  08  | 04  06  01  |
 -----------------------------------------

 Probability vector distribution:
0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00
0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00
```

Figure 30: Final result of SSS algorithm

The end result of this process either gives us a solved puzzle, as seen in Figure 30, or no result. In the next chapter, we provide an overview of the data used for testing the implementations and a detailed comparison of the results.

CHAPTER 7

Test Data and Test Results

7.1 Collection of test data

The test data for the algorithms were collected from various websites [15] [16] [17]. The puzzles have been classified by the respective websites into various difficulty levels. Though the specific criteria for classification were not provided, in general it is based on a combination of number of prefilled cell nodes and the initial distribution of those nodes.

The difficulty levels range from ‘Very Easy’ to ‘Impossible’, but for our analysis we have restricted ourselves to ‘Easy’, ‘Medium’ and ‘Hard’ puzzles only. Additionally, in order to avoid ambiguous results, we have examined only those puzzles that have unique solutions. The input data we use are a set of comma-delimited text files, where each file contains a Sudoku puzzle.

Results presented in this chapter are based on tests that have been run on 15 ‘4 x 4’ puzzles, 150 ‘9 x 9’ puzzles, 100 ’16 x 16’ puzzles and 30 ‘25 x 25’ puzzles. Additional tests results (that are not included here) have also contributed to the final analysis.

7.2 Comparison of the three algorithms

In all the algorithms, the initial probability distribution is set equally for all the missing nodes. We then follow several iterations of elimination to redistribute the probabilities, based on what we know about other cell nodes grouped in the same constraint. In order to avoid the adverse effects of biases due to large number of loops in the graph; and taking into consideration

the existing time complexity of the algorithms, we restrict the maximum number of iterations to a constant value (1000) in all three algorithms.

We note that the analysis we describe in the next section is based on a combination of the data presented and additional tests that were run.

7.2.1 Sequential MP Algorithm

Table 3: Results of Sequential message passing algorithm

Puzzle size	Difficulty level	Number of puzzles	Number of complete/correct solutions	Percentage of accurate results
4x4	Easy	15	15	100%
9x9	Easy	50	2	4%
	Medium	50	3	6%
	Hard	50	7	14%
16x16	Easy	50	0	0%
	Medium and Hard	50	1	2%
25x25	Easy	15	0	0%
	Medium and Hard	15	0	0%

The Sequential Message Passing algorithm always visits the missing nodes in the same order. It traverses the matrix one row at a time from left to right. When a missing node is identified, we check for messages from the constraint nodes that it is associated with. These messages contain information from other cell nodes involved in that particular constraint; regarding the values they hold or can hold. If any of these cells' value is already known, then an outgoing message from that cell will simply be the fixed probability vector, irrespective of the incoming messages. Also, this vector will always be a Kronecker delta function [1] [2]. This then avoids any loopy belief propagation in this particular loop.

Thus, it is clear that the distribution of value cell nodes greatly influences the degree of loopy belief propagation. If we were to visit a missing node that is associated with many value

nodes initially, then there is a greater possibility of resolving its value. It can then, in turn, assist in resolving the value of other missing nodes that it is associated with. Hence, the order in which we visit missing cell nodes is also of great importance and as expected the results that we see in Table 3 reinforce this theory. Most of the puzzles were caught in loops and never converged to a solution.

7.2.2 Randomized MP Algorithm

Table 4: Results of Randomized message passing algorithm

Puzzle size	Difficulty level	Number of puzzles	Number of complete/correct solutions (Average of 50 runs)	Percentage of accurate results
4x4	Easy	15	15	100%
9x9	Easy	50	40	80%
	Medium	50	38	76%
	Hard	50	37	74%
16x16	Easy	50	15	30%
	Medium and Hard	50	7	14%
25x25	Easy	15	1	6.66%
	Medium and Hard	15	0	0%

The Randomized Message Passing algorithm is aimed at overcoming the limitations of the Sequential algorithm. Since we saw that visiting missing cell nodes in the same order lead to biases due to loopy belief propagation, in this algorithm we visit the nodes in random order. In this manner we increase the chances of resolving missing cells associated with favorable value nodes first and they in turn can help identify other missing node values.

Table 4 summarizes the results of running the Randomized MP algorithm multiple times. The reason for multiple runs is because this is still a randomized approach. It may or may not converge. If a problem has ‘m’ missing cell nodes then these nodes can be visited in ‘m!’ ways.

It is to be noted that the fixed order in which the Sequential MP algorithm visits the missing cells is one of these 'm!' possibilities.

Thus, the Randomized MP algorithm too suffers from loopy belief propagation but has better chances of finding a solution due to its arbitrary approach. This is reflected in Table 4; we see a much better accuracy though the results are still not perfect.

7.2.3 Sinkhorn Balancing Algorithm

Table 5: Results of Sinkhorn balancing algorithm

Puzzle size	Difficulty level	Number of puzzles	Number of complete/correct solutions	Percentage of accurate results
4x4	Easy	15	15	100%
9x9	Easy	50	50	100%
	Medium	50	49	98%
	Hard	50	41	82%
16x16	Easy	50	6	12%
	Medium and Hard	50	1	2%
25x25	Easy	15	0	0%
	Medium and Hard	15	0	0%

The Sinkhorn balancing approach is based on grouping belief vectors that are associated within a particular constraint. As was the case with the MP algorithms (and unlike the traditional solutions that perform logical elimination over a large search space), the Sinkhorn algorithm too makes use of a probabilistic representation and approach to solve the discrete constraint satisfaction problem. This greatly reduces the potential solution space to be considered.

However, as mentioned in [3], the Sinkhorn method has an added advantage over belief propagation as it is not affected by graph cycles and loopy belief propagation. It also has a lower time complexity. While the MP algorithms seem to produce good results only for the easier

problems, the Sinkhorn balancing approach is able to solve most of the problem except the extremely difficult ones.

While the Sinkhorn algorithm almost always converges to a solution, the rate of convergence is greatly influenced by multiple factors such as the puzzle size, difficulty level and choice of tolerance function during Sinkhorn balancing. The higher the value of any of these factors, the more number of iterations it takes for the algorithm to converge. The results presented in Table 5 are not completely bias-free as the maximum number of iterations was capped at 1000 to keep it consistent with the MP algorithms. While this limit was enough to accommodate most of the '9 x 9' puzzles, it was not sufficient for the larger puzzles. However, when the maximum iteration count was increased to 100000, an additional 18 puzzles in the '16 x 16 Easy' category, 3 puzzles in the '16 x 16 Medium and Hard' category and 1 puzzle in the '25 x 25 Easy' category were solved.

Similarly, for the experiment documented in Table 5, we used a tolerance function of 0.001 and a maximum iteration count of 1000 during Sinkhorn balancing. As described in Figure 12, the tolerance function (ϵ) acts as the convergence criterion for the Sinkhorn Balancing algorithm. Varying this value also affects the convergence rate of the Sinkhorn Sudoku Solution algorithm. The number of iterations to solve the puzzle is directly proportional to the tolerance function and inversely proportional to the number of Sinkhorn iterations.

Based on the results obtained, we have summarized the relative performance of the three algorithms in Figure 31.

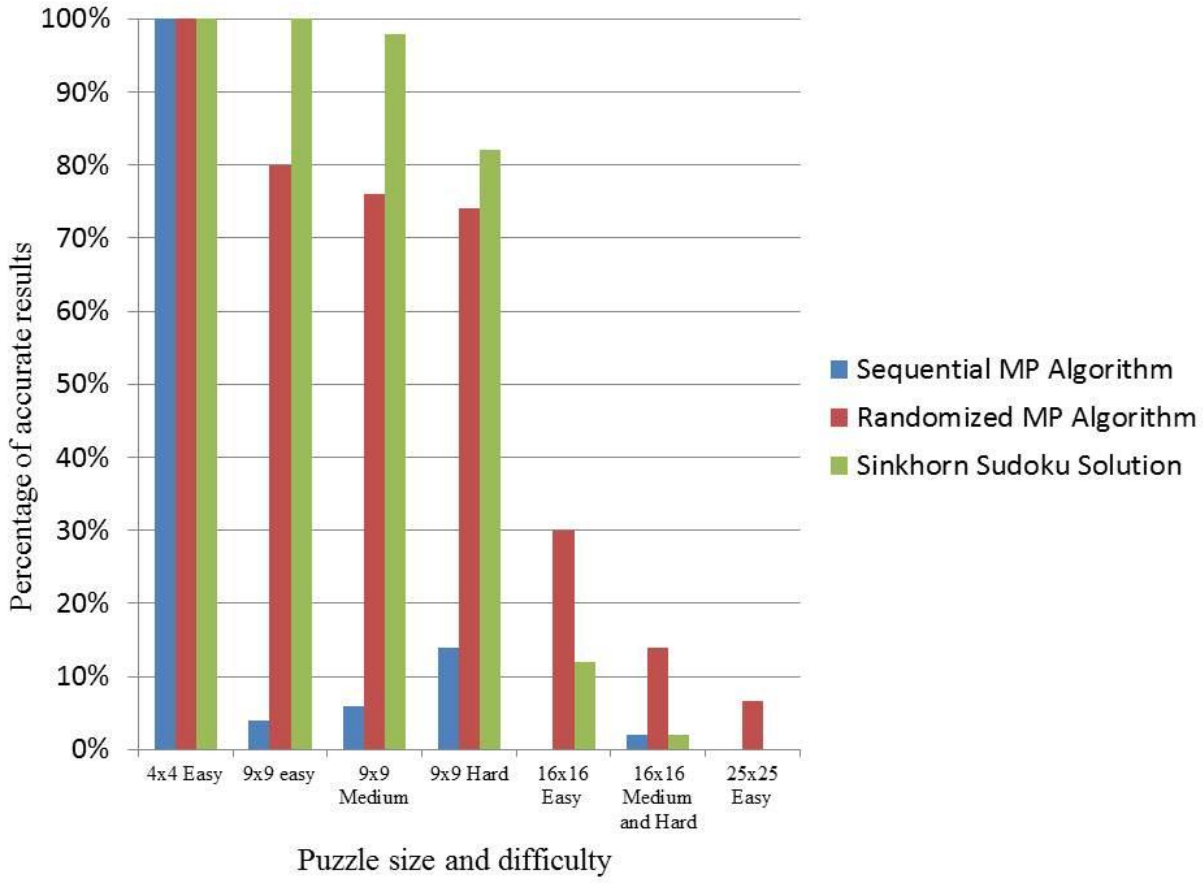


Figure 31: Comparison of 3 algorithms

Based on Figure 31, we see that for all puzzles the Sequential MP algorithm fairs quite poorly, whereas the Randomized MP shows much better performance. The algorithm based on Sinkhorn Balancing shows the best performance for all difficulty levels within the standard '9 x 9' puzzle set. Also, considering additional results not documented here, the SSS algorithm was found to produce valid solutions more consistently than the other two algorithms.

7.3 Convergence rate of the three algorithms

The general trend observed in the convergence rates of the three algorithms shows that, in the case of MP algorithms, if a solution is obtainable then, the number of iterations to reach that solution is not greatly affected by the size or difficulty level of the problem. On the other hand, the Sinkhorn algorithm's convergence rate is quite closely related to the nature of the problem. These observations are depicted in the figures that follow. Figures 33 and 35 compare the effect on convergence rate due to the difficulty level and Figure 37 highlights the impact on convergence rate when the size of the puzzle is increased. In all three figures, the horizontal axis shows the number of iterations and vertical axis shows number of unsatisfied constraint.

a) Standard '9 x 9' puzzle

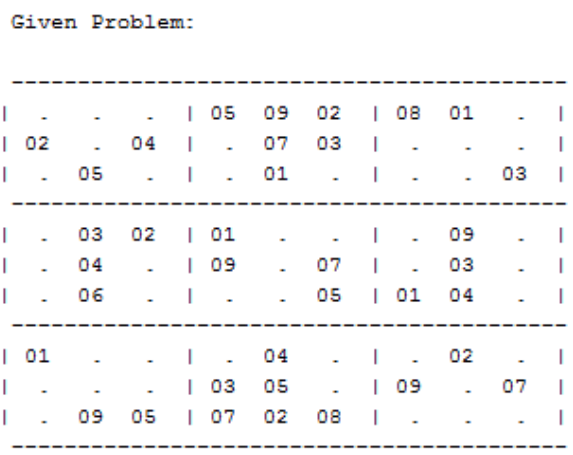


Figure 32: Given puzzle - '9 x 9' Easy

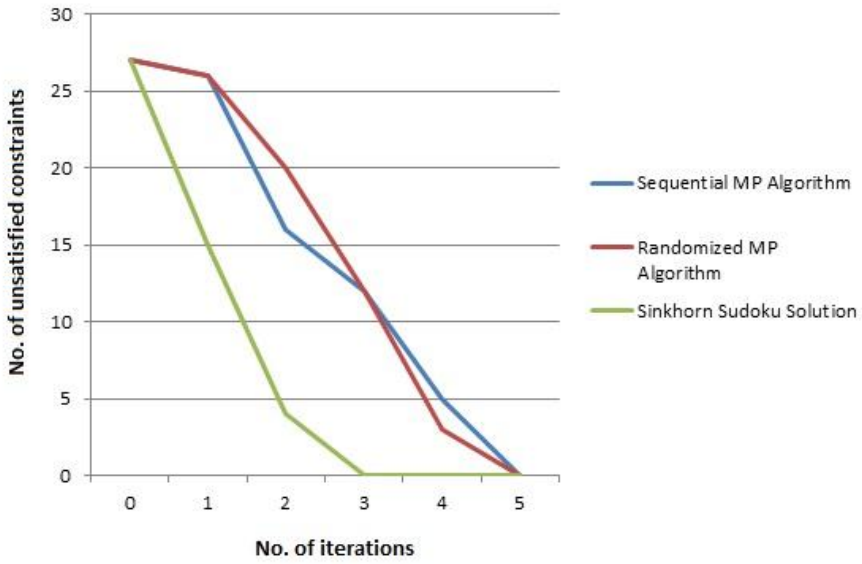


Figure 33: Convergence rate for given puzzle.

We see from Figure 33 that the Sinkhorn Sudoku Solution algorithm has a better convergence rate for a standard '9 x 9' puzzle. Also, while both MP algorithms take the same number of iterations to solve this particular problem instance, the rate at which the constraints are satisfied varies.

b) Hard '9 x 9' puzzle

```
Given Problem:

 -----------------------------------------
|  .   .   .  | 03  07   .  |  .   .  05  |
| 08   .   .  |  .  05  01  | 03   .   .  |
|  .  05   .  |  .   .   .  |  .  06  02  |
 -----------------------------------------
| 09  04   .  |  .   .   .  |  .   .   .  |
|  .   .   .  | 07   .  08  |  .   .   .  |
|  .   .   .  |  .   .   .  |  .  05  04  |
 -----------------------------------------
| 01  06   .  |  .   .   .  |  .  04   .  |
|  .   .  03  | 01  02   .  |  .   .  07  |
| 05   .   .  |  .  06  04  |  .   .   .  |
 -----------------------------------------
```

Figure 34: Given puzzle - '9 x 9' Hard

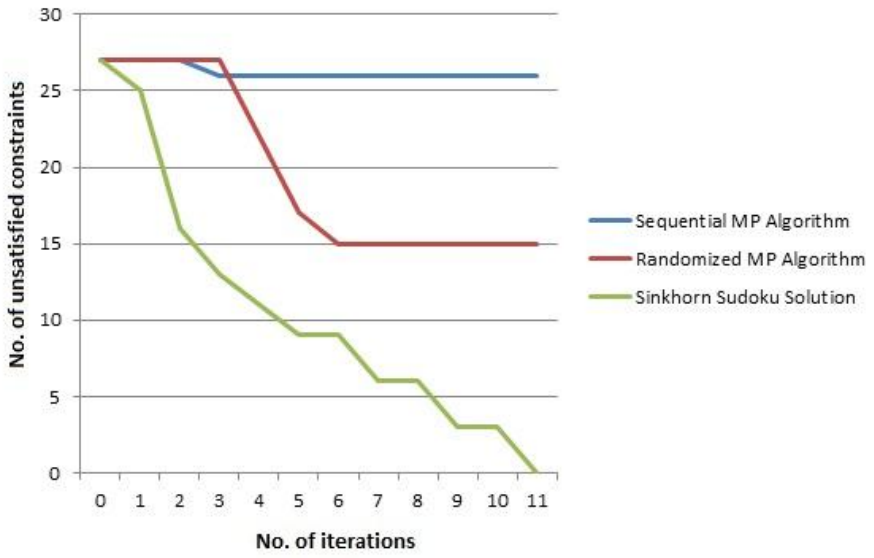


Figure 35: Convergence rate for given puzzle.

We see from Figure 35 that for a hard '9 x 9' puzzle, the Sinkhorn Sudoku Solution algorithm is still able to arrive at a solution, even though the convergence rate is higher than that of a standard '9 x 9' puzzle. The MP algorithms on the other hand fail to arrive at a valid solution due to the biases introduced by loopy belief propagation in the bipartite graph.

c) Standard '16 x 16' puzzle

Given Problem:

.	09	.	.	.	.	.	.	.	13	07	.	03	.	.	.
.	.	02	.	11	06	03	04	.	09	.	.	.	08	10	15
.	16	.	10	.	.	12	.	.	04	.	03	06	.	13	.
.	.	06	08	02	16	.	.	.	10	05	.	.	01	07	09
12	10	05	.	.	.	.	.	.	.	04	06	15	.	03	14
.	01	.	.	.	04	11	.	16	.	.	14	.	05	06	.
.	.	08	14	05	15	.	03	.	.	.	.	.	12	.	.
15	.	.	06	10	09	.	08	.	.	12	.	01	11	16	.
.	.	.	.	.	.	.	.	02	.	.	16	13	06	01	.
.	05	.	16	06	.	08	.	.	.	.	01	.	.	11	.
.	15	11	.	01	.	.	.	.	.	.	.	09	14	12	07
.	.	01	.	03	10	15	07	09	.	.	11	.	.	.	.
04	11	.	05	.	.	.	.	01	14	03	10	02	13	15	.
13	.	.	.	.	.	.	02	.	11	.	.	12	07	.	06
.	.	.	.	07	01	.	11	.	.	.	13	.	03	04	08
08	.	03	12	.	14	.	13	04	05	06	07	11	.	.	.

Figure 36: Given puzzle - '16 x 16' Easy

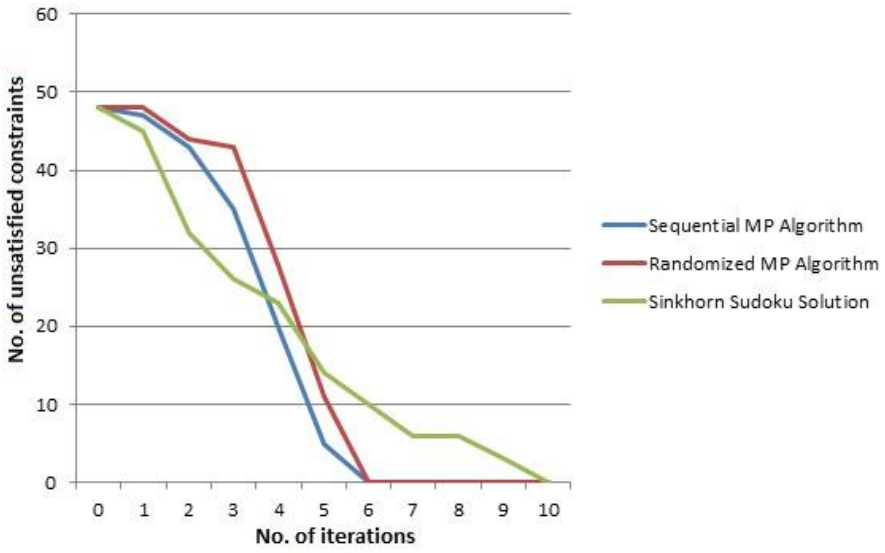


Figure 37: Convergence rate for given puzzle

Figure 37 shows the convergence rates of the three algorithms when given a standard ’16 x 16’ puzzle as input. Keeping the difficulty level the same, if we were to increase the puzzle size, we see that the Sinkhorn Sudoku Solution takes longer to converge. However, it is still able to arrive at a valid solution. The MP algorithms on the other hand take a fewer number of iteration to solve the puzzle.

Thus, based on Figures 33, 35 and 37 we see that MP algorithms are affected by the difficulty level of the puzzle. However, in cases where they do converge the size of the puzzle does not affect the number of iterations to find a solution. On the other hand, the algorithm based on Sinkhorn Balancing is far more consistent and is able to find a valid solution in most cases. However, its convergence rate is affected by both the size and difficulty level of the puzzle.

Figures 38, 39 and 40 depict the convergence of the Sequential MP, Randomized MP and SSS algorithms for 15 puzzles from the '9 x 9 Easy' category. The horizontal axis shows the number of iterations and the vertical axis depicts the number of unsatisfied constraints. All three algorithms had the same maximum iterations limit of 100 and the Sinkhorn tolerance limit was set at 0.001.

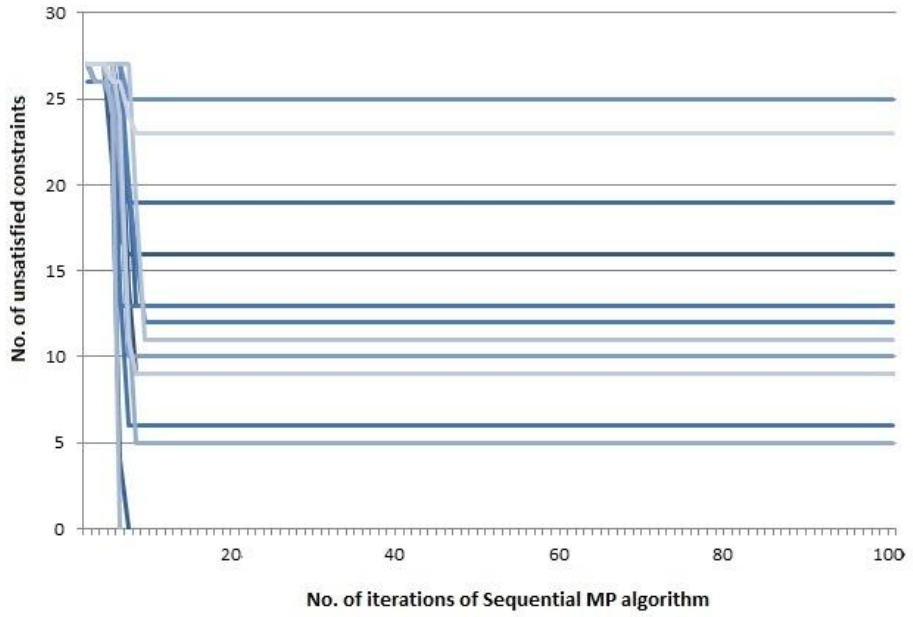


Figure 38: Convergence of Sequential MP algorithm

The Sequential Message Passing algorithm fails to arrive at a valid solution for most puzzles, as shown in Figure 38. For the few instances where the puzzle is solved, the number of iterations it takes does not vary greatly.

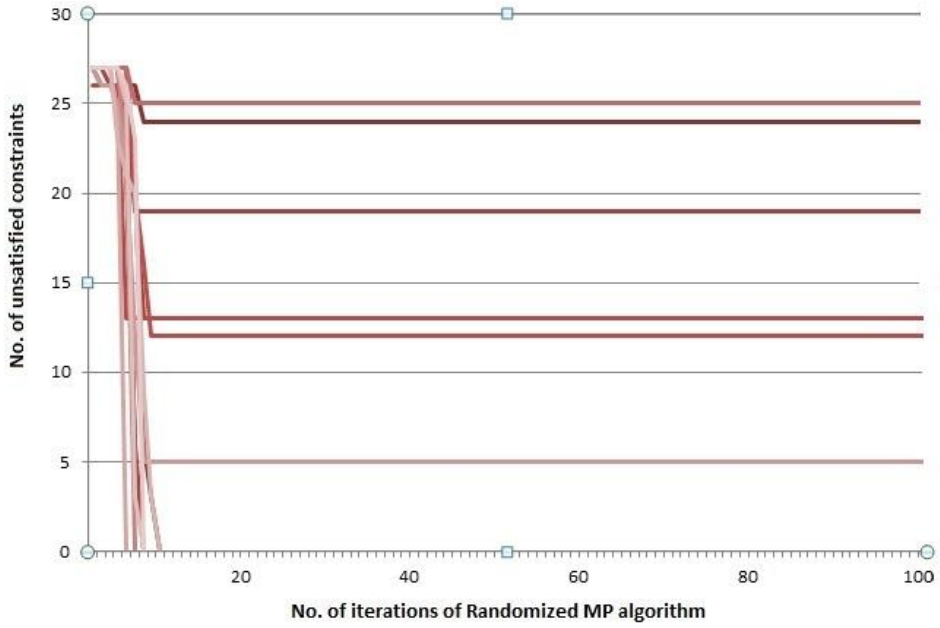


Figure 39: Convergence of Randomized MP algorithm

The Randomized Message Passing algorithm shows much better performance than the Sequential MP algorithm, as shown in Figure 39. However, the convergence rate is consistent with the other Message Passing algorithm.

In Figure 40, we have shown the convergence rate of the Sinkhorn based solution. As can be observed, it shows the best performance in terms of arriving at a valid solution. But unlike the MP algorithms, the SSS algorithm has a bigger and more diverse convergence rate.

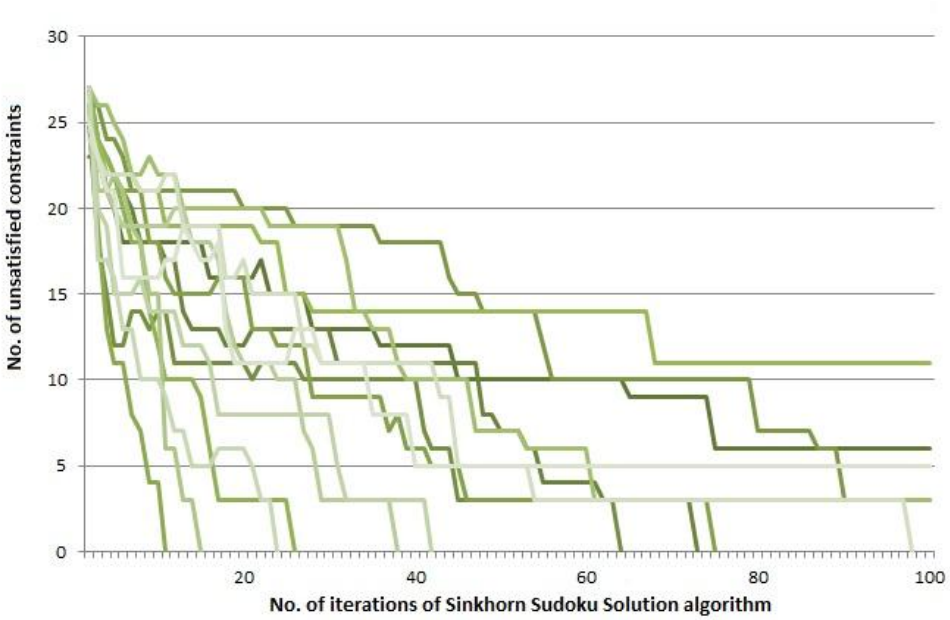


Figure 40: Convergence of SSS algorithm

Having seen the various test results, in the final chapter we summarize our findings and define the scope of future work.

CHAPTER 8

Conclusion and Future work

Sudoku, being a NP-Complete problem, there is no optimal technique that runs in polynomial time to solve these puzzles. However, the search space can be largely reduced by avoiding special tricks and instead taking an algorithm that performs general inference. The three algorithms compared in this work can be applied to any similar constraint satisfaction problem.

Based on the results discussed in this work, the next question that arises is regarding the best choice of algorithm among these three. This decision is largely dependent on the problem at hand. While the MP algorithm, especially the Randomized MP algorithm, might appear more effective for larger puzzles, it is possible that it was influenced by the test data set used and the fact that it is an averaged result of multiple runs of the algorithm. Similarly, the SSS algorithm was definitely influenced by the choice of maximum iterations limit. It is to be noted that the presence or absence of loops tend to greatly affect the results of the MP algorithms, whereas Sinkhorn balancing bypasses this issue and can hence be considered more consistent as compared to the MP algorithms.

In the case of the MP algorithms, addressing the influence of loopy belief propagation and biases is the primary focus area for future work. On the other hand, Sinkhorn balancing offers great scope in terms of addressing multiple constraint problems similar to the Sudoku puzzle. It is a straightforward approach that can be adapted to other areas of study such as the LDPC decoding problem, especially ones whose Tanner graphs have many cycles [3].

REFERENCES

[1] Khan, S; Jabbari, S; Jabbari, S; Ghanbarinejad, M, "Solving Sudoku Using Probabilistic Graphical Models", data last retrieved January 1, 2014, from http://webdocs.cs.ualberta.ca/~jabbaria/Documents/Solving%20Sudoku%20Using%20Probabilistic%20Graphical%20Models.pdf .

[2]Moon, T.K; Gunther, J.H, "Multiple Constraint Satisfaction by Belief Propagation: An Example Using Sudoku", data last retrieved January 1, 2014, from, http://www.di.ens.fr/~fbach/courses/fall2012/sudoku.pdf .

[3] Moon, T.K; Gunther, J.H; Kupin, J.J, "Sinkhorn Solves Sudoku", *IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 55, NO. 4*, April 2009.

[4] Probabilistic Graphical Models, Coursera & Stanford – Daphne Koller, data last retrieved January 1, 2014, from https://class.coursera.org/pgm-003 .

[5] Koller, D; Friedman, N, "Probabilistic Graphical Models", MIT press ISBN 978-0262013192.

[6] Goldberger, J., "Solving Sudoku Using Combined Message Passing Algorithms", *Technical Report TRBIU-ENG-2007-05-03,* Engineering School, Bar-Ilan Univ. (2007).

[7] Glossary of Graph Theory. (n.d.). In *Wikipedia*. Data last retrieved February 11, 2014, from http://en.wikipedia.org/wiki/Glossary_of_graph_theory .

[8] Factor Graphs. (n.d.). In *Wikipedia*. Data last retrieved February 11, 2014, from http://en.wikipedia.org/wiki/Factor_graph .

[9] Belief Propagation. (n.d.). In *Wikipedia*. Data last retrieved February 11, 2014, from http://en.wikipedia.org/wiki/Belief_propagation .

[10] Factor Graphs. Data last retrieved February 11, 2014, from https://capocaccia.ethz.ch/capo/wiki/2012/bssexperiments12 .

[11] Mathematics of Sudoku. (n.d.). In *Wikipedia*. Data last retrieved April 1, 2014, from http://en.wikipedia.org/wiki/Mathematics_of_Sudoku .

[12] Aaronson, L, "Sudoku science," IEEE *Spectrum*, Feb. 2006.

[13] McGuire, G; Tugemann, B; Civario, G, "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration" data last retrieved April 3, 2014, from, http://www.math.ie/McGuire_V2.pdf .

[14] Sudoku Puzzle. Data last retrieved March 26, 2014, from http://www.menneske.no/sudoku/.

[15] Sudoku-Download. Data last retrieved March 26, 2014, from http://www.sudoku-download.net/index.php .

[16] Web Sudoku. Data last retrieved March 10, 2014, from http://www.websudoku.com/ .

[17] Low-density parity-check code. (n.d.). In *Wikipedia*. Data last retrieved April 11, 2014, from http://en.wikipedia.org/wiki/Low-density_parity-check_code