FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 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

Institute: Computer Science Institute of Charles University

Supervisor: RNDr. Vít Jelínek, Ph.D., Computer Science Institute of Charles University

Abstract: Binary matrices not containing a smaller matrix as a submatrix have become 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 helping me to improve my program by suggesting different approaches and new functionality and for his patience and willingness to spend many hours correcting my thesis. I would also like to thank him for the knowledge I have not implemented a program no one would ever use.

Contents

Preface

Theses at the Faculty of Mathematics and Physics of Charles University in Prague usually fit into one of three categories:

1. Theoretical thesis
2. Experimental thesis
3. Implementation thesis

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

- Design of algorithms for generating a special binary matrix
- Making the algorithms run fast on inputs that are usual for researchers
- Implementation of the algorithms to provide a practical tool

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

Implemented program is available here: Kučera [2016].

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, where the first index is zero and we put rows before columns. Example can be found in Figure 2.

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 $f : L(P) \to L(M)$, 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)
- $\forall l, l' \in L(P)$: if lines l and l' intersect and there is a one-entry at the intersection, then there is a one-entry at the intersection of $f(l)$ and $f(l')$.

Otherwise, it *avoids* the pattern P.

$$P = \begin{array}{cc} & \begin{array}{cc} 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \end{array} \quad M_1 = \begin{array}{cc} & \begin{array}{ccc} 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \quad M_2 = \begin{array}{cc} & \begin{array}{ccc} 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

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 describe their properties. 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.

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 thesis is, for given $n \in \mathbb{N}$ and set of forbidden patterns $\mathcal{P}$, to generate a uniformly random $M \in \mathcal{M}_n(\mathcal{P})$.

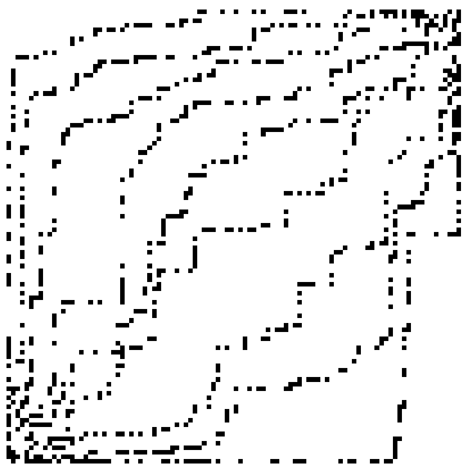

Figure 2: Example of a generated matrix avoiding I_{10} (unit matrix). Black dots are one-entries and white are zero-entries. As you can see, matrices avoiding a pattern can have a nice structure.

Generating random matrices

Notation 3. Let A be a set and $a \in A$ be its element. By $a \in_R A$ we denote that a is a uniformly random element of A.

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

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

Testing avoidance

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

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

Some technical details are explained in Chapter 5 to make reading the code (Kučera [2016]) easier for the reader. The last chapter (Chapter 6) contains the 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] or Karlin and Taylor [1975].

1.1 Markov chains

Definition 4. Let $\mathcal{S}$ be a finite set of states and for every $i, j \in \mathcal{S}$, let $p_{i,j}$ be a 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 $X_0, X_1, \ldots$, where $X_i \in \mathcal{S}$ for every i a *Markov chain* if

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

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. Let $p_{i,j}^{(k)} = Pr(X_{t+k} = j | X_t = i)$ denote the k-step transition probabilities for $k = 0, 1, \cdots$ and $i, j \in \mathcal{S}$. An irreducible Markov chain has *period* D if D is the greatest common divisor of $\{k \geq 1 | p_{i,i}^{(k)} > 0\}$ for some $i \in \mathcal{S}$ (equivalently, for all $i \in S$). A chain is called *aperiodic* if its period is 1. In particular, if an irreducible chain has $p_{i,i}^{(1)} > 0$ for some i, then it is aperiodic.

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 $i, j \in S$, the limit $\lim_{k\to\infty} p_{i,j}^{(k)}$ exists and is independent of i; call it* equilibrium distribution *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 aperiodic 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:

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

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

Symmetry

If i and j differ in more that one bit, we will never get one from the other by flipping one bit; therefore, both $p_{i,j} = p_{j,i} = 0$. If i and j differ in exactly one bit, we have $p_{i,j} = p_{j,i} = \frac{1}{n^2}$, because we need to change exactly that one bit to succeed. Otherwise, $i = j$ and we have $p_{i,i} = p_{i,i}$.

Irreducibility

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

Aperiodicity

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

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

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

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

2.1 Sketch of a brute force algorithm

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

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

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

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

2.2 Equivalent mappings

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

Definition 8. We call a line l of a pattern P *important at level* k, with respect to a given permutation $(l_1, \ldots, l_{w+h})$, if one of the conditions is met:

- The line l is adjacent to a line not in $(l_1, \ldots, l_k)$.
- There is a one-entry on the line l at the intersection with line l' that is not in $(l_1, \ldots, l_k)$.

Otherwise the line is *unimportant at level k* for the permutation.

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

At the beginning, all lines are important. After k lines get mapped, a line can become unimportant at level k as all lines that bound it are in $(l_1, \ldots, l_k)$. If a line is unimportant at some level, it stays unimportant at any higher level.

Definition 9. We say two partial mappings of the same level l are *equivalent* if all important lines at level l are mapped to the same lines of the big matrix in both mappings.

$$P = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \qquad M = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

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

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

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

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

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

3.1 Walking pattern

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

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

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

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

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

It can be shown a walking pattern is exactly a matrix avoiding a forbidden pattern $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$.

3.2 Dynamic program

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

The pattern P is a walking pattern, so there is a walk containing all the one-entries of P. We choose one such walk arbitrarily. For each entry of the walk we remember whether its value in P is one or zero and whether the walk continues from the entry vertically, in which case we call it a *vertical entry* or horizontally, calling it a *horizontal entry*. The last entry is an exception and it is neither horizontal nor vertical.

Definition 12. For an element e of M at the position $[i, j]$ ($[0, 0]$ is the first element), 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.

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.

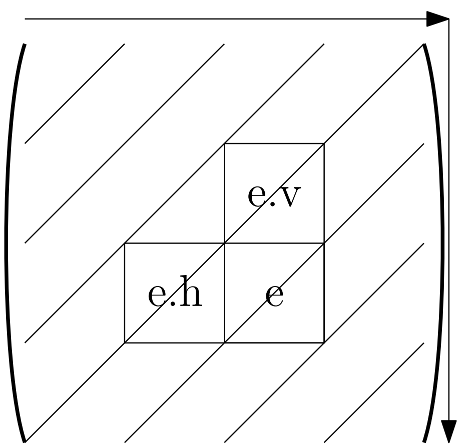

Figure 3.2: Diagonals of an matrix and the order in which the algorithm for walking pattern iterates through them.

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 $c_v = c_h = 0$. When we ask whether w_k can be mapped to e, where e is an element of M, we check whether w_k stands for a one-entry of P and if it does, we require e to be a one-entry too.

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

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

3.2.3 Correctness

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

The function can terminate before recomputing all elements and we have no guarantee about the state of elements that have not been recomputed. If the function finds the pattern ending in entry e, it stops computing at that point, but to prove correctness it is enough to prove the values are correct in $M_{\leq e}$, which has been fully recomputed. If, on the other hand, the function does not find the pattern, it recomputes the whole structure.

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

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

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

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

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

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

3.2.4 Generalization

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

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

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

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

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

4. Improvements to basic algorithms

In this chapter we improve algorithms presented in previous chapters and introduce a parallel method of testing pattern avoidance.

4.1 General pattern

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

4.1.1 Improving memory consumption

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

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

4.1.2 Not mapping empty lines

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

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

4.1.3 Using the last changed position

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

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

- The first row of P is mapped under the r-th row. This prevents any other row to be mapped to the 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 the 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 the r-th one.

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 at 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,

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

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 are 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

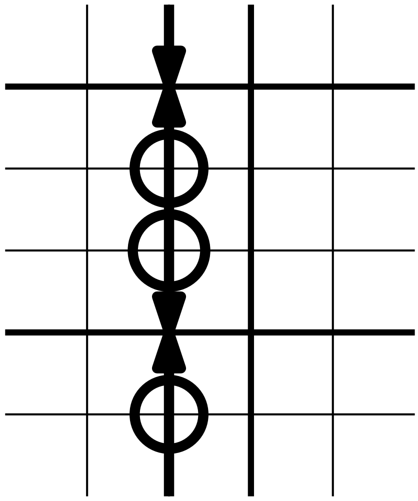

Figure 4.2: Checking whether there is enough one-entries. Bold lines in the picture are mapped and in circles are the positions where we look for one-entries.

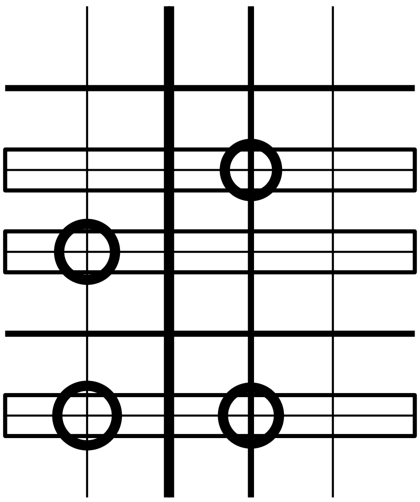

Figure 4.3: Checking whether crossed non-mapped lines can be mapped anywhere. Bold lines are mapped or being mapped, in rectangles are the lines we check and in circles are the positions where we look for one-entries.

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

Orthogonal bounds

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

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.

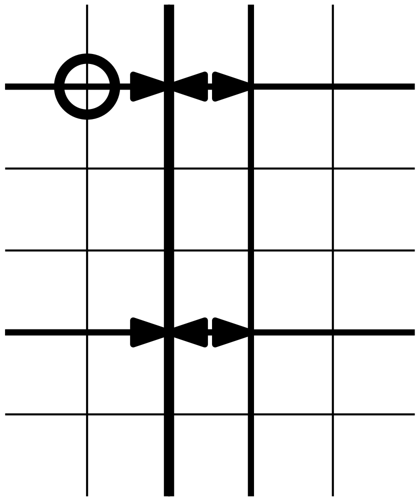

Figure 4.4: Checking whether there is enough one-entries on the orthogonal lines. Bold lines are mapped or being mapped and in circles are the positions where we look for one-entries.

Pattern	n	#iterations	one	rec	orth	time (sec)	memory
1 0 0	100	100,000	yes	yes	yes	122.24	3,004 B
1 1 1	100	100,000	no	no	no	248.65	3,268 B
0 0 1	500	10,000	yes	yes	yes	1,072.53	5,380 B
	500	10,000	no	no	no	2,631.73	15,976 B
1 1 1	100	100,000	yes	yes	yes	1,512.56	3,268 B
1 1 1	100	100,000	no	no	no	672.63	3,268 B
1 0 0	500	10,000	yes	yes	yes	10,276.60	9,128 B
	500	10,000	no	no	no	6,756.96	15,972 B

Table 4.1: Testing additional restrictions of the generated matrix is useful in some cases but it comes with a performance drop in different cases.

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 avoiding some patterns while it was much better not to use any of those restrictions for different pattern, which you can see in Table 4.1. For some patterns it also happened that using additional restrictions was useful for matrix of size 100×100 and for the same patterns it was better not to use them for generating a 500×500 matrix.

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 of MCMC, instead of finding all the partial mappings 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 $n-1$ private copies of patterns class. For $i = 1, \dots, n-1$ it assigns i-th thread, called worker, to a pair of i-th copy of the matrix and i-th copy of patterns. 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 $c < d$ computed the same iteration, it must have been the case an earlier iteration succeeded when task with ID c was computed and after it got removed, task with ID d 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 other tasks get removed as they all had a higher ID – they 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 then 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 is gets recomputed. This is what happens till the end.

4.2.2 Speculative computing

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

The way we deal with these inconveniences is described in Chapter 5 and should be clear from the code (Kučera [2016]) itself.

4.2.3 Reverting and synchronizing in the main thread

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

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

4.3 Walking pattern

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

4.3.1 Using the last changed position

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

Not only that. We also know, using the fact the structure was completely correct before the change, that if the values of both c_v and c_h of an element did not change, the element will not cause the element underneath it to change and we no longer have to recompute 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 recomputing. The function ends either when the pattern is discovered or when the queue becomes empty.

4.3.2 Lazy implementation

Lazy implementation of avoid and revert functions (see Section 5.1.2) is used when the MCMC parallelism (more in Section 4.2) 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. A different, but not less significant reason is that because of concurrency, usually there are several revert calls after one avoid call; therefore, it is better to improve revert calls even with a downside of making avoid calls slightly slower.

The avoid function expects the inner structure of the walking pattern to be in a valid state and that requires some effort. To make lazy revert as fast as 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 is necessary to also compute changes in the positions changed in those lazy revert calls that are postponed.

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

Recompute the whole structure every time

The easiest way to implement lazy avoid is 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.

The weakness 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 is 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 changed.

This gets more complicated when an avoid call in the lazy implementation discovers the pattern in $M_{\leq e}$, because the avoid call returns as soon as the pattern is discovered, without recomputing the whole inner structure. It is still possible to remember some horizontal, vertical and diagonal bounds and use them to restrict the recomputed part of the matrix. However, the improvement is not that significant and we can do better.

Queue of positions to recompute

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

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

Two leveled queue of positions to recompute

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

Pattern	n	#iterations	type	threads	time (sec)	memory
1 1 1	100	100,000	general	1	1,486.61	< 100 KB
1 1 1	100	100,000	general	9	381.56	602 MB
1 0 0	100	100,000	general	17	332.46	1,173 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	569.23	24.3 MB
0 0 1 0 0	100	100,000	general	17	336.78	39.9 MB
0 1 0 0 0	100	100,000	walking	1	5.31	2.9 MB
1 0 0 0 0	100	100,000	walking	9	2.06	4.2 MB
	100	100,000	walking	17	1.91	5.1 MB

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

4.4 Comparison of all methods

We have improved all algorithms and added a parallel version of the MCMC process. The question is, whether our improvements were useful in terms of performance and memory consuming. I have done a few tests and in Table 4.2, Table 4.3 and Table 4.4, you can see that at least in some cases walking pattern does much better than general pattern and that parallel computation was not added without a good reason. We always generate a matrix of size $n \times n$ and if there is one thread, we use the serial variant of MCMC process; otherwise, we use the parallel one.

Pattern	n	#iterations	type	#th	time (sec)	memory
	100	100,000	general	1	1,486.61	< 100 KB
	100	100,000	general	9	381.56	602 MB
	100	100,000	general	17	332.46	1,173 MB
	500	100,000	walking	1	137.74	19 KB
I_{10}	500	100,000	walking	9	38.85	614 MB
	500	100,000	walking	17	31.41	1,210 MB
	5,000	10,000	walking	1	3,588.79	467 MB
	5,000	10,000	walking	9	1,105.49	5,025 MB
	5,000	10,000	walking	17	685.04	9,174 MB

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

5. Technical documentation

In this chapter, we cover those parts of the algorithm that may be hard to understand just from the code (Kučera [2016]). This only means functions that are technically hard, for example functions with unexpected dependencies, side effects and so on. Algorithmic difficult tasks are explained in Chapter 4.

5.1 Classes and API

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

5.1.1 Matrix

A minimalistic template container for storing and accessing matrices.

5.1.2 Pattern

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

- *General_pattern* – more in Chapter 2
- *Walking_pattern* – more in Chapter 3
- *Slow_pattern* – a class using a brute force algorithm to test pattern avoidance.

Besides others, the class declares two important functions:

- *Avoid* – tests whether a given pattern is avoided by a given matrix.
- *Revert* – it is used to revert changes of pattern's inner structures after a call of Avoid function which is unsuccessful.

5.1.3 Patterns

To create an interface between Pattern and MCMCgenerator we use a class called Patterns. It is a container of Pattern instances and allows the user to generate matrices avoiding more than just one forbidden pattern.

It follows the API of Pattern and a call of a function on the class can be imagined as a call of the same function on each element of the container. This is not necessarily what happens, because for instance when we test avoidance of all patterns in Patterns, as soon as we find a pattern which is contained in a given matrix we can stop testing and return false, instead of testing the rest of patterns and still returning false.

5.1.4 Statistics

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

- *Matrix statistics* – these statistics store information about the structure of all matrices that have been generated throughout the MCMC process. An example of that is a histogram of occurrences of one-entries at all positions of the matrix.

- *Performance statistics* – this is what we use to count how many changes were successful and how long did it take to test a change.

To get data from the MCMC process, a method called “add_data” is used. If the user wants to define their own statistics, they can add variables into the class, alter “add_data” function to allow storing the desired data and change functions “print_· · ·” to redefine the output.

5.1.5 MCMCgenerator

MCMCgenerator is a method that for a given number of iterations and a given instance of Patterns approximates a uniformly random matrix avoiding that pattern, following MCMC algorithm shown in Section 1.2. It also takes instances of Statistics classes as arguments to produce more data if requested by the user.

5.2 General_pattern

The general pattern class contains a lot of methods. Most of them are easy to follow and they all should be commented enough in the code (Kučera [2016]). The only part which deserves more attention is the constructor.

5.2.1 Construction

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

Storing the pattern

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

Choosing the line order

After that, we need to choose the right line order (again more in Chapter 4). Exactly one algorithm to determine the order is chosen by the user.

To compute MAX, SUM or TWO order we use a brute force algorithm that checks sequences of line adding and for each it computes how many lines are unimportant. Then it just chooses the order which is the best in chosen metric. To compute DESC order, we sort the lines according to the number of one-entries.

Identifying important lines

In the next step, we find which lines are important at each level with respect to chosen order. What to remember is based on the equivalence introduced in Chapter 2 and the decision not to remember unimportant lines, which we explained in Chapter 4.

Parallel bound indices

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

First, we add lines in a chosen order and second, we do not remember all lines, as some are unimportant. What we want is to have an instant access to the indices of the lines that bound the currently added line in the partial mapping, so we do not need to compute the index over and over again. That is exactly what gets computed when the function "find_parralel_bound_indices" is called. The series of other function calls follows just because we compute the indices for all added lines in the order in which they are going to be added.

Extending order

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

5.3 MCMC parallelism

While the idea behind MCMC parallelism is described in Section 4.2 and the code (Kučera [2016]) is heavily commented, the work done by the main thread may still be hard to understand.

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

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

- pop_front: The main thread deletes the first task (the one with the lowest ID) if one of two things happen:
  - The ID of the task being deleted is equal to I. That means the change computed by the task is being propagated to the generated matrix and there is no need to remember the task anymore. This also increases I, not necessarily by one.
  - The ID of the task being deleted is less than I. This situation happens due to synchronization. The worker was supposed to synchronize a task computed by a different worker that did not have the lowest ID at the time. Therefore, the task needs to be in the list of tasks so we 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 with ID J will not propagate to the generated matrix and there in 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.

5.4 BMP generating

To generate output in a BMP format, I use an open source C++ library called EasyBMP. More information about the library and the author can be found in Macklin [2005].

6. User documentation

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

6.1 Installation

The program is written in C++, using the C++11 standard and is independent of any platform or compiler extensions. To use it, you either just use an executable file (Windows) or build the program using a C++ compiler.

6.1.1 Windows

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

For those who want to alter the code (Kučera [2016]) or recompile the program in Visual Studio, Project File is also added.

6.1.2 Unix, Linux, MacOS

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

6.2 Configuration file

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

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

- input
- pattern
- output
- statistics

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

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

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

6.2.1 Input

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

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

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

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

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

- random_seed: The random seed for the MCMC process.

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

- init_matrix: A *size* by *size* matrix the MCMC process starts with.

Possible value:	*matrix text file path* (6.3.1)
	“zero” - a matrix containing no one-entries
Default value:	“zero”

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

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

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

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

6.2.2 Pattern

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

- pattern_file: A relative path to an input matrix file - the pattern.

Possible value:	*matrix text file path* (6.3.1)
Default value:	"pattern/input.txt"

- pattern_type: The type of the pattern. Determines the method used for testing avoidance.

Possible value:	"general"
	"walking" - see Chapter 3
	"slow" - brute force algorithm for a general pattern
Default value:	"general"

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

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

- map_one_entries: If set to "yes", the map function tests whether there are enough one-entries in between already mapped lines.

Possible value:	"yes"
	"no"
Default value:	"yes"

- map_recursion: If set to "yes" and the map_one_entries is also set to "yes", the map function tests mapping recursively.

Possible value:	"yes"
	"no"
Default value:	"yes"

- map_orthogonal_bounds: If set to "yes", the map function also tests the orthogonal bounds of added line.

Possible value:	"yes"
	"no"
Default value:	"no"

- 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)
	"vector" - std::vector (dynamic array)
Default value:	"hash"

- line_order: Chooses the order in which the lines are being added to the partial mapping. See Section 4.1.4.

Possible value:	"max"
	"two"
	"sum"
	"desc"
	"auto"
	order file path (6.3.3)
Default value:	"max"

6.2.3 Output

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

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

Possible value:	"console"
	matrix bmp file path (6.3.2)
	matrix text file path (6.3.1)
	"no"
Default value:	"no"

- performance_stats: If the serial computation is chosen, the program can output statistics like the percentage of avoid call success, an average duration of one call and an average size of structures. If more patterns are chosen at the same time, the statistics may get misleading as they also count the cases when the first pattern is contained in the matrix and the other patterns are not tested at all.

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

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

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

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

Possible value:	"yes"
	"no"
Default value:	"no"

- patterns_to_console: Prints all the used patterns into the console.

Possible value:	"yes"
	"no"
Default value:	"no"

6.2.4 Statistics

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

- 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

- 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

- 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

- histogram_file: Sets where to output the histogram computed during the MCMC process.

Possible value:	*matrix bmp file path* (6.3.2)
	matrix text file path (6.3.1)
	“console”
	“no”
Default value:	“no”

- 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* (6.3.2)
	matrix text file path (6.3.1)
	“console”
	“no”
Default value:	“no”

6.3 File input and output

In this section, we describe the format of both input and output files.

6.3.1 Matrix text file

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

- two natural numbers specifying the number of rows and columns in this order.

- a sequence of zeros and ones of length rows times columns specifying the matrix from the top left corner one row after another.

Example:

2 3
1 0 1
1 1 0

6.3.2 Matrix bmp file

For an $n \times n$ matrix, the standard bmp file contains $n \times n$ pixels. Black colored pixel stand for a one-entry and white colored pixels for a zero-entry. If the histogram is output as a bmp file, the pixels are grayscaled and the darker a pixel is the more often the entry was a one-entry during the MCMC process.

6.3.3 Order file

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

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

One possible order for the matrix given as an example in Subsection 6.3.1 is this file:

2 1 0 3 4

It first maps the left column, the second and first row after that and finishes the mapping with the middle column and the right one.

6.4 Examples of output

In this section, we show a few histograms that the program generated. It is here to show that random matrices avoiding forbidden patterns may have a nice structure as well as to finish my thesis with beautiful pictures.

We use two walking patterns:

$$P_1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \qquad P_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

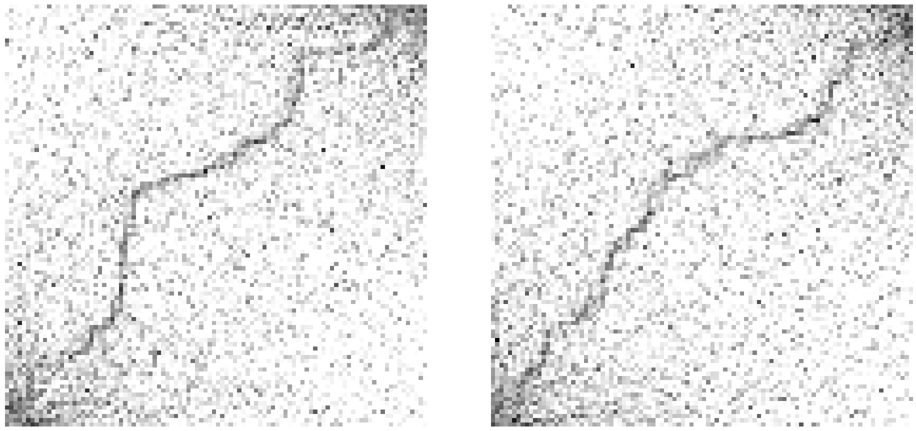

Figure 6.1: A histogram of a generated matrix avoiding pattern P_1.

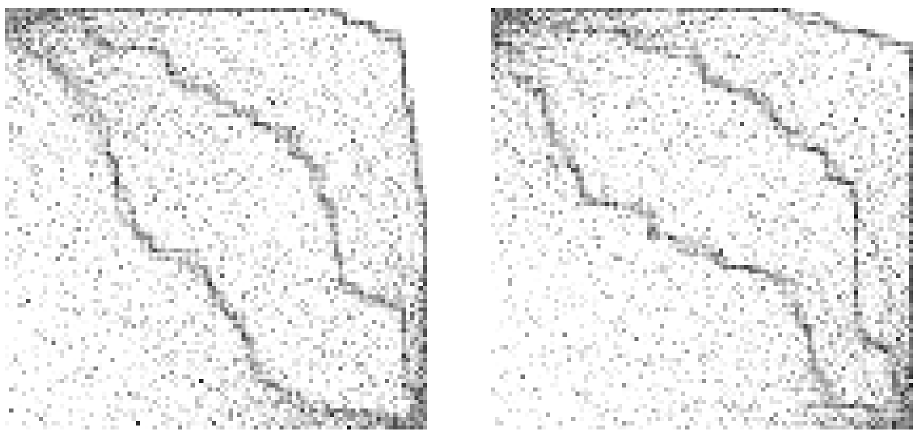

Figure 6.2: A histogram of a generated matrix avoiding pattern P_2 using a different random seed than the previous picture.

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.

Bibliography

Samuel Karlin and Howard M Taylor. *A First Course in Stochastic Processes*, volume 1. Gulf Professional Publishing, 1975.

Stanislav Kučera. The program generating random binary matrices without forbidden patterns, 2016. URL `http://github.com/`.

Paul Macklin. An open source (revised BSD) C++ library, 2005. URL `http://easybmp.sourceforge.net/`.

Neal Madras. *Lectures on Monte Carlo methods*, volume 16. Springer Science & Business, 2002.

Neal Madras and Hailong Liu. Random pattern-avoiding permutations. *Algorithmic Probability and Combinatorics, AMS, Providence, RI*, pages 178–180, 2010.

List of Figures

List of Tables