Assignment 3: Sudoku

Riko Jacob and Matti Karppa

Hand-in date: 2022-11-08 11:59

The assignment is to be solved in groups of 1–3 people.

1 Introduction

Your task is to solve the Sudoku problem in three different ways: by a backtracking algorithm that directly addresses the problem, by reduction to the Exact Cover problem, and by reducing the problem to SAT. You will then need to design and conduct a horse race experiment to compare the scalability of the three different implementations against one another.

You have considerable liberty in desigining the file formats and data structures for your code. You are also expected to use an external SAT solver for comparison.

2 Preliminaries

This section sets up the notation we will use in further parts of the instructions. For a positive integer n, we write $[n] = \{1, 2, \ldots, n\}$. For a set S, we denote by $\mathcal{P}(S) = \{T \mid T \subseteq S\}$ the *power set* of S, that is, the set of all subsets of S. We denote the set of all unordered pairs of integers from 1 to n by $\binom{[n]}{2}$. For example,

$$\binom{[3]}{2} = \{\{1,2\},\{1,3\},\{2,3\}\}\,.$$

We use the following notation for propositional logic: Let p, q, r, and s be variables, and let ϕ and ψ be Boolean expressions. We write $\neg p$ for the negation of p. We say that variables and their negations, that is, p and $\neg p$, are *literals.* We denote the conjunction (logical AND) and disjunction (logical OR) of Boolean expressions by $\phi \wedge \psi$ and $\phi \vee \psi$, respectively. If a Boolean expression consists of a disjunction of literals, we say that it is a *clause.* We say that a Boolean expression is in *Conjunctive Normal Form* (CNF) if it consists of a conjunction of clauses.

For example, $p \vee \neg q$ is a clause, and

$$\psi = (p \vee \neg q) \wedge (r \vee s \vee p) \qquad (1)$$

is in CNF. Furthermore, we write $\phi \implies \psi$ when ϕ *implies* ψ. We remark the following rule regarding implications which is useful when converting formulae to CNF:

$$\phi \implies \psi \iff \neg\phi \vee \psi\,.$$

Here we make use of the logical equivalence symbol $\iff$ or the *if and only if* symbol to denote that the two expressions on the left and right hand side are *equivalent*.

We say that a function assigning true or false values to variables is a *truth assignment*. If there exists a truth assignment that makes an expression evaluate true, we say that the expression is *satisfiable* and that the truth assignment *satisfies* the expression. We will denote the true value by 1 and the false value by 0. For example, the truth assignment $p \mapsto 1$, $q \mapsto 1$, $r \mapsto 0$, $s \mapsto 0$ satisfies the CNF expression ψ in Equation (1), but it does not satisfy the expression

$$\begin{aligned}\phi =&(p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge \\ &(\neg p \vee q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)\,,\end{aligned}$$

which is in fact *unsatisfiable*.

3 Sudoku

A *Sudoku* is a combinatorial puzzle. Let n be a parameter controlling the size of the problem. A Sudoku problem asks whether the *cells* of a partially filled $n^2 \times n^2$ *grid* can be filled to satisfy certain constraints using numbers from 1 to n^2. The cells that have initially been filled with a number are called *clues*. If some of the cells are empty, we say that the grid forms a *partial solution*. We divide the grid into n^2 subgrids of size $n \times n$. Note that the initial puzzle with only clues is also a partial solution. The most common problem size encountered in newspapers and practised by enthusiasts is with $n = 3$ that forms a 9×9 grid.

Following constraints must be followed when filling the grid:

(i) Each number from 1 to n^2 must occur exactly once on each row,

(ii) Each number from 1 to n^2 must occur exactly once on each column,

(iii) Each number from 1 to n^2 must occur exactly once in each $n \times n$ *subgrid*.

The general SUDOKU problem asks the following:

Problem 1 (SUDOKU)**.** Given a partially filled $n^2 \times n^2$ Sudoku grid, can each cell of the grid be filled to satisfy the constraints (i)–(iii)?

Note that we only require that there is *a* solution, we do not necessarily require the solution to be unique. Formulated like this, SUDOKU is known to be NP-complete [YS03].

4 Auxiliary functions and data formats

Before we proceed to actually solving SUDOKU, we will need to construct a few auxiliary functions and specify how we pass and store data.

Task 1. Design a file format for storing SUDOKU problems. Your file format will be used to pass the data into the solvers you will implement. The file format should be sufficiently flexible to allow you to pass partial solutions (so some of the cells are marked empty), and it should support different values for the size parameter n, that is, it should be possible to pass 4×4 sudokus in your format as well as 16×16, or larger. Implement the necessary functions for reading and writing your file format. In practice, it suffices to do this from stdin / to stdout.

Task 2. Design a data structure to store SUDOKU problems in memory. The same design considerations hold as for the file format: Your data structure should be able to also hold empty cells, and be adaptible for any value of n. There are multiple possible solutions, but it is probably a good idea to use a very simple data structure.

Task 3. Implement a function ISSOLVED that determines if a given SUDOKU grid is solved, that is, that it neither contains any empty cells, nor are any of the constraints violated.

Task 4. Implement a function CANPLACE that determines if the number k can be placed in the cell (i, j) in the (fully or partially solved) SUDOKU grid. That is, the function should check that the cell is unoccupied and that placing k would not immediately violate any of the constraints.

5 Simple backtracking algorithm

This is probably one of the simplest non-trivial algorithms for solving sudoku: check that the initial partial solution is not inconsistent with the constraints, then proceed trying all possible numbers in all empty cells, one at a time, until a conflict is found, then backtrack to try the next solution in the last location where a conflict had not been found, and so on. This is essentially a depth-first search.

The algorithm is described in pseudocode in Algorithm 1. The algorithm traverses through the SUDOKU grid in an arbitrary, but fixed, order. The order is determined by the choice of the first invocation of the recursive algorithm, and the choice of (i', j') on line 10 in the pseudocode. If we want to go through the grid in a row-by-row order, for an initial partial solution M, the algorithm should be called by BACKTRACKINGSUDOKUSOLVER$(M, 1, 1)$, and the rule for determining (i', j') would be that if $j < n^2$, then $i' = i$ and $j' = j + 1$, otherwise $i' = i + 1$ and $j' = 1$.

Note that you should check that the initial partial solution is *feasible* before calling the recursive function: otherwise we might miss the possibility that some of the clues are inconsistent as such. With that check being done beforehand, we maintain the invariant that at any level of recursion, the grid contains no conflicts, and if we make it outside the grid, that means all cells have been successfully filled, so we have a solution.

Task 5. Implement the backtracking algorithm of Algorithm 1. Test for the correctness of your implementation.

It is also useful to be able to, not just find a single solution, but to *count* the number of solutions to a particular problem. It is easy to do this by a small modification to Algorithm 1: instead of returning Boolean values, return the total number of solutions, starting from the given partial solution. Whenever we reach the successful end of recursion (line 5), we return that we found exactly one new solution. The loop (lines 8–16) needs to be modified to accumulate the total number of solutions for each subproblem. Instead of an early return, we add the total number of subsolutions on line 12. Finally, on line 17, we simply return the total number of solutions found.

Task 6. Modify your implementation of the backtracking solver to count the total number of solutions. Test for the correctness of your implementation.

6 Input data generation

Finding suitable input data can be challenging. Although the Internet is full of SUDOKU problems, the datasets typically only contain examples of the case $n = 3$, and moreover have been constructed to be human-solvable. They are not very challenging for machine solvers, as they tend to behave in an unusually benign manner for a combinatorial search problem, and usually they are supposed to have the guarantee that they have a unique solution.

Algorithm 1 Pseudocode for the Backtracking SUDOKU solver. The algorithm modifies the grid in place.

```
1: Input: A $n^2 \times n^2$ SUDOKU grid $M$, the coordinates of the next cell to
   try to fill: row $i$ and column $j$.
2: Output: The grid is modified recursively in-place to a suitable solution,
   if possible, and true is returned. Otherwise, the grid is returned to its
   initial state and false is returned.
3: procedure BacktrackingSudokuSolver($M, i, j$)
4:     if $i, j$ is outside the grid then    ▷ We have successfully filled the grid
5:         return true
6:     end if
7:     for $k \leftarrow 1, 2, \ldots, n^2$ do
8:         if CanPlace($M, i, j, k$) then
9:             $M_{i,j} \leftarrow k$    ▷ We can insert the number $k$ at cell $M_{i,j}$ without
   introducing a conflict
10:            Let $(i', j')$ be the next cell.
11:            if BacktrackingSudokuSolver($M, i', j'$) then
12:                return true
13:            end if
14:            Set $M_{i,j}$ empty.
15:        end if
16:    end for
17:    return false
18: end procedure
```

Instead, we want to generate random instances for any parameter value n. With our backtracking solver at hand, we can do so easily. Algorithm 2 shows how to generate Sudokus with unique solutions. The idea is to simply add random numbers to the grid one at a time, and make sure that the SUDOKU can still be solved. Once the entire grid has been filled, we start removing clues one at a time, until the grid has at least two solutions. That is when we have found a random instance with a unique solution. It is easy to modify the algorithm to produce instances with no solutions, or to create instances with multiple solutions.

The check for solutions on line 11 of Algorithm 2 can be done with a proper invocation of the backtracking solver; note that by our definition of the recursion in Algorithm 1, the grid needs to be copied, as the procedure would modify it otherwise. Furthermore, we need to remember to check the grid for initial inconsistency; it may be convenient to create a wrapper function that takes care of this stuff. The number of solutions on line 14 can

Algorithm 2 Pseudocode for generating SUDOKU instances with exactly one unique solution.

```
1:  Input: Parameter value $n$, random number generator seed $s$.
2:  Output: A SUDOKU instance that has a unique solution.
3:  procedure GENERATERANDOMUNIQUE($n, s$)
4:      Initialize the random number generator with the seed $s$.
5:      Initialize the $n^2 \times n^2$ grid $M$ to be all empty.
6:      for $i \leftarrow 1, 2, \ldots, n^2$ do
7:          for $j \leftarrow 1, 2, \ldots, n^2$ do
8:              do
9:                  Draw random integer $k \in [n^2]$.
10:                 $M_{i,j} \leftarrow k$.
11:             while $M$ has no solutions
12:         end for
13:     end for
14:     while $M$ has exactly one solution do
15:         Draw $i, j$ at random.
16:         $k \leftarrow M_{i,j}$.
17:         Set $M_{i,j}$ empty.
18:     end while
19:     $M_{i,j} \leftarrow k$.
20: end procedure
```

be checked with the solution-counting variant of the backtracking algorithm.

Task 7. Implement an input generator that generates instances that have unique solutions for a given value of n, following 2. The generator should support setting a random number seed for reproducibility. You can also create other input generators if you should so choose.

7 Reduction to SAT

We recall the Boolean satisfiability problem:

Problem 2 (SAT)**.** Given a Boolean expression over n variables, does there exist a truth assignment such that the formula evaluates true?

Most SAT solvers, however, operate on the related CNF SAT problem:

Problem 3 (CNF SAT)**.** Given a Boolean expression over n variables in Conjunctive Normal Form (CNF), does there exist a truth assignment such that the formula evaluates true?

The following Boolean expression is in CNF:

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_5 \vee x_6)\,, \tag{2}$$

and Equation (2) is satisfied by the following truth assignment:

$$\begin{aligned} &x_1 \mapsto 0 \quad x_2 \mapsto 1 \quad x_3 \mapsto 1 \\ &x_4 \mapsto 1 \quad x_5 \mapsto 1 \quad x_6 \mapsto 1\,. \end{aligned} \tag{3}$$

SAT solvers generally expect problems to be formulated in CNF, and read the data in DIMACS format. The files commonly have the extension `.cnf` and are formatted according to following rules:

- Comment lines can be placed anywhere in the file, and they begin with the character `c`. Comments are ignored.
- Before any clauses are presented, there will be a *preamble* that consists of the following line:
  `p cnf` n m
  where n is replaced with the number of variables in the problem and m is replaced with the number of clauses.
- This is followed by the clauses. The clauses are presented as space-separated numbers. The numbers correspond to literals, and the numbering of variables starts at 1. That is, the numbers are from 1 to n. Negative literals are prefixed with a `-`. The clause is terminated by a 0.

Equation (2) could be presented in DIMACS format as follows:

```
p cnf 6 3
-1 -2 0
1 3 4 0
2 5 6 0
```

SAT solvers produce their output in a similar format. The rules of the output are as follows:

- Comment lines can be placed anywhere in the file, and they begin with the character `c`. Comments are ignored.
- If the formula is unsatisfiable, there will be a line `s UNSATISFIABLE`.
- If the formula is satisfiable, there will be a line `s SATISFIABLE`, followed by a satisfying truth assignment.

- The truth assignment may be split on multiple lines, each beginning with a `v`. The truth assignment consists of all variables from 1 to n, and the variables that map to 0 are prefixed with a `-`. The truth assignment is terminated with a 0.

For example, the truth assignment of Equation (3) could be presented as follows:

```
s SATISFIABLE
v -1 2 3 4 5 6 0
```

We can reduce a SUDOKU to CNF SAT as follows. Let M be an $n^2 \times n^2$ SUDOKU grid. We start by defining n^6 variables as follows: for each $(i, j, k) \in [n^2]^3$, let the variable $x_{i,j,k}$ be true if and only if the cell $M_{i,j}$ is occupied by the number k.

The constraints are encoded as the following $4n^4 + \frac{n^8-n^6}{2}$ clauses:

(i) For each $(i, k) \in [n^2]^2$:

$$(x_{i,1,k} \vee x_{i,2,k} \vee \cdots \vee x_{i,n^2,k}),$$

(ii) For each $(j, k) \in [n^2]^2$:

$$(x_{1,j,k} \vee x_{2,j,k} \vee \cdots \vee x_{n^2,j,k}),$$

(iii) For each $(i_0, j_0) \in [n]^2$ and $k \in [n^2]$,

$$(x_{(i_0-1)n+1,(j_0-1)n+1,k} \vee x_{(i_0-1)n+1,(j_0-1)n+2,k} \vee \cdots \vee x_{i_0 n, j_0 n,k})$$

(iv) For each $(i, j) \in [n^2]^2$,

$$(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,n^2}), \text{ and}$$

(v) For each $(i, j) \in [n^2]^2$ and each unordered pair $\{k, k'\} \in \binom{[n^2]}{2}$,

$$(\neg x_{i,j,k} \vee \neg x_{i,j,k'}).$$

Constraints (i)–(iii) encode the regular explicit SUDOKU constraints. The n^4 clauses of constraint (i) enforce the fact that each number must occur on some column of each row. Correspondingly, the n^4 clauses of constraint (ii) enforce the fact that each number must occur on some row of each column. Finally, the n^4 clauses of constraint (iii) enforce the fact that each number must occur in some cell of each of the n^2 subgrids of size $n \times n$.

Constraints (iv) and (v) encode the implicit assumption that each cell must contain exactly one number. The n^4 clauses of constraint (iv) enforce that each cell must contain at least one number. The $n^4 \cdot \frac{n^2(n^2-1)}{2} = \frac{n^8-n^6}{2}$ clauses of constraint (v) enforce the fact that if $x_{i,j,k}$ is set, then $x_{i,j,k'}$ cannot be set for any values of i, j, k, k' such that $k \neq k'$; that is, $x_{i,j,k} \implies \neg x_{i,j,k'}$ which when converted into CNF turns into $\neg x_{i,j,k} \vee \neg x_{i,j,k'}$, so by commutativity we only need to worry about the unordered pairs of $k \neq k'$. In other words, the constraint (v) enforces the fact that no cell can contain more than one number.

For example, for $n = 3$, we have 729 variables and 3240 clauses for the structural encoding of the general SUDOKU constraints. Clues can be added as additional clauses that contain unitary literals that enforce the certain number to be in a certain cell. For example, a clause containing only the literal $x_{2,3,4}$ would enforce the condition that $M_{2,3} = 4$.

Task 8. Write a routine that takes a SUDOKU grid as input and outputs the corresponding CNF SAT program in DIMACS format.

To evaluate the correctness of your implementation, you will want to input your DIMACS file into a SAT Solver. A good place to look for a SAT solver is on the website of the annual SAT competition[1]. For example, this year's winner, Armin Bière's Kissat[2] is probably a good choice.

Task 9. Write a routine that decodes the output of a SAT solver to a fully solved SUDOKU grid in the case that the output contains a satisfying truth assignment.

8 Exact Cover

We now turn to a related problem, the EXACT COVER. Let

$$U = \{u_1, u_2, \ldots, u_n\}$$

be some universe, and let

$$\mathcal{F} = \{S_1, S_2, \ldots, S_m\} \subseteq \mathcal{P}(U)$$

be a family of subsets over U. We say that the set S_i *covers* the elements u_j that satisfy $u_j \in S_i$. We extend this notion to a family of sets by saying that the family covers the elements that are included the union of the sets in

[1]`https://satcompetition.github.io/2022/index.html`
[2]`https://github.com/arminbiere/kissat`

the family. Furthermore, we say that the set cover is *exact* if each $u \in U$ is covered by exactly one set in the family. In other words, a family $\mathcal{F} \subseteq \mathcal{P}(U)$ is an exact cover of U if it satisfies the following constraints:

- $\bigcup_{i=1}^{m} \mathcal{F} = S_1 \cup \cdots \cup S_m = U$, and
- $S_i \cap S_j = \emptyset$ for all $i \neq j$.

The Exact Cover problem asks if, given a family of subsets over some universe, there exists a subset of the family that is an exact set cover of the universe:

Problem 4 (Exact Cover)**.** Given a family of sets $\mathcal{F} \subseteq \mathcal{P}(U)$ over some universe U, does there exist a subset $\mathcal{S} = \{S_1, \ldots, S_m\} \subseteq \mathcal{F}$ such that $S_1 \cup \cdots \cup S_m = U$, and $S_i \cap S_j = \emptyset$ for all $i \neq j$?

For a practical example, let the universe be $U = \{a, b, c, d, e, f, g\}$, and let the family of sets be

$$\mathcal{F} = \{\{c, e, f\}, \{a, d, g\}, \{b, c, f\}, \{a, d\}, \{b, g\}, \{d, e, g\}\} . \tag{4}$$

We can see that the family

$$\{\{c, e, f\}, \{a, d\}, \{b, g\}\} \tag{5}$$

is an exact cover of U.

One quite natural way to represent such families of sets is in terms of a binary matrix M. Suppose $|U| = n$ and $|\mathcal{F}| = m$. Then we could represent $\mathcal{F}$ as an $m \times n$ matrix where each column corresponds to a unique element in U and each row corresponds to a set in $\mathcal{F}$. We set the element $M_{ij} = 1$ if and only if $u_j \in \mathcal{F}_i$, and set $M_{i,j} = 0$ otherwise. With this notation, the family of Equation (4) could be represented as the following binary matrix:

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} , \tag{6}$$

and we can see that the exact cover of Equation (4) corresponds to rows 1, 4, and 5

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} ,$$

and computing the sum of these rows yields

$$\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1 & 1\end{bmatrix} ,$$

a row vector of all ones, as it should. Having zeros in the sum would indicate that some element was not covered, and having values larger than one would indicate that some element was covered multiple times.

Task 10. Design a data structure that can represent Exact Cover problems as a binary matrix. It is preferable to keep the data structure as simple as possible (but no simpler than is necessary).

9 Knuth's Algorithm X

In this section, we describe Knuth's Algorithm X [Knu00] that solves the Exact Cover problem in an abstract way. We delay the concrete algorithm until Section 11.

The essence of the algorithm is simple: we choose (any) order in which we go through the columns, and at each step of the recursion, we demand that we include a row in our solution set that covers this particular column. Then we remove any other columns covered by the row, and recurse. If we reach a case where there are no uncovered columns, we are done. Otherwise, if we have no more rows to choose from, we have failed, and we backtrack.

Knuth formulates the algorithm in terms of indeterminism, that is, that there are multiple execution paths the algorithm can choose from when deciding which row to use, and it suffices that one of these paths leads to a solution. As computers are inherently deterministic, we reformulate Knuth's algorithm in pseudocode in Algorithm 3 in a deterministic manner. The algorithm modifies the matrix in-place. When we say that a row or a column is *covered*, we assume it is removed from the matrix for recursive invocations; however, we tacitly assume that we can recover the rows and columns when we *uncover* them. You can think of this happening by making a copy of M with the relevant rows and columns deleted; we delay the description of the data structure that makes this process efficient until Section 10.

Task 11. Using pen and paper, go through the execution of Algorithm 3 on the matrix of Equation 6. Make sure you understand what the algorithm does at each step. You don't have to record this in your report, but understanding the full algorithm will be very difficult unless you master it on a toy example with a trivial data structure.

Algorithm 3 Pseudocode for the Knuth's Algorithm X.

```
1: Input: A binary matrix $M$ whose columns correspond to the elements
   in the universe and rows to the sets in the family of sets. Initially $S = \emptyset$.
2: Output: Store a subset of the rows that is an exact cover of the universe
   in $S$, or set $S$ empty if no such set exists.
3: procedure ALGORITHMX($M, S$)
4:     if $M$ has no columns then
5:         Terminate.        ▷ Problem solved and $S$ contains a valid solution.
6:     else
7:         Choose the next column $j$.                    ▷ Any order will do.
8:         for all rows $i$ such that $M_{ij} = 1$ do
9:             Add $i$ to $S$.
10:            for all columns $j'$ such that $M_{ij'} = 1$ do
11:                Cover the column $j'$.
12:                for all rows $i'$ such that $M_{i'j'} = 1$ do
13:                    Cover the row $i'$.
14:                end for
15:            end for
16:            ALGORITHMX($M, S$)          ▷ Recurse on the reduced matrix.
17:            Remove $i$ from $S$.
18:        end for
19:    end if
20: end procedure
```

10 Dancing Links

The motivation for this section is to describe a data structure that enables one to store sparse binary matrices in such a way that the columns and rows can be easily covered and uncovered. We will follow Knuth [Knu00], but try to include a little bit more details.

The main idea is to represent the matrix as a network of doubly linked list *nodes*. Each node represents a one in the matrix, and is connected as a torus both horizontally and vertically: one can traverse around the matrix along the rows and along the columns, in both directions. In addition to the nodes representing ones, the data structure contains special *column headers*. These are nodes that do not correspond to actual ones in the matrix, but serve as entry-point to individual columns. This allows one to reach ones that are on different rows and do not share ones on the same column. Finally, there is a special column header called the *root* or h that serves as an entry point into the data structure.

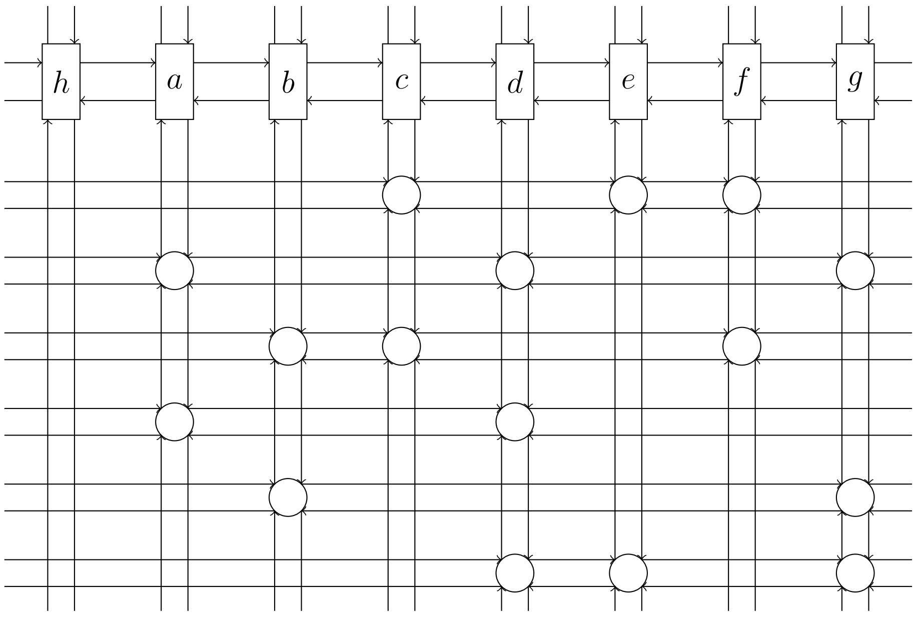

Figure 1: The Dancing Links representation of the matrix of Equation (6). Column headers are marked as square nodes and ordinary nodes representing ones in the matrix are marked with round nodes. The header node is marked h and has no up or down neighbors. The doubly linked lists have a torus structure both horizontally and vertically: the links without arrow ends at the edges of the image extend implicitly to the other side.

The matrix of Equation (6) is shown in this *dancing links* data structure in Figure 1. Note the torus structure of the nodes both horizontally and vertically: the nodes are linked in a ring that extends around the edges of the figure, so traversing sufficiently far in any given direction eventually leads back to the original node. The data structure allows one to traverse along the ones in a row in either direction, or along the ones in a column. Ones in a column that does not have a one on the same row can be reached through the column headers.

Every node x must support the following operations:

- $L[x]$: return the left neighbor of x,
- $R[x]$: return the right neighbor of x,
- $U[x]$: return the up neighbor of x,
- $D[x]$: return the down neighbor of x, and

- $C[x]$: return the column node associated with x.

Furthermore, each column header c must also support the following operations:

- $N[c]$: return the *name* of the column, and
- $S[c]$: return the *size* of the column.

The operation $N[c]$ is used when decoding solutions, as it recovers the identity of the column covered by a particular one. The operation $S[c]$ is useful for recording the number of ones in the particular column, mainly for the purpose of creating heuristics for choosing the order in which to cover columns. If such heuristics are not used, this operation is not needed. Note that column headers are also nodes.

Initially, we might want to think that when we create a self-standing node, it is connected only to itself: $L[x] = R[x] = U[x] = D[x] = x$. Since we need to construct the toruses, it may be useful to implement *hooking* operations as auxiliary functions. Such functions would connect a new node into a pre-existing torus. For example, let us initialize three nodes x, y, z, as seen in Figure 2a; all links from all three nodes point to the nodes themselves. Let us define the operation `hookRight` that adds a new node to the torus to the right of the left hand operand. We would define the operation x`.hookRight(`y`)` as the following sequence of operations:

$$R[y] \leftarrow R[x]; \quad L[R[y]] \leftarrow y; \quad L[y] \leftarrow x; \quad R[x] \leftarrow y;$$

If we apply a x`.hookRight(`y`)` operation to add the node y to the (single-node) torus of x, to the right hand side of x, we get the result of Figure 2b. Furthermore, if we continue and apply y`.hookRight(`z`)` to add the third node z to the torus to the right of y, we get the result of Figure 2c. It is instructive to work this out with pen and paper. Hooking nodes in other directions (such as left, up, down) is analoguous. You will need to implement at least two hooking functions (left/right and up/down) to be able to construct the matrix.

Furthermore, we will need to be able to *link* and *unlink* nodes along either direction of the torus. Unlinking is used when a column or a node is covered: it disconnects the node from the torus. Linking does the reverse operation and restores the node into the link. It is imperative that the links of the node that is severed from the torus are not zeroed out; the entire point of Dancing Links is to maintain this information and to easily be able to restore severed connections.

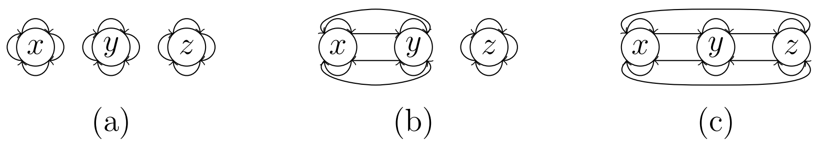

Figure 2: (a) Three freshly initialized nodes x, y, z. (b) The result of x.`hookRight(`y`)`. (c) The result of y.`hookRight(`z`)`.

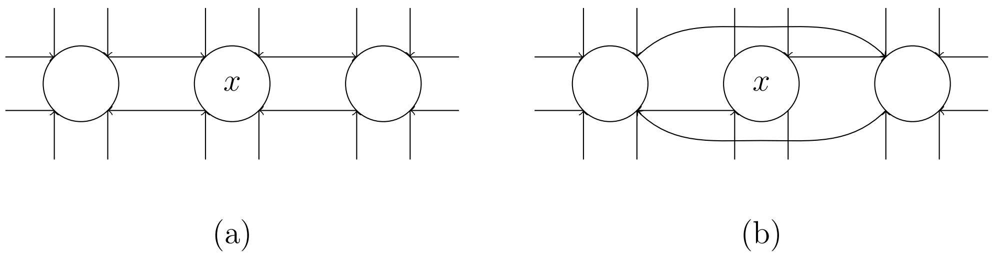

Figure 3: (a) Three nodes before unlinking. (b) The nodes after unlinking the middle node x.

Unlinking the node x along the left–right axis is done by the following sequence of operations:

$$R[L[x]] \leftarrow R[x]; \quad L[R[x]] \leftarrow L[x]; \tag{7}$$

Linking is done by the reversing the operations:

$$L[R[x]] \leftarrow x; \quad R[L[x]] \leftarrow x; \tag{8}$$

Linking and unlinking along the up–down axis is done analoguously.

Figure 3a shows three nodes along the left–right torus. Figure 3b shows the result after performing the `unlink(`x`)` operation. Note that, although the neighbors of x no longer have a way of reaching x along the left–right axis, x still remembers its former neighbors, so if we were to apply `link(`x`)`, we would get back to the situation of Figure 3a.

Task 12. Go through the hooking and linking examples with a pen and paper. Make sure you understand how the links work. You do not have to record this in your report.

Task 13. Design and implement the data structure for ordinary nodes and column headers. Note that you have both kinds of nodes in your toruses.

We are now ready to implement the full matrix. One way to construct such a matrix is by first initializing the root node, then constructing the column headers for each column, and hooking them in place. Then, we go through the input matrix one row at a time, initialize a node for each one in the row, and hook them into their correct locations, both left–right and up–down.

Task 14. Using the auxiliary node data structure, as you implemented in Task 13, implement the Dancing Links matrix representation of a binary matrix, akin to Figure 1. You should implement a function that converts a regular binary matrix into this data structure. The order of rows need not be preserved, but you should implement functionality to record the name of the columns, such that the columns can be identified from any given node in the matrix.

Task 15. Implement a function that prints the content of the matrix in a human-readable form, taking a Dancing Links matrix as input. Since the order of rows is lost, and one cannot tell by direct observation if a particular row has been printed yet, storing the references ot the printed nodes in a `HashSet` might be useful.

11 DLX

We are now ready to describe the main algorithm, *Dancing Links X (DLX)*. This is a practical variant of Knuth's Algorithm X that was described in Section 9 using the Dancing Links data structure from Section 10.

The main procedure of our algorithm, SEARCH, is presented in pseudocode in Algorith 4. Essentially, this is just a restatement of Algorithm 3 but filling missing pieces with the Dancing Links data structure. The auxiliary procedures are presented in Algorithm 5.

The termination condition for the algorithm is checking if the root h is a neighbor of itself. If this is the case, then all columns have been covered and we are done. Otherwise we choose an uncovered column, and try all possible rows one at a time and cover the columns that have ones on that row, and then recurse. If we have not terminated our execution, we will backtrack, undo our coverings, and continue with the next row.

The procedure COVER in Algorithm 5 unlinks the present column from the list of column headers, and then unlinks all rows that have a one on that column. The procedure UNLINKLEFTRIGHT corresponds to the sequence of Equation (7) and UNLINKUPDOWN can be implemented analoguously. The size of the column is decremented to keep track of the number of ones in any particular column which is used by the CHOOSE function.

The procedure UNCOVER essentially just undoes everything COVER does by executing the inverse operations in a backwards order. Similarly, the procedure LINKLEFTRIGHT corresponds to the sequence of Equation (8) and LINKUPDOWN can be implemented analoguously.

The function CHOOSE can be used for choosing the next column to cover. It serves to minimize the branch factor: we choose the column that has the least number of ones in it. This means that we minimize the number of branches in the search tree, and is a useful heuristic. In the context of SUDOKU, this is somewhat analoguous to choosing the next empty cell to fill that has the fewest number of possible non-contradictory values to choose from.

Finally, the procedure PRINT goes through the stack and prints the row. Every element of the stack is an element (in an unknown order) of some row, and each element is from a different row. So, we iterate over all of the neighbors of the element, and query the corresponding column header for its name, and print this out. Printing here is in an abstract sense: printing can be understood as the recovery of the identity of the columns of the row, and as such can result in the creation of a new data structure, such as a list of lists that contains the solution.

Task 16. Implement the DLX algorithm as per Algorithms 4 and 5. Test it for correctness.

It is easy to modify the DLX algorithm to count the solutions instead of just recovering a solution. This happens by returning a 1 on line 5 of Algorithm 4, as we have found one particular solution, and in other cases we iterate over all rows and sum the number of solutions. This is similar to the modification of Algorithm 1 to count the solutions.

Task 17. Modify DLX to create a variant that counts the number of solutions.

12 Reduction from Sudoku to Exact Cover

We are now ready to construct a DLX-based SUDOKU solver. The solver works by transforming a SUDOKU grid into a binary matrix that represents an EXACT COVER problem.

Each row of the binary matrix corresponds to a particular row, column, and value, so there are n^6 rows in total, much like the number of variables for

Algorithm 4 Pseudocode for the main SEACH routine of DLX.

```
1: Input: Root node of our Dancing Links data structure $h$, and a stack $S$ that is initially empty that will be used to record the present candidate solution.
2: Output: Upon termination, the stack $S$ will store the reference to one node from each row included in the solution if a solution is found, or the stack will be empty.
3: procedure SEARCH($h, S$)
4:     if $R[h] = h$ then
5:         PRINT(S) and terminate.    ▷ There are no columns, so we have found a valid solution.
6:     else
7:         Choose the next column $c$.
8:         COVER($c$).
9:         for $r \leftarrow D[c], D[D[c]], \ldots,$ while $r \neq c$ do ▷ Iterate over all rows that have a one on column $c$.
10:            Push $r$ into $S$.
11:            for $j \leftarrow R[r], R[R[r]], \ldots,$ while $j \neq r$ do
12:                COVER($j$).    ▷ Cover the other columns covered by the row $r$.
13:            end for
14:            SEARCH($h, S$).
15:            Pop $r$ from $S$ and let $c \leftarrow C[r]$.
16:            for $j \leftarrow L[r], L[L[r]], \ldots,$ while $j \neq r$ do
17:                UNCOVER($j$).    ▷ Undo.
18:            end for
19:        end for
20:        UNCOVER($c$).    ▷ Undo.
21:    end if
22: end procedure
```

the SAT reduction. There are $4n^4$ columns, corresponding to four different kinds of constraints with n^4 constraints each. The constraints are:

(i) For each $i, k \in [n^2]$, the number k must be present on row i,

(ii) For each $j, k \in [n^2]$, the number k must be present on column j,

(iii) For each $i_0, j_0 \in [n]$ and each $k \in [n^2]$, the number k must be present in the $n \times n$ subgrid (i_0, j_0), and

(iv) For each $i, j \in [n^2]$, there must be a number in cell (i, j).

Algorithm 5 Pseudocode for auxiliary routines of DLX.

```
1: function CHOOSE($h$)                                    ▷ Choose the next column to cover
2:     $s \leftarrow \infty$.
3:     for $j \leftarrow R[h], R[R[h]], \ldots,$ while $j \neq h$ do
4:         if $S[j] < s$ then
5:             $c \leftarrow j$.                            ▷ Select the minimum-sized column.
6:             $s \leftarrow S[j]$.
7:         end if
8:     end for
9:     return $c$.
10: end function
11: procedure COVER($c$)                                             ▷ Covers a column
12:     UNLINKLEFTRIGHT($c$).
13:     for $i \leftarrow D[c], D[D[c]], \ldots,$ while $i \neq c$ do           ▷ Iterate over rows.
14:         for $j \leftarrow R[i], R[R[i]], \ldots,$ while $j \neq i$ do
15:             UNLINKUPDOWN($j$).                         ▷ Remove ones from the row.
16:             $S[C[j]] \leftarrow S[C[j]] - 1$.     ▷ Decrement the size of the column.
17:         end for
18:     end for
19: end procedure
20: procedure UNCOVER($c$)                                          ▷ Undoes covering.
21:     for $i \leftarrow U[c], U[U[c]], \ldots,$ while $i \neq c$ do
22:         for $j \leftarrow L[i], L[L[i]], \ldots,$ while $j \neq i$ do
23:             $S[C[j]] \leftarrow S[C[j]] + 1$.
24:             LINKUPDOWN($j$).
25:         end for
26:     end for
27:     LINKLEFTRIGHT($c$).
28: end procedure
29: procedure PRINT($S$)
30:     while $S$ is non-empty do
31:         Pop $n$ from $S$.
32:         $j \leftarrow n$.
33:         do
34:             Print $N[C[j]]$.
35:             $j \leftarrow R[j]$.
36:         while $j \neq n$
37:     end while
38: end procedure
```

Constraint (iv) works much like the constraints (iv) and (v) in the SAT program. However, since EXACT COVER mandates that a number 1 can be present only on one row per each column, this implicitly codes the fact that there cannot be multiple numbers in any cell.

We construct the matrix by going through all of the rows. Each row corresponds to a triple $(i, j, k) \in [n^2]^3$. We then set exactly four columns to 1 and leave others zero: the column corresponding to the constraint (i) and i and k, and so on.

Clues can be encoded by zeroing out any rows that are inconsistent with the clues. If we have a clue (i, j, k), that is, that the number k occupies the cell (i, j) in the grid, we would zero-out all rows (i, j, k') for all $k' \neq k$.

Task 18. Implement a procedure that converts a SUDOKU grid into a EXACT COVER binary matrix.

Task 19. Implement a SUDOKU solver that takes in a SUDOKU grid as input, converts it into an EXACT COVER binary matrix, constructs a Dancing Links matrix, applies DLX, and then decodes the solution, and returns it as a filled SUDOKU grid.

13 Horse race

We have now implemented three different solutions to SUDOKU: the basic backtracking algorithm, a reduction to SAT and the use of an external SAT solver, and a solver based on DLX. It is time to compare these three approaches.

Task 20. Design and perform an experiment where you compare the three ways to solve a SUDOKU in terms of runtime and scalability with respect to the parameter size n. Use your best judgment in reporting the results in your report.

14 Report

Write a report in LaTeX, following the advice given on the lectures and the feedback you got from your report in the first assignment. The report must not exceed **6 pages**, excluding possible tables, figures, and references. Return the report and the code you produced as a **single zip file** through LearnIT. Your group only needs to return one report.

References

[Knu00] Donald E. Knuth. “Dancing links”. In: arXiv:cs/0011047 (2000).

[YS03] Takayuki Yato and Takahiro Seta. “Complexity and Completeness of Finding Another Solution and Its Application to Puzzles”. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A.5 (2003).