5-13-2015

Using Probabilistic Graphical Models to Solve NP-complete Puzzle Problems

Fengjiao Wu
SJSU

Using Probabilistic Graphical Models to Solve NP-complete Puzzle Problems

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

Fengjiao Wu

April 2015

The Designated Project Committee Approves the Project Titled

Using Probabilistic Graphical Models to Solve NP-complete Puzzle Problems

by

Fengjiao Wu

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSE STATE UNIVERSITY

April 2015

Dr. Sami Khuri Department of Computer Science

Dr. Teng Moh Department of Computer Science

Dr. Thomas Austin Department of Computer Science

ABSTRACT

Using Probabilistic Graphical Models to Solve NP-complete Puzzle Problems

Probabilistic Graphical Models (PGMs) are commonly used in machine learning to solve problems stemming from medicine, meteorology, speech recognition, image processing, intelligent tutoring, gambling, games, and biology. PGMs are applicable for both directed graph and undirected graph. In this work, I focus on the undirected graphical model. The objective of this work is to study how PGMs can be applied to find solutions to two puzzle problems, sudoku and jigsaw puzzles. First, both puzzle problems are represented as undirected graphs, and then I map the relations of nodes to PGMs and Belief Propagation (BP). This work represents the puzzle grid as a bipartite graph, which contains disjoint sets S and C such that the graph's edges connect vertices in S only with vertices in C, and vice versa. S contains all the cells. C contains all constraint groups. Then, I apply the well-known sum-product message passing (MP) algorithm, which is also known as BP. In the jigsaw puzzle problem, I aim to reconstruct an image from a collection of square image patches. I use the neighborhood pairwise compatibility and local evidence similarity to evaluate the correctness of a reconstruction. The sudoku and the jigsaw puzzle problems are known to be NP-complete [16, 6]. In this work, I implement the algorithms mentioned above, and show that PGMs are quite successful in rapidly tackling these two problems. I am able to solve 90% of hard sudoku puzzles within 17 rounds of MP. The jigsaw images that are reconstructed by the chosen algorithm are reasonable.

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to our beloved Dr. Sami Khuri, Professor and Chair of Computer Science Department, for providing me this opportunity to work with him. His professional guidance, technical support, and continuous encouragement motivated me to challenge myself and make progress throughout this project. I am obliged to Natalia Khuri for her willingness to spend so much time and provide invaluable suggestions in all my endeavors.

I would also like to thank my committee members, Dr. Teng Moh and Dr. Thomas Austin for their time and advices.

Table of Contents

List of Figures

List of Tables

1. Introduction

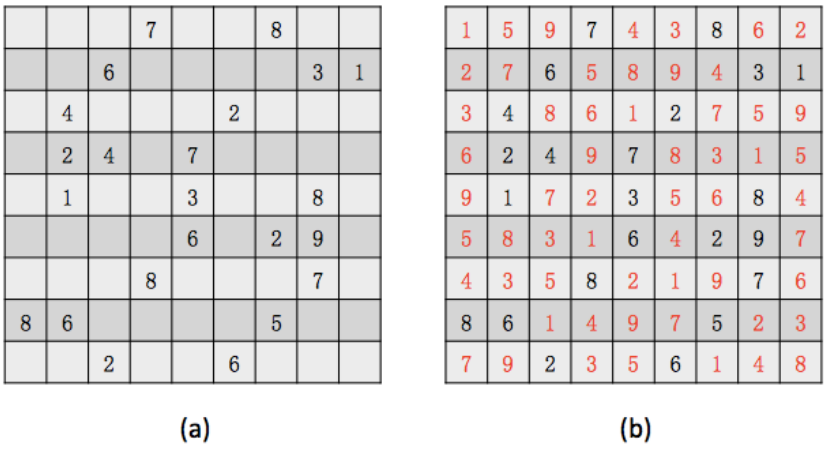

Figure 1. A 9×9 sudoku puzzle. (a) The puzzle problem. (b) The solution.

Probabilistic Graphical Models (PGMs) use a graph-based representation as the basis for compactly encoding a complex distribution over a high-dimensional space. I explore how PGMs can be applied to undirected Markov Networks. This work focuses on two puzzle problems: sudoku and jigsaw puzzles. First, I represent the two puzzles as undirected graphs called Markov networks. Then, I map the relations of nodes to Markov Random Fields (MRFs) and perform belief propagation. MRFs work well with the Neighborhood System and Clique. I apply MRFs to our PGMs using the notations defined in chapter 3. S is a set of lattice points containing all the points, which are denoted as s. I denote X as the value, and I define a set of neighbors of each S, which are denoted as c. In an $N \times N$ sudoku puzzle, each cell is a point in S, and each point has (N-1) neighbors. As to the jigsaw grids, a patch location is treated as a point in MRFs, and each point has up to four neighbors: up, down, left, and right (corner points have two neighbors and other edging nodes have four neighbors). In this way, the two puzzle problems are mapped to MRFs, and I can perform the theory of message passing to them accordingly.

Sudoku is a popular puzzle game. An $N \times N$ sudoku puzzle is a grid of cells partitioned into N smaller blocks of N elements. The objective is to fill an $N \times N$ grid with digits so that each column, each row, and each of the $\sqrt{N} \times \sqrt{N}$ sub-grid (all-different constraints) contains all of the digits from 1 to N, as shown in Figure 1. Sudoku puzzles are mathematical problems, and many people are interested in solving sudoku puzzles in their daily lives. Usually, people use brute-force algorithm and randomly pick a proper number to fill the empty cell, or they choose more reasonable numbers by inspection. Sudoku has been proven to be NP-complete [15]. In mathematics, we treat sudoku puzzles as an instance of the graph-coloring problem [14].

I represent sudoku puzzles as bipartite graphs, which are defined as graphs whose vertices can be divided into two disjoint sets, S and C, so that the graph's edges connect vertices in S only to vertices in C, and vice versa. S contains all the $N \times N$ cells in sudoku grids, and C contains all the $3 \times N$ constraints in sudoku grids. In doing so, all the cells in an $N \times N$ sudoku puzzle can be mapped to s $\in$ S by assigning them labels from 1 through N^2 in a row-scan order, and all the constraints can be labeled from 1 to $3 \times N$ in a row, column, and small sub-grid order. In this work, I focus on solving 9×9 sudoku puzzles.

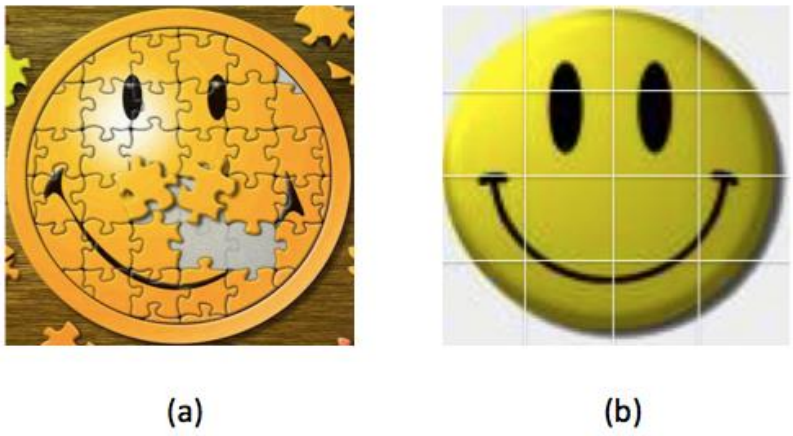

Figure 2. Jigsaw puzzles. (a) A jigsaw puzzle with patches with unique shapes. (b) A jigsaw puzzle with square patches.

In the jigsaw puzzle problem, the objective is to reconstruct an image from a collection of none overlapping image patches, which are sampled from an original image. Jigsaw puzzles are very challenging games, and require expertise to solve. When it comes to mathematics, jigsaw puzzle problems are proven to be NP-complete [6] when the pairwise affinity is unreliable, since here I only deal with square patches. In reality, most jigsaw puzzles provide patches with unique shapes, which contains crucial pairwise affinity information as shown in Figure 2 (a). The jigsaw puzzle discussed in this work contains only square patches as shown in Figure 2 (b).

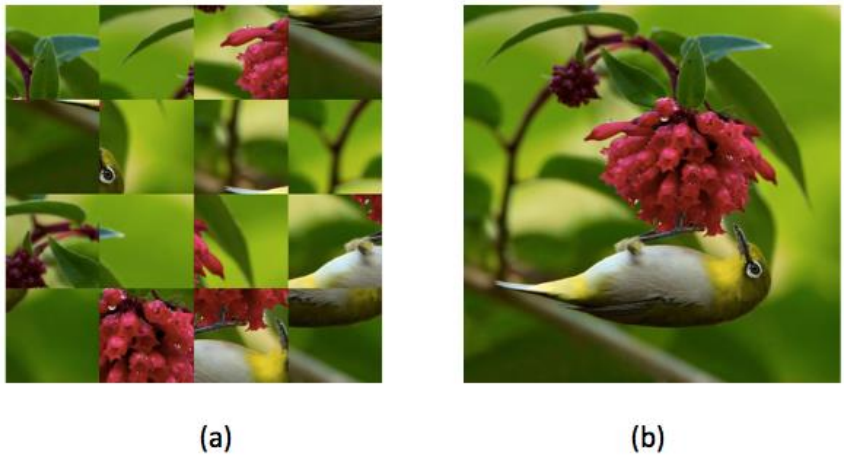

Figure 3. A 4×4 jigsaw puzzle. (a) 16 unordered patches (b) The reconstructed image.

Figures 3 shows an example of a jigsaw with 16 patches. The way that I map the jigsaw puzzle to PGMs is to treat the location as nodes and patches as labels. Hence, the problem is reduced to finding a patch configuration that is most likely on the graph.

In the next chapter, I will first explain the concept of PGMs, and then show the existing approaches for solving the two puzzle problems: sudoku and jigsaw puzzle.

2. Background

2.1 Probabilistic Graphical Models (PGMs)

Probabilistic graphical models are graphs in which nodes represent random variables, and the (lack of) arcs represent conditional independence assumptions. Hence, they provide a compact representation of joint probability distributions. Graphical models, also called Markov Random Fields (MRFs) or Markov networks, have a simple definition of independence: two (sets of) nodes A and B are conditionally independent given a third set, C, if all paths between the nodes in A and B are separated by a node in C [11]. By contrast, directed graphical models also called Bayesian Networks or Belief Networks (BNs), have a notion of independence, which takes into account the directionality of the arcs, as I explain below. Undirected graphical models are more popular with physics and vision communities, and directed models are more popular with the AI and statistics communities. (It is possible to have a model with both directed and undirected arcs, which is called a chain graph.)

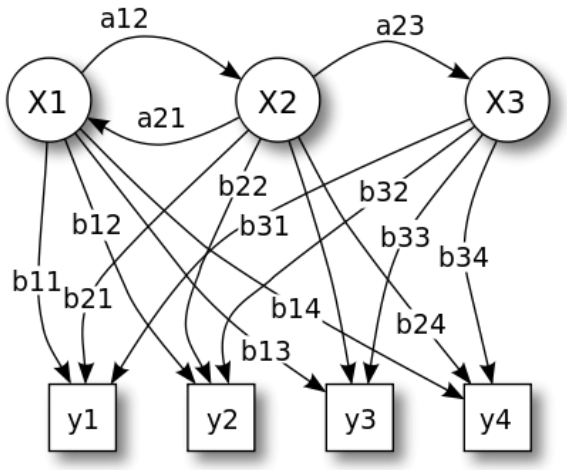

Figure 4. A PGM example [14].

In undirected graph, variables might not be in a "causality" relation, but they can be correlated, such as the pixels in a neighborhood in an image. An undirected graph over a set of random variables $\{X_1, \ldots, X_n\}$ is called an undirected graphical model or Markov random field (MRF) or Markov network, as shown in Figure 4.

2.2 Existing Works

Sudoku puzzles have been solved by computer-based solutions with humanly specified tricks. The most accurate method is brute-force, but with high time cost, which is $O(N^M)$, where N is the size of the grid and M is the number of empty cells. This solution is guaranteed to find the correct result as long as the sudoku puzzles are solvable. However, for some hard sudoku puzzles, which have 50 to 60 empty cells in a 9×9 puzzle, it will be very time consuming. The idea of using PGMs to solve sudoku is greatly motivated by the success of applying PGMs and

BP to decode the low-density parity-check (LDPC) code [2]. In LDPC decoding, the information about received bits that are contained in the relative constraint is aggregated through a neighborhood that contains the previous received information. Similarly, each sudoku cell is mapped with 3 constraints and thus 3 groups of neighborhood systems. Since sudoku and LDPC decoding share similar structures, I am able to apply BP to sudoku as well.

Todd K. Moon and Jacob H. Gunther demonstrated how BP works for solving sudoku puzzles in [3], and provided a successful instance solved by BP and a biased failed instance due to the loopy propagation caused by loops in sudoku puzzles. Sheehan Khan et al., proved that a heuristic algorithm that combines BP and Sinkhorn balancing will avoid the increasing impact of loopy propagation along with the size of sudoku puzzles [4]. They also combine the sum-product and max-product, which are two traditional methods when using BP, thus reduced the time complexity from either methods from $O(N^M)$ to $O(N!\,N^4)$ without compromising accuracy. In my work, I use the time saving Elimination step to fill all cells with unique values, and then apply traditional sum-product algorithm for BP. I apply logarithms to solve the underflow problem, which occurs during MP. This work successfully shows that PGMs and BP work well with hard sudoku puzzles. By increasing the number of BP, we can solve more puzzles.

There are many problems in society, which can be mapped to the jigsaw puzzle problem, for example, speech descrambling [7], reassembling archeological relics [8], and documentation fragments [9]. The jigsaw puzzle problems that I use in this work are square patches without pairwise or neighborhood affinity derived from the shape of the patches. This kind of jigsaw puzzle problems are even harder and technically challenging, and have already been proven to be NP-complete [6]. To be able to effectively apply PGMs and MP, I first need to define pairwise affinity and local evidence. Pairwise affinity is the metric for measuring neighborhood relationships, which are importance, since MRFs contain nodes and links. By checking the individual pixel from a patch, I am able to find a more probable match for the patch, assuming that nearby patch borders share similar pixel information. I need to measure four neighborhood

probabilities for each patch pair: up, down, left, and right. Note that I have a combination of N choose 2. In addition, the direction of patches is fixed, which makes this problem more manageable. Local evidence is the factor that can provide the framework of the original image. I can never rebuild the original image without the knowledge of the layout.

In Cho's paper [5], the author came up with PGMs combined with two strategies of building local evidence, building sparse-and-accurate evidence, and building dense-and-noisy evidence. In this work, I choose to use the dense-and-noisy evidence. The jigsaw puzzles that people solve are provided with a small version of the original image. The reference version can always lead to a low-resolution image, which is of the same size as the original image that I need to construct. Then I can take the low-resolution image as our local evidence. The performance of this approach heavily relies on the texture of the image, on how I make the low-resolution image, and on which metric I use to evaluate the result. This work successfully shows that I can get a reasonable reconstructed image after applying PGMs and MP to the jigsaw puzzles.

In the next chapter, I will show the models built for solving the sudoku and jigsaw puzzles. I will also cover the notations used in the models, and I will explain how I compute the probabilities.

3. Building the Models

3.1 PGM of Sudoku Puzzle

The goal of solving sudoku is to fill numbers from 1 to 9 into a 9×9 sudoku puzzle, and a number can only appear once in each row, each column, and each 3×3 sub grid. Each row, each column, and each 3×3 sub grid must hold a permutation of the numbers of 1 to 9. As mentioned earlier in this work, I focus on two sets for sudoku puzzle: S for cells and C for constraints. The cells are partially filled, and the rest of S is empty. I assume that sudoku puzzles considered in this work have unique solutions. The notations I use in this work are shown in Figure 5. The numbers filled in cells are the ordering number, which is in row-scan order.

The constraints are represented by $C_m \in C$, $m \in \{1,\ldots,27\}$, where $C_m = \{0,1\}$. I denote the contents of cell n by $S_n \in \{1,\ldots,9\}$, meaning that numbers from 1 to 9 are candidates for cell n, with $n \in \{1, 2, \ldots, 81\}$. Cells are numbered in row-scan order as shown in Figure 5.

Figure 5. A 9×9 sudoku puzzle with cell labels and constraints.

Each constraint contains and controls 9 cells as shown in Figure 6, and all the 9 cells must take different numbers from 1 to 9 to guarantee that every number appears in each constraint (9 cells) only once. Constraint c_1 controls 9 cells, S_1 to S_9. Each cell is controlled by 3 constraints, for example, cell S_1 belongs to C_1, C_{10}, and C_{19}.

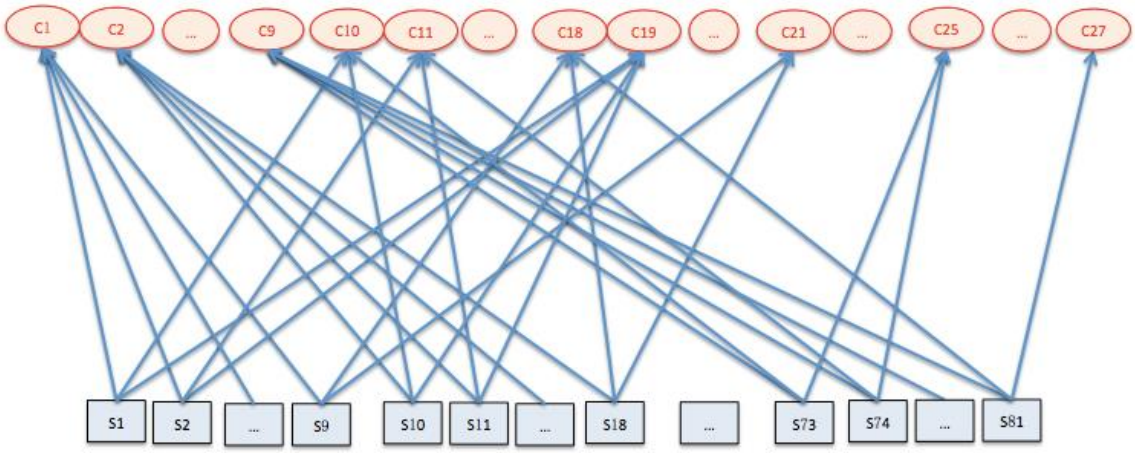

Figure 6. Factor graph associated with a 9×9 sudoku puzzle.

I use the same notations as defined in [3]. I define the set of cells associated with one constraint C_m as N_m, and the set of constraints associated with one cell S_n as M_n. Therefor, each N_m has 9 elements, and each M_n has 3 elements. By $N_{m,n}$, we mean $N_{m\backslash n}$, excluding n from the set. As shown in Figure 6,

$N_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

$N_{10} = \{1, 10, 19, 28, 37, 46, 55, 44, 73\}$,

$N_{19} = \{1, 2, 3, 10, 11, 12, 19, 20, 21\}$,

$M_1 = \{1, 10, 19\}$,

$M_2 = \{1, 11, 19\}$,

$M_{81} = \{9, 18, 27\}$,

$N_{1,3} = \{1, 2, 4, 5, 6, 7, 8, 9\}$,

$N_{10,10} = \{1, 19, 28, 37, 46, 55, 44, 73\}$.

Each cell has a message vector in the form of p_n,

$p_n = [P(S_n=1)\ P(S_n=2)\ \ldots\ P(S_n=N)]$, which is the probability vector of cell S_n.

If the cell is originally filled with number k, $k \in \{1,\ldots,9\}$, then the k^{th} element in the probability vector is 1 and all other elements are 0. For example, if k = 1, then,

$p_n = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$.

In another way, if a cell has three candidate numbers, which are 1, 2, and 3 and all three are legal, then,

$p_n = \frac{1}{3}$ [1 1 1 0 0 0 0 0 0] = [$\frac{1}{3}$ $\frac{1}{3}$ $\frac{1}{3}$ 0 0 0 0 0 0].

3.2 Jigsaw Puzzle

3.2.1 MFR of Jigsaw Puzzle

For a jigsaw puzzle with size $M \times N$, where M is the number of rows in the puzzle grid and N is the number of nodes (patches) in each row, the number of patches is $M \times N$.

To solve jigsaw puzzle problems, I require the square patches cut from the original image, and the reference layout of the image. I denote locations as nodes and patches as labels, and then map the jigsaw puzzle to MRFs. Our goal becomes filling all the nodes with labels, which is finding a mapping configuration that is most likely to the original mapping. Cho et al [12] solved the patch transformation problem, which is very similar to the jigsaw puzzle problem, by assuming the existence of a low-resolution image.

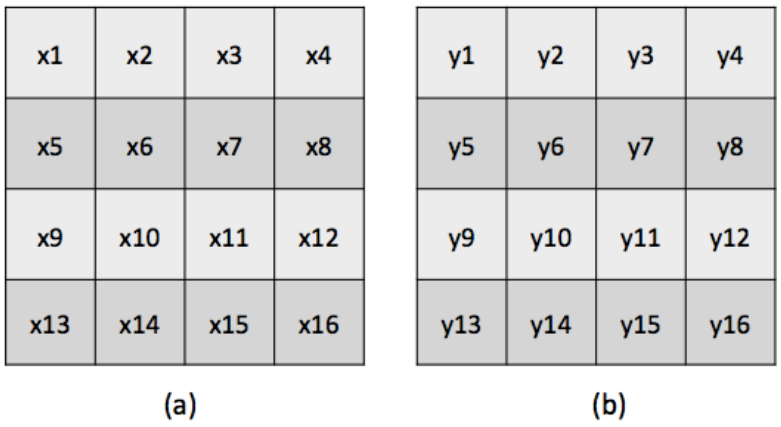

Figure 7. A jigsaw puzzle instance. (a) The reconstructed image grid. (b) The original image grid.

I denote the reconstructed image as X = $\{x_1,\ldots,x_{16}\}$, and denote the original one as Y = $\{y_1, \ldots, y_{16}\}$. Figure 7 is a 4×4 jigsaw puzzle, with 16 nodes and 16 labels. Both image, X and Y have the same framework, corresponding nodes, and labels. As can be seen in Figure 7, each location node has at most 4 neighborhood nodes. Neighbors always share one common edge. For example, node 1 and node 2 are neighbors, since the right edge of node 1 is also the left edge of node 2. Node 1 and node 5 are not neighbors, since they do not share a common edge. Indices are represented by i and j, and the neighborhood set of a node i is represented by N(i).

N(1) = {2, 5},

N(6) = {2, 5, 7, 10}.

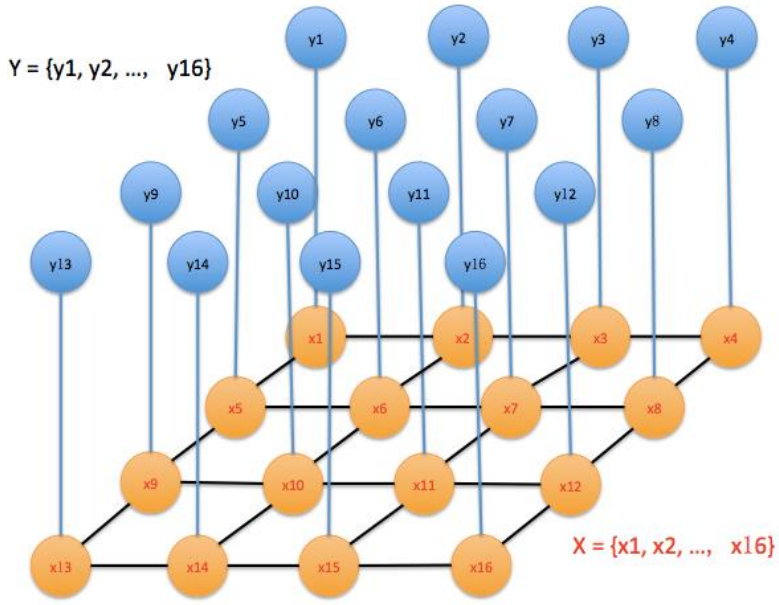

Figure 8. MRF for a 4×4 jigsaw puzzle.

The links in Figure 8 represent direct probabilistic dependency between node pairs. Y = $\{y_1,\ldots,y_{16}\}$ includes all the observed nodes derived from the low resolution image. X =

$\{x_1,\ldots,x_{16}\}$ contains all the nodes and the pairwise affinity relationships between the nodes. Each node in the reconstructed image has relationships with its at most 4 hidden neighborhood nodes and one observed node.

To make the reconstructed image as similar as possible to the original image, I need to maximize the following equation,

$$P(x;y) = \frac{1}{z}\prod_{i=1}^{N} p(y_i|x_i)p_{i,\square}(x_j|x_i)p(x_i)E(x) \quad (1)$$

where:

Z: Normalization constant

N(i): Markov blanket of a node i, set of neighborhood nodes of node i

$p(y_i|x_i)$: Local evidence used to evaluate image x to have a similar scene structure as y, also called DataCost

$p_{i,j}(x_j|x_i)$: Probability of placing a patch x_j in the neighborhood of another patch xi, also called SmoothnessCost

$p(x_i)$: In most cases, this term is modeled as uniform distribution

E(x): Exclusion term that discourages patches from being used more than once

To make it easier to understand, I borrow the notations from [13], which represents the MRF model by an energy equation,

$$energy(Y,X) = \sum_i DataCost(y_i,x_i) + \sum_{j=N(i)} SmoothnessCost(x_i,x_j) \quad (2)$$

The energy function evaluates the cost of going from image X to image Y. This is a minimization problem.

The DataCost function returns the cost of assigning a label value xi to data yi, and the SmoothnessCost function returns the dissimilarity of neighborhood nodes.

3.2.2 Patch Compatibility

In this part, I define the metric of evaluating dissimilarity between two neighborhood patches, as $p_{i,j}(x_j|x_i)$ mentioned in Equation 1 and the SomoothnessCost used in the Equation 2.

The two neighborhood patches have four kinds of relationships: up, down, left, and right, and they share a common edge. I assume that the color difference along the nearby boundaries should stay most similar if they are neighbors in the original image Y.

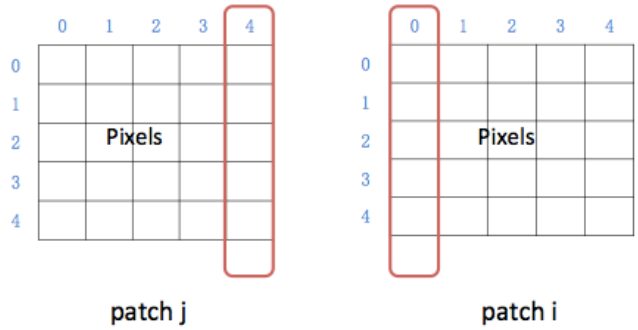

Figure 9. Left-right relationship between patch i and patch j.

Patch i and patch j are considered as $K \times K \times 3$ matrices, where K is the number of pixels in each row and each column, and 3 is the number of color space. I can either use RGB or LAB color space. I choose to use LAB color space as mentioned in [12]. I use u to represent the right most column of patch j, and v to represent the left most column of patch i. In the example as shown in Figure 9, K=5, so the matrix is of size $5 \times 5 \times 3$. Patch i is put as the right neighbor of patch j. The left-right dissimilarity equation is,

$$DLR(x_j, x_i) = \sum_{k=1}^{K} \sum_{l=1}^{3} (x_j(k, u, l) - x_i(k, v, l))^2 \qquad (3)$$

In the MRFs, probabilities are used to represent all the dependencies. Therefore, I need to convert the distance to a probability number between 0 and 1,

$$P_{i,j}(x_j|x_i) \propto \exp(-\frac{D(x_j,x_i)}{2\sigma_c^2}) \quad (4)$$

where

The up-down relationship compatibility of any patch pair is computed in the same way.

3.2.3 Local Evidence

The low-resolution image is used as our local evidence, and it provides the image layout. If the layout is unknown, it will be extremely hard to reconstruct the correct image with only patch compatibility. Local evidence provides the resource to compute the DataCost mentioned in the Equation 2.

$$p(y_i|x_i = l) \propto exp(-\frac{(y_i - m(l))^2}{2\sigma_e^2}) \quad (5)$$

where:

m(l) is the mean color of patch l and $\sigma_e = 0.4$.

I explore two ways to compute the local evidence.

The first solution is to use the mean color values (R, G, B) of the whole patch, and then take the average distance.

$$y_i(R) = \frac{1}{N}\sum_{r=1}^{N} R_r \ , \ y_i(G) = \frac{1}{N}\sum_{r=1}^{N} G_r \ , \ y_i(B) = \frac{1}{N}\sum_{r=1}^{N} B_r \quad (6)$$

$$ml(R) = \frac{1}{N}\sum_{r=1}^{N} R_r \ , \ ml(G) = \frac{1}{N}\sum_{r=1}^{N} G_r \ , \ ml(B) = \frac{1}{N}\sum_{r=1}^{N} B_r \quad (7)$$

By applying Equation 6 and Equation 7 to Equation 5, I convert the mean color distance to a probability number between 0 and 1,

$$p(y_i|x_i = l) = \exp\left(-\frac{\frac{1}{3}\sum_{c=\{R,G,B}(y_i(c)-ml(c))^2}{2\sigma_e{}^2}\right) \quad (8)$$

where:

i: Location

y_i: Original patch at location i

m(l): The patch I want to put at location i

N: Number of pixels ($N = 25 \times 25$ for this example)

σ_e: 0.4

The second solution is to compute the mean color values (R, G, B) of each pixel, and then take the average.

$$y_i = \frac{1}{3}\sum_{c=\{R,G,B\}} c\,, \qquad m(l)_i = \frac{1}{3}\sum_{c=\{R,G,B\}} c \quad (9)$$

By applying Equation 9 to Equation 5, I convert the mean color distance to a probability number between 0 and 1, and get the final equation for computing the DataCost,

$$p(y_i|x_i = l) = \exp\left(-\frac{\frac{1}{N}\sum_{i=1}^{N}(y_i-m(l)_i)^2}{2\sigma_e{}^2}\right) \quad \text{or}$$

$$p(y_i|x_i = l) = \frac{1}{N}\sum_{i=1}^{N}\exp(-\frac{(y_i-m(l))^2}{2\sigma_e{}^2}) \quad (10)$$

where:

i: Pixel at one patch

N: Number of pixels ($N = 25 \times 25$ for this example).

In this chapter, I have built the models and introduced the notations, which are used for sudoku and jigsaw puzzles. In the next chapter, I will show the message passing and solution check steps for solving the puzzle problems.

4. Message Passing and Solution Check

4.1 Sudoku Puzzle

4.1.1 Message Passing

Before MP, I introduce the concept of elimination. From the perspective of a human being, he or she always first fills the cells that have only one possible value candidate. When there are no more cells with one value candidate available, he or she will fill in more complicated cells by taking guesses.

This elimination step is implemented by checking all the constraints associated with the cell and excluding the numbers that are already taken by eight other neighbors in the same constraint. At least one value will be left. If the number of possible candidates is one, then the cell is filled. After checking and filling all 81 cells, I am able to perform another round if at least one cell is filled with a number from the previous round. Thus, I reduce the number of empty cells and increase the accuracy of probabilistic messages used in the Belief Propagation Step.

Belief Propagation is implemented by sending probabilistic messages between adjacent nodes. Each constraint node in C is connected to N cells from S, and each cell from S is connected to three constraint nodes from C.

The message that constraint C_m sends to cell S_n is the probability of satisfying constraint C_m when cell S_n takes the value x,

$r_{mn}(x) = P(C_m \text{ is satisfied}|S_n = x)$,

$r_{mn}(x) = P(C_m|S_n = x)$.

In fact, the value x is one of the values in vector [1 2 3 4 5 6 7 8 9], so the message is passed from a constraint to a cell in the form of a message vector.

$r_{mn} = [r_{mn}(1)\ r_{mn}(2)\ r_{mn}(3)\ r_{mn}(4)\ r_{mn}(5)\ r_{mn}(6)\ r_{mn}(7)\ r_{mn}(8)\ r_{mn}(9)]$

$$r_{mn}(x) = \sum_{\substack{\{n' \in N_{m,\square}\} \\ n=x, n'=x_{n'} \text{ all unique}}} \prod_{l \in N_{m,n}} q_{lm}(x_l) \qquad (11)$$

The probability message that cell S_n sends to constraint C_m is the probability that all other constraints associated with cell S_n besides C_m are satisfied when cell S_n takes value x,

$q_{nm}(x) = P(S_n = x|\text{all the constraints except } C_m \text{ involving } S_n \text{ are satisfied})$

$q_{nm}(x) = P(S_n = x|\{C_{m'}, m' \in M_{n,m}\})$.

Similarly, the message that passed from cell S_n is also a message vector.

$$q_{n,m}(x) = P(n = x) \prod_{m' \in M_{n,m}} r_{m'n}(x) \qquad (12)$$

By following Equation 11 and Equation 12, constraints and cells can send their messages back and forth. Every cell is reachable to three constraints, so it can send a message to each of the constraints with aggregated messages from two other constraints. After a

constraint receives messages from all the nine cells, which are under its control, the constraint can send a message to each of the nine cells with an aggregated neighborhood message of all nine cells. In this way, one round of message passing is performed.

$$q_n(x) = P(n = x) \prod_{m \in Mn} r_{mn}(x) \quad (13)$$

I should update $q_n(x)$ with Equation 3 after each round. In this work, I treat $q_n(x)$ as $p_n(x)$, meaning $q_n(x)$ is the updated version of $p_n(x)$. Especially in the solution check part, the probabilistic message that I reference is $q_n(x)$.

Each message m is a vector, which should contain all possible numbers from {1, 2, 3, 4, 5, 6, 7, 8, 9}, meaning the possibility of one number can be filled into the specific cell. The message passing is done in this way. After several rounds of message passing, I can get a belief vector of each cell taking nine numbers. Then in the solution check part, I assign the number with the highest belief to a cell, until all the empty cells are successfully filled, or the algorithm cannot move forward any further.

PGMs of sudoku puzzles are cyclic graphs, under which circumstance the loopy belief propagation is likely leading to a potentially biased result. Without cycles in graphs, belief propagation theory is supposed to give a good result after a sufficiently large number of message passing steps. However, experience has shown that the results are usually still useful [10]. This work also successfully proves that BP works well for our problem. The only challenge that I deal with is underflow, which occurs along with the increasing number of message passing. I solve this problem by applying logarithms on the probabilistic messages. I will explain the detailed solution in chapter 6.

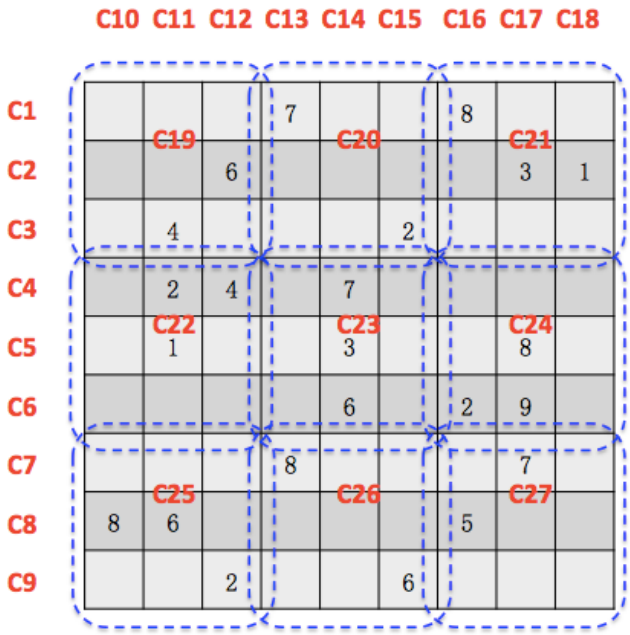

Figure 10. A 9×9 sudoku puzzle.

Initialization of the probabilistic message is the first problem that I need to solve. The probabilistic messages sent between cells and constraints are taking each other as factors. I need to start performing MP either with constraint to cell messages or cell to constraint messages. Figure 10 shows a 9×9 sudoku instance, which contains 58 empty cells. In this work, I choose to start from Equation 1. Take cell S_1 as an example,

$M_1 = \{1, 10, 19\}$,

$r_{1,1} = [\frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ 0\ 0\ \frac{1}{7}]$,

$r_{10,1} = [\frac{1}{8}\ \frac{1}{8}\ \frac{1}{8}\ \frac{1}{8}\ \frac{1}{7}\ \frac{1}{8}\ \frac{1}{8}\ 0\ \frac{1}{8}]$,

$r_{19,1} = [\frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ 0\ \frac{1}{7}\ 0\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}]$,

$q_{1,1} = [\frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\ 0\ \frac{1}{6}\ 0\ \frac{1}{6}\ 0\ \frac{1}{6}]$,

$q_{1,10} = [\frac{1}{5}\ \frac{1}{5}\ \frac{1}{5}\ 0\ \frac{1}{5}\ 0\ 0\ 0\ \frac{1}{5}]$,

$q_{1,19} = [\frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ \frac{1}{7}\ 0\ 0\ \frac{1}{7}]$,

$p_1 = [\frac{1}{5}\ \frac{1}{5}\ \frac{1}{5}\ 0\ \frac{1}{5}\ 0\ 0\ 0\ \frac{1}{5}]$.

4.1.2 Solution Check

After several rounds of message passing are performed, and the final p_n is updated, I start to do the solution check, which is to try to guess which value is the perfect candidate for a blank cell.

$$x' = \text{ArgMax}[\ p_n\ (S_n=x)\ ,\ x] \tag{14}$$

The value, which makes the highest relative probability, has the biggest potential to be put into the cell. According to Equation 4, I start performing the solution check by actually checking the values of p_n of all blank cells in S. In this work, I introduce the concept of relative probability from [4]. The relative probability means that the probability of a cell takes a value divided by the summation of other related empty cells (20 relatives from three constraints) taking the same value. There is still one question left; which cell to be filled first? The author did not explain clearly which neighborhood to consider or which cell to assign first in the original paper. I decided to find a cell, which has the highest probability element among at most 81×9 probabilities, and chose a value for this cell. Then, I focus on the cell I find, and try to consider all the three constraints related to the cell.

Each time after I set the wining value to the according cell, I update the relative constrains by removing and distributing the probability of the value from neighborhood cells, which also take this value as a candidate. I keep checking and assigning values to cells until

there is no blank cell any more, meaning I succeed, or there is a conflict, meaning I fail. The procedure is shown in Figure 11.

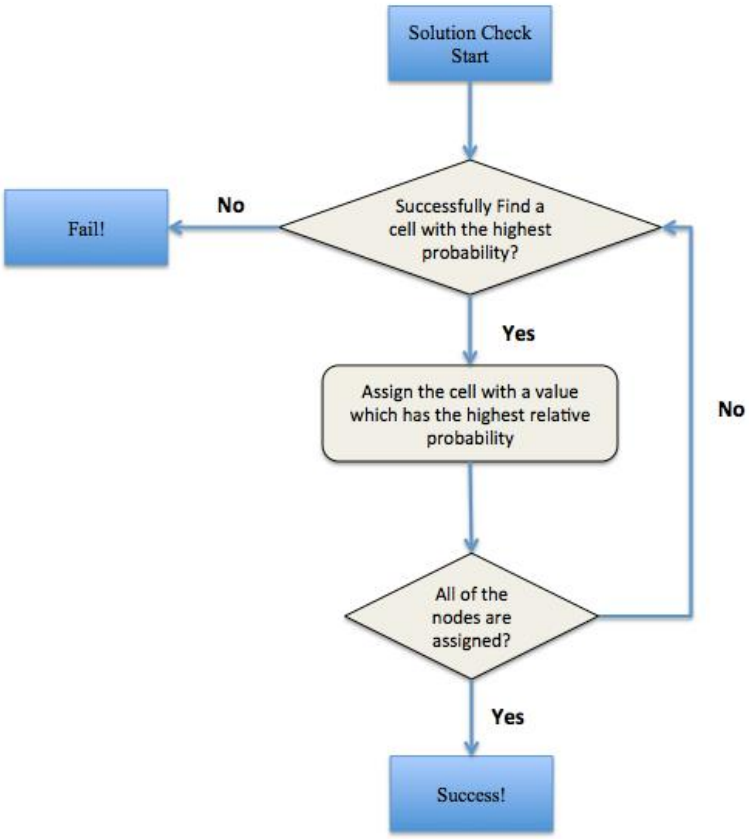

Figure 11. Solution check procedure for sudoku puzzles.

4.2 Jigsaw Puzzle

4.2.1 Message Passing

Unlike the sudoku puzzle problem, there is no constraint in the model of the jigsaw puzzle, and messages cannot be sent back and forth as in the sudoku puzzle. The neighborhood relationships that exist in the jigsaw puzzle happen between neighboring nodes,

which are locations. Nodes can send probabilistic messages to their neighboring nodes, and receive messages from them accordingly.

I denote i and j as locations, and x_i and x_j as the patch label values that the locations take. For example, node j sends a message to node i, with node i taking patch label xi. The assumption is that, I have the messages of node j's neighbors besides node i send to node j. In addition, node j takes patch label x_j. All the patches excluding the current patch xi are candidates for x_j. Node j sends aggregated messages, which are probabilities of node j taking all the possible patches, to node i.

x1	x2	x3	x4
x5	x6	x7	x8
x9	x10	x11	x12
x13	x14	x15	x16

Figure 12. A 4×4 jigsaw puzzle.

As shown in Figure 12, Xi is capable of taking any patch label, so the message that node j sends to node i is in the format of a vector, $m_{ji} = [m_{ji}(x1)\ m_{ji}(x2)\ \ldots\ m_{ji}(x16)]$.

The message that is sent from node j to a node i is,

$$m_{ji}(x_i) \propto \sum_{x_j} p_{i,}\square(x_i|x_j)p(y_i|x_j)\prod_{l\in N(j)\setminus i} m_{lj}(x_j) \qquad (15)$$

N(j): Markov blanket of a node j, set of neighborhood nodes of node j

N(j)\i: Markov blanket of a node j besides i, since i is the receiver of this message

$p(y_i|x_i)$: Local evidence used to evaluate image x to have a similar scene structure as y,

also called DataCost

$p_{i,j}(x_j|x_i)$: Probability of placing a patch x_j in the neighborhood of another patch x_i, also called SmoothnessCost

The message is a vector, since all the patches are possible to any locations.

4.2.2 Solution Check

I perform a solution check, which is to begin to decide which patch to put for locations, after several rounds of message passing. By computing the nodes' marginal probabilities, which are called beliefs, I make the decision of which patch to take. The belief is computed by gathering all messages from node i's neighborhood nodes and the local evidence,

$$b_i(x_i) = p(y_i|x_i) \prod_{j \in N(i)} m_{ji}(x_i) \quad (16)$$

I need to decide which patch to be assigned to the image first or which location to be filled first. There are two approaches to perform the solution check. The first approach is to choose a location, which has the highest belief, and then assign a patch to it. The second approach is to randomly pick a patch and assign the patch to a location, which makes the highest belief among all the unassigned locations. I choose to use the first solution, since the highest belief that I get is the global highest one, which also supports the decision of assigning the winning location. The solution check is performed 16 times until all the nodes are assigned by a patch label. The procedure is shown in Figure 13.

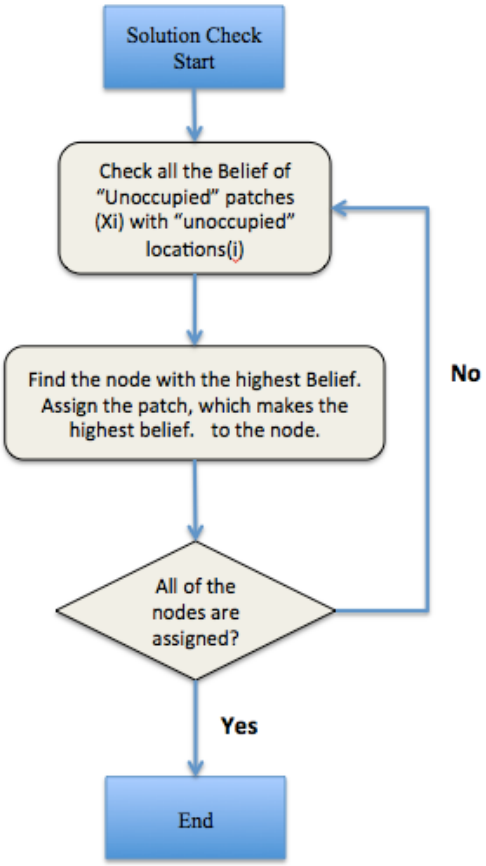

Figure 13. Solution check procedure for jigsaw puzzles.

I explain the different message passing and solution check strategies I use for solving sudoku and jigsaw puzzles in this chapter. In chapter 5, I will analyze the evaluations for both puzzle problems.

5. Evaluation and Performance Metrics

5.1 Required Number of MP Rounds for Sudoku

The only way to evaluate the performance of belief propagation applied on sudoku puzzles is to check if it can provide a successful solution, which means all the cells are filled without collisions.

In theory, the more message passing rounds I perform within the limitation of loopy belief propagation, the closer I get to the correct solution. In this work, I test all the 40 hard sudoku puzzles by performing message passing round from one to many, and perform solution check after each message passing round to check if the specific number of message passing round is enough or not.

5.2 Evaluation Metrics for Jigsaw

Jigsaw puzzles problems have been solved through many approaches, but there aren't any conventional methods of evaluation. In this work, I use the direct comparison and neighborhood comparison.

5.2.1 Direct Comparison

I compare all the patches in the reconstructed image directly with patches in the original image, and then count the number of differences. The direct comparison is a stricter metric of

evaluating a reconstructed image than the neighborhood comparison. I choose to use this metric to show our result, since the reconstructed image is more visually pleasing to the human eyes.

5.2.2 Neighborhood Comparison

For each patch, I consider the four neighbors in the reconstructed image to check if the belief propagation algorithms have assigned the correct neighborhood patches.

In the next chapter, I will explain how I implement the two algorithms in Java. I will also cover the challenges, which I encounter during implementation, and solutions that I use to conquer these challenges.

6. Implementation

6.1 Data Structures for Solving Sudoku Puzzle Problems

The algorithm for solving sudoku puzzles is implemented in Java, and the source code is attached to the appendix. I store sudoku puzzles in the format of matrices in a file, as shown in Figure 14, where zeros mean blank cells. The algorithm reads one sudoku puzzle from the file, and start reading the next after the MP and solution check are done on the first puzzle. I will use multiple multi- dimensional arrays to store the puzzle and messages that I need to use during the MP and solution check.

When a new sudoku puzzle is read into a two-dimension array: int [][] matrix = new int [9][9], it will be serialized into a one- dimensional array: int [] S = new int [81]. While serializing, the neighborhood relationships M and N are calculated. I use a two-dimensional array N to store 27 constraints and their related cells: int [][] N = new int [27][9], as shown in the Table 1. I use another two-dimension array M to store 81 cells and their related constraints: int [][] M = new int [81][3], as shown in Table 2. All the indices in the tales start from 0.

After I get all the mapping relationships, I am able to compute the initialized the messages as mentioned in chapter 3. I use a two-dimensional array to store the probabilities of a cell takes a value: double [][] P = new double [81][9], where the first index denotes the cell and the second index denotes the value it takes. The initialized messages are shown in Table 3, taking the instance of Figure 15 as an example.

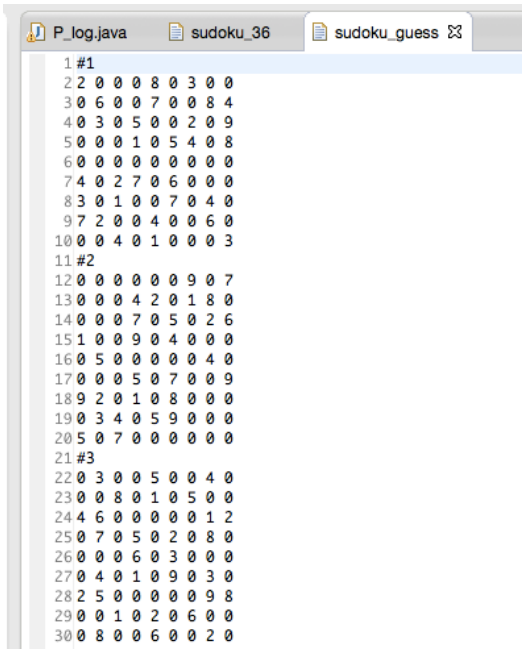

Figure 14. The file that contains sudoku puzzles.

			7			8		
		6					3	1
	4				2			
	2	4		7				
	1			3			8	
				6		2	9	
			8				7	
8	6					5		
		2			6			

Figure 15. A Sudoku instance.

Table 1. The constraints' neighborhood relationships.

Constraint	9 Cells								
0	0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16	17
2	18	19	20	21	22	23	24	25	26
3	27	28	29	30	31	32	33	34	35
4	36	37	38	39	40	41	42	43	44
5	45	46	47	48	49	50	51	52	53
6	54	55	56	57	58	59	60	61	62
7	63	64	65	66	67	68	69	70	71
8	72	73	74	75	76	77	78	79	80
9	0	9	18	27	36	45	54	63	72
10	1	10	19	28	37	46	55	64	73
11	2	11	20	29	38	47	56	65	74
12	3	12	21	30	39	48	57	66	75
13	4	13	22	31	40	49	58	67	76
14	5	14	23	32	41	50	59	68	77
15	6	15	24	33	42	51	60	69	78
16	7	16	25	34	43	52	61	70	79
17	8	17	26	35	44	53	62	71	80
18	0	1	2	9	10	11	18	19	20
19	3	4	5	12	13	14	21	22	23
20	6	7	8	15	16	17	24	25	26
21	27	28	29	36	37	38	45	46	47
22	30	31	32	39	40	41	48	49	50
23	33	34	35	42	43	44	51	52	53
24	54	55	56	63	64	65	72	73	74
25	57	58	59	66	67	68	75	76	77
26	60	61	62	69	70	71	78	79	80

Table 2. Cells' neighborhood relationships.

Cell	Constriants		
0	0	9	18
1	0	10	18
2	0	11	18
3	0	12	19
4	0	13	19
5	0	14	19
6	0	15	20
7	0	16	20
8	0	17	20
9	1	9	18
10	1	10	18
11	1	11	18
12	1	12	19
13	1	13	19
14	1	14	19
15	1	15	20
16	1	16	20
17	1	17	20
18	2	9	18
19	2	10	18
20	2	11	18
21	2	12	19
22	2	13	19
23	2	14	19
24	2	15	20
25	2	16	20
26	2	17	20

Cell	Constriants		
27	3	9	21
28	3	10	21
29	3	11	21
30	3	12	22
31	3	13	22
32	3	14	22
33	3	15	23
34	3	16	23
35	3	17	23
36	4	9	21
37	4	10	21
38	4	11	21
39	4	12	22
40	4	13	22
41	4	14	22
42	4	15	23
43	4	16	23
44	4	17	23
45	5	9	21
46	5	10	21
47	5	11	21
48	5	12	22
49	5	13	22
50	5	14	22
51	5	15	23
52	5	16	23
53	5	17	23

Cell	Constriants		
54	6	9	24
55	6	10	24
56	6	11	24
57	6	12	25
58	6	13	25
59	6	14	25
60	6	15	26
61	6	16	26
62	6	17	26
63	7	9	24
64	7	10	24
65	7	11	24
66	7	12	25
67	7	13	25
68	7	14	25
69	7	15	26
70	7	16	26
71	7	17	26
72	8	9	24
73	8	10	24
74	8	11	24
75	8	12	25
76	8	13	25
77	8	14	25
78	8	15	26
79	8	16	26
80	8	17	26

Table 3. Initialized probabilistic messages.

P	Value
P[0][1]	0.2
P[0][2]	0.2
P[0][3]	0.2
P[0][5]	0.2
P[0][9]	0.2
P[1][3]	0.333333333
P[1][5]	0.333333333
P[1][9]	0.333333333
P[2][1]	0.25
P[2][3]	0.25
P[2][5]	0.25
P[2][9]	0.25
P[4][1]	0.25
P[4][4]	0.25
P[4][5]	0.25
P[4][9]	0.25
P[5][1]	0.2
P[5][3]	0.2
P[5][4]	0.2
P[5][5]	0.2
P[5][9]	0.2
P[7][2]	0.25
P[7][4]	0.25
P[7][5]	0.25
P[7][6]	0.25
P[8][2]	0.2
P[8][4]	0.2
P[8][5]	0.2
P[8][6]	0.2
P[8][9]	0.2
P[9][2]	0.25
P[9][5]	0.25

P	Value
P[9][7]	0.25
P[9][9]	0.25
P[10][5]	0.25
P[10][7]	0.25
P[10][8]	0.25
P[10][9]	0.25
P[12][4]	0.333333333
P[12][5]	0.333333333
P[12][9]	0.333333333
P[13][4]	0.25
P[13][5]	0.25
P[13][8]	0.25
P[13][9]	0.25
P[14][4]	0.25
P[14][5]	0.25
P[14][8]	0.25
P[14][9]	0.25
P[15][4]	0.333333333
P[15][7]	0.333333333
P[15][9]	0.333333333
P[18][1]	0.2
P[18][3]	0.2
P[18][5]	0.2
P[18][7]	0.2
P[18][9]	0.2
P[20][1]	0.166666667
P[20][3]	0.166666667
P[20][5]	0.166666667
P[20][7]	0.166666667
P[20][8]	0.166666667
P[20][9]	0.166666667
P[21][1]	0.2

P	Value
P[21][3]	0.2
P[21][5]	0.2
P[21][6]	0.2
P[21][9]	0.2
P[22][1]	0.25
P[22][5]	0.25
P[22][8]	0.25
P[22][9]	0.25
P[24][6]	0.333333333
P[24][7]	0.333333333
P[24][9]	0.333333333
P[25][5]	0.5
P[25][6]	0.5
P[26][5]	0.25
P[26][6]	0.25
P[26][7]	0.25
P[26][9]	0.25
P[27][3]	0.25
P[27][5]	0.25
P[27][6]	0.25
P[27][9]	0.25
P[30][1]	0.333333333
P[30][5]	0.333333333
P[30][9]	0.333333333
P[32][1]	0.25
P[32][5]	0.25
P[32][8]	0.25
P[32][9]	0.25
P[33][1]	0.333333333
P[33][3]	0.333333333
P[33][6]	0.333333333
P[34][1]	0.333333333
P[34][5]	0.333333333
P[34][6]	0.333333333

I perform the MP with the initialized probabilities stored in P. The messages of constraint-to-variable are stored in a three-dimensional array: double [][][] R = new double [27][81][9], where the first index denotes the constraint, the second index denotes the cell, and the third index denotes the value, which the receiver cell takes. The variable-to-constraint messages are stored in a three-dimensional array: double [][][] Q = new double [27][81][9], where the three indices are similar to the indices in R. I start sending the constraint-to-variable messages first, as shown in Equation 11. After all the Rs are updated, I perform the variable-to-constraint message passing, which results in updated Qs. Then, I update the Ps, which are posteriori beliefs. All the steps mentioned above are called one round of MP.

MP can is run for many iterations until a predefined number of iterations is reached.

	9	5	7	1	3	8	4	2
2	8	6	5	4	9	7	3	1
1	4	3	6	8	2	9	5	
5	2	4	9	7	8	1	6	3
6	1	9	2	3	5	4	8	7
3	7	8	4	6	1	2	9	5
4	7	1	8	2		6	7	9
8	6	7	3	9		5	2	4
9	3	2		5	6		1	8

Figure 16. The unfinished puzzle by one round of MP.

As mentioned in chapter 4, I use relative probabilities in the solution check part, so I use a two-dimensional array to store the computed relative probabilities: double [][] R_P = new double [81][9], where the first index denotes all the cells, and the second index denotes the values taken by the cells.

If I perform the solution check after one round of MP is performed on the example instance, six cells are left empty when I am met by an obstacle and cannot go any further, which is shown in Figure 16. I test by adding one more round of MP each time, and get the result, as shown in the Table 4.

Table 4. Number of left unassigned cells along with increasing number of MP.

MP Round	0	1	2	3	4	5	6	7	8	9	10	11
Count of Unassigned Cells	58	6	10	10	8	8	7	5	3	4	2	0

I successfully fill all the cells of this sudoku puzzle after 11 rounds of MP, as shown in Figure 17.

1	5	9	7	4	3	8	6	2
2	7	6	5	8	9	4	3	1
3	4	8	6	1	2	7	5	9
6	2	4	9	7	8	3	1	5
9	1	7	2	3	5	6	8	4
5	8	3	1	6	4	2	9	7
4	3	5	8	2	1	9	7	6
8	6	1	4	9	7	5	2	3
7	9	2	3	5	6	1	4	8

Figure 17. The completed puzzle by 11 rounds of MP.

6.2 Implementation for Solving Jigsaw Puzzle Problems

6.2.1 Image Instance

To solve jigsaw problems, a collection of image patches cut from the original image and a low-resolution version of the original image are needed. I produce both of the two resources by an image. For example, I cut the image instance as shown in Figure 18 (a) into 16 patches as shown in Figure 18 (b).

The size of the original image is 100×100 pixels, thus makes the size of each patch 25×25 pixels. I have the values of 100×100 pixels, so I am able to manipulate the pixel values of 16 patches according to their locations in the original image.

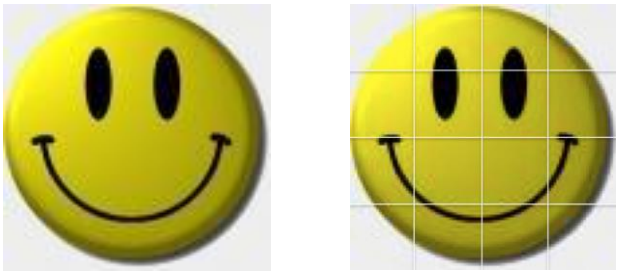

Figure 18. An 100×100 pixels image instance. (a) Original image. (b) 16 patches derived from (a).

6.2.2 Patch Compatibility

As mentioned in chapter 4, I denote the location nodes as $X = \{x_1,\ldots,x_{16}\}$ in a row scan order, denote the patches as $\{1,\ldots,16\}$, and denote the probability of a node taking a patch as $p(x_i=i)$. Figure 19 shows an example of node x1 taking patch 1.

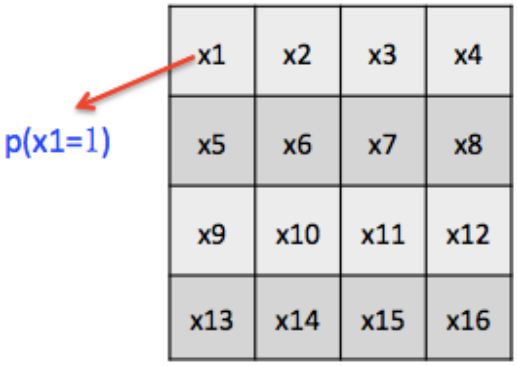

Figure 19. Location node x_i takes patch 1.

First, I store the neighborhood relationships into a two-dimensional array: int N[][] = new int[PATCH_NUMBER][4], where PATCH_NUMBER means the number of patches and 4 means the indices of the patch's neighbors in the order of up, down, left, and right. The contents stored in N are shown in Table 5. All the indices start from 0, and -1 means the neighbor of that position does not exist.

For example, location 0 has two neighbors, which are to the right and under location 0,

N[0] = {-1, 4, -1, 1}.

Location 5 and location 9 have four neighbors,

N[5] = {1, 9, 4, 6}.

N[9] = {5, 13, 8 10}.

Table 5. Neighborhood relationships.

Location	Up	Down	Left	Right
0	−1	4	−1	1
1	−1	5	0	2
2	−1	6	1	3
3	−1	7	2	−1
4	0	8	−1	5
5	1	9	4	6
6	2	10	5	7
7	3	11	6	−1
8	4	12	−1	9
9	5	13	8	10
10	6	14	9	11
11	7	15	10	−1
12	8	−1	−1	13
13	9	−1	12	14
14	10	−1	13	15
15	11	−1	14	−1

Second, I need to compute the patch compatibility as mentioned in chapter 3. I use a three-dimensional array to store the patch compatibility: double P_ji[][][] = new double [PATCH_NUMBER][PATCH_NUMBER][4], where the first index is patch j, the second index is patch i, and the third index means patch j is put to the up, down, left, and right directions to patch j. I compute the patch compatibility before I perform the MP. The results are shown in Table 6 and Table 7.

Table 6. Patch compatibility with patch 0.

	j	P/D[j][0][0]	P/D[j][0][1]	P/D[j][0][2]	P/D[j][0][3]
D	0	497.3379864	162107.4848	80137.85257	187079.6658
	2	41212.54623	91.89852385	137516.141	165433.0617
	3	149074.3071	63869.78191	101671.0197	180002.275
	4	45374.06255	70345.65167	126034.3808	185998.0737
	5	43607.90763	6261.39571	110911.4797	210.3932271
	6	48329.47404	4740.511643	103172.9205	9943.294368
	7	149079.1735	9207.008041	119832.9623	115446.8622
	8	87526.8354	123300.1638	37490.34832	98853.48229
	9	47686.92506	34656.66184	96416.61357	106326.5719
	10	105539.104	7522.235034	123343.6802	95020.10745
	11	149048.6926	55020.03185	104660.8241	131951.3455
	12	142083.229	158147.2886	21499.93774	101778.1261
	13	121041.3828	85809.03405	21534.05467	107166.6614
	14	170406.6198	80574.70475	21506.35918	97984.6128
	15	149079.8795	119053.0001	21486.30342	94632.24257
P	0	0.99999985	0.996256205	2.45E-94	0.999013045
	2	0.99998757	0.999997874	2.32E-161	0.999127193
	3	0.999955038	0.998523281	1.72E-119	0.999050364
	4	0.999986314	0.998373676	5.98E-148	0.999018748
	5	0.999986847	0.999855135	2.77E-130	0.99999889
	6	0.999985423	0.999890321	3.03E-121	0.999947519
	7	0.999955036	0.999786992	1.05E-140	0.999390835
	8	0.999973601	0.997151164	1.61E-44	0.999478368
	9	0.999985617	0.99919844	2.36E-113	0.999438945
	10	0.999968168	0.999825967	8.31E-145	0.999498591
	11	0.999955045	0.998727764	5.54E-123	0.999303778
	12	0.999957146	0.996347497	7.68E-26	0.99946294
	13	0.999963492	0.998016532	7.01E-26	0.999434513
	14	0.999948604	0.998137411	7.55E-26	0.999482952
	15	0.999955036	0.997249159	7.97E-26	0.999500637

Table 7. Patch compatibility with patch 5.

	j	P/D[j][5][0]	P/D[j][5][1]	P/D[j][5][2]	P/D[j][5][3]
D	0	49662.31798	160404.6876	186111.6945	167328.8949
	1	6261.39571	43607.90763	210.3932271	110911.4797
	2	81696.1535	4073.180705	9731.597088	142583.2542
	3	179889.2088	67265.40596	114132.8747	153495.985
	4	74.39637088	79285.63033	107649.0297	41320.38979
	6	11443.75395	76.63602209	96335.49463	106462.4621
	7	179955.0282	3012.559015	139940.3869	71045.23499
	8	35574.85619	112444.7502	101636.6895	77744.46658
	9	8412.534586	23987.64928	131619.1143	6.136414403
	10	56479.15746	1794.425045	119374.7208	1859.514141
	11	179834.4232	42876.57386	97342.21343	113834.2768
	12	92695.6025	155938.4362	179083.7478	63324.56355
	13	82216.91327	73212.58079	178766.7819	42141.42892
	14	120461.1408	63806.83289	178291.3787	45561.43027
	15	179880.7614	102401.3067	178690.4145	100858.6992
P	0	0.999351521	0.973186176	0.998974024	0.975937825
	1	0.999918217	0.992638064	0.99999884	0.983985342
	2	0.998933454	0.999310055	0.999946327	0.979459469
	3	0.997653045	0.98866688	0.999370696	0.977904874
	4	0.999999028	0.986655238	0.999406436	0.994003454
	6	0.999850533	0.999987014	0.999468801	0.984622777
	7	0.997652187	0.999489665	0.999228454	0.989711942
	8	0.999535429	0.981127095	0.999439578	0.988747304
	9	0.999890121	0.995943647	0.999274316	0.999999107
	10	0.999262541	0.999695989	0.999341803	0.999729366
	11	0.997653758	0.99276108	0.999463251	0.983566802
	12	0.998789942	0.97392295	0.999012748	0.990824826
	13	0.998926659	0.98767108	0.999014495	0.993884668
	14	0.998427773	0.989246449	0.999017114	0.99339002
	15	0.997653155	0.982798216	0.999014915	0.985426245

Table 6 and Table 7 show the patch compatibilities of all other patches to patch 0 and patch 5. The first half part is D[j][i][], which is computed with Equation 3, and the second half is P[j][i][], which is computed based on the values from D with Equation 4.

6.2.3 Local Evidence and Patch Dissimilarity

Third, I compute the patch dissimilarity between the patches that are in the original image and the patches that are in the low-resolution image with Equation 5. I use a two-dimensional array to store the patch dissimilarity: double P_YX[][] = new double[PATCH_NUMBER][PATCH_NUMBER], where the first index represents the patch in the low-resolution image and the second index represents the patch in the original image.

There are two ways to compute the Patch Dissimilarity, as mentioned in chapter 3. The first approach is to take the average color of the whole patch, as shown in Equation 8. The results are shown in Table 8.

P_YX[0][0] is the patch dissimilarity between the first patch in the low-resolution image and the first patch in the original image. The two patches share the same location, so P_YX[0][0] is supposed to be the highest among all the P_YX[0][i], i = {1,…,15}. Table 8 shows part of the P_YX[][], where the second index is in {0, 1, 5, 9}.

Table 8. Patch dissimilarities between patches in the original image with patches 0, 1, 5, and 9 in the low-resolution image, computed by Equation 8.

i	P_YX[0][i]	P_YX[1][i]	P_YX[5][i]	P_YX[9][i]
X_i=0	0.998980729	0.005526281	9.86E-06	6.78E-05
X_i=1	0.004767242	0.995103836	0.145254112	0.478744206
X_i=2	9.51E-04	0.517971432	0.584255866	0.684200782
X_i=3	0.595693328	0.002743421	1.94E-05	5.75E-05
X_i=4	0.005996038	0.780195847	0.035613483	0.214111691
X_i=5	1.11E-05	0.134004128	0.999889228	0.664714582
X_i=6	1.35E-06	0.037012756	0.837732467	0.335044363
X_i=7	2.54E-04	0.200398498	0.67496748	0.443287085
X_i=8	5.54E-04	0.409490005	0.670381133	0.665416819
X_i=9	7.89E-05	0.454094927	0.643648985	0.998463179
X_i=10	1.89E-05	0.219010073	0.937790184	0.858330379
X_i=11	1.94E-05	0.031459993	0.466619623	0.145347246
X_i=12	0.589625127	0.002621787	1.85E-05	5.46E-05
X_i=13	1.64E-04	0.101224302	0.531804661	0.257376438
X_i=14	5.10E-05	0.048904268	0.475738809	0.173167384
X_i=15	0.356456659	0.002309948	3.51E-05	7.04E-05

The result of the second approach is shown in Table 9. Similar to Table 8, part of the P_YX[][] is shown, where the second index is in {0, 1, 5, 9}.

Table 9. Patch dissimilarities between patches in the original image with patches 0, 1, 5, and 9 in the low-resolution image, computed by Equation 10.

i	P_YX[0][i]	P_YX[1][i]	P_YX[5][i]	P_YX[9][i]
X_i=0	0.569660491	0.005730888	1.83E-16	5.13E-28
X_i=1	0.038091857	0.398860364	0.022261843	0.001680164
X_i=2	0.054203159	0.043076088	0.159821569	0.208356586
X_i=3	0.276965667	0.024105957	0.001597599	0.018942013
X_i=4	0.025893361	0.317988402	0.015923656	0.00105132
X_i=5	4.34E-17	0.050188961	0.364432937	0.018617752
X_i=6	2.51E-26	0.021875205	0.095433469	0.338025328
X_i=7	0.031764963	0.01963699	0.052553865	0.042099207
X_i=8	0.04673934	0.052300022	0.131594316	0.214218618
X_i=9	3.79E-20	0.024523967	0.10135803	0.810051929
X_i=10	6.06E-26	0.023527443	0.066818164	0.136746633
X_i=11	0.007993247	0.031762112	0.03251628	0.030824731
X_i=12	0.251263288	0.01307282	0.009698372	0.004346259
X_i=13	0.028489031	0.009552744	0.042566428	0.034704425
X_i=14	0.013730906	0.005137693	0.0408317	0.009828956
X_i=15	0.130128936	0.002575371	0.01042798	0.003032355

The probabilistic messages are computed with Sum-Product method as shown in Equation 15. I use a three-dimensional array to store the messages: double M_ji_Xi[][][] = new double[4][PATCH_NUMBER][PATCH_NUMBER], where the first index represents the neighborhood location j, the second index is location i, and the third index represents the patch assigned to location i. First, I initialize all the messages to 1 [13].

As I mentioned in chapter 4, the messages j sends to i are in the format of a message vector. For example, M_ji_Xi[1][0][0] is the message that location 1 sends messages to location 0 when location 0 takes patch 0. The procedure of how I use sum-product to compute this message is shown in Table 10.

Table 10. The procedure of sum-product computation of location 1 sending messages to location 0, with normalization.

X_1	$P_{0,1}(X_0=0\|X_1)$	$P(Y_1\|X_1)$	Product	SUM
$X_1=0$	0(Exclude)	0.005526281	0	
$X_1=1$	0.999999732	0.995103836	**0.995103569**	
$X_1=2$	0.999983321	0.517971432	**0.517962793**	
$X_1=3$	0.999977154	0.002743421	**0.002743358**	
$X_1=4$	0.999981294	0.780195847	**0.780181252**	
$X_1=5$	0.999973248	0.134004128	**0.134000543**	
$X_1=6$	0.999975186	0.037012756	**0.037011837**	
$X_1=7$	0.999972675	0.200398498	**0.200393022**	
$X_1=8$	0.999918488	0.409490005	**0.409456626**	
$X_1=9$	0.999955424	0.454094927	**0.454074685**	
$X_1=10$	0.999973139	0.219010073	**0.219004191**	
$X_1=11$	0.999939373	0.031459993	**0.031458086**	
$X_1=12$	0.999922038	0.002621787	**0.002621582**	
$X_1=13$	0.999926398	0.101224302	**0.101216852**	
$X_1=14$	0.999935402	0.048904268	**0.048901109**	3.936439239/16=
$X_1=15$	0.999906863	0.002309948	**0.002309733**	**0.246027452**

MP will be performed for several rounds before I conduct the solution check. The procedure of how I compute the beliefs is shown in Table 11. For example, location 0 receives two message vectors from its neighbors, location 1 and location 4. Along with the patch dissimilarity with patch 0 in the low-resolution image, the belief is computed by multiplication.

Table 11. The procedure of computing beliefs of location 0 taking patches in {0,…,15}.

Location 0	j=1	j=4	Y=0	Belief
X_0=0	0.246027452	0.166962934	0.998980729	0.041035596
X_0=1	0.183980645	0.11478779	0.004767242	0.000100678
X_0=2	0.213982035	0.153757722	9.51E-04	3.12892E-05
X_0=3	0.239679744	0.167083558	0.595693328	0.023855459
X_0=4	0.197289591	0.105110248	0.005996038	0.000124341
X_0=5	0.236121501	0.163184361	1.11E-05	4.27698E-07
X_0=6	0.19698203	0.164646213	1.35E-06	4.37837E-08
X_0=7	0.227736383	0.163844639	2.54E-04	9.4776E-06
X_0=8	0.220758638	0.157805642	5.54E-04	1.92997E-05
X_0=9	0.198418918	0.152223646	7.89E-05	2.3831E-06
X_0=10	0.232668625	0.162237845	1.89E-05	7.13431E-07
X_0=11	0.238064649	0.166736584	1.94E-05	7.70065E-07
X_0=12	0.245985458	7.04E-06	0.589625127	1.02043E-06
X_0=13	0.240046826	1.93E-05	1.64E-04	7.58757E-10
X_0=14	0.242944846	2.41E-07	5.10E-05	2.98274E-12
X_0=15	0.239731295	1.11E-05	0.356456659	9.48421E-07

Solution check part is implemented according to the procedure in Figure 13. I keep assigning patches to locations until all the locations are filled.

I am able to reconstruct the image accurately with one round of MP, since the fewer patches I cut the image into, the easier I get to the correct result. Then, I cut the same image instance into 25 patches and get the correct result after two rounds of MP.

Figure 20 shows the reconstructed image with one round of MP, with 23 out of 25 patches being put in the correct location. Figure 21 shows the reconstructed image with 2 rounds of MP, all of the 25 patches being reconstructed correctly. If I perform more than two rounds of MP to this problem, the accuracy will deteriorate due to loopy belief propagation.

Figure 20. Reconstructed image with one round of MP.

Figure 21. Reconstructed image with two rounds of MP.

6.3 Challenges and Solutions during Implementation

Underflow occurs during MP when solving both sudoku and jigsaw puzzles.

All the possibilities are between 0 and 1. Since I need to perform MP for several rounds, gathering messages (possibilities) from constraints or nodes, the possibilities and messages decrease sharply and get to underflow after two or three rounds.

I use the logarithm to avoid the underflow. If all the probabilities are expressed in the format of logarithm, then the speed of decreasing will be slower than previously.

I use logarithm to initialize the P as mentioned before. I use a and b to denote the original probabilities. I use x and y to denote the probabilities after applying the logarithm. Since I already have the x and y in hand, I need to perform the calculations of Equation 11 and Equation 12, which contain addition and multiplication. The following are the transformations of SUM and PRODUCT:

$x = \lg(a)$, $y = \lg(b)$

$$\lg(ab) = \lg(a) + \lg(b) = x + y \quad (17)$$

$$\lg(a+b) = \lg(a\times(1+\frac{b}{a})) = \lg(b\times(\frac{a}{b}+1)) = \lg(b) + \lg((\frac{a}{b}+1))$$

$$x - y = \lg(\frac{a}{b})$$

$$2^{x-y} = \frac{a}{b}$$

$$lg(2^{x-y}+1) = lg(\frac{a}{b}+1)$$

$$\lg(a+b) = y + lg(2^{x-y}+1) \quad (18)$$

If (x-y) is big enough to make $(2^{x-y} + 1)$ overflow, then I could ignore the 1. Therefore,

$$\lg(a+b) = y + lg(2^{x-y}) = y + (x-y) = x \quad (19)$$

Equation 17 is used when I perform the multiplication. In addition, Equation 18 is used if $(2^{x-y} + 1)$ is within the limitation of Double, and Equation 19 is applied if $(2^{x-y} + 1)$ leads to overflow.

During the jigsaw puzzle solving, the messages sent between locations are aggregated based on the number of locations in each round of MP. After several rounds of message passing, the message will become incredibly large and finally overflow. I perform normalization for each message. Since the message is gathered from N possibilities, I use the Mean method to normalize the message. As shown in Table 10, I take the value of message dividing N as our final message value, where N is the number of patches in this example.

I explain how I implement the two algorithms with two example instances respectively, in this chapter. In the next chapter, I will show the results that I get by running the puzzle instances.

7. Results

7.1 Sudoku Experimental Results

In order to verify our implementation of solving sudoku puzzles, I choose 40 hard 9x9 sudoku puzzle problem instances [5] for testing; each puzzle has 50 to 60 unassigned cells.

First, I test all the puzzles with one round of MP and only two puzzles are solved. Then, I test all the puzzles with two rounds of MP and five puzzles are solved, including the two puzzles that have been solved in the previous test. More puzzles are solved along with the increasing number of MP iterations. I keep testing and get 35 solved puzzles with 11 rounds of MP. The PGM-based algorithm is able to solve 36 out of 40 (90%) sudoku puzzles within 17 rounds of MP. Figure 22 shows the success rate of solved puzzles along with the increasing number of MP.

Usually, even if I perform more rounds of MP than needed, the puzzles are still successfully solved. For example, a puzzle is solved with five rounds of MP for the first time and it will stay being solved when I perform more than five rounds of MP. More rounds of MP lead to more accurate results.

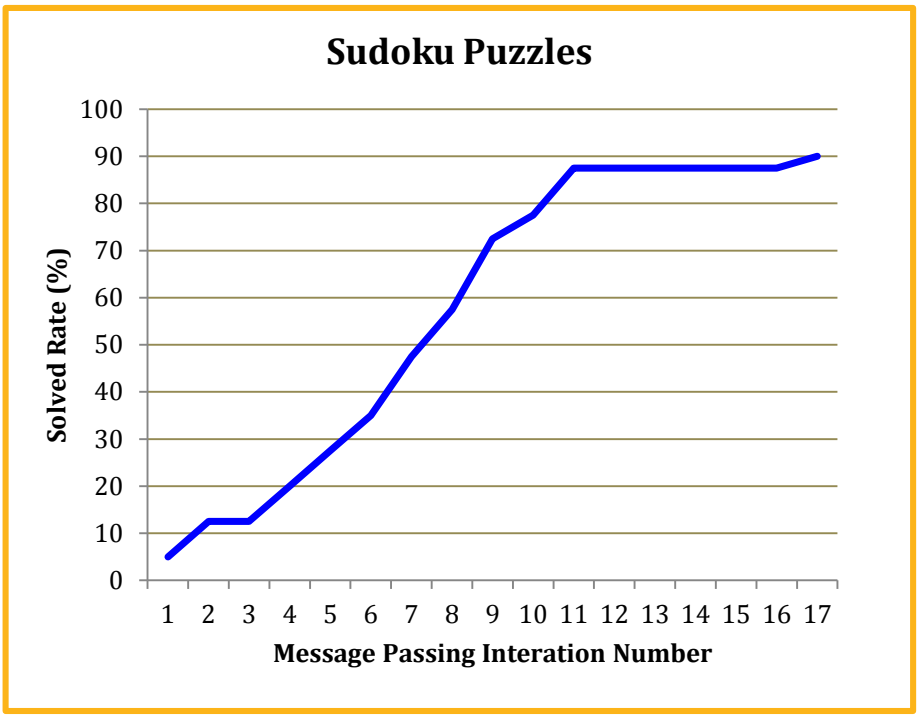

Figure 22. Experimental results for 40 9×9 hard sudoku puzzles.

7.2 Jigsaw Experimental Results

To test our implementation, I use problem instances with 100 patches and 400 patches partitioned from 200×200 pixels images respectively. The more patches the images are cut, the harder it becomes to get the correct result. I use the direct comparison to evaluate the result I get, as mentioned in chapter 5.

I encounter the overflow problem during implementation and use the mean method to deal with the overflow, as mentioned in chapter 6. Along with more rounds of MP, the accuracy of the reconstructed image will increase until the loopy belief propagation happens, as mentioned in chapter 6. I choose the peak value to be our final result. After testing, the

PGM-based algorithm is able to reconstruct 100 and 400 patch images with average correct rates, 83% and 30% respectively, as shown in Figure 23.

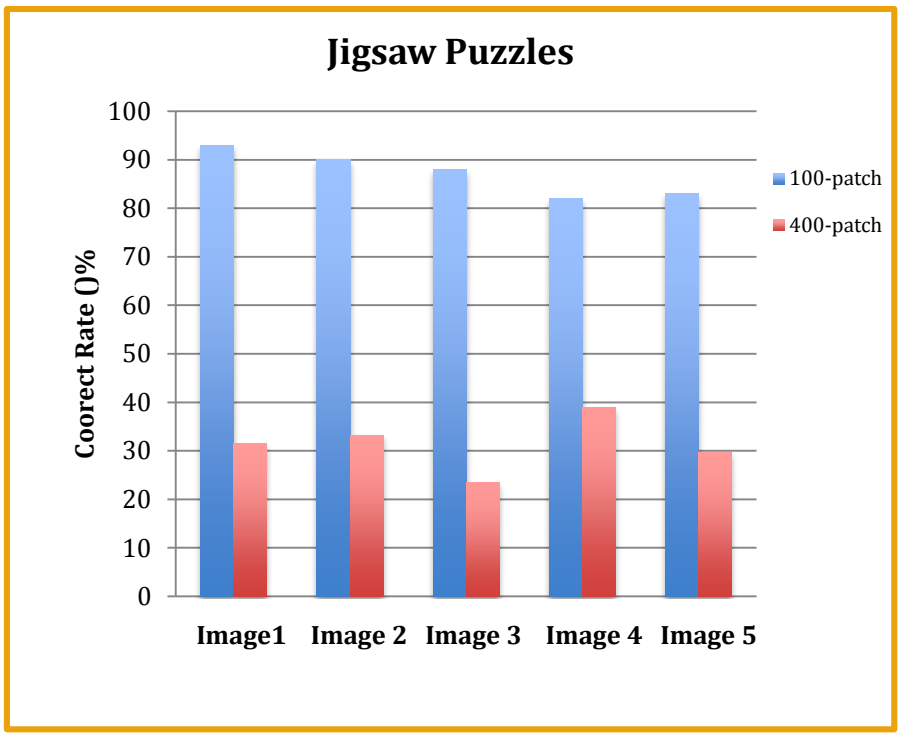

Figure 23. Experimental results for five 200×200 pixels images, with 100 and 400 patches respectively.

Both algorithms work well with our puzzle instances and show that PGMs can lead to good results even with loopy belief propagation. In the next chapter, I will conclude our work and explore new topics based on our results.

8. Conclusion and Future Work

8.1 Conclusion

In conclusion, this work showed how sudoku and jigsaw puzzle problems could be represented as PGMs and successfully demonstrated that the PGM-based algorithms rapidly find good solutions. In this work, I provide the data structures of our implementations and list the details of how I compute probabilistic messages to get to the solution step by step.

The sudoku puzzle problems represent a series of problems, which are constraint related, so the PGMs can also be applied to those problems as well. The patch compatibility I use in this work is proven to be valuable to evaluate the image edge-compatible. Moreover, I focus on square patches, thus making the reconstruction of puzzle problems with unique shapes to be more accurate. In this work, I combine the sum-product MP and logarithm to slow down the underflow and demonstrate that even with the loopy belief propagation, MP still works well for both puzzle problems. In addition, by applying the sum-product method, the time complexity is reduced from $O(N^M)$ to $O(N!\,N^4)$ for solving sudoku puzzle problems, where N is the size of the puzzle and M is the number of empty cells in the puzzle.

8.2 Future work

This work showed that along with more rounds of MP, more sudoku puzzles are solved. Some puzzles are first solved with five rounds of MP, while some other puzzles are first solved with 11 rounds of MP. An interesting question is: what is the number of MP rounds a specific sudoku puzzle needs? Similar to sudoku puzzle problems, the number of MP rounds needed for solving a specific jigsaw puzzle is unknown unless I test it with the program. I choose the peak value before the algorithm goes to loopy belief propagation. Therefore, finding the number of MP rounds, which makes the peak, becomes another interesting question.

To avoid the increasing impact of loopy propagation, applying Sinkhorn balancing to solve the two puzzle problems as mentioned in [4] and comparing with our current solution is a good recommendation.

9. References

[1] Charles A. Bouman. Markov Random Fields and Stochastic Image Models. IEEE, 74(4):532--551, 1986

[2] R. G. Gallager, "Low-Density Parity-Check Codes," IRE Trans. on Info. Theory, vol. IT-8, pp. 21–28, Jan. 1962.

[3] MOON, T. K., AND GUNTHER, J. H. Multiple constraint satisfaction by belief propagation: An example using sudoku. Adaptive and Learning Systems, 2006 IEEE Mountain Workshop on (2006), 122 – 126.

[4] Sheehan Khan, Shahab Jabbari. Solving sudoku Using Probabilistic Graphical Models. 2007.

[5] Taeg Sang Cho, Shai Avidan, William T. Freeman etc. A probabilistic image jigsaw puzzle solver. IEEE Conference 2010.

[6] E. D. Demaine and M. L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. Graphs and Combinatorics, 23, 2007.

[7] Y.-X. Zhao, M.-C. Su, Z.-L. Chou, and J. Lee. A puzzle solver and its application in speech descrambling. In ICCEA, 2007.

[8] B. J. Brown, C. Toler-Franklin, D. Nehab, M. Burns, D. Dobkin, A. Vlachopoulos, C. Doumas, and T. W. Szymon Rusinkiewicz. A system for high-volume acquisition and matching of fresco fragments: Reassembling Tehran wall paintings. ACM TOG (SIGGRAPH), 2008.

[9] L. Zhu, Z. Zhou, and D. Hu. Globally consistent reconstruction of ripped-up documents. IEEE TPAMI, 2008.

[10] Y. Weiss, "Correctness of Local Probability Propagation in Graphical Models with Loops," Neural Computation, vol. 12, pp. 1–41, 2000.

[11] Kevin Murphy. A Brief Introduction to Graphical Models and Bayesian Networks. 1998.

[12] T. S. Cho, M. Butman, S. Avidan, and W. T. Freeman. The patch transform and its applications to image editing. IEEE CVPR, 2008.

[13] Nghia Ho. Loopy belief propagation, Markov Random Field, stereo vision. 2012.

[14] Jack. Stanford course on Probabilistic Graphical Models. 2002.

[15] BONDY, J., AND MURTY, U. Graph Theory With Applications. North-Holland, 1982.

[16] YATO, T., AND SETA, T. Complexity and completeness of finding another solution and its application to puzzles. IEICE Trans Fundam Electron Commun Comput Sci E86-A, 5 (2003), 1052 – 1060.

[17] sudoku puzzle resource: http://mypuzzle.org/sudoku

[18] M. Makridis and N. Papamarkos. A new technique for solving a jigsaw puzzle. In IEEE ICIP, 2006.

[19] H. Freeman and L. Garder. Apictorial jigsaw puzzles: the computer solution of a problem in pattern recognition. IEEE TEC, (13):118–127, 1964.

[20] SIMONIS, H. Building industrial applications with constraint programming. Constraints in computational logics: theory and applications (2001), 271 – 309.

10. Appendix

10.1 Java Source Code for Solving Sudoku Puzzles

```
package sudoku;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.Date;

public class SudokuPGM {

	public static int PROPAGATION_Size = 0;
	public int[][] matrix = new int[9][9];
	public int[] S = new int[81]; // scan from first row, then second row
	public int[][] N_Current = new int[27][9];
	public int[][] N = new int[27][9];// [index of constraint][cell index]
	public int[][] M = new int[81][3];// [index of S][constraint index]
	public double[][] P = new double[81][9]; //
	public double[][][] R = new double[27][81][9];
	public double[][][] Q = new double[27][81][9];
	public double[][] R_P = new double[81][9];
	public int SUDOKU = 0;
	public int incompleteCount = 81;
	public int success = 0;
	public int fail = 0;

	public static void main(String args[]) {
		String pathname = "sudoku_left";
		for (int i = 18; i <= 40; i++) { // set the MP round size
			Date start = new Date();
			SudokuPGM sudoku = new SudokuPGM();
			SudokuPGM.PROPAGATION_Size = i;
			sudoku.read(pathname);
			System.out.println("\n\nSuccess: " + sudoku.success);
			System.out.println("Fail: " + sudoku.fail);
			Date end = new Date();
			System.out.println(end.getTime() - start.getTime());
		}
	}
	// initialize the matrix, S, N, M
	public void initialMatrix() {
		// read from file
		incompleteCount = 81;
		int index = 0;
		for (int i = 0; i < 27; i++) {
			for (int j = 0; j < 9; j++) {
				N_Current[i][j] = -1;
				N[i][j] = -1;
			}
		}
		for (int i = 0; i < 9; i++) {
			for (int j = 0; j < 9; j++) {
				S[index] = matrix[i][j];
				int boxN = getBoxNumber(i, j);
```

```
                M[index][0] = i;
                M[index][1] = j + 9;
                M[index][2] = boxN;

                addToN(index, i);
                addToN(index, j + 9);
                addToN(index, boxN);

                // only store the real values of index into its constraints N
                if (S[index] != 0) {
                        modifyN_Current(index);
                }
                for (int v = 0; v < 9; v++) {// initialize the probability of
                                                                        // 1-9 to 1
                        for (int C = 0; C < 3; C++) {
                                Q[M[index][C]][index][v] = Math.log10(1);
                                R[M[index][C]][index][v] = Math.log10(1);
                        }
                }
                index++;
        }
    }
}

public void read(String pathname) {
    try {
        System.out.println("MP Round: " + PROPAGATION_Size);
        File filename = new File(pathname);
        InputStreamReader reader = new InputStreamReader(
                new FileInputStream(filename));
        BufferedReader br = new BufferedReader(reader);

        String line = "";
        line = br.readLine();
        int i = 0;
        while (line != null) {
            if (!line.contains("#")) {
                String[] m = line.split(" ");
                for (int j = 0; j < 9; j++) {
                    matrix[i % 9][j] = Integer.parseInt(m[j]);
                }
                if (i % 9 == 8) {
                    SUDOKU += 1;
                    System.out.print("P" + SUDOKU + ": ");
                    initialMatrix();
                    int setCount = 1;// loop until there is no updating in
                                                        // one round
                    while (setCount > 0) {
                        setCount = eliminate();
                        // System.out.print("SetCount:"+setCount);
                    }
                    if (incompleteCount > 0) {
                        for (int n = 0; n < 81; n++) {
                            if (S[n] == 0)
                                getProbability(n);
                        }
                        propagation();
                        int go = 1;
                        int loop = incompleteCount;
                        for (int g = 1; g <= loop && go > 0; g++) {
                            go = guess();
                        }
                    }
                    if (incompleteCount == 0) {
                        success++;
                        System.out.println(" Success!");
```

```
                        } else {
                            fail++;
                            System.out.println(" Fail!    " + incompleteCount
                                    + " unset cells.");
                        }
                    }
                    i++;
                }
                line = br.readLine();
            }
            br.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
}

// return 1 if success; return -1 if fail
public int guess() {
        // printP();
        double maxP = Math.log10(0);
        int maxIndex = 0;
        // boolean flagg = true;
        for (int n = 0; n < 81; n++) {// find the max p one
            if (S[n] == 0) {
                for (int v = 0; v < 9; v++) {
                    if (P[n][v] > Math.log10(0) && P[n][v] < Math.log10(1)) {
                        if (P[n][v] > maxP) {
                            maxP = P[n][v];
                            maxIndex = n;
                        }
                    }
                }
            }
        }
        if (S[maxIndex] != 0) {
            return -1;
        }
        // go to deal with the max one
        double maxRP = Math.log10(0);
        int value = 0;
        int[] relatedCells = new int[20];
        int r_index = 0;
        for (int i = 0; i < 20; i++) {
            relatedCells[i] = -1;
        }
        for (int C = 0; C < 3; C++) {// in 3 constriants
            int m = M[maxIndex][C];
            for (int i = 0; i < 9; i++) {// 9 cells
                if (S[N[m][i]] == 0) {
                    boolean flag = true;
                    for (int ii = 0; ii <= (r_index - 1) && flag; ii++) {
                        if (relatedCells[ii] == N[m][i]) {
                            flag = false;
                        }
                    }
                    if (flag) {
                        relatedCells[r_index] = N[m][i];
                        r_index++;
                    }
                }
            }
        }
        // System.out.println("\nFor: " + maxIndex);
        for (int v = 0; v < 9; v++) {// find other possible values
            if (P[maxIndex][v] > Math.log10(0)) {// compute Relative probability
```

```
                double sum = Math.log10(0);
                int i = 0;
                while (relatedCells[i] != -1) {
                        sum = sum_log(sum, P[relatedCells[i]][v]);
                        i++;
                }
                R_P[maxIndex][v] = P[maxIndex][v] - sum;
                if (R_P[maxIndex][v] > maxRP) {
                        maxRP = R_P[maxIndex][v];
                        value = v + 1;
                }
        }
    }
    if (value == 0) {
        return -1;
    }
    S[maxIndex] = value;
    modifyN_Current(maxIndex);
    for (int v = 0; v < 9; v++) {// update the current cell
        P[maxIndex][v] = Math.log10(0);
    }
    for (int i = 0; i < 20; i++) {
        relatedCells[i] = -1;
    }
    r_index = 0;
    for (int C = 0; C < 3; C++) {// in 3 constriants
        int m = M[maxIndex][C];
        for (int i = 0; i < 9; i++) {// 9 cells
            if (S[N[m][i]] == 0 && P[N[m][i]][value - 1] > Math.log10(0)
                    && N[m][i] != maxIndex) { // right cell
                boolean flag = true;
                for (int ii = 0; ii <= (r_index - 1) && flag; ii++) {
                    if (relatedCells[ii] == N[m][i]) {
                        flag = false;
                    }
                }
                if (flag) {
                    relatedCells[r_index] = N[m][i];
                    r_index++;
                }
            }
        }
    }
    int i = 0;
    while (relatedCells[i] != -1) {
        int num = 0;
        for (int v = 0; v < 9; v++) { // possible values
            if (P[relatedCells[i]][v] > Math.log10(0)) {
                num++;
            }
        }
        double distribute = P[relatedCells[i]][value - 1]
                - Math.log10(num - 1);
        P[relatedCells[i]][value - 1] = Math.log10(0);// uodate that cell
        for (int v = 0; v < 9; v++) {
            if (P[relatedCells[i]][v] > Math.log10(0)) {
                sum_log(P[relatedCells[i]][v], distribute);
            }
        }
        i++;
    }
    return 1;
}

public void propagation() {
    for (int loop = 0; loop < PROPAGATION_Size; loop++) {
```

```
            messagePassing();
        }
}

public void messagePassing() {
        // R[m][n][x]=* # (1 - q[m][n'][value]) other 0 cell related in the same C
        for (int n = 0; n < 81; n++) {
                if (S[n] == 0) {
                        for (int x = 0; x < 9; x++) {
                                if (P[n][x] != Math.log10(0)) {// possible values!
                                        for (int C = 0; C < 3; C++) {
                                                int m = M[n][C];
                                                R[m][n][x] = Math.log10(1);
                                                String vp = getPossibleVP_R(m, n, x);
                                                String[] possibleVP = vp.split(",");
                                                String v = possibleVP[0];
                                                String[] p = possibleVP[1].trim().split(" ");
                                                double sum = permutate(n, "", v, p, Math.log10(0),
                                                                m);
                                                R[m][n][x] = sum;
                                        }// R
                                }
                        }
                }// S indexS == 0
        }// indexS

        // Q[m][n][x]= P(n=x) * # R[m'][n][x], m'= other two
        for (int n = 0; n < 81; n++) {
                if (S[n] == 0) {
                        for (int x = 0; x < 9; x++) {
                                if (P[n][x] != Math.log10(0)) {// possible values!
                                        for (int C = 0; C < 3; C++) {
                                                int m = M[n][C];
                                                Q[m][n][x] = P[n][x];// P(n=x)
                                                for (int C_other = 0; C_other < 3; C_other++) {
                                                        if (C_other != C) {
                                                                int m_2 = M[n][C_other];
                                                                Q[m][n][x] = Q[m][n][x] + R[m_2][n][x];
                                                        }
                                                }
                                                // Q[m][n][x] = Math.log10(Q[m][n][x]);
                                        }
                                }
                        }
                }// S indexS == 0
        }// indexS

        // P[n][x]
        for (int n = 0; n < 81; n++) {
                if (S[n] == 0) {
                        for (int x = 0; x < 9; x++) {
                                if (P[n][x] != Math.log10(0)) {// possible values!
                                        for (int C = 0; C < 3; C++) {
                                                P[n][x] = P[n][x] + R[M[n][C]][n][x];
                                        }
                                        // P[n][x] = Math.log10(P[n][x]);
                                }
                        }
                }// S indexS == 0
        }// indexS
}

public double permutate(int n, String pre, String last, String[] position,
                double sum, int m) {
        if (last.length() == 0) {
                double product = Math.log10(1);
```

```
            for (int i = 0; i < pre.length(); i++) {
                int value = Integer.parseInt(pre.substring(i, i + 1));
                int indexS = Integer.parseInt(position[i]);
                product = product + Q[m][indexS][value];
                // if(n==66){System.out.println("Permutation: "+indexS+":"+value+"    P:"+P[indexS][value]);}
            }
            sum = sum_log(sum, product);
            // if(n==66){System.out.println(product+" Product-SUM: "+sum);}
            return sum;
        }
        for (int i = 0; i < last.length(); i++) {
            sum = permutate(n, pre + last.substring(i, i + 1),
                    last.substring(0, i) + last.substring(i + 1), position,
                    sum, m);
        }
        return sum;
    }

    public String getPossibleVP_R(int m, int indexS, int value) {

        String result = "";
        for (int i = 0; i < 9; i++) {
            boolean flag = true;
            int index_NC = 0;
            while (N_Current[m][index_NC] != -1 && flag) {
                if ((i + 1) == S[N_Current[m][index_NC]]) {
                    flag = false;
                }
                index_NC++;
            }
            if (flag && i != value) {
                result += String.valueOf(i);
            }
        }
        result += ",";
        for (int i = 0; i < 9; i++) {
            boolean flag = true;
            int index_NC = 0;
            while (N_Current[m][index_NC] != -1 && flag) {
                if (N[m][i] == N_Current[m][index_NC]) {
                    flag = false;
                }
                index_NC++;
            }
            if (flag && N[m][i] != indexS) {
                result += String.valueOf(N[m][i]);
                result += " ";
            }
        }
        return result.trim();
    }

    // get probability from 3 constraints related to a node
    public void getProbability(int index) {

        for (int i = 0; i < 9; i++) {// initialize the probability of 1-9 to 1
            P[index][i] = Math.log10(1);
        }
        for (int C = 0; C < 3; C++) {
            getProbabilityFromEachConstraints_PRODUCT(index, M[index][C]);
        }
        int vCount = 0;
        for (int i = 0; i < 9; i++) {
            if (P[index][i] > Math.log10(0)) {
                vCount++;
            }
```

```
        }
        if (vCount != 0) {
            for (int i = 0; i < 9; i++) {
                P[index][i] = P[index][i] - Math.log10(vCount);
            }
        }
        getProbabilityForQ(index);
        // Q[[M[index][0]][index][0-8]=0;
    }

    // c1,c2: constraints number
    public void getProbabilityForQ(int indexS) {
        for (int C = 0; C < 3; C++) {
            for (int v = 0; v < 9; v++) {
                Q[M[indexS][C]][indexS][v] = Math.log10(1);
            }
            for (int C_other = 0; C_other < 3; C_other++) {
                if (C_other != C) { // only concern about other two constraints
                    getProbabilityFromEachConstraints_Q(indexS, M[indexS][C],
                            M[indexS][C_other]);
                }
            }
            int vCount = 0;
            for (int i = 0; i < 9; i++) {
                if (Q[M[indexS][C]][indexS][i] > Math.log10(0)) {
                    vCount++;
                }
            }
            if (vCount != 0) {
                for (int i = 0; i < 9; i++) {
                    Q[M[indexS][C]][indexS][i] = Q[M[indexS][C]][indexS][i]
                            - Math.log10(vCount);
                }
            }
        }
    }

    public void getProbabilityFromEachConstraints_Q(int index, int m,
            int realConstraintNum) {
        int num = 0;
        while (N_Current[realConstraintNum][num] != -1) {
            int indexS = N_Current[realConstraintNum][num];
            Q[m][index][S[indexS] - 1] = Math.log10(0);
            num++;
        }
    }

    public void getProbabilityFromEachConstraints_PRODUCT(int index,
            int constraintNum) {
        int num = 0;// get the number of values in the constraints
        while (N_Current[constraintNum][num] != -1) {
            int indexS = N_Current[constraintNum][num];// index of S
            // System.out.println(indexS + ": " + S[indexS]);
            P[index][S[indexS] - 1] = Math.log10(0);
            num++;
        }
    }

    // add value to all the constrains
    public void modifyN_Current(int index) {
        incompleteCount -= 1;
        for (int C = 0; C < 3; C++) {
            addToN_Current(M[index][C], index);
        }
    }
```

```
	public void addToN(int indexS, int N_number) {
		int index = 0;
		while (N[N_number][index] != -1) {// find the next place to put new
			index++;
		}
		N[N_number][index] = indexS;
	}

	// add real value to one constraint
	public void addToN_Current(int constraintNum, int indexS) {
		int index = 0;
		while (N_Current[constraintNum][index] != -1) {
			// System.out.println(SUDOKU+"
"+constraintNum+"]["+index+":"+N_Current[constraintNum][index]+
			// "Want to put"+indexS+":"+S[indexS]);
			index++;
		}
		N_Current[constraintNum][index] = indexS;
	}

	public int getBoxNumber(int i, int j) {
		int number = 0;
		if (i % 9 <= 2) {
			number = 18;
		} else if (i % 9 <= 5) {
			number = 21;
		} else {
			number = 24;
		}
		if (j % 9 >= 3 && j % 9 <= 5) {
			number += 1;
		} else if (j % 9 > 5) {
			number += 2;
		}
		return number;
	}

	public void print() {
		int e = 0;
		for (int i = 0; i < 9; i++) {
			for (int j = 0; j < 9; j++) {
				System.out.print(S[e++] + "	");
			}
			System.out.println();
		}
		System.out.println();
	}

	// loop to get the probability
	public int eliminate() {
		int setCount = 0; // count the number of values set in this round
		double maxP = Math.log10(0);
		int targetIndex = -1;
		int targetValue = 0;
		for (int i = 0; i < 81; i++) {
			if (S[i] == 0) {
				getProbability(i);
				int count = 0; // number of values with probability != 0
				int value = 0; // when count == 1, this value is unique
				for (int j = 0; j < 9; j++) {
					if (P[i][j] != Math.log10(0)) {
						if (P[i][j] > maxP) {
							maxP = P[i][j];
							targetIndex = i;
							targetValue = j + 1;
						}
```

```
                        count++;
                        value = j + 1;
                    }
                }
                if (count == 1) {
                    S[i] = value;
                    modifyN_Current(i);
                    setCount++;
                }
            }
        }
        return setCount;
    }

    public void printP() {
        for (int i = 0; i < 81; i++) {
            for (int v = 0; v < 9; v++) {
                if (S[i] == 0 && P[i][v] > Math.log10(0))
                    System.out.println("P[" + i + "][" + (v + 1) + "]" + " "
                            + Math.pow(10, P[i][v]));
            }
        }
    }

    public void printN() {
        System.out.println("Constraints: ");
        for (int i = 0; i < 27; i++) {
            System.out.print(i + "       ");
            for (int j = 0; j < 9; j++) {
                System.out.print(N[i][j] + "       ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public void printM() {
        System.out.println("Cells: ");
        for (int i = 0; i < 81; i++) {
            System.out.print(i + "       ");
            for (int j = 0; j < 3; j++) {
                System.out.print(M[i][j] + "      ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public double sum_log(double a, double b) {
        if (a == Double.NEGATIVE_INFINITY) {
            return b;
        } else if (b == Double.NEGATIVE_INFINITY) {
            return a;
        } else {
            double x, y, c = 0;
            if (a > b) {
                x = a;
                y = b;
            } else {
                x = b;
                y = a;
            }
            double decide = Math.pow(10, x - y);
            if ((decide + 1) == Double.POSITIVE_INFINITY) {// overflow
                c = x;
            } else {
```

```
                    decide += 1;
                    c = y + Math.log10(decide);
                }
                return c;
            }
        }
}
```

10.2 Java Source Code for Solving Jigsaw Puzzles

```
package jigsaw;
import java.awt.Color;
public class ColorSpaceConvert {
    public double[] D50 = {96.4212, 100.0, 82.5188};
    public double[] D55 = {95.6797, 100.0, 92.1481};
    public double[] D65 = {95.0429, 100.0, 108.8900};
    public double[] D75 = {94.9722, 100.0, 122.6394};
    public double[] whitePoint = D65;
    public double[] chromaD50 = {0.3457, 0.3585, 100.0};
    public double[] chromaD55 = {0.3324, 0.3474, 100.0};
    public double[] chromaD65 = {0.3127, 0.3290, 100.0};
    public double[] chromaD75 = {0.2990, 0.3149, 100.0};
    public double[] chromaWhitePoint = chromaD65;
    public double[][] M     = {{0.4124, 0.3576,    0.1805},
                               {0.2126, 0.7152,    0.0722},
                               {0.0193, 0.1192,    0.9505}};
    public double[][] Mi    = {{ 3.2406, -1.5372, -0.4986},
                               {-0.9689,    1.8758,    0.0415},
                               { 0.0557, -0.2040,    1.0570}};

    public ColorSpaceConvert() {
       whitePoint = D65;
       chromaWhitePoint = chromaD65;
    }

    public ColorSpaceConvert(String white) {
       whitePoint = D65;
       chromaWhitePoint = chromaD65;
       if (white.equalsIgnoreCase("d50")) {
          whitePoint = D50;
          chromaWhitePoint = chromaD50;
       }
       else if (white.equalsIgnoreCase("d55")) {
          whitePoint = D55;
          chromaWhitePoint = chromaD55;
       }
       else if (white.equalsIgnoreCase("d65")) {
          whitePoint = D65;
          chromaWhitePoint = chromaD65;
       }
       else if (white.equalsIgnoreCase("d75")) {
          whitePoint = D75;
          chromaWhitePoint = chromaD75;
       }
    }

    /**
     * @param H Hue angle/360 (0..1)
     * @param S Saturation (0..1)
     * @param B Value (0..1)
     * @return RGB values
     */
```

```
public int[] HSBtoRGB(double H, double S, double B) {
    int[] result = new int[3];
    int rgb = Color.HSBtoRGB((float) H, (float) S, (float) B);
    result[0] = (rgb >> 16) & 0xff;
    result[1] = (rgb >> 8) & 0xff;
    result[2] = (rgb >> 0) & 0xff;
    return result;
}

public int[] HSBtoRGB(double[] HSB) {
    return HSBtoRGB(HSB[0], HSB[1], HSB[2]);
}

/**
  * Convert LAB to RGB.
  * @param L
  * @param a
  * @param b
  * @return RGB values
  */
public int[] LABtoRGB(double L, double a, double b) {
    return XYZtoRGB(LABtoXYZ(L, a, b));
}

/**
  * @param Lab
  * @return RGB values
  */
public int[] LABtoRGB(double[] Lab) {
    return XYZtoRGB(LABtoXYZ(Lab));
}

public double[] LABtoXYZ(double L, double a, double b) {
    double[] result = new double[3];

    double y = (L + 16.0) / 116.0;
    double y3 = Math.pow(y, 3.0);
    double x = (a / 500.0) + y;
    double x3 = Math.pow(x, 3.0);
    double z = y - (b / 200.0);
    double z3 = Math.pow(z, 3.0);

    if (y3 > 0.008856) {
        y = y3;
    }
    else {
        y = (y - (16.0 / 116.0)) / 7.787;
    }
    if (x3 > 0.008856) {
        x = x3;
    }
    else {
        x = (x - (16.0 / 116.0)) / 7.787;
    }
    if (z3 > 0.008856) {
        z = z3;
    }
    else {
        z = (z - (16.0 / 116.0)) / 7.787;
    }

    result[0] = x * whitePoint[0];
    result[1] = y * whitePoint[1];
    result[2] = z * whitePoint[2];

    return result;
```

```
}
public double[] LABtoXYZ(double[] Lab) {
   return LABtoXYZ(Lab[0], Lab[1], Lab[2]);
}

public double[] RGBtoHSB(int R, int G, int B) {
   double[] result = new double[3];
   float[] hsb = new float[3];
   Color.RGBtoHSB(R, G, B, hsb);
   result[0] = hsb[0];
   result[1] = hsb[1];
   result[2] = hsb[2];
   return result;
}

public double[] RGBtoHSB(int[] RGB) {
   return RGBtoHSB(RGB[0], RGB[1], RGB[2]);
}
public double[] RGBtoLAB(int R, int G, int B) {
   return XYZtoLAB(RGBtoXYZ(R, G, B));
}
public double[] RGBtoLAB(int[] RGB) {
   return XYZtoLAB(RGBtoXYZ(RGB));
}


public double[] RGBtoXYZ(int R, int G, int B) {
   double[] result = new double[3];
   // convert 0..255 into 0..1
   double r = R / 255.0;
   double g = G / 255.0;
   double b = B / 255.0;

   // assume sRGB
   if (r <= 0.04045) {
      r = r / 12.92;
   }
   else {
      r = Math.pow(((r + 0.055) / 1.055), 2.4);
   }
   if (g <= 0.04045) {
      g = g / 12.92;
   }
   else {
      g = Math.pow(((g + 0.055) / 1.055), 2.4);
   }
   if (b <= 0.04045) {
      b = b / 12.92;
   }
   else {
      b = Math.pow(((b + 0.055) / 1.055), 2.4);
   }

   r *= 100.0;
   g *= 100.0;
   b *= 100.0;

   // [X Y Z] = [r g b][M]
   result[0] = (r * M[0][0]) + (g * M[0][1]) + (b * M[0][2]);
   result[1] = (r * M[1][0]) + (g * M[1][1]) + (b * M[1][2]);
   result[2] = (r * M[2][0]) + (g * M[2][1]) + (b * M[2][2]);

   return result;
}
public double[] RGBtoXYZ(int[] RGB) {
   return RGBtoXYZ(RGB[0], RGB[1], RGB[2]);
```

```
}
public double[] xyYtoXYZ(double x, double y, double Y) {
   double[] result = new double[3];
   if (y == 0) {
      result[0] = 0;
      result[1] = 0;
      result[2] = 0;
   }
   else {
      result[0] = (x * Y) / y;
      result[1] = Y;
      result[2] = ((1 - x - y) * Y) / y;
   }
   return result;
}

public double[] xyYtoXYZ(double[] xyY) {
   return xyYtoXYZ(xyY[0], xyY[1], xyY[2]);
}
public double[] XYZtoLAB(double X, double Y, double Z) {
   double x = X / whitePoint[0];
   double y = Y / whitePoint[1];
   double z = Z / whitePoint[2];
   if (x > 0.008856) {
      x = Math.pow(x, 1.0 / 3.0);
   }
   else {
      x = (7.787 * x) + (16.0 / 116.0);
   }
   if (y > 0.008856) {
      y = Math.pow(y, 1.0 / 3.0);
   }
   else {
      y = (7.787 * y) + (16.0 / 116.0);
   }
   if (z > 0.008856) {
      z = Math.pow(z, 1.0 / 3.0);
   }
   else {
      z = (7.787 * z) + (16.0 / 116.0);
   }
   double[] result = new double[3];
   result[0] = (116.0 * y) - 16.0;
   result[1] = 500.0 * (x - y);
   result[2] = 200.0 * (y - z);

   return result;
}
public double[] XYZtoLAB(double[] XYZ) {
   return XYZtoLAB(XYZ[0], XYZ[1], XYZ[2]);
}

public int[] XYZtoRGB(double X, double Y, double Z) {
   int[] result = new int[3];

   double x = X / 100.0;
   double y = Y / 100.0;
   double z = Z / 100.0;

   // [r g b] = [X Y Z][Mi]
   double r = (x * Mi[0][0]) + (y * Mi[0][1]) + (z * Mi[0][2]);
   double g = (x * Mi[1][0]) + (y * Mi[1][1]) + (z * Mi[1][2]);
   double b = (x * Mi[2][0]) + (y * Mi[2][1]) + (z * Mi[2][2]);

   // assume sRGB
   if (r > 0.0031308) {
```

```
            r = ((1.055 * Math.pow(r, 1.0 / 2.4)) - 0.055);
        }
        else {
            r = (r * 12.92);
        }
        if (g > 0.0031308) {
            g = ((1.055 * Math.pow(g, 1.0 / 2.4)) - 0.055);
        }
        else {
            g = (g * 12.92);
        }
        if (b > 0.0031308) {
            b = ((1.055 * Math.pow(b, 1.0 / 2.4)) - 0.055);
        }
        else {
            b = (b * 12.92);
        }
        r = (r < 0) ? 0 : r;
        g = (g < 0) ? 0 : g;
        b = (b < 0) ? 0 : b;
        // convert 0..1 into 0..255
        result[0] = (int) Math.round(r * 255);
        result[1] = (int) Math.round(g * 255);
        result[2] = (int) Math.round(b * 255);
        return result;
    }
    public int[] XYZtoRGB(double[] XYZ) {
        return XYZtoRGB(XYZ[0], XYZ[1], XYZ[2]);
    }
    public double[] XYZtoxyY(double X, double Y, double Z) {
        double[] result = new double[3];
        if ((X + Y + Z) == 0) {
            result[0] = chromaWhitePoint[0];
            result[1] = chromaWhitePoint[1];
            result[2] = chromaWhitePoint[2];
        }
        else {
            result[0] = X / (X + Y + Z);
            result[1] = Y / (X + Y + Z);
            result[2] = Y;
        }
        return result;
    }
    public double[] XYZtoxyY(double[] XYZ) {
        return XYZtoxyY(XYZ[0], XYZ[1], XYZ[2]);
    }

    public static void main(String args[]){
     ColorSpaceConvert csc = new ColorSpaceConvert("D65");
     csc.RGBtoLAB(110, 71, 146);
     }
  }

package jigsaw;

//Use Low resolution picture, no log      M/PatcchNUmber
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import javax.imageio.ImageIO;
//    http://html-color-codes.info/
```

```
//RGB # color
//http://www.colorspire.com/rgb-color-wheel/
//online RGB LAB
//http://www.easyrgb.com/index.php?X=CALC#Result

public class JigsawPGM {
	public int MP_Round = 3;
	public int Current_MP = 0;
	public String fileName = "100_0.jpg";// 200_1.jpg";
	public String fileName_low = "100_00.jpg";
	public String newFileName = "1_new";
	// public int logSIZE = 10;

	public int HEIGHT;
	public int WIDTH;
	public int Pixels;
	public int PATCH_SIZE = 25;

	public int PATCH_NUMBER;
	public int ROW_SIZE;// 20patches in each row
	public int COL_SIZE;// 20 patches in each column
	public LAB[][] LABData;
	// public L
	public int[][] RGBInteger;
	public int[][] RGBInteger_low;
	public BufferedImage image = null;
	public BufferedImage image_low = null;
	public int N[][]; // up down left right
	public static double M_ji_Xi_0[][][]; // Message: v- Neighbor Relation
	                                      // (N[i][v]=j) -- Current Location
	                                      // i-- Current Patch Xi
	public static double M_ji_Xi_1[][][];
	public static double P_ji[][][]; // patch dissimilarity: patch j, patch i,
	                                 // relation: patch j is at the UDLR of
	                                 // patch i
	public int LOCATION_STATE[]; // [location]=patch,
	public boolean PATCH_STATE[];
	public Queue<Integer> Guess = new LinkedList<Integer>();
	public double P_YX[][];
	public int OverFlow = 0;

	// public double [][][] original ;
	public void init() {

		try {
			image = ImageIO.read(new File(fileName));// ye 11851.jpg
			                                         // 10375/01.jpg
			image_low = ImageIO.read(new File(fileName_low));

			HEIGHT = image.getHeight();
			WIDTH = image.getWidth();
			Pixels = HEIGHT * WIDTH;

			// PATCH_SIZE = getGCD(HEIGHT, WIDTH);
			// PATCH_SIZE = 50;
			ROW_SIZE = WIDTH / PATCH_SIZE;
			COL_SIZE = HEIGHT / PATCH_SIZE;
			PATCH_NUMBER = ROW_SIZE * COL_SIZE;
			// PATCH_ROW =
			N = new int[PATCH_NUMBER][4];
			System.out.println(HEIGHT + " " + WIDTH + ": " + PATCH_NUMBER);
			System.out.println("MP: " + MP_Round);
			M_ji_Xi_0 = new double[4][PATCH_NUMBER][PATCH_NUMBER];
			M_ji_Xi_1 = new double[4][PATCH_NUMBER][PATCH_NUMBER];
			P_ji = new double[PATCH_NUMBER][PATCH_NUMBER][4];
			LOCATION_STATE = new int[PATCH_NUMBER];
```

```
            PATCH_STATE = new boolean[PATCH_NUMBER];
            P_YX = new double[PATCH_NUMBER][PATCH_NUMBER];

            initNeighbor();

            // System.out.println(image_low.getHeight()+" "+image_low.getWidth());
            RGBInteger = new int[HEIGHT][WIDTH];
            RGBInteger_low = new int[HEIGHT][WIDTH];
            LABData = new LAB[HEIGHT][WIDTH];// =new
                                                    // String[image.getHeight()][image.getWidth()];
                                                    // // where we'll put the image
            ColorSpaceConvert csc = new ColorSpaceConvert("D65");
            String HexString = "";
            for (int i = 0; i < HEIGHT; i++) {
                  for (int j = 0; j < WIDTH; j++) {
                        RGBInteger[i][j] = image.getRGB(j, i);
                        RGBInteger_low[i][j] = image_low.getRGB(j, i);
                        HexString = Integer.toHexString(RGBInteger[i][j]);
                        int R = Integer.parseInt(HexString.substring(2, 4), 16);
                        int G = Integer.parseInt(HexString.substring(4, 6), 16);
                        int B = Integer.parseInt(HexString.substring(6, 8), 16);
                        LABData[i][j] = new LAB();
                        LABData[i][j].lab = csc.RGBtoLAB(R, G, B);
                  }
            }

            YCompatibility_RGB();
            setAnchorPatch();
            patchCompatibility();
            process();

      } catch (IOException e) {
            System.out.println("No file");
      }
}

public void process() {
      for (int d = 0; d < 4; d++) {
            for (int i = 0; i < PATCH_NUMBER; i++) {
                  for (int Xi = 0; Xi < PATCH_NUMBER; Xi++) {
                        M_ji_Xi_0[d][i][Xi] = 0;
                        M_ji_Xi_1[d][i][Xi] = 0;
                  }
            }
      }
      Current_MP = 0;

      while (Current_MP <= MP_Round) {
            initialMessage(Current_MP == 0 ? false : true);
            setAnchorPatch();
            guess();
            // permutationGuess();
            File newFile = new File(newFileName + Current_MP + ".jpg");
            try {
                  ImageIO.write(image, "JPEG", newFile);
            } catch (IOException e) {
                  e.printStackTrace();
            }
            Current_MP++;
      }
}

public void setAnchorPatch() {
      for (int i = 0; i < PATCH_NUMBER; i++) {
            LOCATION_STATE[i] = -1;
            PATCH_STATE[i] = false;
```

```
        }
    }

    public void initNeighbor() {
        for (int i = 0; i < PATCH_NUMBER; i++) {

            N[i][0] = (i - ROW_SIZE >= 0) ? (i - ROW_SIZE) : -1;
            N[i][1] = (i + ROW_SIZE < PATCH_NUMBER) ? (i + ROW_SIZE) : -1;

            N[i][2] = (i % ROW_SIZE == 0) ? -1 : i - 1;
            N[i][3] = ((i + 1) % ROW_SIZE == 0) ? -1 : i + 1;
        }
    }

    // M = SUM(P_ji),j for 399
    public void initialMessage(boolean not_init) {
        // M_ji_Xi[][][]: Message: v- Neighbor Relation (N[i][v]=j) -- Current
        // Location i-- Current Patch Xi
        int SIZE = 8;
        double[] sum = new double[SIZE];

        double[][][] M0 = M_ji_Xi_0, M1 = M_ji_Xi_1;
        if (Current_MP % 2 == 1) {
            M0 = M_ji_Xi_1;
            M1 = M_ji_Xi_0;
        }
        for (int i = 0; i < PATCH_NUMBER; i++) {
            for (int Xi = 0; Xi < PATCH_NUMBER; Xi++) {
                for (int d = 0; d < 4; d++) {
                    M0[d][i][Xi] = 0;// location j = N[i][d];
                    for (int s = 0; s < SIZE; s++) {// initial sum
                        sum[s] = 0;
                    }
                    if (N[i][d] != -1) {
                        for (int Xj = 0; Xj < PATCH_NUMBER; Xj++) {// PATCH
                            double P = P_ji[Xj][Xi][d] * P_YX[N[i][d]][Xj]; //
                            if (not_init) {// Real MP
                                for (int dd = 0; dd < 4; dd++) {// location j =
                                                                // N[i][d];
                                    int l = N[N[i][d]][dd];
                                    if (l != -1 && l != i) {
                                        P = P * M1[dd][N[i][d]][Xj];
                                    }
                                }
                            }
                            int index = Xj % SIZE;

                            sum[index] = sum[index] + P;
                        }// Xj
                        for (int s = 0; s < 4; s++) {
                            sum[s] = sum[s] + sum[s + 4];
                        }
                        M0[d][i][Xi] = (sum[0] + sum[1] + sum[2] + sum[3])
                                / PATCH_NUMBER;
                    }
                }// neighbor
            }// Xi

        }// i

    }

    public void iniGuess() {
        // Guess.add(0);
        for (int i = 0; i < PATCH_NUMBER; i++) {
            if (LOCATION_STATE[i] != -1) {
```

```
                for (int d = 0; d < 4; d++) {
                    if (N[i][d] != -1 && LOCATION_STATE[N[i][d]] == -1)
                        Guess.add(N[i][d]);
                }
            }
        }
    }

    // bi(xi) = II j (m_ji_Xi)
    public void guess() {
        // iniGuess();
        int count = 0;
        int misDistance = 0;
        int total = 0;
        double M[][][] = M_ji_Xi_0;
        if (Current_MP % 2 == 1) {
            M = M_ji_Xi_1;
        }
        // for(int i =0;i<PATCH_NUMBER;i++){//location
        while (total < PATCH_NUMBER) {
            // int i=Guess.poll();
            double maxB = 0;
            int maxPatch = 0;
            int maxLocation = 0;
            for (int i = 0; i < PATCH_NUMBER; i++) {
                if (LOCATION_STATE[i] != -1) {
                    continue;
                }
                for (int Xi = 0; Xi < PATCH_NUMBER; Xi++) {
                    if (PATCH_STATE[Xi]) {
                        continue;
                    }
                    double II = P_YX[i][Xi];
                    for (int d = 0; d < 4; d++) {
                        if (N[i][d] != -1) {
                            II = II + M[d][i][Xi];
                        }
                    }

                    if (II > P_YX[i][Xi] && II >= maxB) {
                        maxB = II;
                        maxPatch = Xi;
                        maxLocation = i;
                    }
                }
            }
            // System.out.println(maxLocation+": "+maxPatch+" "+maxB);
            total++;
            int distance = Math.abs(maxLocation - maxPatch);
            // misDistance+=distance;
            if (maxB > 0 && distance == 0) {
                count++;
                System.out.println("Correct:-- " + maxLocation);
            } else {
                System.out.println(maxLocation + ": " + maxPatch);
            }
            setPatch(maxLocation, maxPatch);
            PATCH_STATE[maxPatch] = true;
            LOCATION_STATE[maxLocation] = maxPatch;

        }
        System.out.println(Current_MP + "  " + count + "     " + converge());
    }

    public void permutationGuess() {
        maxG = 0;
```

```
        maxGuess = new int[PATCH_NUMBER];
        int[] GuessPatch = new int[PATCH_NUMBER];
        int[] patchState = new int[PATCH_NUMBER];
        double[][] B = new double[PATCH_NUMBER][PATCH_NUMBER];
        double M[][][] = M_ji_Xi_0;
        if (Current_MP % 2 == 1) {
            M = M_ji_Xi_1;
        }
        for (int i = 0; i < PATCH_NUMBER; i++) {
            for (int Xi = 0; Xi < PATCH_NUMBER; Xi++) {
                B[i][Xi] = P_YX[i][Xi];
                for (int d = 0; d < 4; d++) {
                    if (N[i][d] != -1) {
                        B[i][Xi] += M[d][i][Xi];
                    }
                }
            }
        }
        int count = 0;
        permutationGuess2(0, B, patchState, GuessPatch);
        for (int i = 0; i < PATCH_NUMBER; i++) {
            setPatch(i, maxGuess[i]);
            if (i == maxGuess[i]) {
                count++;
            }
        }
        System.out.print(count + " out of! ");

    }

    public double maxG = 0;
    public int[] maxGuess;

    public void permutationGuess2(int cur, double[][] B, int[] patchState,
            int[] GuessPatch) {
        if (cur == PATCH_NUMBER) {
            double R = 1;
            for (int i = 0; i < PATCH_NUMBER; i++) {
                R *= B[i][GuessPatch[i]];
            }
            if (R > maxG) {
                maxG = R;
                System.out.print(maxG + ": ");
                for (int i = 0; i < PATCH_NUMBER; i++) {
                    System.out.print(GuessPatch[i]);
                    maxGuess[i] = GuessPatch[i];
                }
                System.out.println();
            }
            return;
        }
        for (int i = 0; i < PATCH_NUMBER; i++) {
            if (patchState[i] != 1) {
                GuessPatch[cur] = i;
                patchState[i] = 1;
                permutationGuess2(cur + 1, B, patchState, GuessPatch);
                patchState[i] = 0;
            }
        }
        return;
    }

    public double converge() {
        double sum = 0;
        for (int i = 0; i < PATCH_NUMBER; i++) {
            for (int j = 0; j < PATCH_NUMBER; j++) {
```

```
                for (int n = 0; n < 4; n++) {
                    sum += Math.abs(M_ji_Xi_1[n][i][j] - M_ji_Xi_0[n][i][j]);
                }
            }
        }
        return sum;
    }

    public void print(double[][][] m) {
        for (int i = 0; i < 1; i++) {
            for (int j = 0; j < PATCH_NUMBER; j++) {
                System.out.print(i + ":      " + j + ":   ");
                System.out.println(m[0][i][j] + "       " + m[1][i][j] + "       "
                        + m[2][i][j] + " " + m[3][i][j] + ";      ");
            }
            System.out.println();
            System.out.println();
        }
    }

    // test if the best neighbor is right according to the original picture
    public void test() {
        int count = 0, total = 0;
        for (int i = 0; i < PATCH_NUMBER; i++) {
            System.out.print(i + ":    ");
            for (int d = 0; d < 4; d++) {
                double max = Double.MIN_VALUE;
                int minJ = -1;
                for (int j = 0; j < PATCH_NUMBER; j++) {
                    // if(j!=i)
                    if (P_ji[j][i][d] > max) {
                        max = P_ji[j][i][d];
                        minJ = j;
                    }
                }
                if (N[i][d] != -1)
                    total++;
                if (minJ == N[i][d])
                    count++;
                System.out.print(N[i][d] + "-" + minJ + "          ");
            }
            System.out.println();
        }
        System.out.println("Count: " + count + " / " + total);
    }

    public static int getGCD(int m, int n) {
        while (m % n != 0) {
            int temp = m % n;
            m = n;
            n = temp;
        }
        return n;
    }

    public void patchCompatibility() {// Up Down Left Right
        double Min[] = new double[4], Min2[] = new double[4], a[] = new double[4];
        for (int i = 0; i < PATCH_NUMBER; i++) {// for patch i
            for (int d = 0; d < 4; d++) {
                Min[d] = Double.MAX_VALUE;
                Min2[d] = Double.MAX_VALUE;
            }
            for (int j = 0; j < PATCH_NUMBER; j++) {
                if (j == i) {
                    for (int r = 0; r < 4; r++)
                        P_ji[j][i][r] = 0;
```

```
                } else {
                        P_ji[j][i][0] = DUD_DDU(j, i, true);
                        P_ji[j][i][1] = DUD_DDU(j, i, false);
                        P_ji[j][i][2] = DLR_DRL(j, i, true);
                        P_ji[j][i][3] = DLR_DRL(j, i, false);
                        for (int d = 0; d < 4; d++) {
                                if (P_ji[j][i][d] < Min[d]) {
                                        Min2[d] = Min[d];
                                        Min[d] = P_ji[j][i][d];
                                } else if (P_ji[j][i][d] > Min[d]
                                                && P_ji[j][i][d] < Min2[d]) {
                                        Min2[d] = P_ji[j][i][d];
                                }
                        }
                }
        }// j
        for (int d = 0; d < 4; d++)
                a[d] = 2 * (Min2[d] - Min[d]) * (Min2[d] - Min[d]);
        for (int j = 0; j < PATCH_NUMBER; j++) {
                if (j != i)
                        for (int d = 0; d < 4; d++) {
                                P_ji[j][i][d] = Math.exp((P_ji[j][i][d] / a[d]) * (-1));
                        }
        }
    }
}

public void YCompatibility_RGB() {
    double RGB_Y[][] = new double[PATCH_NUMBER][3];// Mean color of R G B
    double RGB_X[][] = new double[PATCH_NUMBER][3];

    for (int i = 0; i < PATCH_NUMBER; i++) {
        int row = i / ROW_SIZE * PATCH_SIZE, col = i % ROW_SIZE
                * PATCH_SIZE;
        for (int index = 0; index < 3; index++) {
            RGB_Y[i][index] = 0;
            RGB_X[i][index] = 0;
        }
        String HexS;
        for (int r = 0; r < PATCH_SIZE; r++) {
            for (int c = 0; c < PATCH_SIZE; c++) {
                HexS = Integer.toHexString(RGBInteger[row + r][col + c]);
                RGB_X[i][0] += Integer.parseInt(HexS.substring(2, 4), 16);
                RGB_X[i][1] += Integer.parseInt(HexS.substring(4, 6), 16);
                RGB_X[i][2] += Integer.parseInt(HexS.substring(6, 8), 16);
                HexS = Integer
                        .toHexString(RGBInteger_low[row + r][col + c]);
                RGB_Y[i][0] += Integer.parseInt(HexS.substring(2, 4), 16);
                RGB_Y[i][1] += Integer.parseInt(HexS.substring(4, 6), 16);
                RGB_Y[i][2] += Integer.parseInt(HexS.substring(6, 8), 16);
            }
        }
        // System.out.println(i+" -- "+RGB_Y[i]+":      "+RGB_X[i]);
        for (int index = 0; index < 3; index++) {
            RGB_Y[i][index] = RGB_Y[i][index] / (PATCH_SIZE * PATCH_SIZE)
                    / 100;
            RGB_X[i][index] = RGB_X[i][index] / (PATCH_SIZE * PATCH_SIZE)
                    / 100;
        }
    }
    int fitCount = 0;
    for (int Y = 0; Y < PATCH_NUMBER; Y++) {
        double maxP = Double.NEGATIVE_INFINITY;
        int maxX = -1;
        for (int X = 0; X < PATCH_NUMBER; X++) {
            double sumS = 0;
```

```
                for (int i = 0; i < 3; i++) {
                        sumS += (RGB_Y[Y][i] - RGB_X[X][i])
                                        * (RGB_Y[Y][i] - RGB_X[X][i]);
                }
                P_YX[Y][X] = Math.exp(sumS * (-1) / (2 * 0.16));
                // if(Y==15)System.out.println(P_YX[Y][X]);
                // P_YX[Y][X] = Log(P_YX[Y][X]);

                if (P_YX[Y][X] > maxP) {
                        maxP = P_YX[Y][X];
                        maxX = X;
                }
        }

        if (Y == maxX) {// System.out.println("Fit: "+Y);
                fitCount++;
        }// System.out.println(Y+" Max fit Patch:-- "+maxX);
    }
    System.out.println("Fit Count:-- " + fitCount);
}

public void YCompatibility() {
    double RGB_Y[] = new double[PATCH_NUMBER];
    double RGB_X[] = new double[PATCH_NUMBER];

    for (int i = 0; i < PATCH_NUMBER; i++) {
        int row = i / ROW_SIZE * PATCH_SIZE, col = i % ROW_SIZE
                * PATCH_SIZE;
        RGB_Y[i] = 0;
        RGB_X[i] = 0;
        for (int r = 0; r < PATCH_SIZE; r++) {
            for (int c = 0; c < PATCH_SIZE; c++) {
                RGB_X[i] += Integer.parseInt(
                        Integer.toHexString(RGBInteger[row + r][col + c])
                                .substring(2, 8), 16);
                RGB_Y[i] += Integer.parseInt(
                        Integer.toHexString(
                                RGBInteger_low[row + r][col + c])
                                .substring(2, 8), 16);
            }
        }
        // System.out.println(i+" -- "+RGB_Y[i]+":      "+RGB_X[i]);
        RGB_Y[i] = RGB_Y[i] / (PATCH_SIZE * PATCH_SIZE) / 1000000;
        RGB_X[i] = RGB_X[i] / (PATCH_SIZE * PATCH_SIZE) / 1000000;

    }
    int fitCount = 0;
    for (int Y = 0; Y < PATCH_NUMBER; Y++) {
        double maxP = Double.NEGATIVE_INFINITY;
        int maxX = -1;
        for (int X = 0; X < PATCH_NUMBER; X++) {
            P_YX[Y][X] = Math.exp((RGB_Y[Y] - RGB_X[X])
                    * (RGB_Y[Y] - RGB_X[X]) * (-1) / (2 * 0.16));
            if (Y == 6)
                System.out.println(Y + " -- " + X + ":       "
                        + (RGB_Y[Y] - RGB_X[X]) * (RGB_Y[Y] - RGB_X[X])
                        * (-1) / (2 * 0.16) + "      " + P_YX[Y][X]);
            // P_YX[Y][X] = Log(P_YX[Y][X]);

            if (P_YX[Y][X] > maxP) {
                maxP = P_YX[Y][X];
                maxX = X;
            }
        }
        if (Y == maxX)
            fitCount++;// System.out.println(Y+" Max fit Patch:-- "+maxX);
```

```
        }
        System.out.println("Fit Count:-- " + fitCount);
}

// //i j:patch locations, last col in J, first col in i
public double DLR_DRL(int j, int i, Boolean Left) {
        int rowJ, rowI, colJ, colI;
        if (Left) { // j is the left Neighbor of i: DLR
                rowJ = j / ROW_SIZE * PATCH_SIZE;
                colJ = j % ROW_SIZE * PATCH_SIZE + (PATCH_SIZE - 1);
                rowI = i / ROW_SIZE * PATCH_SIZE;
                colI = i % ROW_SIZE * PATCH_SIZE;
        } else {// j is the right Neighbor of i: DRL
                rowJ = j / ROW_SIZE * PATCH_SIZE;
                colJ = j % ROW_SIZE * PATCH_SIZE;
                rowI = i / ROW_SIZE * PATCH_SIZE;
                colI = i % ROW_SIZE * PATCH_SIZE + (PATCH_SIZE - 1);
        }
        // System.out.println(rowJ +" "+ colJ);
        double result = 0;
        for (int k = 0; k < PATCH_SIZE; k++) {
                for (int l = 0; l < 3; l++) {
                        result += (LABData[rowJ + k][colJ].lab[l] - LABData[rowI + k][colI].lab[l])
                                        * (LABData[rowJ + k][colJ].lab[l] - LABData[rowI + k][colI].lab[l]);
                }
        }
        // System.out.print(j+": "+result);
        return result;
}

public double DUD_DDU(int j, int i, Boolean Up) {
        int rowJ, rowI, colJ, colI;
        if (Up) { // j is the UP Neighbor of i: DUD
                rowJ = j / ROW_SIZE * PATCH_SIZE + (PATCH_SIZE - 1);
                colJ = j % ROW_SIZE * PATCH_SIZE;
                rowI = i / ROW_SIZE * PATCH_SIZE;
                colI = i % ROW_SIZE * PATCH_SIZE;
        } else {// j is the DOWN Neighbor of i: DDU
                rowJ = j / ROW_SIZE * PATCH_SIZE;
                colJ = j % ROW_SIZE * PATCH_SIZE;
                rowI = i / ROW_SIZE * PATCH_SIZE + (PATCH_SIZE - 1);
                colI = i % ROW_SIZE * PATCH_SIZE;
        }
        double result = 0;
        // System.out.println(j+" "+i+" "+Up+" "+rowJ+" "+colJ+" "+rowI+" "+colI);
        for (int k = 0; k < PATCH_SIZE; k++) {
                for (int l = 0; l < 3; l++) {
                        // System.out.println(LABData[rowJ][colJ+k]==null);
                        result += (LABData[rowJ][colJ + k].lab[l] - LABData[rowI][colI
                                        + k].lab[l])
                                        * (LABData[rowJ][colJ + k].lab[l] - LABData[rowI][colI
                                                        + k].lab[l]);
                }
        }
        return result;
}

public void shuffle() {
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < PATCH_NUMBER; i++) {
                list.add(i);
        }
        Collections.shuffle(list);
        for (int i = 0; i < PATCH_NUMBER; i++) {// location
                int source = list.get(i);// get patch number(content in original
                                                                // same location)
```

```
                setPatch(i, source);
        }
}

// set content to Real Image File
public void setPatch(int destination, int source) {
        int rowD = destination / ROW_SIZE * PATCH_SIZE, colD = destination
                        % ROW_SIZE * PATCH_SIZE;
        int rowS = source / ROW_SIZE * PATCH_SIZE, colS = source % ROW_SIZE
                        * PATCH_SIZE;
        for (int i = 0; i < PATCH_SIZE; i++) {
                for (int j = 0; j < PATCH_SIZE; j++) {// [rowD+i][colD+j]
                        image.setRGB(colD + j, rowD + i, RGBInteger[rowS + i][colS + j]);
                }
        }

}

public double sum_log(double a, double b) {

        if (a == Double.NEGATIVE_INFINITY) {
                return b;
        } else if (b == Double.NEGATIVE_INFINITY) {
                return a;
        } else {
                double x, y, c = Double.NEGATIVE_INFINITY;
                if (a > b) {
                        x = a;
                        y = b;
                } else {
                        x = b;
                        y = a;
                }
                double decide = Math.pow(10, x - y) + 1;
                if (decide == Double.POSITIVE_INFINITY) {// overflow
                        OverFlow++;
                        c = x;
                } else {
                        c = y + Log(decide);
                }
                // System.out.println("SUM:"+Math.pow(10,a)+" +    "+Math.pow(10,b)
                // +" = " +Math.pow(10,c));
                return c;
        }
}

public void DLR_DRL_test() {
        double firstMin = Double.MAX_VALUE, secondMin = Double.MAX_VALUE;
        int XI = 10;
        double[] record = new double[ROW_SIZE * COL_SIZE];
        for (int j = 0; j < PATCH_NUMBER; j++) {
                record[j] = DLR_DRL(j, XI, true);
                // double P = Math.exp((d/0.08)*(-1));
                if (record[j] < firstMin) {
                        firstMin = record[j];
                } else if (record[j] > firstMin && record[j] < secondMin) {
                        secondMin = record[j];
                }
        }
        double a = 2 * (secondMin - firstMin) * (secondMin - firstMin);
        for (int i = 0; i < PATCH_NUMBER; i++) {
                double P = Math.exp((record[i] / a) * (-1));
                System.out
                                .println("      " + (secondMin - firstMin) + "          P:    " + P);
        }
}
```

```
	public double Log(double value) {
		return Math.log10(value) / Math.log10(10);
	}

	class LAB {
		public double[] lab;// = new double[3];
		@Override
		public String toString() {
			// TODO Auto-generated method stub
			return lab[0] + "	" + lab[1] + "	" + lab[2];
		}
	}
	// 100*100, 5*5 patch,
	public static void main(String[] args) {
		JigsawPGM IIT = new JigsawPGM();
		IIT.init();
	}
}
```