

Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Stanislav Kučera

Generating random pattern-avoiding matrices

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: RNDr. Vít Jelínek, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

signature of the author

Title: Generating random pattern-avoiding matrices

Author: Stanislav Kučera

Department: Computer Science Institute of Charles University

Supervisor: RNDr. Vít Jelínek, Ph.D., department

Abstract: Abstract.

Keywords: binary matrix pattern-avoiding Markov chain Monte Carlo

Dedication.

Contents

Preface	3
Introduction	5
1 Markov chain Monte Carlo	6
1.1 Markov chain for pattern-avoiding binary matrices	6
2 An algorithm for testing pattern-avoidance of a general pattern	8
2.1 Sketch of a brute force algorithm	8
2.2 Equivalent mappings	8
3 An algorithm for testing pattern-avoidance of a special pattern	10
3.1 Walking pattern	10
3.2 Dynamic program	10
3.2.1 The algorithm	11
3.2.2 Inner structure	12
3.2.3 Correctness	12
3.2.4 Generalization	13
4 Improvements to basic algorithms	14
4.1 General pattern	14
4.1.1 Improving memory consumption	14
4.1.2 Not mapping empty lines	14
4.1.3 Using the last changed position	14
4.1.4 Line order	15
4.1.5 Mapping approaches	16
4.1.6 Using the whole structure in the next iteration	18
4.2 MCMC parallelism	19
4.2.1 Example of the MCMC process for n threads	20
4.2.2 Speculative computing	20
4.2.3 Reverting and synchronizing in the main thread	21
4.3 Walking pattern	21
4.3.1 Using the last changed position	21
4.3.2 Lazy avoid	21
5 Technical documentation	24
5.1 General pattern	24
5.1.1 Construction	24
5.2 Parallel computing	25
5.2.1 MCMC parallelism	25
5.3 Library interface	26
6 User documentation	27
6.1 Configuration file	27
6.1.1 Input	27
6.1.2 Pattern	28

6.1.3	Output	29
6.1.4	Statistics	30
6.2	File input	31
6.2.1	Matrix file	31
6.2.2	Order file	31
6.3	File output	32
6.3.1	Matrix text file	32
6.3.2	Matrix bmp file	32
Conclusion		33
List of Figures		34
List of Tables		35
List of Abbreviations		36
Attachments		37

1 Preface

2 Theses at the faculty of mathematics and physics usually fit into one of three
3 categories:

- 4 1. Theoretical thesis
- 5 2. Experimental thesis
- 6 3. Implementation thesis

7 My thesis does not fit entirely into only one category and it does not try to. The
8 project consists of several similarly important parts which are:

- 9 • Design of the algorithm for generating a special binary matrix
- 10 • Making it run fast on inputs which are usual for researchers
- 11 • Implementing the algorithm to provide practical tool

12 None of these points would make sense alone but together the thesis may be-
13 come very useful for scientists as it provides with a process generating random
14 matrices and it is a common practice to test hypothesis on random data.

Introduction

The area of pattern avoidance is heavily studied for permutations and it also becomes studied more for its generalization - binary matrices.

We denote by $M \in \{0,1\}^{n \times m}$ a binary matrix of size n by m , calling n the number of rows of M - the height of the matrix M and m the number of columns - its width. A line of a matrix is one of its rows or columns and for matrix M , we denote $L(M)$ the ordered set of all lines of M . Its order is given by the natural indexing of rows and columns.

We say a binary matrix M contains a binary matrix P , which we call a “pattern”, as a submatrix, if there is a mapping $f : L(P) \rightarrow L(M)$, such that

- $l \in L(P)$ is a row of P iff $f(l) \in L(M)$ is a row of M
- $\forall l, l' \in L(P) : l < l' \Rightarrow f(l) < f(l')$ (preserves the order)
- $\forall l, l' \in L(P) : \text{if lines } l \text{ and } l' \text{ intersect and there is a one-entry at the intersection, then there is a one-entry at the intersection of } f(l) \text{ and } f(l')$.

otherwise, it avoids the pattern P .

$$\begin{array}{ccc}
 \begin{array}{cc} & \begin{array}{cc} 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \end{array} &
 \begin{array}{ccc} & \begin{array}{ccc} 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} &
 \begin{array}{ccc} & \begin{array}{ccc} 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array}
 \end{array}$$

Figure 1: An example of a matrix containing a pattern and a matrix avoiding the pattern.

Let the pattern P be the left matrix from Figure 1. The middle matrix contains the pattern P , because a mapping $\{(0,0), (1,2), (2,3), (3,4)\}$ satisfies all the conditions. On the other hand the right matrix avoids the pattern as there is no such mapping.

There are other possible definitions of containing but from now on we will always consider a pattern to be contained (or avoided) as a submatrix. We will always denote M the binary matrix for which we test the containing and by P the pattern that is being tested. Moreover, we denote h the height (the number of rows) of P and w its width.

Generating random matrix

Our main goal is to show an iterated algorithm, which, for a high enough number of iterations, a given forbidden pattern and number n , generates a binary matrix $M \in \{0,1\}^{n \times n}$ avoiding the pattern uniformly at random.

To create such an algorithm we generalize an algorithm generating fixed size permutations without a forbidden pattern introduced by Madras and Liu in ?. To achieve randomness it uses Markov chain Monte Carlo method, which is explained in the next section.

47 The process starts with a binary matrix of the correct size avoiding the pattern
48 (zero matrix always works) and in each step it uniformly at random changes one
49 bit of the matrix and tests whether the matrix still avoids the pattern after the
50 change. If it does, the step is over, if it does not, we revert the change (flip the
51 bit back) and end the step.

52 It can be proven that if the Markov chain is chosen carefully, this leads to a
53 uniformly random matrix avoiding the pattern, after a sufficient number of steps.
54 Sadly no-one knows what is “sufficient” in this manner of speaking.

55 **Testing avoidance**

56 The hardest task to do in a step is to test whether the new matrix avoids the
57 pattern. We show a very fast algorithm which only works for a special class
58 of binary matrices as well as a slightly less performing algorithm for a general
59 pattern, which again comes as a generalization of an algorithm for permutations
60 from the article by Madras and Liu.

1. Markov chain Monte Carlo

Our goal to generate a binary matrix avoiding a forbidden pattern that is uniformly random heavily depends on the theory of Markov chains. In this work we only define useful terms and state an important theorem. If you are interested in more details, see [A first course in stochastic processes][An Introduction to Probability Theory and Its Applications][Lectures on Monte Carlo methods].

We shall consider discrete-time Markov chains X_0, X_1, X_2, \dots , where each X_i takes values in a finite or countably infinite state space S . For $k = 0, 1, 2, \dots$, the k -step transition probabilities are:

$$p_{i,j}^{(k)} = \Pr(X_{t+k} = j | X_t = i) \quad (i, j \in S)$$

A Markov chain is **irreducible** if the chain can eventually get from each state to every other state, that is, for every $i, j \in S$ there exists a $k \geq 0$ (depending on i and j) such that $p_{i,j}^{(k)} > 0$.

An irreducible chain has period D if D is the greatest common divisor of $\{k \geq 1 | p_{i,i}^{(k)} > 0\}$ for some $i \in S$ (equivalently, for all $i \in S$). A chain is called **aperiodic** if its period is 1. In particular, if an irreducible chain has $P_{i,i} > 0$ for some i , then it is aperiodic.

A Markov chain is said to be **symmetric** if $P_{i,j} = P_{j,i}$ for every pair of states i and j .

Suppose that an irreducible Markov chain on the finite state space S is symmetric. Then the equilibrium distribution is uniform on S .

1.1 Markov chain for pattern-avoiding binary matrices

Now when we want to generate a binary matrix $M \in \{0, 1\}^{n \times n}$ avoiding pattern P , we create a Markov chain, whose states will be the set of all binary matrices of size $n \times n$ that avoid P .

A step looks like this:

1. Choose $r \in \{0, 1, \dots, n-1\}$ uniformly at random.
2. Choose $c \in \{0, 1, \dots, n-1\}$ uniformly at random.
3. Flip the bit at $M[r, c]$.
4. If M contains P , flip the bit back.

When the process starts with a matrix which avoids P , then after every step it still avoids P . We need to show the Markov chain we just presented satisfies the conditions of the theorem:

Symmetry

Imagine a sequence of bits flipping which changes the i -th matrix to j -th one. The reversed order of the same sequence changes the j -th matrix to the i -th one.

94 Irreducibility

95 As the steps go, it is easy to see we can with non-zero probability create any
96 matrix $M \in \{0, 1\}^{n \times n}$ that avoids P from a zero matrix $0_n = 0^{n \times n}$. When we
97 can get from 0_n to M by a sequence of flip changes a , the reversed sequence is a
98 sequence of steps from any matrix M avoiding P to 0_n . Thus the Markov chain
99 is irreducible.

100 Aperiodicity

101 The Markov chain is irreducible so it suffices to show that there is an i for which
102 $P_{i,i} > 0$. Clearly there is a matrix for which there is atleast one bit that cannot
103 be flipped without creating a pattern and this forces $P_{i,i} = 1$.

2. An algorithm for testing pattern-avoidance of a general pattern

We begin with a very basic algorithm, which we then improve a lot to get a fast algorithm for testing avoidance of a general pattern.

2.1 Sketch of a brute force algorithm

Let $L := L(P) = \{l_1, l_2, \dots, l_{w+h}\}$ be the lines (rows and columns) of the pattern P in a standard ordering. Partial mapping of lines of P is a function f from $L' \subseteq L$ to lines of the big matrix M satisfying two conditions:

- Both $l' \in L'$ and $f(l')$ are rows or they are both columns.
- If $l' \in L'$ and $l'' \in L'$ are both rows or columns and $l' < l''$, then $f(l') < f(l'')$. This means partial mapping keeps the order of the lines.
- If $l' \in L'$ is a row of P and $l'' \in L'$ is a column of P and there is a one-entry at the intersection of l' and l'' , then there is a one-entry at the intersection of $f(l')$ and $f(l'')$.

The basic algorithm we use goes as follows. First it maps l_1 to all possible lines of M , creating partial mappings of $\{l_1\} \subseteq L$. For $k = 2, \dots, w + h$ it takes each partial mapping from the previous iteration and extends it by adding line l_k to the partial mapping in all possible ways. If we manage to map all the lines of P , then M does not avoid it and if at some point there are no partial mappings to extend it means M avoids P .

Note that we do not use even the slightest heuristics like not to map a line having k one-entries to a line with fewer one-entries. The fact we made a useless mapping will be discovered at the time we try to map the k -th orthogonal line (or maybe earlier). This leads to possibly very large number of partial mappings and our next goal will be to reduce this number significantly.

2.2 Equivalent mappings

There are a lot of possible conditions we can check before adding a line to a partial mapping. The most fundamental operation we do to decrease the number of found mapping is melting two equivalent mappings into one.

The idea behind it is easy. There is no need to remember two different mappings if they can be both extended exactly the same way as our function is only supposed to check whether a pattern can be mapped to a big matrix not to find all such mappings.

Definition: A **level** of a partial mapping is the number of lines mapped by the mapping.

140 **Definition:** We call a line l of a pattern **important** in a partial mapping if
 141 one of the conditions is met:

- 142 • An adjacent line of the pattern has not been mapped yet.
- 143 • There is a one-entry on the line l at the intersection with line l' that has
 144 not been mapped yet.

145 . Otherwise the line is **unimportant** in the mapping.

146 At the beginning, when no line is mapped, all lines are important. After some
 147 lines get mapped, a line can become unimportant in the partial mapping as all
 148 lines that bound it are in the mapping as well. If a line is unimportant in a partial
 149 mapping of some level, it will stay unimportant in all extensions of the mapping
 150 we can find.

151 **Definition:** We say two partial mappings of the same level are **equivalent**
 152 if all important lines in the mapping of that level are mapped to the same
 lines of the big matrix in both mappings. For P being the left matrix of Fig-

$$\begin{array}{ccccc}
 & 4 & 5 & 6 & 7 \\
 0 & \left(\begin{array}{cccc} 1 & 1 & 0 & 0 \end{array} \right) & 0 & \left(\begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{array} \right) \\
 1 & \left(\begin{array}{cccc} 0 & 1 & 1 & 1 \end{array} \right) & 1 & \left(\begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{array} \right) \\
 2 & \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \end{array} \right) & 2 & \left(\begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{array} \right) \\
 3 & \left(\begin{array}{cccc} 0 & 1 & 0 & 1 \end{array} \right) & 3 & \left(\begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{array} \right) \\
 & & 4 & \left(\begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{array} \right)
 \end{array}$$

Figure 2.1: An example showing unimportant line and equivalent mappings.

153
 154 ure 2.1 and M being the other matrix in Figure 2.1, in partial mapping $f =$
 155 $\{(1, 1), (2, 2), (3, 4), (5, 6)\}$ line 2 is unimportant because both lines 1 and 3 are
 156 mapped and so is line 5 - the only line to intersect line 2 in a one-entry. Line
 157 3 is important, because there is line 7 intersecting it in one-entry, which is not
 158 mapped.

159 In the same situation as above, consider a different partial mapping $f' =$
 160 $\{(1, 1), (2, 3), (3, 4), (5, 6)\}$, which is a mapping of the same level as f and only
 161 differs from f in mapping line 2. The line 2 is unimportant and by the definition
 162 of equivalent partial mappings, f and f' are equivalent. The idea behind this
 163 notion is simple. It is not important where we map line 2, because it does not
 164 restrict where we can map any other line that has not been mapped yet. This
 165 means that if a partial mapping f can be somehow extended, the equivalent
 166 partial mapping f' can be extended in the same way; therefore, it is sufficient to
 167 only extend one of them in order to find one full mapping. Note that it would
 168 be also sufficient to only extend one of the partial mappings if we were looking
 169 for all full mappings, but, in that case, we would need to keep the information
 170 about where the unimportant lines were mapped to.

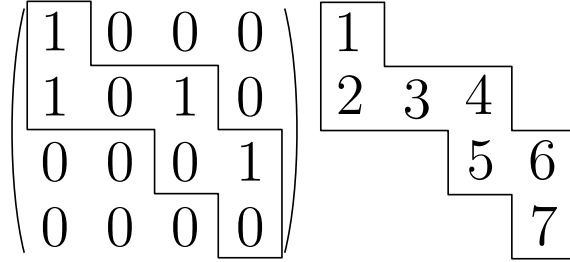
3. An algorithm for testing pattern-avoidance of a special pattern

In the previous chapter, we have seen an algorithm for a general forbidden pattern which, using some heuristics, runs pretty fast. In this chapter, we introduce a special kind of a pattern, satisfying additional conditions, for which we can produce much faster algorithm.

3.1 Walking pattern

We call the specific pattern a walking pattern. The additional condition we want the pattern to satisfy is that there is a walk from one corner to the opposite one and all the one-entries of the pattern are contained on the walk.

The **walk** in a matrix is a sequence of some of its entries beginning in the top left corner and ending in the bottom right one. If an entry at the position $[i, j]$ is in the sequence, the next one is either $[i + 1, j]$ or $[i, j + 1]$. Therefore the length of an arbitrary walk is equal to $w + h - 1$.



In the picture you can see a matrix that is a walking pattern as all the one-entries are included in a walk. Not all entries of a walk need to be one-entries though.

It can be shown a walking pattern with a walk is exactly a matrix avoiding a forbidden pattern

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

3.2 Dynamic program

Now that we know what the walking pattern is, we show an algorithm deciding whether a pattern P is contained in a big matrix M or not.

The pattern P is a walking pattern so there is a walk containing all the one-entries of the pattern. We choose one such walk arbitrarily and index its entries $w_1, w_2, \dots, w_{m+n-1}$ starting from the beginning of the walk. For each entry of the walk we remember whether its value is one or zero and whether the walk continues from the entry vertically, which we call a **vertical entry** or horizontally, calling it a **horizontal entry**.

For an element e of M at the position $[i, j]$, the matrix $M_{\leq e}$ is a $(i+1) \times (j+1)$ submatrix of M consisting of rows with the index smaller than or equal to i and

columns with the index smaller than or equal to j . The element e lies in its bottom right corner. Similarly $M_{\geq e}$ is a $(n - i) \times (n - j)$ submatrix of M consisting of rows with the index greater than or equal to i and columns with index greater than or equal to j . The element e is its first element.

To determine whether P is contained in M we find out for each element e of M what is the longest part of the pattern that can be found in $M_{\leq e}$. If there is an element for which we manage to find the last entry of the pattern, the pattern is contained in the matrix, otherwise it is avoided.

For each element e of M at the position $[i, j]$ we remember two numbers. The number $c_v(e)$ says what is the longest part of the walk in $M_{\leq e}$ with the last entry in j -th column and being a vertical entry. The number $c_h(e)$, symmetrically, says what is the longest part of the walk in $M_{\leq e}$ with the last entry in i -th row and being a horizontal entry.

An observation we make is that if we have a fixed element e of M and any other element e' above e in the same column then if $c_v(e')$ is equal to some k , then $c_v(e)$ is at least k . This means that for e we can find the maximum part of the pattern ending in the column of e and continuing vertically by looking only to elements in that column above e and since this is true for all of them, it is sufficient to only check the value of the element right above e (at the position $[i - 1, j]$). Similarly the argument goes for the value of c_h in horizontal way.

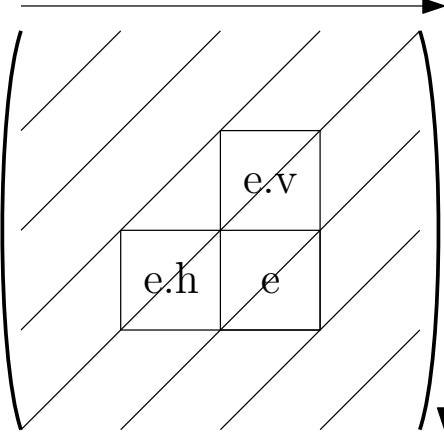
The algorithm iterates through diagonals. A diagonal in this matter of speaking is a subset of elements of M , such that all elements have the same sum of their coordinates. For example, zero diagonal only consists of an element $[0, 0]$, the first diagonal contains elements $[0, 1]$ and $[1, 0]$, and so on.

For simplicity, in the pseudo-code below we do not deal with elements outside M (like $-1, 0$) explicitly. Instead for those elements we just assume the values of c_v and c_h are always equal to zero.

3.2.1 The algorithm

1. For $d = 0, \dots, w + h - 1$
2. For e element of d -th diagonal at the position $[i, j]$
3. $e_v := [i - 1, j]$
4. $e_h := [i, j - 1]$
5. $c_v(e) := c_v(e_v)$
6. $c_h(e) := c_h(e_h)$
7. If $w_{c_v(e)+1}$ can be mapped to e
8. If $c_v(e) + 1 = w + h + 1$
9. Terminate - M contains P as a submatrix
10. If $w_{c_v(e)+1}$ is a vertical entry
11. $c_v(e) := c_v(e) + 1$

239 12. Else
 240 13. $c_h(e) := \max\{c_h(e), c_v(e) + 1\}$
 241 14. If $w_{c_h(e)+1}$ can be mapped to e
 242 15. If $c_h(e) + 1 = w + h + 1$
 243 16. Terminate - M contains P as a submatrix
 244 17. If $w_{c_h(e)+1}$ is a vertical entry
 245 18. $c_v(e) := \max\{c_v(e), c_h(e) + 1\}$
 246 19. Else
 247 20. $c_h(e) := \max\{c_h(e), c_h(e) + 1\}$



248 3.2.2 Inner structure

249 To run properly the algorithm needs two structures. The first one is a description
 250 of the walk, which is just an array of the values of its entries as well as the
 251 information whether the entry is vertical or horizontal. The second structure is
 252 a matrix of the values c_v and c_h as described above.

253 3.2.3 Correctness

254 We need to show that the values of c_v and c_h are always correct for the recomputed
 255 elements after at the end of the function. We proceed by induction.

256 For the first element it is definitely true since there can be only the first entry
 257 of the pattern mapped and we check just that.

258 When we compute an element e of a computed diagonal d , by induction as-
 259 sumption all the diagonals $d' < d$ are correctly computed. In particular, the
 260 values are correct in the diagonal $d - 1$. To compute the correct values of e , we
 261 use the values of two element on the diagonal $d - 1$: e_v , which is right above e and
 262 e_h , which is the first element to the left of e . If e_v or e_h are outside the matrix
 263 then from that direction we cannot expect to find anything more than just the
 264 first entry of the pattern and that is what we check for.

265 Let v be the true length of the longest part of P in $M_{\leq e}$ continuing vertically
 266 in the same column as e . Now if e itself is not an entry of that part of the pattern,
 267 it is a different element e' in the same column. But then the value of $c_v(e')$ is
 268 correctly computed by the inductive hypothesis and it is copied to all element
 269 underneath. Especially e_v gets the value and the algorithm copies the value from
 270 it to e . On the other hand if e is an entry of the part of the pattern we work with,
 271 it is the last entry. The entry right before the last one needs to be mapped to
 272 the same row or column; therefore, either e_v or e_h contain the part of the pattern
 273 shorter by one and the algorithm extends it to a correct value.

274 3.2.4 Generalization

275 The same algorithm, just rotated by 90 degrees, can be also used for a pattern
 276 where all one-entries are contained on a walk from top right corner to the bottom
 277 left one. Indeed the program uses it and if given a walking pattern it determines
 278 by itself which walk it is.

279 On the other hand a direct generalization for a general pattern does not work.
 280 While we can index all entries of the pattern, when trying to map a certain w_k
 281 to an element it is not sufficient to just check whether w_l is above and w'_l to the
 282 left from the element.

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & \boxed{1} \\ 1 & \textcircled{1} & \diamond 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & \boxed{1} \\ 0 & 1 & \textcircled{1} & \diamond 1 \end{pmatrix}$$

283 In the picture, let the matrix on the left side be the pattern P and let M
 284 be the other matrix. The entry in the square can be mapped to the element in
 285 the square and the same holds for entries in the circle but it is not a sufficient
 286 condition for the entry in the triangle to be mapped to the element in the triangle.

287 4. Improvements to basic 288 algorithms

289 4.1 General pattern

290 4.1.1 Improving memory consumption

291 As the algorithm for testing avoidance of a general pattern was described in
292 [chapter2], it creates all possible partial mappings and checks whether at least
293 one can be extended to a full mapping. Note that to compute all the partial
294 mappings of some level l , it only uses mappings of level $l - 1$; therefore, it is
295 enough to only store partial mappings of two levels in memory.

296 In [chapter2] we also introduced the idea of (un)important lines for a partial
297 mapping of level l and equivalence based on not using unimportant lines at all -
298 as they are fully bounded by other already mapped lines. When a line becomes
299 unimportant it stays unimportant till the end of the run; as a result, we can forget
300 where we mapped those lines to save memory. This is not as big of a deal as the
301 previous observation was but note there are cases of patterns in which each line
302 becomes unimportant just two levels after it gets added.

303 4.1.2 Not mapping empty lines

304 An empty line is a row or a column that does not contain any one-entries. Such
305 a line can be mapped to any line and if the algorithm leaves space for it (which
306 it does), we do not need to map it at all.

307 4.1.3 Using the last changed position

308 As the MCMC process works, it always changes one element of the big matrix
309 and asks whether it still avoids the pattern. If it does not and we know that
310 before the change it did, we are sure the changed element $[r, c]$ is a part of the
311 pattern. It is hard to use this fact in the algorithm. It just maps one line after
312 another and we do not know at the beginning to which line the changed position
313 lines should be mapped.

314 What we can do is to enforce that neither the r -th line nor the c -th one
315 get skipped. We will only look at the restriction for rows. The restrictions for
316 columns are symmetrical. There are three situations we want to avoid:

- 317 • The first row of P is mapped under the r -th row. This prevents any other
318 row to be mapped to r -th one and we don't want that.
- 319 • The last row of P is mapped above the r -th row. This again prevents any
320 other row to be mapped to r -th one.
- 321 • Two adjacent rows $l, l + 1$ of P are mapped to $L < L'$ respectively and
322 $L < r < L'$ which leaves no other row to be mapped to r .

323 4.1.4 Line order

324 An important thing, if we want the algorithm to run fast, is to choose a good
325 line order. A line which is unimportant in level l in a line order may easily be
326 important till the nearly last level in a different order.

327 We choose line order to hopefully enforce two things:

- 328 • Make as many unimportant lines as possible. This really allows the equiv-
329 alence based improvements to kick in. The more lines are unimportant the
330 more mappings become equivalent and the faster it is to iterate through all
331 of them.
- 332 • Recognize hopeless partial mappings as soon as possible. A partial mapping
333 gets extended if the line does not break the rule that there is a one-entry
334 where it needs to be. If we map all the rows first, the rule will get broken
335 only after we start to map columns and we probably want to find out sooner.

336

337 In the program a user can either choose their own custom order or one of four
338 algorithms with different main purposes:

- 339 • AUTO - this one tries the other three line orders and chooses the one which
340 shows the best performance over some iterations on a matrix. While this
341 may sound like a good thing to use, it is only so if an initial matrix is chosen
342 and it takes a lot of time since a lot of iterations need to be made in order
343 to make a good sample. I would recommend not to use AUTO order at all
344 and instead to try all the line orders by hand with a number of iterations
345 depending on the pattern and a good initial matrix; for instance, generated
346 with a smaller number of iterations on the same pattern and with any line
347 order.
- 348 • DESC - the lines are ordered in descending order depending on the number
349 of one-entries. This follows the idea to start with the lines that are the
350 hardest to map. Note that this algorithm does poorly if there are a lot of
351 lines with the same number of one-entries (for example an identity matrix).
- 352 • MAX - it orders the lines so that the maximum number of important lines
353 throughout the levels is as small as possible. This focuses straightforwardly
354 to having many unimportant lines, which the program does not remember.
- 355 • SUM - it orders the lines so that the sum of the numbers of the important
356 lines is the smallest possible throughout all levels. The purpose is the same
357 as in the MAX order and quite often it is the case both approaches produce
358 the same order.
- 359 • TWO - it orders the lines so that the maximum number of important lines
360 in two consecutive levels throughout all the levels is as small as possible.
361 This again focuses to having many unimportant lines, which the program
362 does not remember. The constant two is chosen due to the fact general
363 pattern always stores two levels of partial mapping at a time.

364 4.1.5 Mapping approaches

365 The one thing the approaches we will introduce have in common is that they try
 366 to recognize those partial mappings that have no chance to be extended to a full
 367 mapping as early as possible.

368 While the algorithm introduced in [chapter2] finds out the partial mapping
 369 is invalid only at the time it maps two lines having a one-entry at their inter-
 370 section to two lines having a zero-entry at the intersection, different approaches
 371 try to reveal the fact we would end up in the situation earlier by checking more
 372 conditions.

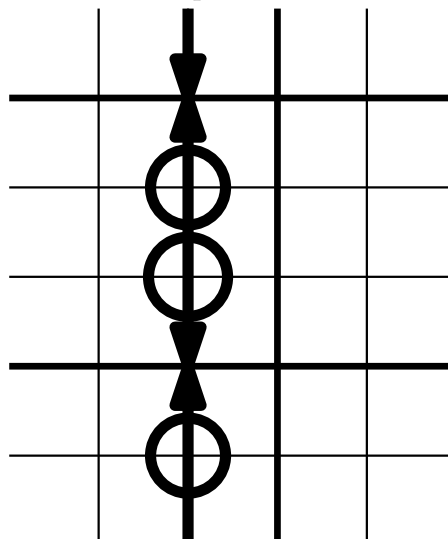
373 Imaging this as the pattern, where $0, \dots, 8$ are indices of the lines:

$$\begin{array}{c} \\ \\ \\ \\ \\ \end{array} \begin{array}{c} 5 \quad 6 \quad 7 \quad 8 \\ \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array} \right) \end{array}$$

374 We are in a situation when only lines 0, 3 and 7 are mapped and line 6 is
 375 currently being mapped. Here are some mapping approaches:

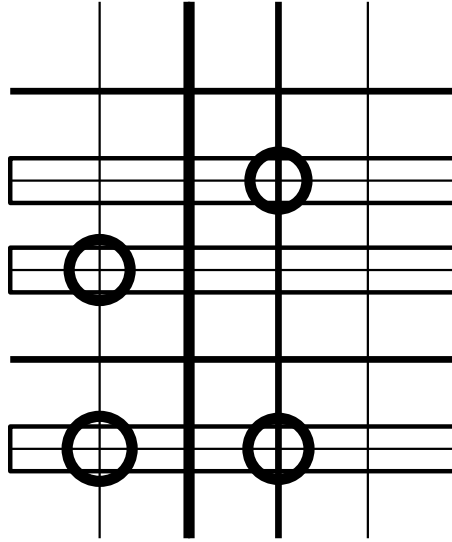
376 Enough one-entries

377 In the situation above, we not only want to check there are one-entries at the
 378 intersections of line 6 with lines 0 and 3, but we also check if there are enough
 379 one-entries in lines between where lines 0 and 3 are mapped so that there is a
 380 hope we can map lines 1 and 2 there and if there is a one-entry below the line
 381 where line 3 is mapped so we can map line 4 there later.



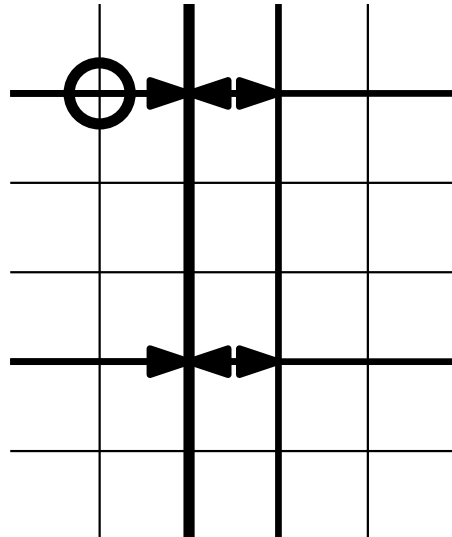
382 Recursive mapping

383 While we were only testing whether there are enough one-entries in between
384 already mapped lines in the previous approach, this time, we also check whether
385 those one-entries can be used for the lines that are intended to be mapped there.
386 For example, when we check there is a one-entry to be used for line 1 later, we
387 also check the line 1 can be mapped to that row, which in this situation means
388 to also check there is a one-entry at the intersection with the line to which the
389 line 7 is mapped.



390 Orthogonal bounds

391 When adding line 6, we check whether there are enough one-entries on the already
392 mapped lines orthogonal to line 6, in between line 6 and the closest mapped lines
393 next to line 6. The same idea as in “Enough one-entries”, but checking different
394 lines.



395 Usage

396 These restrictions on the added lines are not a fixed part of the program. A user
397 can decide which approaches they want to use in the configuration file. This is

398 due to the fact that there is no right path to choose.

399 In the testing that was done for a fixed pattern, we found out it is useful to
400 use all the mentioned restrictions when generating a matrix of size 100×100 , as
401 it turned out to be much faster than without the restrictions. On the other hand,
402 in the same test for a generated matrix of size 500×500 , it was much better not
403 to use any of those restrictions.

404 4.1.6 Using the whole structure in the next iteration

405 It may seem like a good idea to remember all the partial mappings, to propagate
406 them to the next iteration of the MCMC process and alter them depending upon
407 the change.

408 This really can be done. If the change is from zero-entry to one-entry, for each
409 partial mapping we already have we want to try to extend it by the line that just
410 changed and if we manage to do that we then try to extend it to a full mapping
411 in all possible ways if it is a new mapping or do nothing if it is equivalent with a
412 partial mapping of higher level. This can be easily done by means already used
413 in the standard algorithm and may lead to a better performing one.

414 However, if the element gets changed from one-entry to zero-entry we need
415 to go through the partial mappings and delete those that used the currently
416 changed one-entry. This gets a bit messy as we can no longer forget unimportant
417 lines and moreover for each partial mapping we need to remember how many
418 partial mappings of the previous level can be extended to that one, to delete that
419 mapping from the list if there are no longer any mappings extensible to that one.

420 This can all be done, but the whole thing comes with three huge inconven-
421 niences:

- 422 • Memory consumption - there can be a LOT of partial mappings and we
423 need to remember them all. Of course we can still use the equivalence but
424 we need to remember mappings of all levels.
- 425 • The change from one-entry to zero-entry is no longer for free. If this change
426 is done, we already know the pattern is not contained in M , but we still
427 need to do a lot of work to change the structure in order to use it in the
428 next iteration.
- 429 • Reverting - if the change is unsuccessful (the pattern is contained) we need
430 to revert the change which means to completely revert all changes we did to
431 the list of partial mappings. This can be either done by making a backup
432 copy of the whole structure and override the structure if needed, which again
433 is very costly as the structure is huge, or we can remember what partial
434 mappings are new and we go through all partial mappings and remove those
435 new ones. This again means to iterate through the big structure one more
436 time.

437 After realizing these issues it no longer looks useful to me to implement this
438 version of the algorithm.

4.2 MCMC parallelism

To speed up the computations, it is often possible to use parallelism. In this section, we show how to make the MCMC generator parallel, while still allowing both types of the pattern.

While the serial MCMC generator in each iteration changes one element in the generated matrix and checks whether it still avoids forbidden patterns, the parallel one makes several iterations at once, one on each copy of the generated matrix. This means that while iteration x is being computed by a thread, iteration $x + 1$ can at the same time be computed by a different thread. The only issue is that iteration $x + 1$ does not know what is going to be the state of the generated matrix at the time it should start. It expects iteration x to fail - not change the generated matrix at all, counting on the fact at some point it is unlikely a change does not create a mapping of the pattern, and starts with the same matrix as iteration x . If iteration x succeeds though, then the computed iteration $x + 1$ is invalid and the iteration is going to be recomputed again, starting with the altered matrix.

When the parallel version of MCMC generator is chosen and it is assigned n threads, it creates $n - 1$ private copies of the generated matrix and assigns one thread, called worker, to each of them. The last thread, which we call the main thread and which has exclusive access to the master copy of the generated matrix, makes one change of a bit in each private copy of the matrix and makes the corresponding worker check the avoidance.

The job of a worker is only to check if its copy of the matrix still avoids the pattern when one bit is changed. On the other hand all synchronization is left to the main thread. As mentioned before, one iteration of the MCMC process can be recomputed several times. We still want the generator to satisfy the conditions we have for the Markov chain (more in [chapter1]) in order to generate a random matrix. To achieve that, if a computed iteration x succeeds (and changes the generated matrix), all the other computed iterations that would follow after the iteration x become invalid and they all have to be recomputed. The process ends when all iterations get computed.

From now on, we won't be talking about iterations but about tasks. A task is basically one iteration of the MCMC process. The usefulness of this notation comes with an ID - a number, unique per task, assigned to each task, starting with 1 and always increasing. For a pair of consecutive iterations x and $x + 1$ it will always be the case that if task a is the last task to compute iteration x (which means the iteration does not get recomputed ever again after) and task b is the last task to compute iteration $x + 1$, then the ID of a is lower than the ID of b . Also there is no point in which two different tasks would be computing the same iteration at the same time. If tasks with IDs $a < b$ computed the same iteration, it must have been the case an earlier iteration succeeded when task with ID a was computed and after it got removed, task with ID b was assigned to recompute.

At any point in time, we only consider those tasks, that are being computed or those that wait to be processed (not those that have been processed), which means the lowest ID of tasks we consider increases in time.

When a task ends and it has the lowest ID (we can always wait for the task

486 with the lowest ID) we do:

487 • if it fails:

488 – Do nothing - there is no change to propagate to the master copy of the
489 generated matrix and all the tasks with higher ID expected this task
490 to fail, which it did.

491 – This increases the lowest ID by exactly one, as the task we speak of
492 got processed.

493 • if it succeeds:

494 – The main thread propagates the change tested by the task to the
495 master copy of the generated matrix.

496 – All the rest of the task get removed as they all had a higher ID -
497 computed iterations that follow after the one just computed and they
498 expected the task to fail, which it did not.

499 – This increases the lowest ID by more then one, because there are tasks
500 that got removed and one that got processed.

501 4.2.1 Example of the MCMC process for n threads

502 At first, iterations 1 to $n - 1$ are assigned one to each worker as tasks with ID 1 to
503 $n - 1$ with the same order as the order of iterations. If iteration 1 is not successful
504 (which all the other iterations count on), everything is alright. However, if the
505 iteration (its task) is successful, all the results of other tasks (and some of them
506 might have been already finished) are cleared and those iterations get recomputed
507 in tasks n to $2n - 3$ and the worker that computed task with ID 1 is assigned a
508 new task with ID $2n - 2$ - to compute iteration n . The result of the task gets
509 propagated to the master copy of the generated matrix only if all the tasks n to
510 $2n - 3$ fail, else is gets recomputed. This is what happens till the end.

511 4.2.2 Speculative computing

512 It may easily happen that a task not having the lowest ID ends first. In that
513 case we could just wait until it has the lowest ID and process it later. This is
514 not a very efficient approach. Instead we process the task immediately, but we
515 don't propagate the changes to the master copy of the generated matrix until all
516 tasks with lower ID fail and we do not stop the workers processing a task with
517 lower ID. If the task succeeds we remove all the changes computed by tasks with
518 higher ID and override their private copy of the generated matrix. Also it might
519 happen a task with even lower ID succeeds as well. This leads to more and more
520 overriding. Luckily this is the only precarious situation we may encounter and it
521 is not that hard to deal with it, even without copying the possibly huge generated
522 matrix all over the place.

523 The way we deal with these inconveniences is described in [chapter5] and
524 should be clear from the code itself.

525 4.2.3 Reverting and synchronizing in the main thread

526 The speculative computing discussed above is not the only improvement we can
527 make. It turns out to be costly to wake a thread so it computes a trivial function,
528 sets a few atomic variables and falls asleep again. This happens a lot in the
529 MCMC process. Every time a task succeeds it makes other workers revert the
530 changes they computed and synchronize the successful change, which are both
531 trivial functions.

532 To workaround this problem we make a theoretically bad decision which has
533 very nice practical results. All the reverts and synchronizations are computed by
534 the main thread instead of by an appropriate worker. There is no problem with
535 concurrency because the worker is always asleep when a task is to be assigned and
536 using the fact those tasks are really trivial, it does not make the rest of threads
537 wait for the main thread for too long while it computes changes.

538 4.3 Walking pattern

539 While the brute force implementation of an avoid algorithm for a general pattern
540 was improved heavily, the algorithm for a walking pattern is very fast in its nature
541 and cannot be much better. Or can it be?

542 4.3.1 Using the last changed position

543 As the MCMC process works it always changes one element of the big matrix and
544 asks whether it still avoids the pattern. If it does not and we know that before the
545 change it did, we are sure the changed element is a part of the pattern. Knowing
546 that and using the same inductive proof as we did in the proof of correctness of
547 the avoid algorithm (see [chapter2]) it is sufficient to only recompute the part of
548 the inner structure under the changed element and check if the last entry of the
549 pattern can be found there.

550 Not only that. We also know, using the fact the structure was completely
551 correct before the change, that if the values of both c_v and c_h of an element did
552 not change, the element won't cause the element underneath it to change and we
553 no longer have to recompute the other parts of the structure.

554 To use both these facts we replace the cycle through the diagonals by a simple
555 queue, starting at the position of the last changed element and putting more
556 positions in if the values of c_v or c_h are different than they were before. The
557 function ends either when the pattern was discovered or when the queue becomes
558 empty.

559 4.3.2 Lazy avoid

560 Lazy avoid is a variant of avoid function used when the MCMC parallelism is
561 chosen. While all the other types of patterns have a trivial implementation of
562 revert function, when using the walking pattern the inner structure needs to be
563 modified even when reverting. The MCMC parallelism turned out to work much
564 better if the revert calls are handled by the main thread (more in [chapter4]) and
565 it requires the function to run as fast as possible so the other threads are not

566 blocked by the call for too long. That is a reason why functions lazy revert and
567 lazy avoid were created.

568 The avoid function expects the inner structure of the walking pattern (see
569 [chapter3]) to be in a valid state and that requires some effort. To make lazy
570 revert the fastest possible, we postpone the work until the next call of lazy avoid,
571 meaning that lazy avoid then needs to do more things at once. It is no longer
572 sufficient to only compute the submatrix under the position changed last as we
573 did above, but it needs to also compute changes in the positions changed in those
574 lazy revert calls that are postponed.

575 We discuss several approaches, starting with the easiest one and ending with
576 the one that is fast and used in the final implementation.

577 **Recompute the whole structure every time**

578 The easiest way how to implement lazy avoid would be to always recompute the
579 whole inner structure. In that case we do not worry which positions are correct
580 and which are not, because every time we find the pattern, we recomputed all
581 the entries that form it, so we know it really is there. On the other hand, if we
582 manage to recompute the whole structure without finding the last entry of the
583 pattern, it just is not there.

584 The issue is efficiency. If the whole structure was correct and there was a
585 change of the last entry of the matrix it is sufficient to only recompute that one
586 entry. Instead we recompute a possibly very big structure. This results in a very
587 bad performance negating the advantage of parallel computation.

588 **Recompute only a part of the structure diagonal by diagonal**

589 A simple improvement would be to remember the changes done in previous calls
590 of lazy revert and together with the change done in lazy avoid call only recompute
591 the part of the structure that has possibly altered.

592 This gets a bit tricky when lazy avoid call actually discovers the pattern
593 because we cannot be sure the rest of the structure is in a correct order. It is
594 still possible to remember some horizontal, vertical and diagonal bounds and use
595 them to restrict the recomputed part of the matrix. The improvement is not that
596 significant though and we can do better.

597 **Queue of positions to recompute**

598 A different approach is closer to the one used in a standard avoid function. In-
599 stead of going through diagonal one after another, we have a queue of entries-to-
600 recompute. It is no longer sufficient to have a standard queue since in different
601 calls of lazy revert/avoid we can possibly change an entry of different priority (the
602 higher the more important) so we need to have some kind of a priority queue.
603 That is exactly what I tried.

604 Using `std::priority_queue` the function had no more problems with recomput-
605 ing the entries that were not influenced by the changes and used all the benefits
606 mentioned in the previous section. But the container does not come for free and
607 in the end I found out the price I paid for the operations on the priority queue
608 made the whole implementation comparably slow as in the previous attempt.

609 **Two leveled queue of positions to recompute**

610 The final solution comes with the same idea, but a different storage type. As
611 the priority depends upon a diagonal (two entries on the same diagonal can be
612 recomputed in any order) we only remember a priority queue of diagonals and an
613 array of diagonals saying whether a diagonal is already a member of the priority
614 queue. As far as the entries are concerned for every diagonal we have a `std::vector`
615 of entries-to-recompute as well as an array saying whether an entry is already a
616 member of the vector. So finally it is the case that the storage used is not only
617 good theoretically but as the numbers say, also practically. [reference to a table
618 of measurements or something]

619 5. Technical documentation

620 In this chapter, we cover those parts of the algorithm that may be hard to un-
621 derstand just from the code. This only means functions that are technically hard
622 - functions with unexpected dependencies, side effects and so on. Algorithmic
623 difficult tasks are explained in [chapter4].

624 5.1 General pattern

625 The general pattern class contains a lot of function. Most of them are easy to
626 follow and they all should be commented enough in the code. The only part
627 which deserves more attention is the constructor.

628 5.1.1 Construction

629 In the constructor of a general pattern, there are a few function that are easy in
630 nature but as they somehow use each other it is hard not to lose track of their
631 dependencies and results. In order to make this part of the code, which is a very
632 important part indeed, more understandable, we go through the constructor and
633 explain all that is happening in the order it is happening in.

634 Storing the pattern

635 The first thing, which is done right after initialization of variable, is storing the
636 pattern. Instead of storing the pattern in a `Matrix<bool>`, I decided to use to
637 store lines into a number, where in the binary coding a one-entry in the position
638 i means there is a one-entry in the line at the intersection with i -th orthogonal
639 line. This comes handy when computing lines orders. At the same time we also
640 find those lines that are empty (more in [chapter4]) and remember them, because
641 we do not have to map them at all.

642 Choosing the line order

643 After that we need to choose the right line order (again more in [chapter4]).
644 To compute MAX or SUM order we just use a brute force algorithm that checks
645 sequences of line adding and for each it computes how many lines are unimportant.
646 Then it just chooses the order which is the best in chosen metric.

647 What to remember

648 In the next step, we find what do we need to remember in each level of partial
649 mappings with respect to chosen order. As mention earlier, for MAX or SUM
650 order it is already computed when finding the order, but for other variants of
651 orders it is not, so we just compute it every time. What to remember is based
652 on the equivalence introduced in [chapter2] and the decision not to remember
653 unimportant lines (which we explained in [chapter4]).

654 **Parallel bound indices**

655 Now comes the hardest to follow part - precomputing the indices for searching for
656 parallel bounds. The idea behind is simple. When we are adding a new line and
657 we already have a partial mapping, it restricts to where we can add the line. For
658 example, if there are three rows in the pattern and the rows 1 and 3 are mapped,
659 the second one need to be mapped in between those two. The question is, where
660 are those two lines mapped to? First we add in a chosen order and second we do
661 not remember all lines, as some are unimportant. What do we want is to have
662 a instant access to the index of the line, which bounds added line, in the partial
663 mapping so we do not need to compute the index over and over again. That is
664 exactly what gets computed when the function “find_parallel_bound_indices” is
665 called. The series of other function calls follows just because we compute the
666 indices for all added lines in the order in which they are going to be added.

667 **Extending order**

668 The last function, “find_extending_order” just specifies how the next partial map-
669 ping will look like a from where from the previous mapping the values will be
670 copied. Again, unimportant lines play their role here and it may easily be the
671 case from a partial mapping storing k lines, after mapping one more line, we end
672 up with a partial mapping only storing $k - 1$ lines, because two lines become
673 unimportant by adding the line.

674 **5.2 Parallel computing**

675 **5.2.1 MCMC parallelism**

676 While the idea behind MCMC parallelism is described in [chapter4.3] and the
677 code is heavily commented, the work done by the main thread may still be hard
678 to understand.

679 Let I be the ID the process is currently waiting for, that is, the lowest ID of
680 a task that is being tested by a worker. In a structure called “queue” (which is
681 `std::vector<std::deque>`) each worker has a queue of tasks related to it. In the
682 queue, there are tasks that are either being computed or have been computed.
683 The history of tasks is needed to allow reverting changes that should have not
684 happen when the main thread encounters a different successful task with lower
685 ID. There is no need to have a complete history of all tasks computed. There
686 are only those tasks, that have higher ID than I or have lower ID, but those are
687 going to be removed from the “queue” as soon as possible. The name “queue” is
688 not random, it describes the order in which the tasks are being stored - the tasks
689 with lower ID have been inserted earlier and therefore they are at the bottom.

690 Now that we know the most important structure let's see how the main thread
691 works with that and what are the situations.

- 692 • **pop_front**: The main thread deletes the first tasks (the one with the lowest
693 ID) if one of two things happen:
 - 694 – The ID of the task being deleted is equal to I . That means the change
695 computed by the task is being propagated to the generated matrix and

696 there is no need to remember the task anymore. This also increases I ,
697 not necessarily by one.

698 – The ID of the task being deleted is less than I . This situation is due
699 to synchronization. The worker was supposed to synchronize a task
700 computed by a different worker that did not have the lowest ID at the
701 time. Therefore the task needs to be in the list of tasks so we can
702 revert it if needed. If there is no need to revert it and the lowest ID
703 gets greater or equal to the ID of the task, we can just delete it from
704 the “queue”.

705 • `pop_back`: There is only one reason to delete tasks from the end of the
706 “queue” and that is reverting. Imagine there is a task with id J at the
707 end of the “queue”. Now a different worker computes a task with lower
708 ID and finds out the change is successful. This means the task J won’t
709 propagate to the generated matrix and there is no use for it. If it is still
710 being computed, we cannot do much about it, so we just tell the worker to
711 stop computing and deal with it later. If the task is finished, we need to
712 revert it, but only in case the task was successful, because if it was not, it
713 had already been reverted by the worker. So we revert the task if needed
714 and we can just delete it from “queue” as it will never be used.

715 • `emplace_back`: The main thread only inserts new tasks to the end of the
716 “queue” and there are two reasons to insert:

717 – Worker is assigned a completely new task to check the avoidance. In
718 this situation the task is given a new, globally highest ID and we add
719 the task at the end of the list.

720 – The second reason to insert into “queue” are, again, synchronizations.
721 The situation is the same as it was in the case, when we `pop_back` -
722 after we revert all the tasks in the list, we need to synchronize changes
723 that forced reverting and if their ID is not lower or equal to I , we need
724 to add them to the list so they can be reverted if needed.

725 5.3 Library interface

6. User documentation

In the last chapter of the thesis we show how to make the program generate random matrices or to test whether a certain matrix avoids a given forbidden pattern. In the first section we find out how to set a configuration file. After that we go through necessary input and output files and see how they are formatted.

6.1 Configuration file

In order to modify what the program computes, we use a configuration file. There is only command line options to be used and that is a path to the configuration file. If no path is inserted the configuration file is expected to be located in the same directory as the executable file is and its name is “config.txt”.

The file is a standard text file which can be modified by any text editor and is structured into four sections:

- input
- pattern
- output
- statistics

The order of the sections is not fixed and there can be additional empty lines for better readability. In each section there is a list of values that can be set either to arbitrary value or to a specific one. There is at most one command of format “option=value” per line and there might be additional white spaces surrounding the “=” sign.

If an option is set more than once, the latter value is always used. If, on the other hand, an option is not set at all, the default value is used. If there is a line encountered that sets a wrong option, for instance when the user mistypes a valid option, the line is skipped and the user gets a warning in the standard error output.

Let us provide a list of all options for each section together with their default values.

6.1.1 Input

In the first section of the configuration file we set the generating process.

- size: The size of the generated matrix. Results in $M \in \{0, 1\}^{size \times size}$.
Possible value: $s \in \mathbb{N}$
Default value: 100
- iterations: The number of iterations of the MCMC process.
Possible value: $i \in \mathbb{N}$
Default value: 10,000

- 760 • `random_seed`: The random seed for the MCMC process.
Possible value: $s \in \mathbb{N}$
761 “random” - chooses a random seed
Default value: “random”
- 762 • `init_matrix`: A $size \times size$ matrix the MCMC process starts with.
Possible value: *input file path*
763 “zero” - a matrix containing no one-entries
Default value: “zero”
- 764 • `parallel_mode`: choose whether to compute in parallel or serial.
Possible value: “serial”
“mcmc” - more iterations of the MCMC generator are tested in parallel
765 “map” - more partial mappings are being extended in parallel
Default value: “serial”
- 766 • `threads_count`: The number of threads that are going to be used if a parallel
767 mode is chosen.
Possible value: $t \in \mathbb{N}$
768 -1 - chosen according to the number of cores
Default value: 1

769 6.1.2 Pattern

770 It is time to set the options that matter the most - matrix patterns. As you
771 generate a matrix which avoids more than just one pattern, the section [pattern]
772 can be used multiple times, specifying one pattern for each occurrence.

- 773 • `pattern_file`: A path to a input matrix file - the pattern.
Possible value: matrix file path
774 Default value: “input.txt”
- 775 • `pattern_type`: The type of the pattern. Determines the method used for
776 testing avoidance.
Possible value: “general”
“walking” - see [chapter3]
777 “slow” - brute force algorithm for a general pattern
Default value: “general”

778 The next options are only useful if the general pattern type is chosen. It
779 specifies how the mappings are stored as well as what the map function tests.

- 780 • `map_one_entries`: If set to “yes”, the map function not only checks that
781 already mapped lines do cross the currently added one in a one-entry if
782 needed but also tests whether there is enough one-entries in between those
783 already mapped lines. The result is that the map function takes a little
784 longer but it recognizes hopeless mappings earlier.
Possible value: “yes”
“no”
785 Default value: “yes”

- 786 • `map_recursion`: If set to “yes” and the `map_one_entries` is also set to “yes”,
787 the `map` function not only tests there are enough one-entries in between
788 already mapped lines on the currently added one, but for those non-mapped
789 lines crossing the current one in a one-entry, it also tests if the partial
790 mapping has one-entries everywhere they are supposed to be. Although
791 this comes with a big time consumption it can filter a big portion of partial
792 mapping that would later become unfeasible.
Possible value: “yes”
793 “no”
Default value: “yes”
- 794 • `map_orthogonal_bounds`: If set to “yes”, the `map` function also tests the
795 orthogonal bounds of added line - see [chapter2].
Possible value: “yes”
796 “no”
Default value: “no”
- 797 • `map_container`: A container in which the partial mappings are stored.
Possible value: “set” - `std::set` (red-black tree)
“hash” - `std::unordered_set` (hash table)
798 “vector” - `std::vector` (dynamic array)
Default value: “hash”
- 799 • `line_order`: Choose the order in which the lines are being added to the
800 partial mapping. See [chapter2]
Possible value: “max”
“two”
“sum”
801 “desc”
“auto”
“order file path”
Default value: “max”

802 6.1.3 Output

803 There is no reason to generate a matrix noone will ever see. In this section we
804 make the matrix show. As the matrix can be output to console, a text file or a
805 bmp file an option in the section can be set more than once and all the settings
806 make difference.

- 807 • `matrix_output`: The generated matrix can be output as a bmp file in which
808 one-entries are black pixels and zero-entries white. To do that, the file path
809 has to have a pattern “path.bmp”. If a different path is given the file is
810 stored as a matrix text file. It can also be output into a console if “console”
811 is set. In that case it has the text format.
Possible value: “console”
matrix bmp file path
812 matrix text file path
“no”
Default value: “no”

- 813 • `performance_stats`: If the serial computation or the map parallelism is cho-
814 sen, the program can output a statistics like the percentage of avoid call
815 success, how long did one call take on average and what was the average
816 size of structures. Note that this is not very useful as a matrix statistics,
817 see the next section for that. If more patterns are set, the statistics may get
818 misleading as it also counts the cases when the first pattern is contained in
819 the matrix and the other patterns are not tested at all.

820 Possible value: `"console"`
 performance file path
 `"no"`
820 Default value: `"no"`

- 821 • `performance_csv_stats`: The same information as above but formatted to a
822 csv file so the data can be more easily worked with.

823 Possible value: `"console"`
 csv file path
 `"no"`
823 Default value: `"no"`

- 824 • `time_to_console`: Prints how long the computation took into a console.
825 Possible value: `"yes"`
 `"no"`
825 Default value: `"no"`

- 826 • `patterns_to_console`: Prints all the used patterns into the console.
827 Possible value: `"yes"`
 `"no"`
827 Default value: `"no"`

828 6.1.4 Statistics

829 The last section handles the options important for scientists. While generating
830 a random matrix is a great result, on its way the program can also create some
831 statistics, namely make a histogram of occurrences of one-entries in a generated
832 matrix as the MCMC iterates as well as store the matrix with the highest amount
833 of one-entries. As the process usually does not start with a random matrix, the
834 user can decide to only compute the statistics after a certain number of iterations
835 has been done and to only check a small portion of iterations, every 10th for
836 instance, as a single iteration may not make any difference and counting the
837 histogram takes time.

- 838 • `histogram_frequency`: Sets how often the histogram gets refreshed.
839 Possible value: $f \in \mathbb{N}$
 0 - the histogram is not computed at all
839 Default value: 0

- 840 • `histogram_initial`: Sets the initial iteration of the MCMC process when the
841 histogram gets refreshed.
842 Possible value: $i \in \mathbb{N}$
842 Default value: 1,000

- 843 • histogram_final: Sets the last iteration of the MCMC process when the
844 histogram gets refreshed.
 Possible value: $f \in \mathbb{N}$
 -1 - the histogram is computed till the end
 Default value: -1
- 846 • histogram_file: Sets where to output the histogram computed during the
847 MCMC process.
 Possible value: matrix bmp file path
 matrix text file path
 “console”
 “no”
 Default value: “no”
- 849 • max_ones_matrix_file: Sets where to output the matrix that had the most
850 one-entries among all matrices iterated through during the MCMC process.
 Possible value: matrix bmp file path
 matrix text file path
 “console”
 “no”
 Default value: “no”

852 6.2 File input

853 There are only two types of input files expected by the program. Either you want
854 to read a matrix file, which can be a pattern or an initial matrix, or an order file
855 that determines an order in which the lines are going to be mapped if the general
856 pattern is chosen.

857 6.2.1 Matrix file

858 A matrix file is a standard text file having the format as follows:

- 859 • 2 natural numbers specifying the number of rows and columns in this order.
- 860 • a sequence of zeros and ones of length rows×columns specifying the matrix
861 from the top left corner one row after another.

862 **Example:** 2 3
 1 0 1
 1 1 0

863 6.2.2 Order file

864 If you want to choose the order in which the lines are going to be mapped when
865 a general pattern is chosen, it is your responsibility to check that all lines that
866 need to be mapped are mapped. It is for example possible to only map three lines
867 even if the pattern consists of six lines just because there is for example no need

868 to map empty lines at all. Therefore the program does not check the validity of
869 the order and just uses it.

870 Now that the user has been warned, the format of the custom order file is
871 simple. It consist of the indices of the lines of the pattern numbered starting
872 with 0 and starting from the top row and ending with the right column.

873 One possible order for the matrix given as an example in [6.2.1] is this file:

874 2 1 0 3 4

875 First mapping the left column, the second and first row after that and finishing
876 the mapping with the middle column and the right one.

877 6.3 File output

878 Let us now find out what the output files look like.

879 6.3.1 Matrix text file

880 The matrix text file has the same format as the input one. It consists of:

- 881 • 2 natural numbers specifying the number of rows and columns in this order.
- 882 • a sequence of zeros and ones of length rows×columns specifying the matrix
883 from the top left corner one row after another.

884 The matrix is binary except for the one produced as a histogram, which can have
885 higher natural numbers and contains the number of samples as the last number.
886 If you then divide all the entries by the last number, you get a percentage of the
887 entry being a one-entry.

888 6.3.2 Matrix bmp file

889 For an $n \times n$ matrix the standard bmp file contains $n \times n$ pixel of black color
890 meaning a one-entry and a white color for a zero-entry. If the histogram is output
891 as a bmp file, the pixels are greyscaled and the darker a pixel is the more often
892 the entry was a one-entry during the MCMC process.

893 Conclusion

894 List of Figures

895	1	An example of a matrix containing a pattern and a matrix avoiding	
896		the pattern.	4
897	2.1	An example showing unimportant line and equivalent mappings. .	9

898 List of Tables

899 List of Abbreviations

900 Attachments