Dancing Links and Algorithm X to solve Sudoku problems

Patrick Naugle
Computer Science Undergraduate
University of South Florida
Tampa, United States
pnaugle@usf.edu

***Abstract*— The main objective of this experiment is to implement two ideas by Dr. Don Knuth, Dancing Links and Algorithm X, both in the C++ programming language. Knuth refers to this combination as DLX. These components are used to solve exact cover problems in a depth-first, backtracking search. In this project, Knuth's ideas are applied in order to solve Sudoku problems.**

The next objective is to compare the speed of the C++ implementation to that of a naive Sudoku-solving algorithm also implemented in C++. (*Abstract***)**

Keywords—dancing links, Knuth, Sudoku, algorithm (*key words****)***

I. Introduction

Exact cover problems are a subset of constraint satisfaction problems. Exact cover can be represented by a binary matrix in which true values are used to satisfy constraints. The minimal set of true values that satisfies all constraints is called the exact cover. This is the goal of Algorithm X, to find the exact cover for a problem.

II. Knuth's Ideas

A. Algorithm X

Algorithm X is a search algorithm that uses a matrix representation of exact cover to eliminate unnecessary values and to obtain the minimum set of values. Each value is represented as a node in a circular, doubly-linked list in the form of a row-and-column matrix. Each node has a link to the right, to the left, above and below, including column header nodes. Each column in the matrix should have as few nodes as possible to satisfy each constraint. Algorithm X is used in conjunction with Dancing Links in order to efficiently flow through backtracking.

B. Dancing Links

Dancing Links is described as such by Knuth due to the ability to remove and reinstate nodes from the circular, doubly-linked lists that make up the rows and columns of the node-based cover matrix. Removing a node is described as "covering" and reinstating a node is deemed "uncovering." The multi-directional links that each node contains allows for searching to remove two links and if the search proves fruitless, the two remaining links can recover the nodes' positions in the matrix by reattaching the removed links. As an example, a column can be chosen and each row in the column can be explored using the column's downward chain of links and the row nodes' rightward chain of links. If a dead-end is reached, the column can be reinstated by using the column's upward chain and rows' leftward chain.

III. Algorithm X Pseudo Code

Step	Action
1.	If there are no columns, solution is found, return solution
2.	Else, choose a column according to a heuristic
3.	Cover the column and all row nodes in the column along with all nodes in other columns in the same row
4.	Include each node in the column in the solution
5.	Repeat recursively
6.	If led to a dead-end, remove row from solution
7.	Uncover row nodes (and connected nodes) and column

IV. Sudoku

Sudoku puzzles are exact cover problems and pair well with Knuth's DLX. Sudoku puzzles are given as a partially completed nine-by-nine matrix of cells meant to contain a single integer between 1 and 9. Some cells are given and all others are blank. Rows and columns of cells must not contain repeated values, nor can the nine three-by-three sub-matrices. The goal of a Sudoku is to fill in the blank cells while adhering to the previously mentioned constraints.

A. Sudoku Constraints

As described, with Sudoku there are four constraints:

Constraint Number	Constraint Description
1.	Every cell of the Sudoku matrix must contain just one integer between 1 and 9
2.	Every row of the Sudoku matrix must contain only one cell of each integer from 1 to 9
3.	Every column of the Sudoku matrix must contain only a single cell of each integer value from 1 to 9
4.	Every three-by-three sub-matrix, for a total of nine, must contain just one cell with each integer between 1 and 9

B. Exact Cover Matrix

For the exact cover matrix there must be 729 rows, based on the nine rows, nine columns and nine possible integers in the Sudoku problem (9*9*9 = 729), and 324 columns, due to there being 81 cells in the Sudoku matrix and there are four columns (81*4 = 324). Each row in the exact cover matrix satisfies 4 constraint columns. Therefore, for all valid solutions, every column in the matrix will be satisfied by one node, for a total of 81 rows, if there exists a single solution. The implementation of the exact cover matrix for Sudoku was influenced by Bob Hanson [1].

V. DLX vs Naïve in C++

A. DLX

DLX is implemented using a structure for nodes and a class for all other functionality. Nodes have 5 pointers, four for directionality and one for the column in which it belongs. It also has 3 integer values to surmise row and column numbers as well as the count off nodes in a column used by column nodes. The main functionality of the class is to cover, uncover and search along with the ability to convert a string representation of a Sudoku puzzle into a node-based exact cover matrix.

B. Naive

The naïve algorithm is provided by Rafal Szymanski [2]. This algorithm is translated from Java into C++. This algorithm tries all possibilities for each cell and moves forward recursively until it is necessary to backtrack. A major alteration that was implemented includes throwing an exception once a solution has been reached. This was added due to the exceptional amount of time it would take for the algorithm to find its way out of recursion.

C. *Performance Comparison*

Both algorithms have been tested for accuracy and have passed. Using two sets of Sudoku problems from Peter Norvig's website [3], both algorithms are tested for speed.

One set of problems is 50 "easy" Sudoku problems and the other set is 95 "hard" Sudoku problems.

The easy problem set averages about 846 milliseconds with the DLX implementation versus an average of about 27,316 milliseconds for the naïve implementation each across 50 runs. DLX is approximately 32 times faster than the naïve approach with the easy problem set.

The hard problem set averages about 1,536 milliseconds with DLX and over 2 hours with the naïve implementation. DLX is approximately 4,700 times faster than the naïve approach with the hard problem set.

REFERENCES

[1] B. Hanson, "Exact Cover Matrix," stolaf.edu [Online]. Available: https://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm [Accessed Nov. 29, 2019].

[2] R. Szymanski, "NaiveSudokuSolver.java," *github*. September 2014 [Online]. Available: https://github.com/rafalio/dancing-links-java/blob/master/src/dlx/NaiveSudokuSolver.java/. [Accessed Nov. 29, 2019].

[3] P. Norvig, "Solving Every Sudoku Puzzle," *Norvig.com*. [Online]. Available: http://norvig.com/sudoku.html/. [Accessed Nov. 29, 2019].