Applied Algorithms
Lecture 7: Sudoku, Exact Cover, Dancing Links

Matti Karppa

IT-Universitetet i København

2022-10-11 12:15

Sudoku

- Sudoku is a combinatorial puzzle.
- We are given a partially filled $n^2 \times n^2$ **grid** of **cells**.
- We say that the filled cells of the **partial solution** are **clues**.
- We are asked if we can fill the remaining cells with numbers from 1 to n^2 such that
  (i) Each number from 1 to n^2 occurs on each row exactly once,
  (ii) Each number from 1 to n^2 occurs on each column exactly once,
  (iii) Each number from 1 to n^2 occurs in each $n \times n$ **subgrid** exactly once.
- The standard Sudoku corresponds to $n = 3$.
- Note our formulation: what n means, and that we ask if there exists **at least one solution**.
- Formulated like this, Sudoku is known to be **NP**-complete.

SUDOKU example

	1	2	
			4
	2		

SUDOKU example

4	1	2	3
2	3	1	4
1	4	3	2
3	2	4	1

Backtracking algorithm

- Idea: go through empty cells (in any order) and try all possible numbers that do not break constraints, then mover to next cell.
- We maintain the following **invariant**: upon entering any new cell, all number-placements thus far have been valid (not breaking any constraints).
- If at any point we find a cell with no feasible numbers, backtrack.
- If we manage to fill all cells, we're done.
- We must check initially that the partial solution is **feasible** (why?)

Backtracking algorithm pseudocode

```
procedure BacktrackingSudokuSolver(M, i, j)
    if i, j is outside the grid then
        return true
    for k ← 1, 2, ..., n^2 do
        if CanPlace(M, i, j, k) then
            M_{i,j} ← k
            Let (i', j') be the next cell.
            if BacktrackingSudokuSolver(M, i', j') then
                return true
            Set M_{i,j} empty.
    return false
```

Backtracking algorithm example

4	**2**	**6**	**3**	**5**	1	**7**	**9**	8
1	**3**	**49**	2	7	8	**5X**	**6X**	**X**
	7	8			9	1		
2	1	3	4		6			
			8	9	7		1	
				2			5	
3		1					7	5
		7		1				2
	4	5				6		

Backtracking algorithm example

6	**9**	**2**	**5**	**4**	1	**7**	**3**	8
1	**3**	**4**	2	7	8	**5**	**6**	**9**
5	7	8	**3**	**6**	9	1	**2**	**4**
2	1	3	4	**5**	6	**8**	**9**	**7**
4	**5**	**6**	8	9	7	**2**	1	**3**
7	**8**	**9**	**1**	2	**3**	**4**	5	**6**
3	**2**	1	**6**	**8**	**4**	**9**	7	5
8	**6**	7	**9**	1	**5**	**3**	**4**	2
9	4	5	**7**	**3**	**2**	6	**8**	**1**

Exact Cover

- Let $\mathcal{F} \subseteq \mathcal{P}(U)$ be a **family of sets** over some **universe**.
- For a $S \subseteq U$, we say that S **covers** the elements $u \in U$ that satisfy $u \in S$.
- Problem: does there exists a $\mathcal{S} = \{S_1, S_2, \ldots, S_m\} \subseteq \mathcal{F}$ such that
  - $\cup_{i=1}^{m} S_1 \cup \cdots \cup S_m = U$ (every element of the universe is covered), and
  - $S_i \cap S_j = \emptyset$ for $i \neq j$ (the sets are disjoint, so every element is covered **exactly** once)

Exact Cover example

- Let
  - $U = \{a, b, c, d, e, f, g\}$.
  - $\mathcal{F} = \{\{c, e, f\}, \{a, d, g\}, \{b, c, f\}, \{a, d\}, \{b, g\}, \{d, e, g\}\}$.
- Choose $\mathcal{S} = \{\{c, e, f\}, \{a, d\}, \{b, g\}\} \subseteq \mathcal{F}$.
- Then,
  $\bigcup_{S \in \mathcal{S}} = \{c, e, f\} \cup \{a, d\} \cup \{b, g\} = \{a, b, c, d, e, f, g\} = U$.
- Furthermore, $\{c, e, f\} \cap \{a, d\} = \emptyset$, $\{c, e, f\} \cap \{b, g\} = \emptyset$, and $\{a, d\} \cap \{b, g\} = \emptyset$.
- Therefore, $\mathcal{S}$ is an exact cover.

EXACT COVER as a matrix

- We can represent the EXACT COVER problem in matrix form as follows.
- Denote $U = \{u_1, u_2, \ldots, u_n\}$.
- Denote $\mathcal{F} = \{S_1, S_2, \ldots, S_m\}$.
- Let M be an $m \times n$ matrix whose columns correspond to elements in U and rows to elements in $\mathcal{F}$.
- Let $M_{ij} = 1$ iff $u_j \in S_i$.
- Then $\mathcal{S}$ is an exact cover iff the rows corresponding to the elements of $\mathcal{S}$ sum up to a vector of all ones.

Exact Cover matrix example

- As before, let
  - $U = \{a, b, c, d, e, f, g\}$.
  - $\mathcal{F} = \{\{c, e, f\}, \{a, d, g\}, \{b, c, f\}, \{a, d\}, \{b, g\}, \{d, e, g\}\}$.
- The corresponding matrix would be

$$
\begin{array}{cc}
 & \begin{array}{ccccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \end{array} \\
\begin{array}{c} \{\text{c,e,f}\} \\ \{\text{a,d,g}\} \\ \{\text{b,c,f}\} \\ \{\text{a,d}\} \\ \{\text{b,g}\} \\ \{\text{d,e,g}\} \end{array} &
\begin{bmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1
\end{bmatrix}
\end{array}
$$

Exact Cover matrix example

- Now, choosing the proper rows

$$
\begin{array}{c c}
 & \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{array}{c} \{c,e,f\} \\ \{a,d\} \\ \{b,g\} \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}
\end{array}
$$

- And summing them up

$$
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}
\end{array}
$$

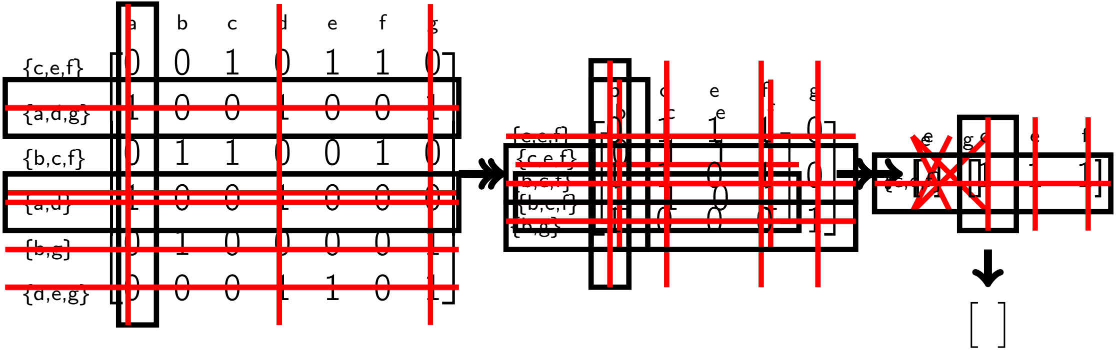
Knuth's Algorithm X

- Abstract algorithm that solves Exact Cover.
- Idea: choose column (element in U) one by one to cover.
- For the chosen column, choose a row that covers it (a set in $\mathcal{F}$).
- Remove all columns that are covered by the row, and all rows that have a one in any of the covered columns.
- Recurse. If all columns are covered, we are done. Otherwise, backtrack and choose another row.

Knuth's Algorithm X pseudocode

procedure AlgorithmX(M, S)
    if M has no columns **then**
        Terminate.
    Choose the next column j.
    for all rows i such that $M_{ij} = 1$ **do**
        Add i to S.
        for all columns j' such that $M_{ij'} = 1$ **do**
            Cover the column j'.
            for all rows i' such that $M_{i'j'} = 1$ **do**
                Cover the row i'.
        AlgorithmX(M, S)
        Remove i from S.

Knuth's Algorithm X example



All columns have been covered!

Dancing Links

- A data structure that makes the basic operations (deleting a row/column, and restoring it afterwards) very efficient for implementing Algorithm X.
- Basic idea: assuming the matrix is sparse (few ones), represent each one as a **node**.
- Nodes are connected as doubly-linked lists horizontally along a row in a torus structure, and likewise vertically along a column.
- In addition, each column contains a **column header** that is a special node that allows access to any other column.
- The entry point into the data structure is a **root** h that is a special column header without any ones.

Nodes

All nodes x 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.

Column headers

Column headers are nodes and support all node operations. In addition, each column header c also supports the following operations:

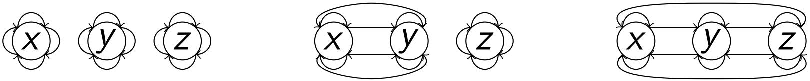
- $N[c]$: return the *name* of the column, and
- $S[c]$: return the *size* of the column.

Connecting nodes

- Upon initial creation, each node is connected only to itself.
- New nodes can be **hooked** into the torus.
- For example, the following sequence hooks the node y to the right side of x:

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

- Example: Hook first y to the right of x, then z to the right of y:



- Other directions are analoguous.

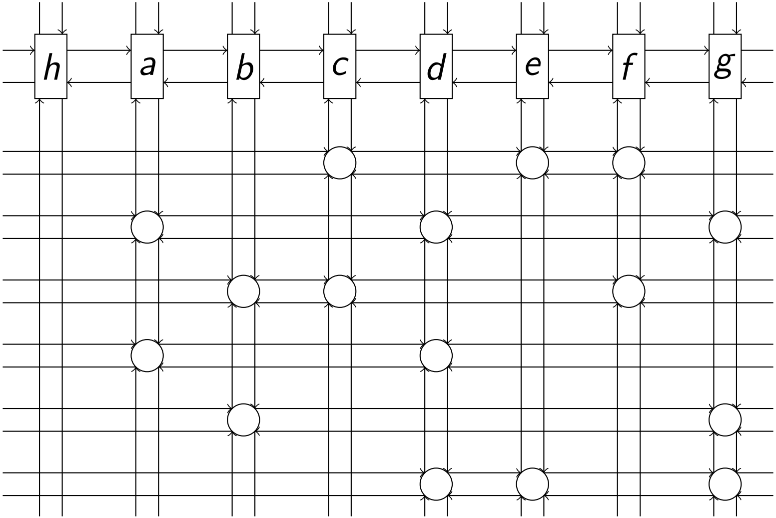
Constructing a Dancing Links matrix

Given as input an $n \times m$ binary matrix M,

- Construct h.
- For each column $j = 1, 2, \ldots, m$, construct the column header c_j and hook it left to h (or the right of the last column header added). It will be useful to store references to the columns in an array for construction of other rows.
- For each row $i = 1, 2, \ldots, n$:
  - Construct node x, add column reference, connect it to up of $C[x]$, hook it to the left of $D[C[x]]$ (or the right of the last node added).

Example matrix

The same matrix as in previous slides, but as a Dancing Links DS:



Linking/Unlinking nodes

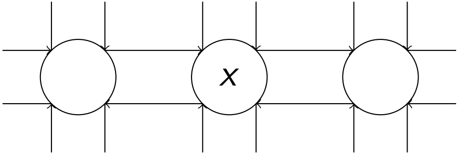
- Nodes can be unlinked from the torus and relinked.
- The node will remember where it was.
- Unlink left/right (up/down is analoguous):

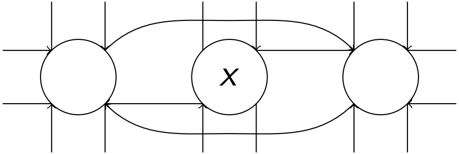
$$R[L[x]] \leftarrow R[x]; \quad L[R[x]] \leftarrow L[x];$$

- Link simply undoes the operation:

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

- Example: before and after unlinking x:





Covering and uncovering a column

- A column can be covered as follows:
- Unlink the column header (left/right) from the header torus.
- For each row that has a one on the column,
  - For each node that has a one on the horizontal torus, unlink the node (up/down) from the vertical torus.
  - In addition, decrement the size of the column in question by one.
- This effectively removes any row that has a one in the column being covered from being accessible from other columns.
- Uncovering simply does the same operations in reverse.

Covering a column pseudocode

procedure Cover(c)
    UnlinkLeftRight(c).
    for $i \leftarrow D[c], D[D[c]], \ldots,$ **while** $i \neq c$ **do**
        for $j \leftarrow R[i], R[R[i]], \ldots,$ **while** $j \neq i$ **do**
            UnlinkUpDown(j).
            $S[C[j]] \leftarrow S[C[j]] - 1.$

Uncovering a column pseudocode

procedure Uncover(c)
    for $i \leftarrow U[c], U[U[c]], \ldots,$ **while** $i \neq c$ **do**
        for $j \leftarrow L[i], L[L[i]], \ldots,$ **while** $j \neq i$ **do**
            $S[C[j]] \leftarrow S[C[j]] + 1.$
            LinkUpDown(j).
    LinkLeftRight(c).

Dancing Links X (DLX)

- Given an Exact Cover binary matrix, find a subset of rows that sum up to all ones.
- Main routine Search implements Knuth's Algorithm X, but uses the cover/uncover functions to remove the columns or rows.
- Column size information can be used to infer the choice of next column that minimizes **branch factor**: choose the column with minimum size (number of uncovered ones).
- We need to maintain the solution in, for example, a stack.
- Upon termination, we **print** the solution (in an abstract sense; we recover the identity of columns in question per row in solution).

Branch-factor minimizing choice of column

function Choose(h) $\triangleright$ Choose the next column to cover
    $s \leftarrow \infty$.
    for $j \leftarrow R[h], R[R[h]], \ldots,$ **while** $j \neq h$ **do**
        if $S[j] < s$ **then**
            $c \leftarrow j$. $\triangleright$ Select the minimum-sized column.
            $s \leftarrow S[j]$.
    return c.

Printing

```
procedure Print(S)
    while S is non-empty do
        Pop n from S.
        j ← n.
        do
            Print N[C[j]].
            j ← R[j].
        while j ≠ n
```

DLX search procedure

procedure Search(h, S)
  if $R[h] = h$ **then** Print(S) and terminate.
  Choose the next column c, then Cover(c).
  for $r \leftarrow D[c], D[D[c]], \ldots,$ **while** $r \neq c$ **do**
    Push r into S.
    for $j \leftarrow R[r], R[R[r]], \ldots,$ **while** $j \neq r$ **do**
      Cover(j).
    Search(h, S).
    Pop r from S and let $c \leftarrow C[r]$.
    for $j \leftarrow L[r], L[L[r]], \ldots,$ **while** $j \neq r$ **do**
      Uncover(j).
  Uncover(c).

Reducing SUDOKU to EXACT COVER

- There will be n^6 rows, one for each (i, j, k) (place number k into cell (i, j)).
- There will be $4n^4$ columns: four different kinds of constraints, n^4 constraints each.
- Each row will contain exactly four or zero ones.
- Constraints:
  (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).
- Clues are handled by zeroing the conflicting rows; for example, if we have k in (i, j), we would zero-out any rows that correspond to $k' \neq k$ for (i, j).

Reducing SUDOKU to SAT

- There will be n^6 variables x_{ijk} (place number k into cell (i,j)), and there will be $4n^4 + (n^8 - n^6)/2$ clauses.
- Clauses:
  (i) For each $(i,k) \in [n^2]^2$: $(x_{i,1,k} \vee x_{i,2,k} \vee \cdots \vee x_{i,n^2,k})$, so each number k must occur on each row i,
  (ii) For each $(j,k) \in [n^2]^2$: $(x_{1,j,k} \vee x_{2,j,k} \vee \cdots \vee x_{n^2,j,k})$, so each number k must occur on each column j,
  (iii) For each $(i_0,j_0) \in [n]^2$, $k \in [n^2]$: $(x_{(i_0-1)n+1,(j_0-1)n+1,k} \vee \cdots \vee x_{i_0 n,j_0 n,k})$, so each number k must occur in each subgrid,
  (iv) For each $(i,j) \in [n^2]^2$: $(x_{i,j,1} \vee x_{i,j,2} \vee \cdots \vee x_{i,j,n^2})$, so each cell must contain a number, and
  (v) For each $(i,j) \in [n^2]^2$ and each unordered pair $(k,k') \in \binom{[n^2]}{2}$: $(\neg x_{i,j,k} \vee \neg x_{i,j,k'})$, so no two numbers can occupy the same cell.
- Clues are handled by adding clauses that contain only the literal x_{ijk} corresponding to the clue.

Mandatory Assignment 3

Mandatory Assignment 3 is on LearnIT. Topic will be solving SUDOKU in three different ways:

(i) Backtracking algorithm,
(ii) Reduction to SAT, and
(iii) Reduction to EXACT COVER and DLX.

Due: 2022-11-08.

There are also some simple exercises on LearnIT that might be useful (especially regarding the reductions)