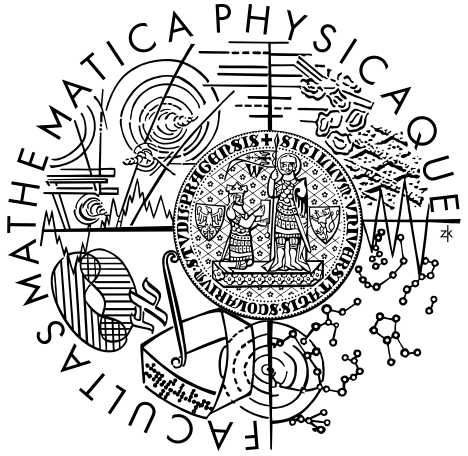
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Stanislav Kučera

Generating random pattern-avoiding matrices

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: RNDr. Vít Jelínek, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2016

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

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

In ..... date ..... signature of the author

Title: Generating random pattern-avoiding matrices

Author: Stanislav Kučera

Department: Computer Science Institute of Charles University

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

Abstract: Abstract.

Keywords: binary matrix pattern-avoiding Markov chain Monte Carlo

Dedication.

Contents

Preface

Theses at the faculty of mathematics and physics 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 the algorithm for generating a special binary matrix
- Making it run fast on inputs which are usual for researchers
- Implementing the algorithm to provide practical tool

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

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 $f : L(P) \to L(M)$, 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)*
- *$\forall l, l' \in L(P)$: 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.*

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

Figure 1: Matrix M_1 contains the pattern P, because 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. We always call M the square binary matrix for which we test the containing and P the pattern (if there is one) that is being tested. Moreover, we denote by h the height (the number of rows) of P and 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 $M \in_R \mathcal{M}_n(\mathcal{P})$.

Generating random matrix

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 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 and despite the fact it is unknown what is the mixing time (the number of iterations required) of a MCMC process, 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 performing 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.

In chapter 5 some technical details are explained to make reading the code easier for reader and to describe user interface. The last chapter (chapter 6) contains user documentation.

1. Markov chain Monte Carlo

Our goal to generate $M \in_R \mathcal{M}(\mathcal{P})$ heavily depends on the theory of Markov chains. 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 3. *We shall consider discrete-time* Markov chain $X_0, X_l, \ldots$, *where* $X_i \in \mathcal{S}$, *for a finite state space* $\mathcal{S}$ *and every* i *(number of steps). The* k*-step transition probabilities are:*

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

Definition 4. *A Markov chain is said to be* symmetric *if* $p_{i,j}^{(1)} = p_{j,i}^{(1)}$ *for every pair of states* i *and* j.

Definition 5. *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* $p_{i,j}^{(k)} > 0$.

Definition 6. *An irreducible chain has* period D *if* D *is the greatest common divisor of* $\{k \geq 1 | p_{i,i}^{(k)} > 0\}$ *for some* $i \in \mathcal{S}$ *(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.*

Theorem 1. *Consider an aperiodic irreducible Markov chain with 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* 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* $\mathcal{S}$.

In other words, the theorems together give us a guarantee that if we choose an irreducible, symmetric and aperiodic Markov chain with state space $\mathcal{S}$ then the probability distribution of X_i converges to uniform distribution on $\mathcal{S}$ independently of the initial state.

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

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

Irreducibility

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

Aperiodicity

The Markov chain is irreducible so it suffices to show that there is an i for which $p_{i,i}^{(1)} > 0$. Clearly there is a matrix for which there is at least one bit that cannot be flipped without creating a pattern and this forces $p_{i,i}^{(1)} > 0$.

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 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 two conditions:

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

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

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 subsection 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 mappings if they can be both extended exactly the same way as our function is only supposed to check whether a pattern can be mapped to a big matrix not to find all such mappings.

Definition 7. *We call a line l of a pattern* important *in a partial mapping if one of the conditions is met:*

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

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

Whether a line is important or not only depends on the permutation, we have chosen for partial functions, because that is what forces the order in which we map lines.

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

Definition 8. *We say two partial mappings of the same level are* equivalent *if all important lines in the mapping of that level 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, matrices in Figure 2.1, in partial mapping $f = \{(1,1),(2,2),(3,4),(5,6)\}$ 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 which, using some heuristics, runs pretty fast. In this chapter, we introduce a special kind of a pattern, satisfying additional conditions, for which we can produce much faster algorithm.

3.1 Walking pattern

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

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

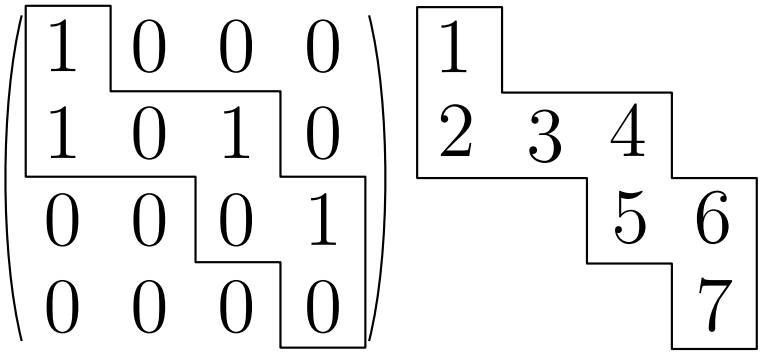


Figure 3.1: An example of a walk and the order of its entries.

In Figure 3.1 you can see a matrix that is a walking pattern as all the one-entries are included in a walk. Not all entries of a walk need to be one-entries though.

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

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

3.2 Dynamic program

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

The pattern P is a walking pattern so there is a walk containing all the one-entries of the pattern. We choose one such walk arbitrarily and index its entries $e_1, e_2, \ldots, e_{h+w-1}$ starting from the beginning of the walk. For each entry of the walk we remember whether its value is one or zero and whether the walk continues from the entry vertically, in which case we call it a **vertical entry** or horizontally, calling it a **horizontal entry**.

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

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

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

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

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

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

3.2.1 The algorithm

1. For $d = 0, \cdots, w + h - 1$
2. For e element of d-th diagonal at the position $[i, j]$
3. $e_v := [i-1, j]$
4. $e_h := [i, j-1]$
5. $c_v(e) := c_v(e_v)$
6. $c_h(e) := c_h(e_h)$

7. If $w_{c_v(e)+1}$ can be mapped to e

8. If $c_v(e) + 1 = w + h + 1$

9. Terminate - M contains P as a submatrix

10. If $w_{c_v(e)+1}$ is a vertical entry

11. $c_v(e) := c_v(e) + 1$

12. Else

13. $c_h(e) := max\{c_h(e), c_v(e) + 1\}$

14. If $w_{c_h(e)+1}$ can be mapped to e

15. If $c_h(e) + 1 = w + h + 1$

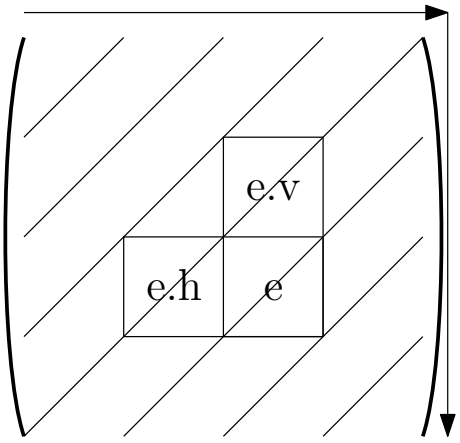
16. Terminate - M contains P as a submatrix

17. If $w_{c_h(e)+1}$ is a vertical entry

18. $c_v(e) := max\{c_v(e), c_h(e) + 1\}$

19. Else

20. $c_h(e) := max\{c_h(e), c_h(e) + 1\}$



3.2.2 Inner structure

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

3.2.3 Correctness

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

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

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

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

3.2.4 Generalization

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

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 it is not sufficient to just check whether w_l is above and w'_l to the left from the element.

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

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

4. Improvements to basic algorithms

4.1 General pattern

4.1.1 Improving memory consumption

As the algorithm for testing avoidance of a general pattern was described in [chapter2], it creates all possible partial mappings and checks whether at least one can be extended to a full mapping. Note that 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.

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

4.1.2 Not mapping empty lines

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

4.1.3 Using the last changed position

As the MCMC process works, it 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 c-th one get skipped. We will only look at the restriction for rows. 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 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.

- 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 [chapter2] 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.

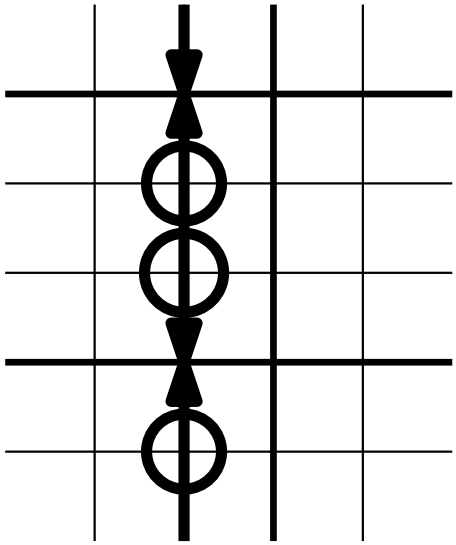
Imaging this as the pattern, where $0, \cdots, 8$ are indices of the lines:

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

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

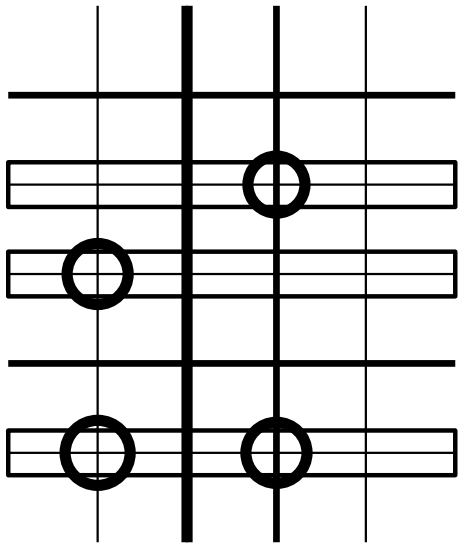
Enough one-entries

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



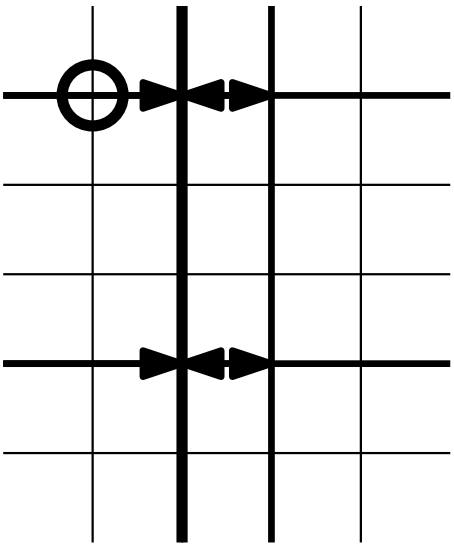
Recursive mapping

While we were only testing whether there are enough one-entries in between already mapped lines in the previous approach, 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 that row, 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

When adding line 6, we check whether there are enough one-entries on the already mapped lines orthogonal to line 6, in between line 6 and the closest mapped lines next to line 6. The same idea as in "Enough one-entries", but checking 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. 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 need to remember them all. Of course we can still use the equivalence but we need to remember mappings of all levels.

- 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 and we go through all partial mappings and remove those new ones. This again means to iterate through the big structure one more time.

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

4.2 MCMC parallelism

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

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

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

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

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

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

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

with the lowest ID) we do:

- if it fails:
  - Do nothing - there is no change to propagate to the master copy of the generated matrix and all the tasks with higher ID expected this task to fail, which it did.
  - This increases the lowest ID by exactly one, as the task we speak of got processed.
- if it succeeds:
  - The main thread propagates the change tested by the task to the master copy of the generated matrix.
  - 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 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 don't 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 a task with lower ID. If the 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 is not that hard to deal with it, even without copying the possibly huge generated matrix all over the place.

The way we deal with these inconveniences is described in [chapter5] and should be clear from the code 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 so it computes a trivial function, sets a few atomic variables and falls 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 workaround this problem we make a theoretically bad decision which has 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 is very fast in its nature and cannot be much better. Or can it be?

4.3.1 Using the last changed position

As the MCMC process works it 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 is a part of the pattern. Knowing that and using the same inductive proof as we did in the proof of correctness of the avoid algorithm (see [chapter2]) it is sufficient to only recompute the part of the inner structure under the changed element 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 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 (more in [chapter4]) 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 [chapter3]) 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 them to restrict the recomputed part of the matrix. The improvement is not that significant though 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 diagonal 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 higher 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 had no more problems with recomputing the entries that were not influenced by the changes and used all the benefits mentioned in the previous section. But the container does not come for free and in the end I found out the price I payed for the operations on the priority queue made the whole implementation comparably slow as in the previous attempt.

Two leveled queue of positions to recompute

The final solution comes with the same idea, but a different storage type. 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. So finally it is the case that the storage used is not only good theoretically but as the numbers say, also practically. [reference to a table of measurements or something]

5. Technical documentation

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

5.1 General pattern

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

5.1.1 Construction

In the constructor of a general pattern, there are a few function 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 a very important part indeed, 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 variable, is storing the pattern. Instead of storing the pattern in a Matrix<bool>, I decided to use 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 lines orders. At the same time we also find those lines that are empty (more in [chapter4]) 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 [chapter4]). To compute MAX or SUM order we just 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.

What to remember

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

Parallel bound indices

Now comes the hardest to follow part - precomputing the indices for searching for parallel bounds. The idea behind 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, the second one need to be mapped in between those two. The question is, where are those two lines mapped to? First we add in a chosen order and second we do not remember all lines, as some are unimportant. What do we want is to have a instant access to the index of the line, which bounds 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" just specifies how the next partial mapping will look like a from where from the previous mapping the values will be copied. Again, unimportant lines play their role here and it may easily be the case 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.

5.2 Parallel computing

5.2.1 MCMC parallelism

While the idea behind MCMC parallelism is described in [chapter4.3] and the code 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 related 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's see how the main thread works with that and what are the situations.

- pop_front: The main thread deletes the first tasks (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 is 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 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". Now a different worker computes a task with lower ID and finds out the change is successful. This means the task J won't 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 just 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 list.

  - The second reason to insert into "queue" are, again, 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.

5.3 Library interface

6. User documentation

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

6.1 Configuration file

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

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

- input
- pattern
- output
- statistics

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

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

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

6.1.1 Input

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

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

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

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

  Possible value: $i \in \mathbb{N}$
  Default value: 10,000

- 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 $size \times size$ matrix the MCMC process starts with.

  Possible value: *input file path*
  "zero" - a matrix containing no one-entries
  Default value: "zero"

- parallel_mode: choose whether to compute in parallel or serial.

  Possible value: "serial"
  "mcmc" - more iterations of the MCMC generator are tested in parall
  "map" - more partial mappings are being extended in parallel
  Default value: "serial"

- threads_count: The number of threads that are going to be used if a parallel mode is chosen.

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

6.1.2 Pattern

It is time to set the options that matter the most - matrix patterns. As you 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 path to a input matrix file - the pattern.

  Possible value: matrix file path
  Default value: "input.txt"

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

  Possible value: "general"
  "walking" - see [chapter3]
  "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.

- map_one_entries: If set to "yes", the map function not only checks that already mapped lines do cross the currently added one in a one-entry if needed but also tests whether there is enough one-entries in between those already mapped lines. The result is that the map function takes a little longer but it recognizes hopeless mappings earlier.

  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 not only tests there are enough one-entries in between already mapped lines on the currently added one, but for those non-mapped lines crossing the current one in a one-entry, it also tests if the partial mapping has one-entries everywhere they are supposed to be. Although this comes with a big time consumption it can filter a big portion of partial mapping that would later become unfeasible.

  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 - see [chapter2].

  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: Choose the order in which the lines are being added to the partial mapping. See [chapter2]

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

6.1.3 Output

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

- 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 “path.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
  matrix text file path
  “no”
  Default value: “no”

- performance_stats: If the serial computation or the map parallelism is chosen, the program can output a statistics like the percentage of avoid call success, how long did one call take on average and what was the average size of structures. Note that this is not very useful as a matrix statistics, see the next section for that. If more patterns are set, the statistics may get misleading as it also counts 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"

6.1.4 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
  matrix text file path
  "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
  matrix text file path
  "console"
  "no"
  Default value: "no"

6.2 File input

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

6.2.1 Matrix file

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

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

Example:

```
2 3
1 0 1
1 1 0
```

6.2.2 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 for example 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 [6.2.1] is this file:

2 1 0 3 4

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

6.3 File output

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

6.3.1 Matrix text file

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

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

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

6.3.2 Matrix bmp file

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

Conclusion

Bibliography

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 173–194, 2010.

List of Figures

List of Tables

List of Abbreviations

Attachments