

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

Title: Generating random pattern-avoiding matrices

Author: Stanislav Kučera

Department: Computer Science Institute of Charles University

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

Abstract: Binary matrices not containing a smaller matrix as a submatrix became an interesting topic recently. In my thesis, I introduce two new algorithms to test whether a big square binary matrix contains a smaller binary matrix together with a process using randomness, which approximates a uniformly random matrix not containing a given matrix. The reason to create such algorithms is to allow researchers test their conjectures on random matrices. Thus, my thesis also contains an effective cross-platform implementation of all mentioned algorithms.

Keywords: binary matrix pattern-avoiding Markov chain Monte Carlo

I would like to thank my supervisor, Vít Jelínek, for his patience and willingness to spend so much time helping me improve the thesis. I also thank to Paul Macklin, author of EasyBMP (<http://easybmp.sourceforge.net/>) – a library for working with bmp files, which I use in my program.

Contents

Preface	3
Introduction	4
1 Markov chain Monte Carlo	7
1.1 Markov chains	7
1.2 Markov chain for pattern-avoiding binary matrices	7
2 An algorithm for testing pattern-avoidance of a general pattern	9
2.1 Sketch of a brute force algorithm	9
2.2 Equivalent mappings	9
3 An algorithm for testing pattern-avoidance of a special pattern	11
3.1 Walking pattern	11
3.2 Dynamic program	11
3.2.1 Inner structures	12
3.2.2 The algorithm	12
3.2.3 Correctness	13
3.2.4 Generalization	14
4 Improvements to basic algorithms	16
4.1 General pattern	16
4.1.1 Improving memory consumption	16
4.1.2 Not mapping empty lines	16
4.1.3 Using the last changed position	16
4.1.4 Line order	17
4.1.5 Mapping approaches	18
4.1.6 Using the whole structure in the next iteration	20
4.2 MCMC parallelism	21
4.2.1 Example of the MCMC process for n threads	23
4.2.2 Speculative computing	23
4.2.3 Reverting and synchronizing in the main thread	23
4.3 Walking pattern	24
4.3.1 Using the last changed position	24
4.3.2 Lazy avoid	24
4.4 Comparison of all methods	25
5 Technical documentation	27
5.1 Classes and API	27
5.1.1 Matrix	27
5.1.2 Pattern	27
5.1.3 Statistics	27
5.2 General_pattern	28
5.2.1 Construction	28
5.3 MCMC parallelism	29

6	User documentation	31
6.1	Installation	31
6.1.1	Windows	31
6.1.2	Unix, Linux, MacOS	31
6.2	Configuration file	31
6.2.1	Input	32
6.2.2	Pattern	32
6.2.3	Output	34
6.2.4	Statistics	34
6.3	File input	35
6.3.1	Matrix file	36
6.3.2	Order file	36
6.4	File output	36
6.4.1	Matrix text file	36
6.4.2	Matrix bmp file	37
	Conclusion	38
	Bibliography	39
	List of Figures	40
	List of Tables	41
	Attachments	42

1 Preface

2 Theses at the Faculty of Mathematics and Physics of Charles University in Prague
3 usually fit into one of three 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 algorithms for generating a special binary matrix
- 10 • Making the algorithms run fast on inputs that are usual for researchers
- 11 • Implementation of the algorithms to provide a practical tool

12 One part would not make sense without others, but together, the thesis may
13 become a very useful tool for scientists interested in matrices with forbidden
14 patterns as the thesis provides them with a process of generating random pattern-
15 avoiding matrices.

Introduction

Throughout the thesis, we will be concerned with binary matrices and something called *pattern*, which will also be a binary matrix.

Definition 1. We let $M \in \{0, 1\}^{m \times n}$ denote a *binary matrix* of size m by n . The *height* of M , denoted by m , is the number of rows of M and n is its width (the number of columns).

Definition 2. A *line* of a matrix is one of its rows or columns and we denote by $L(M)$ the ordered set of all lines of M . Its order is given by the standard indexing of rows and columns.

Definition 3. 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 if and only if $f(l) \in L(M)$ 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 .

$$P = \begin{smallmatrix} & 2 & 3 \\ 0 & \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\ 1 & \end{smallmatrix} \quad M_1 = \begin{smallmatrix} & 3 & 4 & 5 \\ 0 & \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \\ 2 & \end{smallmatrix} \quad M_2 = \begin{smallmatrix} & 3 & 4 & 5 \\ 0 & \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \\ 2 & \end{smallmatrix}$$

Figure 1: Matrix M_1 contains the pattern P , because all the conditions are satisfied by mapping $\{(0, 0), (1, 2), (2, 3), (3, 4)\}$. On the other hand, matrix M_2 avoids P as there is no such mapping.

The interesting cases are square matrices of size n by n , where n is big (going to infinity) and the size of a pattern (not necessarily square matrix) is small (constant). Even for a constant size forbidden pattern it is hard to determine the number of matrices of size n that avoid it or to characterize, what properties they have. That is why it is useful to have a tool generating random matrices. Sometimes we consider matrices avoiding more than just one forbidden pattern, in which case we denote the set of all forbidden matrices by \mathcal{P} . When a matrix avoids \mathcal{P} , it avoids every $P \in \mathcal{P}$.

Notation 1. We denote by $\mathcal{M}_n(\mathcal{P})$ a set of all binary matrices of size n by n avoiding \mathcal{P} as submatrices.

Notation 2. We always call M the square binary matrix, for which we test the containing, and P the pattern (if there is only one) that is being tested. Moreover, we denote by h the height (the number of rows) of P and by w its width.

45 The area of pattern avoidance has been heavily studied for permutations and
 46 it also becomes more popular for their generalization – binary matrices. In most
 47 of the areas in combinatorics, it is useful to explore properties of random objects
 48 and a lot of attention is directed towards random matrices when considering
 49 pattern avoidance. The goal of the thesis is, for given $n \in \mathbb{N}$ and set of forbidden
 patterns \mathcal{P} , to generate a uniformly random $M \in_R \mathcal{M}_n(\mathcal{P})$.

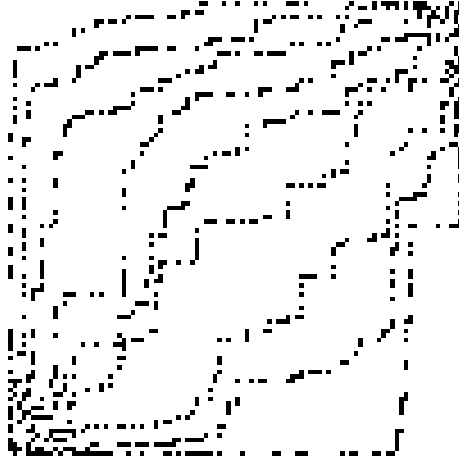


Figure 2: Example of a generated matrix avoiding I_{10} (unit matrix). Black dots are one-entries and white are zero-entries. As you can see, matrices avoiding a pattern can have a nice structure.

50

51 Generating random matrices

52 One way to get $M \in_R \mathcal{M}_n(\mathcal{P})$ is to choose a matrix of required size completely
 53 at random, for such, test whether it avoids the pattern and simply repeat the
 54 process until we find one, which does. However, in the most interesting cases,
 55 only a small fraction of all matrices avoid the pattern and the process takes too
 56 long, to be practically useful.

57 For generating random permutations avoiding forbidden pattern, a different
 58 technique was introduced in Madras and Liu [2010]. It uses a randomized pro-
 59 cess called Markov chain Monte Carlo, which we will abbreviate by MCMC. It
 60 is an iterative process, which for a well chosen Markov chain (more in Chapter
 61 1) approximates a random object. The algorithm by Madras and Liu was devel-
 62 oped for permutations (permutation matrices) and it cannot be used for general
 63 matrices. In Section 1.2 we show how to adapt the algorithm, which will lead
 64 us to a MCMC algorithm that approximates $M \in_R \mathcal{M}_n(\mathcal{P})$. To produce a good
 65 approximation the process needs to do a lot of iterations and despite the fact
 66 it is unknown what is the mixing time (the number of iterations required) of a
 67 MCMC process, in practice, the method does better than the trivial algorithm.

68 Testing avoidance

69 In each step of our MCMC process, we need to test whether a matrix avoids a
 70 pattern. We will show a very fast algorithm that only works for a special class of
 71 binary matrices (explained in Chapter 3) together with a slightly less performing

72 algorithm for a general pattern, which, again, comes as a generalization of an
73 algorithm for permutations from the article by Madras and Liu and is described
74 in Chapter 2.

75 In Chapter 4, we improve both our algorithms and introduce a parallel version
76 of MCMC process, which further increases the performance of matrix generating.

77 Some technical details are explained in Chapter 5 to make reading the code
78 easier for reader. The last chapter (Chapter 6) contains user documentation.

1. Markov chain Monte Carlo

Our goal to generate $M \in_R \mathcal{M}(\mathcal{P})$ heavily depends on the theory of Markov chains. We only define useful terms and state two important theorems. If you are interested in more details, see Madras [2002].

1.1 Markov chains

Definition 4. Let \mathcal{S} be a finite set of states and for every $i, j \in \mathcal{S}$ $p_{i,j}$ prescribed probability of a change of state from i to j . Also let X_0 be a random variable with values from \mathcal{S} . We call a sequence X_0, X_1, \dots , where $X_i \in \mathcal{S}$ for every i a *Markov chain* if

$$Pr(X_{t+1} = j | X_t = i) = p_{i,j} \quad (i, j \in \mathcal{S})$$

Definition 5. A Markov chain is said to be *symmetric* if $p_{i,j} = p_{j,i}$ for every pair of states i and j .

Definition 6. A Markov chain is *irreducible* if the chain can eventually get from each state to every other state, that is, for every $i, j \in \mathcal{S}$ there exists a $k \geq 0$ (depending on i and j) such that $Pr(X_k = j | X_0 = i) > 0$.

Definition 7. If an irreducible chain has $p_{i,i} > 0$ for some i , then it is *aperiodic*.

Let $p_{i,j}^{(k)} = Pr(X_{t+k} = j | X_t = i)$ denote the k -step transition probabilities for $k = 0, 1, \dots$ and $i, j \in \mathcal{S}$. The transition probability matrix is $P = (p_{i,j})$.

Next we state two theorems allowing us to expect Markov chains to converge to a uniformly random state in \mathcal{S} even if the initial state X_0 is not random. Both theorems can be found in Madras [2002].

Theorem 1. Consider an aperiodic irreducible Markov chain with finite state space \mathcal{S} . For every $i, j \in \mathcal{S}$, the limit $\lim_{k \rightarrow \infty} p_{i,j}^{(k)}$ exists and is independent of i ; call it π_j . Furthermore,

$$\sum_{j \in \mathcal{S}} \pi_j = 1 \quad \wedge \quad \sum_{i \in \mathcal{S}} \pi_i p_{i,j}^{(1)} = \pi_j$$

for every $j \in \mathcal{S}$.

Theorem 2. Suppose that an irreducible Markov chain on the finite state space \mathcal{S} is symmetric. Then the equilibrium distribution is uniform on \mathcal{S} .

1.2 Markov chain for pattern-avoiding binary matrices

To generate a binary matrix $M \in \{0, 1\}^{n \times n}$ avoiding patterns in \mathcal{P} , we create a Markov chain, whose states space is $\mathcal{M}_n(\mathcal{P})$. After sufficiently many iterations (m) of MCMC process we set $M := X_m \in \mathcal{M}_n(\mathcal{P})$. We always begin with an initial matrix X_0 and the process looks like this:

- 104 1. For $i := 1, 2, \dots, m$:
- 105 2. Set $X_i := X_{i-1}$.
- 106 3. Choose $r \in_R \{0, 1, \dots, n-1\}$ uniformly at random.
- 107 4. Choose $c \in_R \{0, 1, \dots, n-1\}$ uniformly at random.
- 108 5. Flip the bit at $X_i[r, c]$.
- 109 6. If X_i contains \mathcal{P} , flip the bit back.

110 If the process starts with a matrix X_0 that avoids \mathcal{P} , then after every step it
 111 still avoids \mathcal{P} . Note that an iteration does not change the matrix if the condition
 112 6 is satisfied. We need to show the Markov chain we presented meets all the
 113 conditions of both theorems:

114 Symmetry

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

117 Irreducibility

118 As the steps go, it is easy to see we can with non-zero probability create any
 119 matrix $M_1 \in \mathcal{M}_n(\mathcal{P})$ from the zero matrix $0_n = 0^{n \times n}$ by choosing the one-entries
 120 of M_1 . When we can get from 0_n to M_2 by a sequence of flip changes, the reversed
 121 sequence is a sequence of steps from $M_2 \in \mathcal{M}_n(\mathcal{P})$ to 0_n . Thus, with non-zero
 122 probability we can always reach M_2 from M_1 ; therefore, the Markov chain is
 123 irreducible.

124 Aperiodicity

125 The Markov chain is irreducible so it suffices to show that there is an i for which
 126 $p_{i,i} > 0$. Clearly, there is a matrix for which there is at least one bit that cannot
 127 be flipped without creating a pattern (for example the one with the maximum
 128 number of one-entries) and this forces $p_{i,i} > 0$.

129 2. An algorithm for testing 130 pattern-avoidance of a general 131 pattern

132 In this chapter and Chapter 3 we show algorithms for testing whether a pattern
133 P is contained in a square binary matrix M .

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

136 2.1 Sketch of a brute force algorithm

137 Let $L = (l_1, l_2, \dots, l_{w+h})$ be a permutation of lines (rows and columns) of the
138 pattern P and $k \in [w+h]$. *Partial mapping of level k* of lines of P is a function
139 f from $L' := \{l_1, l_2, \dots, l_k\} \subseteq L$ to lines of the big matrix M satisfying three
140 conditions:

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

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

153 The algorithm can be improved in two ways. Firstly, we can try to recognize
154 unextendable partial mappings earlier than at the moment a line can no longer be
155 mapped, for example by counting whether there is enough one-entries in between
156 already mapped lines (more in Section 4.1.5). Secondly, which is going to be
157 fundamental for us, we can try not to remember more copies of different mappings
158 that can be extended in the same way.

159 2.2 Equivalent mappings

160 There is no need to remember two different partial mappings of the same level
161 if they can be both extended exactly the same way, because our function is only
162 supposed to check whether a pattern can be mapped to a big matrix not to find
163 all such mappings.

164 **Definition 8.** We call a line l of a pattern P *important* for chosen permutation
165 of lines of P , if one of the conditions is met:

- 166 • An adjacent line of the pattern has not been mapped yet.
- 167 • There is a one-entry on the line l at the intersection with line l' that has
- 168 not been mapped yet.
- 169 . Otherwise the line is *unimportant* for the permutation.

170 Whether a line is important or not only depends on the permutation, so if
 171 we have a line unimportant in a partial mapping of level k , it is unimportant in
 172 every partial mapping of level k .

173 At the beginning, when no line is mapped, all lines are important. After some
 174 lines get mapped, a line can become unimportant in the partial mapping as all
 175 lines that bound it are in the mapping as well. If a line is unimportant in a partial
 176 mapping of some level, it will stay unimportant in all extensions of the mapping
 177 we can find.

178 **Definition 9.** We say two partial mappings of the same level are *equivalent* if
 179 all important lines in the mapping of that level are mapped to the same lines of
 180 the big matrix in both mappings.

$$\begin{array}{c}
 \begin{array}{c}
 4 \quad 5 \quad 6 \quad 7 \\
 0 \left(\begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline 2 & 0 & 1 & 0 \\ \hline 3 & 0 & 1 & 0 \\ \hline \end{array} \right) \\
 \end{array}
 \quad
 \begin{array}{c}
 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
 0 \left(\begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 1 \\ \hline 2 & 1 & 1 & 0 & 0 \\ \hline 3 & 0 & 1 & 1 & 0 \\ \hline 4 & 1 & 1 & 1 & 0 \\ \hline \end{array} \right) \\
 \end{array}
 \end{array}$$

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

181 For P and M , binary matrices in Figure 2.1, in partial mapping of level 4
 182 $f = \{(1, 1), (2, 2), (3, 4), (5, 6)\}$, line 2 is unimportant because both lines 1 and 3
 183 are mapped and so is line 5 - the only line to intersect line 2 in a one-entry. Line
 184 3 is important, because there is line 7 intersecting it in one-entry, which is not
 185 mapped.

186 In the same situation as above, consider a different partial mapping $f' =$
 187 $\{(1, 1), (2, 3), (3, 4), (5, 6)\}$, which is a mapping of the same level as f and only
 188 differs from f in mapping line 2. The line 2 is unimportant and by the definition
 189 of equivalent partial mappings, f and f' are equivalent. The idea behind this
 190 notion is simple. It is not important where we map line 2, because it does not
 191 restrict where we can map any other line that has not been mapped yet. This
 192 means that if a partial mapping f can be somehow extended, the equivalent
 193 partial mapping f' can be extended in the same way; therefore, it is sufficient to
 194 only extend one of them in order to find one full mapping. Note that it would
 195 be also sufficient to only extend one of the partial mappings if we were looking
 196 for all full mappings, but, in that case, we would need to keep the information
 197 about where the unimportant lines were mapped to.

198 3. An algorithm for testing 199 pattern-avoidance of a special 200 pattern

201 In the previous chapter, we have seen an algorithm for a general forbidden pattern.
202 In this chapter, we introduce a special kind of a pattern, satisfying additional
203 conditions, for which we can produce a much faster algorithm.

204 3.1 Walking pattern

205 **Definition 10.** A *walk* in a matrix P is a sequence of some of its entries, begin-
206 ning in the top left corner and ending in the bottom right one. If an entry at the
207 position $[i, j]$ is in the sequence, the next one is either $[i + 1, j]$ or $[i, j + 1]$. Let w
208 denote the width of P and h denote its height, the length of an arbitrary walk is
209 equal to $w + h - 1$ and we denote elements of the sequence by $w_1, w_2, \dots, w_{w+h-1}$.

210 **Definition 11.** We call a binary matrix P a *walking pattern* if there is a walk in
211 P such that all the one-entries of P are contained on the walk.

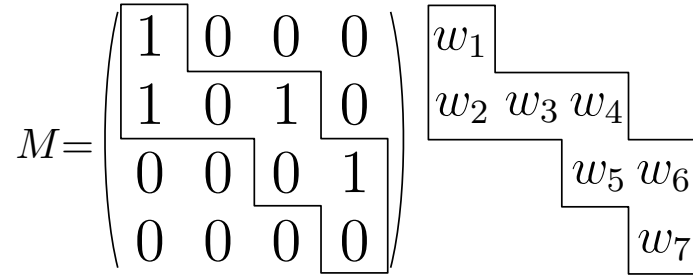


Figure 3.1: An example of a walk W in matrix M and the order of entries in W .

212 In Figure 3.1 matrix M is a walking pattern as all the one-entries are included
213 in a walk. We can also see that not all entries of a walk need to be one-entries.

214 It can be shown a walking pattern is exactly a matrix avoiding a forbidden
215 pattern $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

216 3.2 Dynamic program

217 Next, we show an algorithm deciding whether a walking pattern P is contained
218 in a big matrix M or not.

219 The pattern P is a walking pattern, so there is a walk containing all the one-
220 entries of P . We choose one such walk arbitrarily. For each entry of the walk we
221 remember whether its value in P is one or zero and whether the walk continues
222 from the entry vertically, in which case we call it a *vertical entry* or horizontally,
223 calling it a *horizontal entry*.

Definition 12. 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 then lies in the 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 biggest index k such that there exists a mapping of $P_{\leq w_k}$ to $M_{\leq e}$. If there is an element for which we manage to find the whole pattern ($k = w+h-1$), P is contained in M ; otherwise, it is avoided.

3.2.1 Inner structures

The algorithm uses two structures. For each w_k we remember whether it is a one-entry or zero-entry in P and whether it is a vertical entry or horizontal entry.

The second structure is a matrix of the same size as M . For each element e at the position $[i, j]$ we store two numbers. The number $c_v(e)$ is the biggest index k such that w_k is a vertical entry and there is a mapping of $P_{\leq w_k}$ to $M_{\leq e}$, in which w_k is being mapped to the j -th column. The number $c_h(e)$, symmetrically, is the biggest index k such that w_k is a horizontal entry and there is a mapping of $P_{\leq w_k}$ to $M_{\leq e}$, in which w_k is being mapped to the i -th row.

3.2.2 The algorithm

Definition 13. A *diagonal* of the matrix M is a subset of elements of M , such that all elements have the same sum of their coordinates.

For example, the zero diagonal only consists of the element $[0, 0]$, the first diagonal contains elements $[0, 1]$ and $[1, 0]$, and so on.

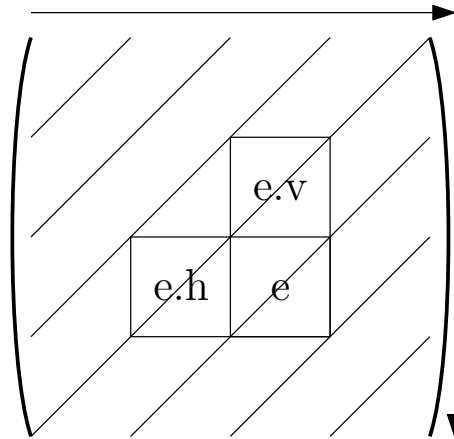


Figure 3.2: Diagonals of an matrix and the order in which the algorithm for walking pattern iterates through them.

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

251 $[-1, i]$ is a vertical entries and $[i, -1]$ is a horizontal for every $i \geq 0$. When we ask
 252 whether w_k can be mapped to e , where e is an element of M , we check whether
 253 w_k stands for a one-entry of P and if it does, we require e to be a one-entry too.

254 For an $n \times n$ matrix M the algorithm works as follows:

- 255 1. For $d = 0, \dots, 2n - 2$
- 256 2. For e element of d -th diagonal at the position $[i, j]$
- 257 3. $e_v := [i - 1, j]$
- 258 4. $e_h := [i, j - 1]$
- 259 5. $c_v(e) := c_v(e_v)$
- 260 6. $c_h(e) := c_h(e_h)$
- 261 7. If $w_{c_v(e_v)+1}$ can be mapped to e
- 262 8. If $c_v(e_v) + 1 = w + h - 1$
- 263 9. Terminate - M contains P as a submatrix
- 264 10. If $w_{c_v(e_v)+1}$ is a vertical entry
- 265 11. $c_v(e) := c_v(e_v) + 1$
- 266 12. Else
- 267 13. $c_h(e) := \max\{c_h(e), c_v(e_v) + 1\}$
- 268 14. If $w_{c_h(e_h)+1}$ can be mapped to e
- 269 15. If $c_h(e_h) + 1 = w + h - 1$
- 270 16. Terminate - M contains P as a submatrix
- 271 17. If $w_{c_h(e_h)+1}$ is a vertical entry
- 272 18. $c_v(e) := \max\{c_v(e), c_h(e_h) + 1\}$
- 273 19. Else
- 274 20. $c_h(e) := \max\{c_h(e), c_h(e_h) + 1\}$

275 3.2.3 Correctness

276 The first observation we make is that for every element e of M and any element
 277 e' above e in the same column $c_v(e') \leq c_v(e)$. This holds because whenever
 278 we manage to map $P_{\leq w_k}$ to $M_{\leq e'}$, then the same mapping maps $P_{\leq w_k}$ to $M_{\leq e}$.
 279 Similarly, it also holds for every e element of M and any element e' to the left of
 280 e in the same row that $c_h(e') \leq c_h(e)$.

281 The function can terminate before recomputing all elements and we have no
 282 guarantee about the state of elements, which have not been recomputed. If the
 283 function finds the pattern ending in entry e , it stops computing at that point, but

284 to prove correctness it is enough to prove the values are correct in $M_{\leq e}$, which
 285 has been fully recomputed. If, on the other hand, the function does not find the
 286 pattern, it recomputes the whole structure.

287 We need to show that the values of c_v and c_h are always correct for the
 288 recomputed elements at the end of the function. We proceed by induction on
 289 diagonals.

290 For the first diagonal it is definitely true since there can only be mapped w_1
 291 and we check that on lines 7 and 14.

292 When we are recomputing the values of $c_v(e)$ and $c_h(e)$ of an element e in the
 293 diagonal d , by induction hypothesis, all elements in diagonals $d' < d$ are correctly
 294 recomputed. Let cor denote the correct value of $c_v(e)$ as it is defined and com be
 295 the computed value. We need to show $cor = com$.

296 We can already see $cor \geq com$ because it holds after setting $c_v(e)$ on line 5
 297 and we only increase it, if we manage to find an extension of a mapping, in which
 298 case there really is a mapping; therefore, cor is greater or equal to the updated
 299 value.

300 To prove $cor \leq com$ we proceed by contradiction. Let us assume $cor > com$.
 301 It means there is a mapping of $P_{\leq w_{cor}}$ to $M_{\leq e}$ we have never found. Every such
 302 mapping has to map w_{cor} to e , because if it did not, the mapping would be
 303 possible even for diagonal $d - 1$, which is recomputed correctly and the value cor
 304 would be copied to com on line 5. Let us assume that w_{cor-1} is a vertical entry
 305 (else we proceed analogously). If $P_{\leq w_{cor}}$ can be mapped to $M_{\leq e}$ and w_{cor-1} is a
 306 vertical entry, then $P_{\leq w_{cor-1}}$ can be mapped to $M_{\leq e_v}$ and w_{cor-1} must be mapped
 307 to the same column as e . That means that $c_v(e_v) \geq cor - 1$. If $c_v(e_v) = cor - 1$
 308 and from knowing w_{cor} can be mapped to e , $com \geq c_v(e) \geq c_v(e_v) + 1 = cor$
 309 because of line 11. Otherwise $c_v(e_v) > cor - 1$, but then even from line 5 we get
 310 $com \geq cor$, resulting in contradiction.

311 To prove $c_h(e)$ has the correct value, we proceed symmetrically.

312 3.2.4 Generalization

313 The algorithm, with a few minor changes, can be also used for a pattern where all
 314 one-entries are contained on a walk from the top right corner to the bottom left
 315 one. The program supports both rotations of a walk and when walking pattern
 316 is chosen it automatically decides which variant to use.

317 On the other hand, a direct generalization for a general pattern does not work.
 318 While we can index all entries of the pattern, when trying to map a certain w_k
 319 to an element e of M , it is not sufficient to only check whether w_l is above and
 320 w'_l to the left from e .

321 In Figure 3.3, the entry of P in the square can be mapped to the element
 322 of M in the square and the same holds for entries in the circle but it is not a
 323 sufficient condition for the entry of P in the kite to be mapped to the element of
 324 M in the kite.

$$P = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & \boxed{1} \\ 1 & \textcircled{1} & \diamond 1 \end{pmatrix} \quad M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & \boxed{1} \\ 0 & 1 & \textcircled{1} & \diamond 1 \end{pmatrix}$$

Figure 3.3: The algorithm testing avoidance for walking patterns cannot be easily generalized for all patterns.

325 4. Improvements to basic 326 algorithms

327 In this chapter we improve algorithms presented in previous chapters and intro-
328 duce a parallel method of testing pattern avoidance.

329 4.1 General pattern

330 We start by improving the brute force algorithm from Chapter 2.

331 4.1.1 Improving memory consumption

332 The algorithm creates all possible partial mappings and checks whether at least
333 one can be extended to a full mapping (mapping all lines of the pattern). To
334 compute all the partial mappings of some level l , it only uses mappings of level
335 $l-1$; therefore, it is enough to only store partial mappings of two levels in memory
336 at any time.

337 In Chapter 2 we also introduced the notion of (un)important lines and equiva-
338 lence based on not using unimportant lines at all (they are fully bounded by other
339 already mapped lines). When a line becomes unimportant, it stays unimportant
340 till the end of the test; as a result, we can forget where we mapped those lines to
341 save memory and only remember where we mapped important lines.

342 4.1.2 Not mapping empty lines

343 **Definition 14.** An *empty* line is a row or a column that does not contain any
344 one-entries.

345 An empty line can be mapped to any line and we do not need to map it at
346 all, as long as the algorithm does not map two lines surrounding an empty one
347 to two consecutive lines.

348 4.1.3 Using the last changed position

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

355 What we can do is to enforce that neither the r -th line nor the $n + c$ -th
356 one (c -th column) get skipped. We only look at the restriction for rows as the
357 restrictions for columns are symmetrical. There are three situations we want to
358 avoid:

- 359 • The first row of P is mapped under the r -th row. This prevents any other
360 row to be mapped to r -th one and we do not want that.

- The last row of P is mapped above the r -th row. This again prevents any other row to be mapped to r -th one.
- Two adjacent rows $l, l + 1$ of P are mapped to $L < L'$ respectively and $L < r < L'$ which leaves no other row to be mapped to r .

4.1.4 Line order

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

We choose line order to hopefully enforce two things:

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

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

- AUTO - this one tries the other three line orders and chooses the one which shows the best performance over some iterations on a matrix. While this may sound like a good thing to use, it is only so if an initial matrix is chosen and it takes a lot of time since a lot of iterations need to be made in order to make a good sample. I would recommend not to use AUTO order at all and instead to try all the line orders by hand with a number of iterations depending on the pattern and a good initial matrix; for instance, generated with a smaller number of iterations on the same pattern and with any line order.
- DESC - the lines are ordered in descending order depending on the number of one-entries. This follows the idea to start with the lines that are the hardest to map. Note that this algorithm does poorly if there are a lot of lines with the same number of one-entries (for example an identity matrix).
- MAX - it orders the lines so that the maximum number of important lines throughout the levels is as small as possible. This focuses straightforwardly to having many unimportant lines, which the program does not remember.
- SUM - it orders the lines so that the sum of the numbers of the important lines is the smallest possible throughout all levels. The purpose is the same as in the MAX order and quite often it is the case both approaches produce the same order.

- TWO - it orders the lines so that the maximum number of important lines in two consecutive levels throughout all the levels is as small as possible. This again focuses to having many unimportant lines, which the program does not remember. The constant two is chosen due to the fact general pattern always stores two levels of partial mapping at a time.

4.1.5 Mapping approaches

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

While the algorithm introduced in Chapter 2 finds out the partial mapping is invalid only at the time it maps two lines having a one-entry at their intersection to two lines having a zero-entry at the intersection, different approaches try to reveal the fact we would end up in the situation earlier by checking more conditions. Let P from Figure 4.1 be the forbidden pattern and imagine a situation,

$$P = \begin{matrix} & \begin{matrix} 5 & 6 & 7 & 8 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

Figure 4.1: Pattern P on which we demonstrate mapping approaches.

in which only lines 0, 3 and 7 are mapped and line 6 is currently being mapped. There are a few necessary conditions we can check:

Enough one-entries

The first condition is that there is enough one-entries in between mapped lines, which is schematically shown in Figure 4.2. We check whether there is enough one-entries on lines in between those lines, where lines 0 and 3 are mapped, so that there is a hope we can map lines 1 and 2 there. Similarly, we check whether there is a one-entry below the line, where line 3 is mapped so we can map line 4 there later.

Recursive mapping

While we were only testing whether there are enough one-entries in between already mapped lines in the previous approach, as you can see in Figure 4.3, this time we also check whether those one-entries can be used for the lines that are intended to be mapped there. For example, when we check there is a one-entry to be used for line 1 later, we also check the line 1 can be mapped to the row

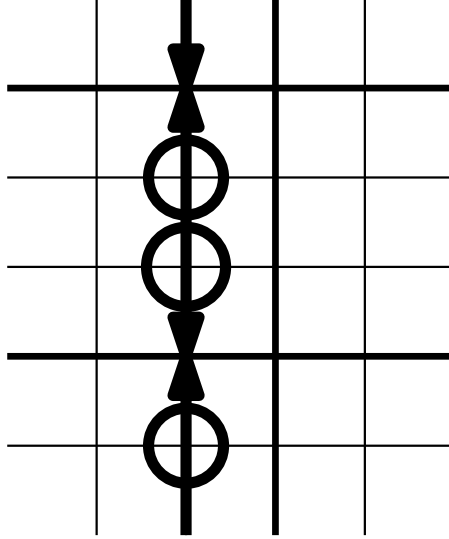


Figure 4.2: Checking whether there is enough one-entries. Bold lines in the picture are mapped and in circles are the positions where we look for one-entries.

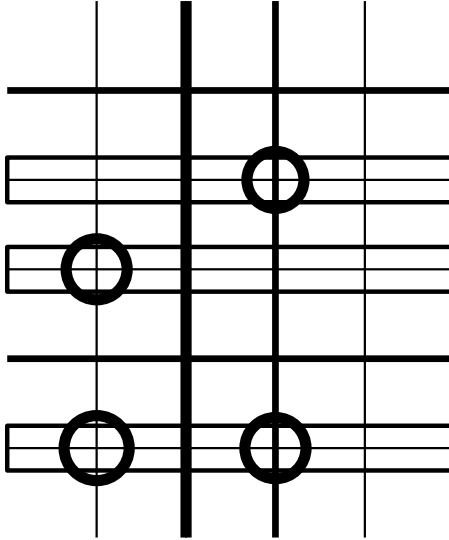


Figure 4.3: Checking whether crossed non-mapped lines can be mapped anywhere. Bold lines are mapped or being mapped, in rectangles are the lines we check and in circles are the positions where we look for one-entries.

430 with one-entry, which in this situation means to also check there is a one-entry
 431 at the intersection with the line to which the line 7 is mapped.

432 **Orthogonal bounds**

433 As shown in Figure 4.4, when we are adding line 6, we check whether there is
 434 enough one-entries on the already mapped lines orthogonal to line 6 between line
 435 6 and the closest mapped lines next to line 6. The idea is same as in “Enough
 436 one-entries”, but we check different lines.

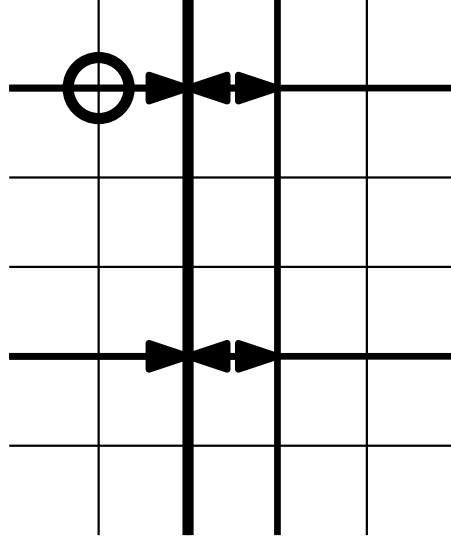


Figure 4.4: Checking whether there is enough one-entries on the orthogonal lines. Bold lines are mapped or being mapped and in circles are the positions where we look for one-entries.

Pattern	n	#iterations	one	rec	orth	time (sec)	memory
1 1 1	100	100 000	yes	yes	yes	1 512.56	3 268 B
1 1 1	100	100 000	yes	yes	no	1 504.46	3 268 B
1 1 1	100	100 000	no	no	no	672.63	3 268 B
1 1 1	50	100 000	yes	yes	yes	186.367	1 480 B
1 1 1	50	100 000	no	no	no	99.1557	3 004 B
1 1 1	50	100 000	yes	yes	yes	186.367	3 680 B
1 1 1	50	100 000	no	no	no	99.1557	3 680 B
1 0 0	500	10 000	yes	yes	yes	10276.6	9 128 B
	500	10 000	no	no	no	6756.96	15 972 B

Table 4.1: My caption

Usage

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

When testing that was done for a fixed pattern, we found out it is useful to use all the mentioned restrictions when generating a matrix of size 100×100 , as it turned out to be much faster than without the restrictions. On the other hand, in the same test for a generated matrix of size 500×500 , it was much better not to use any of those restrictions, which you can see in Table 4.1.

4.1.6 Using the whole structure in the next iteration

It may seem like a good idea to store all the partial mappings. In the next iteration, instead of finding all the partial mapping again, we only alter the mappings we remember. Let i be the number of the iteration we are in, and e be the element.

If the element e is changed from zero-entry to one-entry, for each partial

mapping we have stored in previous iterations, we want to try to extend it only by the line that just changed. If we manage to extend a partial mapping, we then try to extend it to a full mapping in all possible ways (not only by using changed lines). When the new line in such a mapping becomes unimportant, we can stop looking for all possible extensions if the mapping is equivalent with a different one, which comes from previous iterations. This can be easily done by means already used in the standard algorithm.

However, if the element e gets changed from one-entry to zero-entry we need to go through the partial mappings and delete all those that use e . This complicates the algorithm as we can no longer forget unimportant lines. Moreover, for each partial mapping we need to remember how many partial mappings of the previous level can be extended to that one, to delete that mapping from the list if there are no longer any mappings extensible to it.

This can all be done, but it comes with three huge inconveniences:

- Memory consumption - there can be a lot of partial mappings and we need to remember them all. We need to remember mappings of all levels and while we can still use the equivalence when extending a mapping, we need to also store all equivalent mappings for the purposes of deleting.
- The change from one-entry to zero-entry is no longer for free. If this change is done, we already know the pattern is not contained in M , but we still need to do a lot of work to change the structure in order to use it in the next iteration.
- Reverting - if the change is unsuccessful (the pattern is contained) we need to revert the change which means to completely revert all changes we did to the list of partial mappings. This can be either done by making a backup copy of the whole structure and override the structure if needed, which again is very costly as the structure is huge, or we can remember what partial mappings are new (or deleted) and we go through all partial mappings and remove (add) those. This means to iterate through the big structure one more time for every unsuccessful change.

After realizing these issues it no longer looks useful to me and this version of the algorithm is not a part of the implementation.

4.2 MCMC parallelism

To speed up 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 version 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 issue is that the 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, 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, then the computed iteration $x + 1$ is invalid and the iteration is going to be recomputed again, starting with the correct 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 Section 1.2) in order to approximate 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.

For the sake of clarity, from now on, we will not 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 then 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 with the lowest ID) we do:

- if it fails:
 - Do nothing - there is no change to propagate to the master copy of the generated matrix and all the tasks with higher ID expected this task to fail, which it did.
 - This increases the lowest ID by exactly one, as the task we speak of got processed.
- if it succeeds:
 - The main thread propagates the change tested by the task to the master copy of the generated matrix.

- 539 – All the rest of the task get removed as they all had a higher ID -
- 540 computed iterations that follow after the one just computed and they
- 541 expected the task to fail, which it did not.
- 542 – This increases the lowest ID by more then one, because there are tasks
- 543 that got removed and one that got processed.

544 4.2.1 Example of the MCMC process for n threads

545 At first, iterations 1 to $n - 1$ are assigned one to each worker as tasks with ID 1 to
 546 $n - 1$ with the same order as the order of iterations. If iteration 1 is not successful
 547 (which all the other iterations count on), everything is alright. However, if the
 548 iteration (its task) is successful, all the results of other tasks (and some of them
 549 might have been already finished) are cleared and those iterations get recomputed
 550 in tasks n to $2n - 3$ and the worker that computed task with ID 1 is assigned a
 551 new task with ID $2n - 2$ - to compute iteration n . The result of the task gets
 552 propagated to the master copy of the generated matrix only if all the tasks n to
 553 $2n - 3$ fail, else is gets recomputed. This is what happens till the end.

554 4.2.2 Speculative computing

555 It may easily happen that a task not having the lowest ID ends first. In that case,
 556 we could just wait until it has the lowest ID and process it later. This is not a
 557 very efficient approach. Instead we process the task immediately, but we do not
 558 propagate the changes to the master copy of the generated matrix until all tasks
 559 with lower ID fail and we do not stop the workers processing tasks with lower
 560 ID. When a task succeeds we remove all the changes computed by tasks with
 561 higher ID and override their private copy of the generated matrix. Also it might
 562 happen a task with even lower ID succeeds as well. This leads to more and more
 563 overriding. Luckily this is the only precarious situation we may encounter and it
 564 can be dealt with, even without copying the possibly huge generated matrix.

565 The way we deal with these inconveniences is described in Chapter 5 and
 566 should be clear from the code itself.

567 4.2.3 Reverting and synchronizing in the main thread

568 The speculative computing discussed above is not the only improvement we can
 569 make. It turns out to be costly to wake a thread to compute a trivial function,
 570 to set a few atomic variables and to fall asleep again. This happens a lot in the
 571 MCMC process. Every time a task succeeds it makes other workers revert the
 572 changes they computed and synchronize the successful change, which are both
 573 trivial functions.

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

580 4.3 Walking pattern

581 While the brute force implementation of an avoid algorithm for a general pattern
582 was improved heavily, the algorithm for a walking pattern (see Chapter 3) is very
583 fast in its nature and cannot be improved. Or can it be?

584 4.3.1 Using the last changed position

585 The MCMC process always changes one element e of the big matrix and asks
586 whether it still avoids the pattern. If it does not and we know that before the
587 change it did, we are sure e is a part of the pattern (a one-entry of the pattern
588 is mapped to it). Knowing that and using the same inductive proof as we did in
589 the proof of correctness of the avoid algorithm (see Chapter 2) it is sufficient to
590 only recompute the part of the inner structure under e and check if the last entry
591 of the pattern can be found there.

592 Not only that. We also know, using the fact the structure was completely
593 correct before the change, that if the values of both c_v and c_h of an element did
594 not change, the element will not cause the element underneath it to change and
595 we no longer have to recompute other parts of the structure.

596 To use both these facts we replace the cycle through the diagonals by a sim-
597 ple queue, starting at the position of the last changed element and putting more
598 positions in if the values of c_v or c_h are different than they were before recomput-
599 ing. The function ends either when the pattern is discovered or when the queue
600 becomes empty.

601 4.3.2 Lazy avoid

602 Lazy avoid is a variant of avoid function used when the MCMC parallelism (more
603 in Section 4.2) is chosen. While all the other types of patterns have a trivial
604 implementation of revert function, when using the walking pattern, the inner
605 structure needs to be modified even when reverting. The MCMC parallelism
606 turned out to work much better if the revert calls are handled by the main thread
607 and it requires the function to run as fast as possible so the other threads are not
608 blocked by the call for too long. That is a reason why functions lazy revert and
609 lazy avoid were created.

610 The avoid function expects the inner structure of the walking pattern to be
611 in a valid state and that requires some effort. To make lazy revert the fastest
612 possible, we postpone the work until the next call of lazy avoid, meaning that
613 lazy avoid then needs to do more things at once. It is no longer sufficient to only
614 compute the submatrix under the position changed last as we did above, but it
615 needs to also compute changes in the positions changed in those lazy revert calls
616 that are postponed.

617 We discuss several approaches, starting with the simplest one and ending with
618 the one that is fast and used in the final implementation.

619 Recompute the whole structure every time

620 The easiest way to implement lazy avoid is to always recompute the whole inner
621 structure. In that case, we do not worry which positions are correct and which

622 are not, because every time we find the pattern, we recomputed all the entries
623 that form it, so we know it really is there.

624 The weakness is efficiency. If the whole structure was correct and there was a
625 change of the last entry of the matrix it is sufficient to only recompute that one
626 entry. Instead we recompute a possibly very big structure. This results in a very
627 bad performance negating the advantage of parallel computation.

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

629 A simple improvement is to remember the changes done in previous calls of lazy
630 revert and together with the change done in lazy avoid call only recompute the
631 part of the structure that has possibly changed.

632 This gets more complicated when lazy avoid call discovers the pattern in $M_{\leq e}$,
633 because we cannot be sure the rest of the structure (everything under the diagonal,
634 where e is present) is in a correct order. It is still possible to remember some
635 horizontal, vertical and diagonal bounds and use them to restrict the recomputed
636 part of the matrix. However, the improvement is not that significant and we can
637 do better.

638 **Queue of positions to recompute**

639 A different approach is closer to the one used in a standard avoid function. Instead
640 of going through diagonals one after another, we have a queue of entries-to-
641 recompute. It is no longer sufficient to have a standard queue since in different
642 calls of lazy revert/avoid we can possibly change an entry of different priority
643 (the smaller diagonal the more important) so we need to have some kind of a
644 priority queue. That is exactly what I tried.

645 Using `std::priority_queue`, the function has no more problems with recomput-
646 ing the entries that were not influenced by the changes and uses all the benefits
647 mentioned in the previous section. But the container does not come for free and
648 in the end it turns out the price we pay for the operations on the priority queue
649 make the whole implementation comparably slow as in the previous attempts.

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

651 The final solution comes with the same idea, but a different storage. As the prior-
652 ity depends upon a diagonal (two entries on the same diagonal can be recomputed
653 in any order) we only remember a priority queue of diagonals and an array of
654 diagonals saying whether a diagonal is already a member of the priority queue.
655 As far as the entries are concerned, for every diagonal we have a `std::vector` of
656 entries-to-recompute as well as an array saying whether an entry is already a
657 member of the vector. Finally, it is the case that the storage used is not only
658 good theoretically but as the numbers say, also practically.

659 **4.4 Comparison of all methods**

660 We have improved all algorithms and added a parallel version of the MCMC
661 process. The question is, whether our improvements were useful in terms of
662 performance and memory consuming. I have done a few tests and in Table 4.2,

Pattern	n	#iterations	type	threads	time (sec)	memory
1 1 1	100	100 000	general	1	1486.61	< 100 KB
1 1 1	100	100 000	general	9	381.565	602 MB
1 0 0	100	100 000	general	17	332.465	1173 MB

Table 4.2: A table showing the difference between using parallel and serial MCMC process.

Pattern	n	#iter	type	#th	time (sec)	memory
0 0 0 0 1	100	100 000	general	1	3187.17	3.9 MB
0 0 0 1 0	100	100 000	general	9	569.237	24.3 MB
0 0 1 0 0	100	100 000	general	17	336.781	39.9 MB
0 1 0 0 0	100	100 000	walking	1	5.30831	2.9 MB
1 0 0 0 0	100	100 000	walking	9	2.06325	4.2 MB
	100	100 000	walking	17	1.91516	5.1 MB

Table 4.3: A table comparing general and walking pattern performance on the same pattern.

663 Table 4.3 and Table 4.4, you can see that at least in some cases walking pattern
664 does much better than general pattern and that parallel computation was not
665 added without a good reason. We always generate a matrix of size $n \times n$ and if
666 there is one thread, we use the serial variant of MCMC process; otherwise, we
667 use the parallel one.

Pattern	n	#iterations	type	#th	time (sec)	memory
I_{10}	100	100 000	general	1	1486.61	< 100 KB
	100	100 000	general	9	381.565	602 MB
	100	100 000	general	17	332.465	1173 MB
	500	100 000	walking	1	137.74	19 KB
	500	100 000	walking	9	38.8501	614 MB
	500	100 000	walking	17	31.4098	1210 MB
	5 000	10 000	walking	1	3588.79	467 MB
	5 000	10 000	walking	9	1105.49	5025 MB
	5 000	10 000	walking	17	685.041	9174 MB

Table 4.4: A table comparing general pattern and walking pattern, as well as usage of parallel version of MCMC process.

668 5. Technical documentation

669 In this chapter, we cover those parts of the algorithm that may be hard to un-
670 derstand just from the code. This only means functions that are technically
671 hard, for example functions with unexpected dependencies, side effects and so
672 on. Algorithmic difficult tasks are explained in Chapter 4.

673 5.1 Classes and API

674 First we list important classes of the program and explain their purpose.

675 5.1.1 Matrix

676 A minimalistic template container for storing and accessing matrices.

677 5.1.2 Pattern

678 An abstract class defining the interface of patterns. Three classes inherit from
679 the class:

- 680 • *General_pattern* – more in Chapter 2
- 681 • *Walking_pattern* – more in Chapter 3
- 682 • *Slow_pattern* – a class using a brute force algorithm to test pattern avoid-
683 ance.

684 For the case multiple patterns are chosen at the same time, every such Pattern
685 is stored in a container called Patterns, which is a class creating an interface
686 between Pattern and MCMCgenerator.

687 5.1.3 Statistics

688 To acquire, store and output statistics of the generating process, we use classes
689 in the file Statistics.hpp. There are two kinds of classes:

- 690 • *Matrix statistics* – these statistics store information about the structure of
691 all matrices that have been generated throughout the MCMC process. An
692 example of that is a histogram of occurrences of one-entries at all positions
693 of the matrix.
- 694 • *Performance statistics* – this is what we use to count how many changes
695 were successful and how long did it take to test a change.

696 If user wants to define their own statistics, they can add variables into the
697 class, alter “add_data” function to allow storing the desired data and change
698 function “print...” to define the output.

699 5.2 General_pattern

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

703 5.2.1 Construction

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

709 Storing the pattern

710 The first thing, which is done right after initialization of variables, is storing the
711 pattern. Instead of storing the pattern in a `Matrix<bool>`, I decided to store
712 lines into a number, where in the binary coding a one-entry in the position i
713 means there is a one-entry in the line at the intersection with i -th orthogonal
714 line. This comes handy when computing line orders. At the same time we also
715 find those lines that are empty (more in Chapter 4) and remember them, because
716 we do not have to map them at all.

717 Choosing the line order

718 After that, we need to choose the right line order (again more in Chapter 4).
719 To compute MAX, SUM or TWO order we use a brute force algorithm that
720 checks sequences of line adding and for each it computes how many lines are
721 unimportant. Then it just chooses the order which is the best in chosen metric.
722 To compute DESC order, we sort the lines according to the number of one-entries.

723 What to remember

724 In the next step, we find which lines are important in each level of partial map-
725 pings with respect to chosen order, because what to remember is based on the
726 equivalence introduced in Chapter 2 and the decision not to remember unimpor-
727 tant lines, which we explained in Chapter 4.

728 Parallel bound indices

729 Now comes the hardest to follow part – precomputing the indices for searching
730 for parallel bounds. The idea is simple. When we are adding a new line and we
731 already have a partial mapping, it restricts to where we can add the line. For
732 example, if there are three rows in the pattern and the rows 1 and 3 are mapped,
733 then line 2 needs to be mapped in between those two. The question is, where are
734 those two lines mapped to?

735 First, we add in a chosen order and second we do not remember all lines, as
736 some are unimportant. What do we want is to have an instant access to indices
737 of lines, which bounds added line in the partial mapping, so we do not need to

738 compute the index over and over again. That is exactly what gets computed when
 739 the function “find_parallel_bound_indices” is called. The series of other function
 740 calls follows just because we compute the indices for all added lines in the order
 741 in which they are going to be added.

742 **Extending order**

743 The last function, “find_extending_order” specifies how we store an extended
 744 partial mapping. Again, unimportant lines play their role here and it may easily
 745 be the case from a partial mapping storing k lines, after mapping one more
 746 line, we end up with a partial mapping only storing $k - 1$ lines, because two
 747 lines become unimportant by adding the line. This means we not only copy the
 748 previous mapping and add the new mapped line but also remove unimportant
 749 lines. This function precomputes which values are going to be copied and which
 750 are going to be skipped.

751 **5.3 MCMC parallelism**

752 While the idea behind MCMC parallelism is described in Section 4.2 and the
 753 code is heavily commented, the work done by the main thread may still be hard
 754 to understand.

755 Let I be the ID the process is currently waiting for, that is, the lowest ID of
 756 a task that is being tested by a worker. In a structure called “queue” (which is
 757 `std::vector<std::deque>`) each worker has a queue of tasks assigned to it. In the
 758 queue, there are tasks that are either being computed or have been computed.
 759 The history of tasks is needed to allow reverting changes that should have not
 760 happen when the main thread encounters a different successful task with lower
 761 ID. There is no need to have a complete history of all tasks computed. There
 762 are only those tasks, that have higher ID than I or have lower ID, but those are
 763 going to be removed from the “queue” as soon as possible. The name “queue” is
 764 not random, it describes the order, in which the tasks are being stored – the tasks
 765 with lower ID have been inserted earlier and therefore they are at the bottom.

766 Now that we know the most important structure let us see how the main
 767 thread works with it. This is a list of operations changing “queue” and the
 768 situations, in which we perform them:

- 769 • `pop_front`: The main thread deletes the first task (the one with the lowest
 770 ID) if one of two things happen:
 - 771 – The ID of the task being deleted is equal to I . That means the change
 772 computed by the task is being propagated to the generated matrix and
 773 there is no need to remember the task anymore. This also increases I ,
 774 not necessarily by one.
 - 775 – The ID of the task being deleted is less than I . This situation happens
 776 due to synchronization. The worker was supposed to synchronize a
 777 task computed by a different worker that did not have the lowest ID
 778 at the time. Therefore, the task needs to be in the list of tasks so we
 779 can revert it later, if needed. If there is no need to revert it and the

lowest ID gets greater or equal to the ID of the task, we can just delete it from the “queue”.

- `pop_back`: There is only one reason to delete tasks from the end of the “queue” and that is reverting. Imagine there is a task with id J at the end of the “queue”. A different worker computes a task with lower ID and finds out the change is successful. This means the task J will not propagate to the generated matrix and there is no use for it. If it is still being computed, we cannot do much about it, so we tell the worker to stop computing and deal with it later. If the task is finished, we need to revert it, but only in case the task was successful, because if it was not, it had already been reverted by the worker. So we revert the task if needed and we can just delete it from “queue” as it will never be used.
- `emplace_back`: The main thread only inserts new tasks to the end of the “queue” and there are two reasons to insert:
 - Worker is assigned a completely new task to check the avoidance. In this situation, the task is given a new, globally highest ID and we add the task at the end of the “queue”.
 - The second reason to insert into “queue” are synchronizations. The situation is the same as it was in the case, when we `pop_back` – after we revert all the tasks in the list, we need to synchronize changes that forced reverting and if their ID is not lower or equal to I , we need to add them to the list so they can be reverted if needed.

802 6. User documentation

803 In the last chapter of the thesis, we first describe how to install the program and
804 then show how to make the program generate random matrices or to test whether
805 a certain matrix avoids a given forbidden pattern.

806 6.1 Installation

807 The program is written in C++ and should be compilable on any standard plat-
808 form. To use it, you either just use an executable file (Windows) or build the
809 program using C++ compiler.

810 6.1.1 Windows

811 Windows users can run the application using an executable file “matrix-win.exe”
812 either directly, in which case the default configuration file will be used, or using
813 command line with an optional parameter specifying the configuration file.

814 6.1.2 Unix, Linux, MacOS

815 Users on other platforms than Windows can build the solution using command
816 line easily by running “./build.sh”, which uses G++ compiler. Compiler can be
817 switched by rewriting g++ to some other variant (for example clang) in build.sh
818 file. This leads to creating a an executable file “matrix.exe” which can be run
819 with an optional parameter specifying the configuration file.

820 6.2 Configuration file

821 In order to modify what the program computes, we use a configuration file. The
822 configuration file can be chosen when running the program in command line
823 and relative path to it is the first (and only) option. If no path is inserted, the
824 configuration file is expected to be located in the same directory as the executable
825 file and its name is “config.txt”.

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

- 828 • input
- 829 • pattern
- 830 • output
- 831 • statistics

832 The order of the sections is not fixed and there can be additional empty lines for
833 better readability. In each section, there is a list of values that can be set either
834 to arbitrary value or to a specific one. There is at most one command of format

835 “option=value” per line and there might be additional white spaces surrounding
836 the “=” sign.

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

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

844 6.2.1 Input

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

- 846 • size: The size of the generated matrix. Results in $M \in \{0, 1\}^{size \times size}$.

Possible value: $s \in \mathbb{N}$
847 Default value: 100

- 848 • iterations: The number of iterations of the MCMC process.

Possible value: $i \in \mathbb{N}$
849 -1 - tests avoidance of the initial pattern
Default value: 10,000

- 850 • random_seed: The random seed for the MCMC process.

Possible value: $s \in \mathbb{N}$
851 “random” - chooses a random seed
Default value: “random”

- 852 • init_matrix: A $size \times size$ matrix the MCMC process starts with.

Possible value: *matrix file path*
853 “zero” - a matrix containing no one-entries
Default value: “zero”

- 854 • parallel_mode: A choice to compute in parallel or serial.

Possible value: “serial”
855 “mcmc” - more in Section 4.2
Default value: “serial”

- 856 • threads_count: The number of threads used if a parallel mode is chosen.

Possible value: $t \in \mathbb{N}$
857 -1 - chosen according to the number of cores
Default value: 1

858 6.2.2 Pattern

859 In this section, we set the options that matter the most – matrix patterns. As
860 we are allowed to generate a matrix which avoids more than just one pattern,
861 the section [pattern] can be used multiple times, specifying one pattern for each
862 occurrence.

863 • `pattern_file`: A relative path to a input matrix file - the pattern.
 Possible value: *matrix file path*
 864 Default value: `"pattern/input.txt"`

865 • `pattern_type`: The type of the pattern. Determines the method used for
 866 testing avoidance.
 Possible value: `"general"`
 `"walking"` - see Chapter 3
 867 `"slow"` - brute force algorithm for a general pattern
 Default value: `"general"`

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

870 First we can decide what mapping approaches to use. More about them in
 871 Section 4.1.5.

872 • `map_one_entries`: If set to `"yes"`, the map function tests whether there is
 873 enough one-entries in between already mapped lines.
 Possible value: `"yes"`
 874 `"no"`
 Default value: `"yes"`

875 • `map_recursion`: If set to `"yes"` and the `map_one_entries` is also set to `"yes"`,
 876 the map function tests mapping recursively.
 Possible value: `"yes"`
 877 `"no"`
 Default value: `"yes"`

878 • `map_orthogonal_bounds`: If set to `"yes"`, the map function also tests the
 879 orthogonal bounds of added line.
 Possible value: `"yes"`
 880 `"no"`
 Default value: `"no"`

881 • `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)
 882 `"vector"` - `std::vector` (dynamic array)
 Default value: `"hash"`

883 • `line_order`: Chooses the order in which the lines are being added to the
 884 partial mapping. See Section 4.1.4.
 Possible value: `"max"`
 `"two"`
 `"sum"`
 885 `"desc"`
 `"auto"`
 order file path
 Default value: `"max"`

886 6.2.3 Output

887 In this section, we specify, where to output the generated matrix or statistics
888 files. As the matrix can be output to console, a text file or a bmp file, each
889 option in the section can be set more than once and every occurrence will make
890 a new output.

- 891 • `matrix_output`: The generated matrix can be output as a bmp file in which
892 one-entries are black pixels and zero-entries white. To do that, the file path
893 has to have a pattern “*.bmp”. If a different path is given the file is stored
894 as a matrix text file. It can also be output into a console if “console” is set.
895 In that case it has the text format.

Possible value: “console”
matrix bmp file path
matrix text file path
“no”

Default value: “no”

- 897 • `performance_stats`: If the serial computation is chosen, the program can
898 output a statistics like the percentage of avoid call success, how long did
899 one call take on average and what was the average size of structures. If more
900 patterns are chosen at the same time, the statistics may get misleading as
901 they also count the cases when the first pattern is contained in the matrix
902 and the other patterns are not tested at all.

Possible value: “console”
performance file path
“no”

Default value: “no”

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

Possible value: “console”
csv file path
“no”

Default value: “no”

- 907 • `time_to_console`: Prints how long the computation took into a console.

Possible value: “yes”
“no”

Default value: “no”

- 909 • `patterns_to_console`: Prints all the used patterns into the console.

Possible value: “yes”
“no”

Default value: “no”

911 6.2.4 Statistics

912 The last section handles the options important for scientists. While generating
913 a random matrix is a great result, on its way the program can also create some

914 statistics, namely make a histogram of occurrences of one-entries in a generated
 915 matrix as the MCMC iterates as well as store the matrix with the highest amount
 916 of one-entries. As the process usually does not start with a random matrix, the
 917 user can decide to only compute the statistics after a certain number of iterations
 918 has been done and to only check a small portion of iterations, every 10th for
 919 instance, as a single iteration may not make any difference and counting the
 920 histogram takes time.

- 921 • `histogram.frequency`: Sets how often the histogram gets refreshed.
 Possible value: $f \in \mathbb{N}$
 0 - the histogram is not computed at all
 Default value: 0
- 923 • `histogram.initial`: Sets the initial iteration of the MCMC process when the
 histogram gets refreshed.
 Possible value: $i \in \mathbb{N}$
 Default value: 1,000
- 926 • `histogram.final`: Sets the last iteration of the MCMC process when the
 histogram gets refreshed.
 Possible value: $f \in \mathbb{N}$
 -1 - the histogram is computed till the end
 Default value: -1
- 929 • `histogram.file`: Sets where to output the histogram computed during the
 MCMC process.
 Possible value: matrix bmp file path
 matrix text file path
 "console"
 "no"
 Default value: "no"
- 932 • `max_ones_matrix.file`: Sets where to output the matrix that had the most
 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"

935 6.3 File input

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

940 6.3.1 Matrix file

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

- 942 • 2 natural numbers specifying the number of rows and columns in this order.
- 943 • a sequence of zeros and ones of length rows×columns specifying the matrix
- 944 from the top left corner one row after another.

945 **Example:** 2 3
 1 0 1
 1 1 0

946 6.3.2 Order file

947 If you want to choose the order in which the lines are going to be mapped when
948 a general pattern is chosen, it is your responsibility to check that all lines that
949 need to be mapped are mapped. It is for example possible to only map three lines
950 even if the pattern consists of six lines just because there is for example no need
951 to map empty lines at all. Therefore the program does not check the validity of
952 the order and just uses it.

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

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

957 2 1 0 3 4

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

960 6.4 File output

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

962 6.4.1 Matrix text file

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

- 964 • 2 natural numbers specifying the number of rows and columns in this order.
- 965 • a sequence of zeros and ones of length rows×columns specifying the matrix
- 966 from the top left corner one row after another.

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

971 **6.4.2 Matrix bmp file**

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

Conclusion

We have seen, how to use Markov chain Monte Carlo method to approximate a random matrix of given size avoiding a given set of forbidden patterns. We have used a heavily improved brute force algorithm to test avoidance of a general pattern and a simple dynamic programming algorithm for testing avoidance of a pattern, where all one-entries are contained on a walk in the pattern. To even more improve performance of the program, we have used parallel computing in the layer of the MCMC process.

The question is, whether we can do better? We probably cannot asymptotically improve the walking pattern avoidance algorithm (more in Chapter 3), because it is already linear in the number of elements of the matrix and computes only those elements that have changed. On the other hand, the algorithm is only useful for a small portion of all possible patterns. A natural question is, whether we can do as good for other classes of matrices?

In Section 4.1.6, we mentioned an alternative algorithm for testing avoidance of general patterns. Although it is probably not a very good algorithm for big matrices, because it needs to store all possible partial mappings, it may still turn out to be much better for smaller matrices, which are still big enough to see a structure in them, than the implemented algorithm is.

To use parallelism, we decided to use the MCMC layer of the program. There were several reasons to do so. First of all, it is not dependent on the avoidance pattern algorithm; therefore, if someone comes with a better one, it can still use the parallel version of MCMC. Secondly, when the matrix gets complex enough, the probability a change of one element is going to be successful is very small; thus, we have chosen to iterate through unsuccessful changes faster over the possibility to test avoidance faster.

A different approach would be to make avoidance testing parallel. In walking pattern, we could recompute multiple elements of the same diagonal at once, because they do not influence each other. In the algorithm for general patterns, we could generate new partial mappings in parallel, since they are independent. The only problem there is deciding whether two mappings are equivalent.

1007 Bibliography

1008 Neal Madras. *Lectures on monte carlo methods*, volume 16. Springer Science &
1009 Business, 2002.

1010 Neal Madras and Hailong Liu. Random pattern-avoiding permutations. *Algo-*
1011 *rithmic Probability and Combinatorics, AMS, Providence, RI*, pages 178–180,
1012 2010.

1013 List of Figures

1014	1	Matrix M_1 contains the pattern P , because all the conditions	
1015		are satisfied by mapping $\{(0, 0), (1, 2), (2, 3), (3, 4)\}$. On the other	
1016		hand, matrix M_2 avoids P as there is no such mapping.	4
1017	2	Example of a generated matrix avoiding I_{10} (unit matrix). Black	
1018		dots are one-entries and white are zero-entries. As you can see,	
1019		matrices avoiding a pattern can have a nice structure.	5
1020	2.1	An example showing unimportant line and equivalent mappings. .	10
1021	3.1	An example of a walk W in matrix M and the order of entries in W .	11
1022	3.2	Diagonals of an matrix and the order in which the algorithm for	
1023		walking pattern iterates through them.	12
1024	3.3	The algorithm testing avoidance for walking patterns cannot be	
1025		easily generalized for all patterns.	15
1026	4.1	Pattern P on which we demonstrate mapping approaches.	18
1027	4.2	Checking whether there is enough one-entries. Bold lines in the	
1028		picture are mapped and in circles are the positions where we look	
1029		for one-entries.	19
1030	4.3	Checking whether crossed non-mapped lines can be mapped any-	
1031		where. Bold lines are mapped or being mapped, in rectangles are	
1032		the lines we check and in circles are the positions where we look	
1033		for one-entries.	19
1034	4.4	Checking whether there is enough one-entries on the orthogonal	
1035		lines. Bold lines are mapped or being mapped and in circles are	
1036		the positions where we look for one-entries.	20

List of Tables

1037			
1038	4.1	My caption	20
1039	4.2	A table showing the difference between using parallel and serial	
1040		MCMC process.	26
1041	4.3	A table comparing general and walking pattern performance on	
1042		the same pattern.	26
1043	4.4	A table comparing general pattern and walking pattern, as well as	
1044		usage of parallel version of MCMC process.	26

1045 Attachments