

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

Contents

Preface	3
Introduction	5
1 Markov chain Monte Carlo	6
1.1 Markov chains	6
1.2 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 Inner structures	11
3.2.2 The algorithm	11
3.2.3 Correctness	12
3.2.4 Generalization	13
4 Improvements to basic algorithms	15
4.1 General pattern	15
4.1.1 Improving memory consumption	15
4.1.2 Not mapping empty lines	15
4.1.3 Using the last changed position	15
4.1.4 Line order	16
4.1.5 Mapping approaches	17
4.1.6 Using the whole structure in the next iteration	19
4.2 MCMC parallelism	20
4.2.1 Example of the MCMC process for n threads	21
4.2.2 Speculative computing	22
4.2.3 Reverting and synchronizing in the main thread	22
4.3 Walking pattern	22
4.3.1 Using the last changed position	22
4.3.2 Lazy avoid	23
5 Technical documentation	25
5.1 Classes and API	25
5.1.1 Matrix	25
5.1.2 Pattern	25
5.1.3 MCMCgenerator	25
5.1.4 Statistics	25
5.2 General pattern	26
5.2.1 Construction	26
5.3 Parallel computing	27
5.3.1 MCMC parallelism	27

5.4	Library interface	28
6	User documentation	29
6.1	Installation	29
6.1.1	Windows	29
6.1.2	Unix, Linux, iOS	29
6.2	Configuration file	29
6.2.1	Input	30
6.2.2	Pattern	30
6.2.3	Output	32
6.2.4	Statistics	32
6.3	File input	33
6.3.1	Matrix file	34
6.3.2	Order file	34
6.4	File output	34
6.4.1	Matrix text file	34
6.4.2	Matrix bmp file	35
	Conclusion	36
	Bibliography	37
	List of Figures	38
	List of Tables	39
	List of Abbreviations	40
	Attachments	41

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 • Implementing the algorithms to provide practical tool

12 One point 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 with a process of generating random pattern-
15 avoiding matrices.

Introduction

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

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

Figure 1: Matrix M_1 contains the pattern P , because a mapping $\{(0, 0), (1, 2), (2, 3), (3, 4)\}$ satisfies all the conditions. 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. 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 $P \in \mathcal{P}$.

Definition 2. We denote by $\mathcal{M}_n(\mathcal{P})$ a set of all binary matrices of size n by n avoiding \mathcal{P} as submatrices.

Definition 3. 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.

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

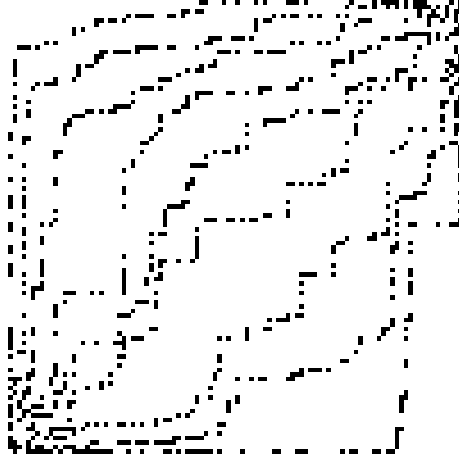


Figure 2: Example of a generated matrix avoiding I_{10} (unit matrix). Black dots are one-entries and white are zero-entries. As you can see, matrices avoiding a pattern can have a nice structure.

48 Generating random matrix

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

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

65 Testing avoidance

66 In each step of our MCMC process we need to test whether a matrix avoids a
 67 pattern. We will show a very fast algorithm that only works for a special class of
 68 binary matrices (explained in Chapter 3) together with a slightly less performing
 69 algorithm for a general pattern, which, again, comes as a generalization of an
 70 algorithm for permutations from the article by Madras and Liu and is described
 71 in Chapter 2.

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

74 In Chapter 5 some technical details are explained to make reading the code
 75 easier for reader and to describe user interface. The last chapter (Chapter 6)
 76 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. In this work 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 theorem can be found in Madras [2002].

Theorem 1. Consider an aperiodic irreducible Markov chain with 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, if \mathcal{S} is finite, then

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

for every $j \in \mathcal{S}$. That is, if we write π to denote the row vector whose entries are π_i , then $\pi P = \pi$.

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:

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

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

113 Symmetry

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

116 Irreducibility

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

123 Aperiodicity

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

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

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

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

135 2.1 Sketch of a brute force algorithm

136 Let $L = (l_1, l_2, \dots, l_{w+h-1})$ be a permutation of lines (rows and columns) of the
137 pattern P and $k \in [w + h - 1]$. *Partial mapping of level k* of lines of P is a
138 function f from $L' := \{l_1, l_2, \dots, l_k\} \subseteq L$ to lines of the big matrix M satisfying
139 three conditions:

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

146 The basic algorithm we use goes as follows. First it maps l_1 to all possible lines
147 of M , creating partial mappings of $\{l_1\} \subseteq L$. For $k = 2, \dots, w + h - 1$ it takes
148 each partial mapping from the previous iteration and extends it by adding line l_k
149 to the partial mapping in all possible ways. If we manage to map all the lines of
150 P , then M does not avoid it and if at some point there are no partial mappings
151 to extend it means M avoids P .

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

158 2.2 Equivalent mappings

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

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

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

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

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

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

$$P = \begin{matrix} & 4 & 5 & 6 & 7 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{matrix} \quad M = \begin{matrix} & 5 & 6 & 7 & 8 & 9 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{matrix}$$

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

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

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

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

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

203 3.1 Walking pattern

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

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

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \begin{array}{c} w_1 \\ w_2 \quad w_3 \quad w_4 \\ \quad w_5 \quad w_6 \\ \quad \quad w_7 \end{array}$$

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

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

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

215 3.2 Dynamic program

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

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

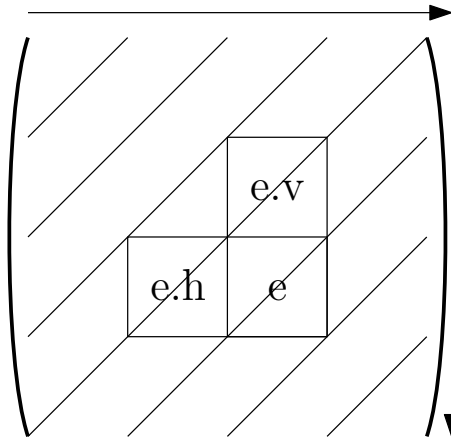


Figure 3.2: Diagonals of an matrix and the order in which the algorithm for walking pattern iterates through them.

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

250 $[-1, i]$ is a vertical entries and $[i, -1]$ is a horizontal for every $i \geq 0$. When we ask
 251 whether w_k can be mapped to e , where e is an element of M , we check whether
 252 w_k stands for a one-entry of P and if it does, we require e to be a one-entry too.
 253 For an $n \times n$ matrix M the algorithm works as follows:

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

274 3.2.3 Correctness

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

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

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

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

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

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

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

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

310 To prove $c_h(e)$ has a correct value, we proceed symmetrically.

311 3.2.4 Generalization

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

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

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

324 4. Improvements to basic 325 algorithms

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

328 4.1 General pattern

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

330 4.1.1 Improving memory consumption

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

336 In Chapter 2 we also introduced the idea of (un)important lines for a partial
337 mapping of level l and equivalence based on not using unimportant lines at all -
338 as they are fully bounded by other already mapped lines. When a line becomes
339 unimportant it stays unimportant till the end of the run; as a result, we can
340 forget where we mapped those lines to save memory.

341 4.1.2 Not mapping empty lines

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

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

347 4.1.3 Using the last changed position

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

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

- 358 • The first row of P is mapped under the r -th row. This prevents any other
359 row to be mapped to r -th one and we don't 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 four 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.

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

405 4.1.5 Mapping approaches

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

409 While the algorithm introduced in Chapter 2 finds out the partial mapping
 410 is invalid only at the time it maps two lines having a one-entry at their inter-
 411 section to two lines having a zero-entry at the intersection, different approaches
 412 try to reveal the fact we would end up in the situation earlier by checking more
 413 conditions.

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

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

Figure 4.1: TODO comment.

414
 415 We are in a situation when only lines 0, 3 and 7 are mapped and line 6 is
 416 currently being mapped. Here are some mapping approaches:

417 Enough one-entries

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

423 Recursive mapping

424 While we were only testing whether there are enough one-entries in between
 425 already mapped lines in the previous approach, this time, we also check whether

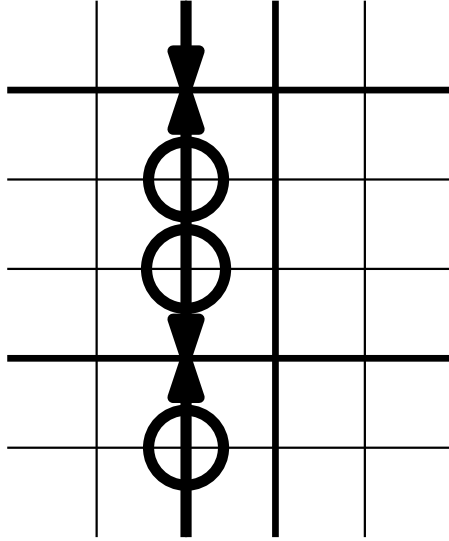


Figure 4.2: TODO comment.

426 those one-entries can be used for the lines that are intended to be mapped there.
 427 For example, when we check there is a one-entry to be used for line 1 later, we
 428 also check the line 1 can be mapped to that row, which in this situation means
 429 to also check there is a one-entry at the intersection with the line to which the
 line 7 is mapped.

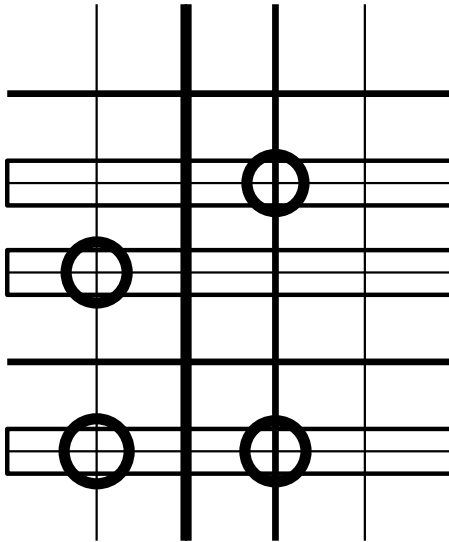


Figure 4.3: TODO comment.

430

431 **Orthogonal bounds**

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

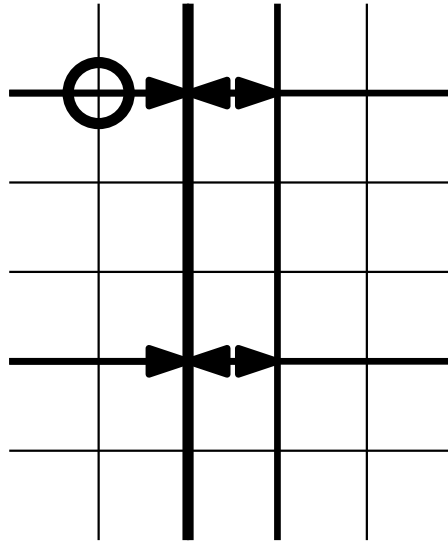


Figure 4.4: TODO comment.

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

In the 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.

4.1.6 Using the whole structure in the next iteration

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

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

However, if the element gets changed from one-entry to zero-entry we need to go through the partial mappings and delete those that used the currently changed one-entry. This gets a bit messy as we can no longer forget unimportant lines and 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 that one.

This can all be done, but the whole thing comes with three huge inconveniences:

- Memory consumption - there can be a LOT of partial mappings and we

464 need to remember them all. Of course we can still use the equivalence but
465 we need to remember mappings of all levels.

- 466 • The change from one-entry to zero-entry is no longer for free. If this change
467 is done, we already know the pattern is not contained in M , but we still
468 need to do a lot of work to change the structure in order to use it in the
469 next iteration.
- 470 • Reverting - if the change is unsuccessful (the pattern is contained) we need
471 to revert the change which means to completely revert all changes we did to
472 the list of partial mappings. This can be either done by making a backup
473 copy of the whole structure and override the structure if needed, which again
474 is very costly as the structure is huge, or we can remember what partial
475 mappings are new and we go through all partial mappings and remove those
476 new ones. This again means to iterate through the big structure one more
477 time.

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

480 4.2 MCMC parallelism

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

484 While the serial MCMC generator in each iteration changes one element in
485 the generated matrix and checks whether it still avoids forbidden patterns, the
486 parallel one makes several iterations at once, one on each copy of the generated
487 matrix. This means that while iteration x is being computed by a thread, iteration
488 $x + 1$ can at the same time be computed by a different thread. The only issue is
489 that iteration $x + 1$ does not know what is going to be the state of the generated
490 matrix at the time it should start. It expects iteration x to fail - not change the
491 generated matrix at all, counting on the fact at some point it is unlikely a change
492 does not create a mapping of the pattern, and starts with the same matrix as
493 iteration x . If iteration x succeeds though, then the computed iteration $x + 1$
494 is invalid and the iteration is going to be recomputed again, starting with the
495 altered matrix.

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

502 The job of a worker is only to check if its copy of the matrix still avoids the
503 pattern when one bit is changed. On the other hand all synchronization is left to
504 the main thread. As mentioned before, one iteration of the MCMC process can be
505 recomputed several times. We still want the generator to satisfy the conditions we
506 have for the Markov chain (more in Section 1.2) in order to generate a random
507 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 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.
 - All the rest of the task get removed as they all had a higher ID - computed iterations that follow after the one just computed and they expected the task to fail, which it did not.
 - This increases the lowest ID by more than one, because there are tasks that got removed and one that got processed.

4.2.1 Example of the MCMC process for n threads

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

552 4.2.2 Speculative computing

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

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

566 4.2.3 Reverting and synchronizing in the main thread

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

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

579 4.3 Walking pattern

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

583 4.3.1 Using the last changed position

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

591 Not only that. We also know, using the fact the structure was completely
592 correct before the change, that if the values of both c_v and c_h of an element did

not change, the element won't cause the element underneath it to change and we no longer have to recompute the other parts of the structure.

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

4.3.2 Lazy avoid

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

The avoid function expects the inner structure of the walking pattern (see Chapter 3) to be in a valid state and that requires some effort. To make lazy revert the fastest possible, we postpone the work until the next call of lazy avoid, meaning that lazy avoid then needs to do more things at once. It is no longer sufficient to only compute the submatrix under the position changed last as we did above, but it needs to also compute changes in the positions changed in those lazy revert calls that are postponed.

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

Recompute the whole structure every time

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

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

Recompute only a part of the structure diagonal by diagonal

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

This gets a bit tricky when lazy avoid call actually discovers the pattern because we cannot be sure the rest of the structure is in a correct order. It is still possible to remember some horizontal, vertical and diagonal bounds and use

636 them to restrict the recomputed part of the matrix. The improvement is not that
637 significant though and we can do better.

638 **Queue of positions to recompute**

639 A different approach is closer to the one used in a standard avoid function. In-
640 stead of going through diagonal 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 (the
643 higher the more important) so we need to have some kind of a priority queue.
644 That is exactly what I tried.

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

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

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

660 5. Technical documentation

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

665 5.1 Classes and API

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

667 5.1.1 Matrix

668 A minimalistic template container for storing and accessing matrices.

669 5.1.2 Pattern

670 An abstract class defining the interface of patterns. Three classes inherit from
671 the class:

- 672 • *General pattern* – more in Chapter 2
- 673 • *Walking pattern* – more in Chapter 3
- 674 • *Slow pattern* – a class using a brute force algorithm to test pattern avoid-
675 ance.

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

679 5.1.3 MCMCgenerator

680 A class that for given Patterns and given matrix repeats MCMC process described
681 in Section 1.2 using given number of iterations.

682 5.1.4 Statistics

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

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

691 5.2 General pattern

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

695 5.2.1 Construction

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

701 Storing the pattern

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

709 Choosing the line order

710 After that we need to choose the right line order (again more in Chapter 4).
711 To compute MAX or SUM order we just use a brute force algorithm that checks
712 sequences of line adding and for each it computes how many lines are unimportant.
713 Then it just chooses the order which is the best in chosen metric.

714 What to remember

715 In the next step, we find what do we need to remember in each level of partial
716 mappings with respect to chosen order. As mention earlier, for MAX or SUM
717 order it is already computed when finding the order, but for other variants of
718 orders it is not, so we just compute it every time. What to remember is based
719 on the equivalence introduced in Chapter 2 and the decision not to remember
720 unimportant lines (which we explained in Chapter 4).

721 Parallel bound indices

722 Now comes the hardest to follow part - precomputing the indices for searching for
723 parallel bounds. The idea behind is simple. When we are adding a new line and
724 we already have a partial mapping, it restricts to where we can add the line. For
725 example, if there are three rows in the pattern and the rows 1 and 3 are mapped,
726 the second one need to be mapped in between those two. The question is, where
727 are those two lines mapped to? First we add in a chosen order and second we do
728 not remember all lines, as some are unimportant. What do we want is to have
729 a instant access to the index of the line, which bounds added line, in the partial

730 mapping so we do not need to compute the index over and over again. That is
731 exactly what gets computed when the function “find_parallel_bound_indices” is
732 called. The series of other function calls follows just because we compute the
733 indices for all added lines in the order in which they are going to be added.

734 Extending order

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

741 5.3 Parallel computing

742 5.3.1 MCMC parallelism

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

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

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

- 759 • `pop_front`: The main thread deletes the first tasks (the one with the lowest
760 ID) if one of two things happen:
 - 761 – The ID of the task being deleted is equal to I . That means the change
762 computed by the task is being propagated to the generated matrix and
763 there is no need to remember the task anymore. This also increases I ,
764 not necessarily by one.
 - 765 – The ID of the task being deleted is less than I . This situation is due
766 to synchronization. The worker was supposed to synchronize a task
767 computed by a different worker that did not have the lowest ID at the
768 time. Therefore the task needs to be in the list of tasks so we can
769 revert it if needed. If there is no need to revert it and the lowest ID
770 gets greater or equal to the ID of the task, we can just delete it from
771 the “queue”.

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

- 782 • `emplace_back`: The main thread only inserts new tasks to the end of the
783 “queue” and there are two reasons to insert:
 - 784 – Worker is assigned a completely new task to check the avoidance. In
785 this situation the task is given a new, globally highest ID and we add
786 the task at the end of the list.
 - 787 – The second reason to insert into “queue” are, again, synchronizations.
788 The situation is the same as it was in the case, when we `pop_back` -
789 after we revert all the tasks in the list, we need to synchronize changes
790 that forced reverting and if their ID is not lower or equal to I , we need
791 to add them to the list so they can be reverted if needed.

792 5.4 Library interface

793 6. User documentation

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

797 6.1 Installation

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

801 6.1.1 Windows

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

805 6.1.2 Unix, Linux, iOS

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

811 6.2 Configuration file

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

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

- 818 • input
- 819 • pattern
- 820 • output
- 821 • statistics

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

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

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

834 6.2.1 Input

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

- 836 • size: The size of the generated matrix. Results in $M \in \{0, 1\}^{size \times size}$.
 Possible value: $s \in \mathbb{N}$
 837 Default value: 100
- 838 • iterations: The number of iterations of the MCMC process.
 Possible value: $i \in \mathbb{N}$
 839 Default value: 10,000
- 840 • random_seed: The random seed for the MCMC process.
 Possible value: $s \in \mathbb{N}$
 841 “random” - chooses a random seed
 Default value: “random”
- 842 • init_matrix: A $size \times size$ matrix the MCMC process starts with.
 Possible value: *matrix file path*
 843 “zero” - a matrix containing no one-entries
 Default value: “zero”
- 844 • parallel_mode: A choice to compute in parallel or serial.
 Possible value: “serial”
 845 “mcmc” - more in Section 4.2
 Default value: “serial”
- 846 • threads_count: The number of threads if a parallel mode is chosen.
 Possible value: $t \in \mathbb{N}$
 847 -1 - chosen according to the number of cores
 Default value: 1

848 6.2.2 Pattern

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

- 853 • pattern_file: A relative path to a input matrix file - the pattern.
 Possible value: *matrix file path*
 854 Default value: “pattern/input.txt”

855 • `pattern_type`: The type of the pattern. Determines the method used for
856 testing avoidance.

 Possible value: `“general”`
 `“walking”` - see Chapter 3
857 `“slow”` - brute force algorithm for a general pattern
 Default value: `“general”`

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

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

862 • `map_one_entries`: If set to `“yes”`, the map function tests whether there is
863 enough one-entries in between already mapped lines.

 Possible value: `“yes”`
864 `“no”`
 Default value: `“yes”`

865 • `map_recursion`: If set to `“yes”` and the `map_one_entries` is also set to `“yes”`,
866 the map function tests mapping recursively.

 Possible value: `“yes”`
867 `“no”`
 Default value: `“yes”`

868 • `map_orthogonal_bounds`: If set to `“yes”`, the map function also tests the
869 orthogonal bounds of added line.

 Possible value: `“yes”`
870 `“no”`
 Default value: `“no”`

871 • `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)
872 `“vector”` - `std::vector` (dynamic array)
 Default value: `“hash”`

873 • `line_order`: Choose the order in which the lines are being added to the
874 partial mapping. See Section 4.1.4.

 Possible value: `“max”`
 `“two”`
 `“sum”`
875 `“desc”`
 `“auto”`
 `“order file path”`
 Default value: `“max”`

876 6.2.3 Output

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

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

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

Default value: `“no”`

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

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

Default value: `“no”`

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

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

Default value: `“no”`

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

Possible value: `“yes”`
897 `“no”`

Default value: `“no”`

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

Possible value: `“yes”`
899 `“no”`

Default value: `“no”`

900 6.2.4 Statistics

901 The last section handles the options important for scientists. While generating
902 a random matrix is a great result, on its way the program can also create some
903 statistics, namely make a histogram of occurrences of one-entries in a generated

904 matrix as the MCMC iterates as well as store the matrix with the highest amount
 905 of one-entries. As the process usually does not start with a random matrix, the
 906 user can decide to only compute the statistics after a certain number of iterations
 907 has been done and to only check a small portion of iterations, every 10th for
 908 instance, as a single iteration may not make any difference and counting the
 909 histogram takes time.

- 910 • 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
- 912 • 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
- 915 • 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
- 918 • 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"
- 921 • 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"

924 6.3 File input

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

929 6.3.1 Matrix file

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

- 931 • 2 natural numbers specifying the number of rows and columns in this order.
- 932 • a sequence of zeros and ones of length rows×columns specifying the matrix
- 933 from the top left corner one row after another.

934 **Example:** 2 3
 1 0 1
 1 1 0

935 6.3.2 Order file

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

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

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

946 2 1 0 3 4

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

949 6.4 File output

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

951 6.4.1 Matrix text file

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

- 953 • 2 natural numbers specifying the number of rows and columns in this order.
- 954 • a sequence of zeros and ones of length rows×columns specifying the matrix
- 955 from the top left corner one row after another.

956 The matrix is binary except for the one produced as a histogram, which can have
957 higher natural numbers and contains the number of samples as the last number.

958 If you then divide all the entries by the last number, you get a percentage of the
959 entry being a one-entry.

960 **6.4.2 Matrix bmp file**

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

965 Conclusion

966 Bibliography

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

969 Neal Madras and Hailong Liu. Random pattern-avoiding permutations. *Algo-*
970 *rithmic Probability and Combinatorics, AMS, Providence, RI*, pages 178–180,
971 2010.

972 List of Figures

973	1	Matrix M_1 contains the pattern P , because a mapping $\{(0, 0), (1, 2), (2, 3), (3, 4)\}$	
974		satisfies all the conditions. On the other hand, matrix M_2 avoids	
975		P as there is no such mapping.	4
976	2	Example of a generated matrix avoiding I_{10} (unit matrix). Black	
977		dots are one-entries and white are zero-entries. As you can see,	
978		matrices avoiding a pattern can have a nice structure.	5
979	2.1	An example showing unimportant line and equivalent mappings. .	9
980	3.1	An example of a walk W in matrix M and the order of entries in W .	10
981	3.2	Diagonals of an matrix and the order in which the algorithm for	
982		walking pattern iterates through them.	11
983	3.3	The algorithm testing avoidance for walking patterns cannot be	
984		easily generalized for all patterns.	14
985	4.1	TODO comment.	17
986	4.2	TODO comment.	18
987	4.3	TODO comment.	18
988	4.4	TODO comment.	19

989 List of Tables

990 List of Abbreviations

991 Attachments