Exact Cover

Fangxin Lin (fl2571)
Xinxin Zhao (xz3061)

December 2021

Contents

1 Introduction 3

2 Exact Cover 3

2.1 Definition 3

2.2 Application - Sudoku 3

2.3 Application - N-Queens 3

3 Dancing Links and Algorithm X 4

3.1 Algorithm X 4

3.2 Dancing Links 4

4 Implementation and Performance 5

4.1 Persistent Adaption of Dancing Links 5

4.2 Sequential Solver 5

4.3 Performance - Sequential Solver 6

4.4 Parallel Solver for N-Queens 6

4.5 Performance - Parallel Solver 6

A Code Listing 7

1 Introduction

Exact Cover is an NP-complete problem and has many applications. To solve it efficiently, Knuth proposed Algorithm X in 2000, a DFS-based algorithm with a data structure named Dancing Links. We will implement a parallel solver for Exact Cover in Haskell based on Dancing Links and Algorithm X.

Since Sudoku and N-Queens can reduce to Exact Cover problem directly, we will implement and test on solvers for Sudoku and N-Queens problems.

2 Exact Cover

2.1 Definition

Given a matrix of 0s and 1s, find a set of rows containing exactly one 1 in each column. In the following example, all columns are satisfied if we choose rows 1, 4, and 5.

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

2.2 Application - Sudoku

To reduce a Sudoku puzzle to an Exact Cover problem, we need:

- Rows ($< 9^3$ in total): For every cell, try every possible number. Every row corresponds to a three-elements tuple of the cell's coordinates and the digit inside it.
- Columns ($4 \times 9 \times 9 = 324$ in total):
  - Position limit: Each cell can only contain one number.
  - Row limit: For each row, every number appears exactly once.
  - Column limit: For each row, every number appears exactly once.
  - Area limit: For each area (3×3), every number appears exactly once.

2.3 Application - N-Queens

To reduce the N-Queens problem to an Exact Cover problem, we need:

- Rows (n^2 in total): Every row corresponds to placing a queen to a position on the board.
- Columns ($n + n + 2(2n - 2) = 6n - 4$ in total):
  - Row limit: Each row can only place a queen.
  - Column limit: Each column can only place a queen.
  - Diagonal limit (don't need to satisfy all): Each diagonal can only place a queen.

3 Dancing Links and Algorithm X

3.1 Algorithm X

1. Find a column with a minimum number of 1s, attempt to remove each row R (e.g. 4$^{\text{th}}$ row in the left matrix) with 1 in that column.
2. Remove all columns with 1 in row R (1$^{\text{st}}$ and 4$^{\text{th}}$ columns in the left matrix).
3. Remove all rows with 1 in removed columns (2$^{\text{nd}}$, 4$^{\text{th}}$ and 6$^{\text{th}}$ rows in the left matrix).
4. Solve the sub-problems until the matrix becomes empty.

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

3.2 Dancing Links

In short, dancing links is a doubly orthogonal circular linked list. The advantage of this data structure is that it can remove and restore a row or column efficiently, and look up columns from a row or vice versa. Fig. 1 is a picture of a dancing links.

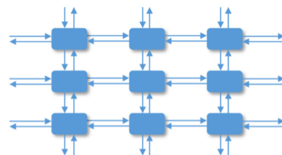


Figure 1: an illustration of a dancing link structure

4 Implementation and Performance

4.1 Persistent Adaption of Dancing Links

Though we can implement dancing links in the same way using `Data.Array` or `Data.Vector` with the same time complexity, but mutable data structure is not suitable for parallel programming. Thus we use `Data.Set` and `Data.Map` to build an immutable data structure to provide the same functionalities as dancing links, but at the cost of $O(\log n)$ overhead.

We define the following data structure, where `val::a` in `DLXRow` is meta-information relating to the actual problem.

```
type DLX a = M.Map Int (DLXColumn a)
type DLXColumn a = S.Set (DLXRow a)
data DLXRow a = DLXRow {
    columns :: S.Set Int,
    val :: a
} deriving (Show)

removeRow :: Ord a => DLX a -> DLXRow a -> DLX a
removeRow dlx row = foldl (flip $ M.adjust (S.delete row)) dlx (columns row)

removeColumn :: Ord a =>  DLX a -> Int -> DLX a
removeColumn dlx idx = M.delete idx $
    S.foldl removeRow dlx (M.findWithDefault S.empty idx dlx)
```

4.2 Sequential Solver

We implement two solvers: one is to find a valid solution while another is to find all solutions. Function `candidates` will select a column heuristically to satisfy next, and the solver will attempt to select each row in that columns and solve the sub-problems recursively.

```
candidates :: Ord a => DLX a -> [DLXRow a]
candidates dlx = S.toList $ minimumBy (comparing S.size) $ M.elems dlx

solve :: Ord a => DLX a -> Maybe [a]
solve dlx | M.null dlx = Just []
solve dlx = msum . map (\row -> fmap (val row :)
                       (solve (selectRow dlx row))) $ candidates dlx

solveAll :: Ord a => DLX a -> [[a]]
solveAll dlx | M.null dlx = [[]]
solveAll dlx = candidates dlx >>=
                 (\row -> map (val row :) (solveAll (selectRow dlx row)))
```

4.3 Performance - Sequential Solver

We test our Sudoku solver against 1,000 most difficult (17-clue) Sudoku problems, comparing with a very fast solver in Haskell Wiki and a solver from a past student of this class. As shown in table 1, our solver is comparable with the very fast solver, while our algorithm can generalize to other exact cover problems.

Table 1: Benchmark on 1,000 17-clue Sudoku problems[1]

	Our	Peter's Solver[2]	Very Fast Solver[3]
Time (sec)	5.79	3,320.00[4]	1.02

Because the runtime of our Sudoku solver is too fast to parallelize, so we will only parallelize with our N-Queens solver.

4.4 Parallel Solver for N-Queens

Our N-Queens solver aims to count the number of solutions for a given N. Since our solver is based on DFS, we can simply use `parMap` to parallelize the process of solving sub-problems. And to prevent producing too many sparks, we also add a parameter `level` to control the maximum depth of parallelism.

```haskell
solveRCCount :: Ord a => Int -> DLX a -> Int
solveRCCount c dlx | M.null dlx || fst (M.findMin dlx) > c = 1
solveRCCount c dlx = sum $ map (solveRCCount c . selectRow dlx) $ candidatesRC c dlx

solveRCCountPar :: Ord a => Int -> Int -> DLX a -> Int
solveRCCountPar _ c dlx | M.null dlx || fst (M.findMin dlx) > c = 1
solveRCCountPar level c dlx =
    let solver = if level == 0 then solveRCCount else solveRCCountPar (level - 1) in
        sum $ parMap rseq (solver c . selectRow dlx) (candidatesRC c dlx)
```

4.5 Performance - Parallel Solver

We test our parallel solver with different numbers of threads. According to fig. 2, the performance of parallelism is pretty close to the ideal situation regardless of the depth of parallelism. Thanks to Haskell and `Control.Parallel.Strategies`, we can conclude our parallelization is good enough by invoking the `parMap` without touching the underlying mechanism.

From fig. 3, the balance of parallelism is slightly better if increasing the depth of parallelism, but the runtime and overall performance does not improve notably.

[1] dataset: link, 5-run average, 4C8T 1.3GHz i7-1065G7

[2] `http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/sodoku.pdf`

[3] `https://wiki.haskell.org/Sudoku#Very_fast_Solver`

[4] In approximation.

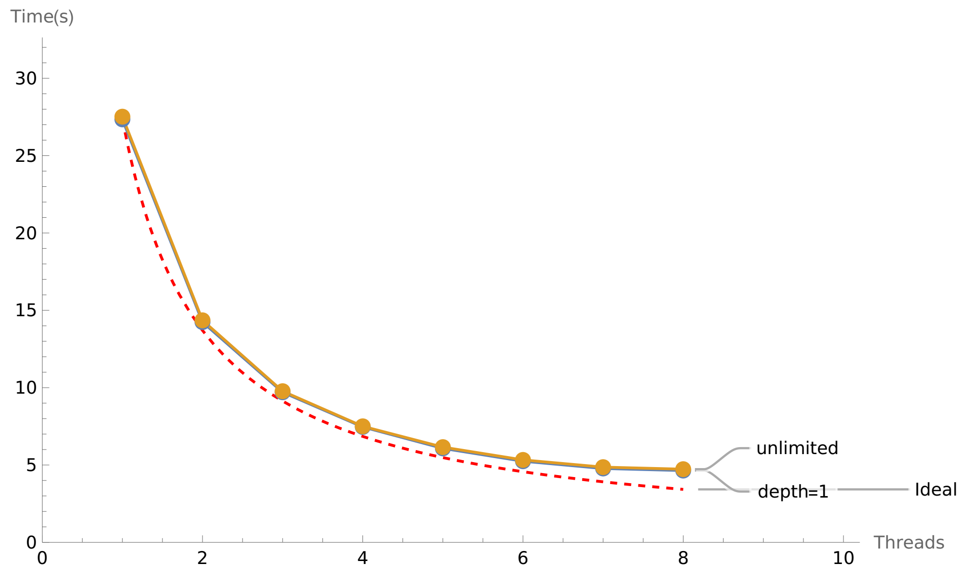
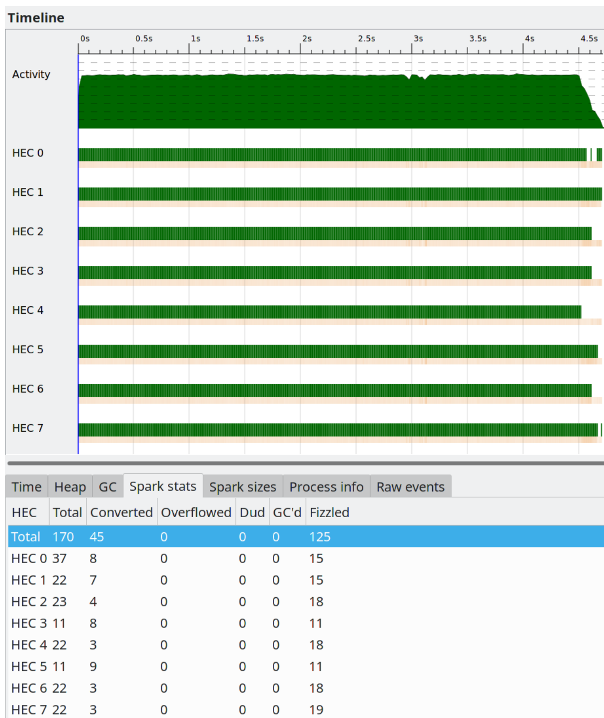
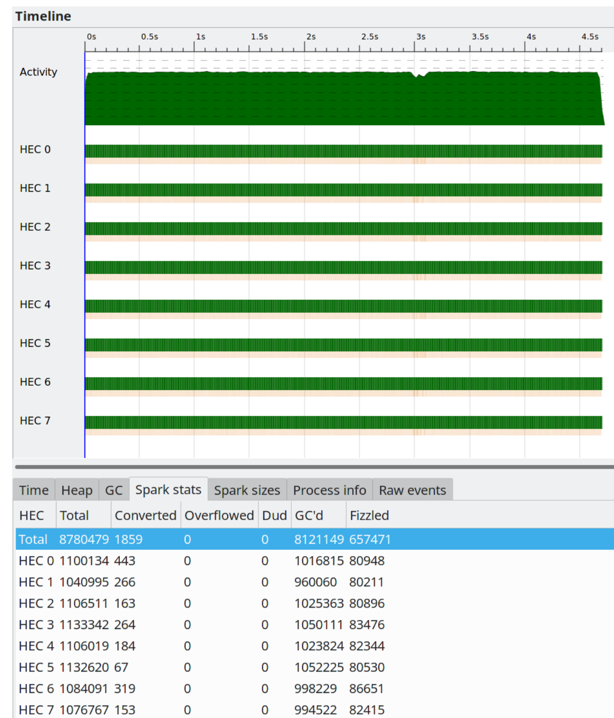


Figure 2: N-Queens(N=14),3.60GHz 8C16T i9-9900KF,5-run average



(a) limit parallel depth to 1



(b) unlimited parallel depth

Figure 3: ThreadScope

A Code Listing

You can visit `https://github.com/zerolfx/exact-cover` to see the updated code of this project.

ExactCover.hs

```haskell
module ExactCover where

import qualified Data.Map as M
import qualified Data.Set as S
import Data.List (minimumBy)
import Control.Monad (msum)
import Data.Ord (comparing)
import Data.Function (on)


type DLX a = M.Map Int (DLXColumn a)
type DLXColumn a = S.Set (DLXRow a)
data DLXRow a = DLXRow {
    columns :: S.Set Int,
    val :: a
} deriving (Show)

instance Ord a => Ord (DLXRow a) where
    compare = comparing val

instance Eq a => Eq (DLXRow a) where
    (==) = (==) `on` val


removeRow :: Ord a => DLX a -> DLXRow a -> DLX a
removeRow dlx row = foldl (flip $ M.adjust (S.delete row)) dlx (columns row)

removeColumn :: Ord a =>  DLX a -> Int -> DLX a
removeColumn dlx idx = M.delete idx $ S.foldl removeRow dlx (M.findWithDefault S.empty idx

selectRow :: Ord a => DLX a -> DLXRow a -> DLX a
selectRow dlx row = foldl removeColumn dlx (columns row)

candidates :: Ord a => DLX a -> [DLXRow a]
candidates dlx = S.toList $ minimumBy (comparing S.size) $ M.elems dlx

solve :: Ord a => DLX a -> Maybe [a]
solve dlx | M.null dlx = Just []
solve dlx = msum . map (\row -> fmap (val row :) (solve (selectRow dlx row))) $ candidates

solveAll :: Ord a => DLX a -> [[a]]
solveAll dlx | M.null dlx = [[]]
solveAll dlx = candidates dlx >>= (\row -> map (val row :) (solveAll (selectRow dlx row)))

buildFromRow :: Ord a => DLXRow a -> DLX a
buildFromRow row = M.fromList [(i, S.singleton row) | i <- S.toList (columns row)]
```

```
buildFromRows :: Ord a => [DLXRow a] -> DLX a
buildFromRows = foldl (\m row -> M.unionWith S.union m (buildFromRow row)) M.empty
```

Sudoku.hs

```
module Sudoku (sudoku, sudokuAll, sudokuAllPar) where

import ExactCover
import qualified Data.Set as S
import qualified Data.Map as M
import Data.Char (ord, chr)
import Control.Parallel.Strategies (using, parList, rseq)

choices :: String -> [(Int, Int, Int)]
choices s =
    [(x, y, ord c - ord '1') | (i, c) <- zip [0..] s, c /= '.', let (x, y) = (i `div` 9, i
    [(x, y, v) | (i, c) <- zip [0..] s, c == '.', let (x, y) = (i `div` 9, i `mod` 9), v <

areaPos :: (Int, Int) -> Int
areaPos (x, y) = 3 * (x `div` 3) + (y `div` 3)

genRow :: (Int, Int, Int) -> DLXRow (Int, Int, Int)
genRow (x, y, v) = DLXRow (S.fromList [x * 9 + v, 1000 + y * 9 + v, 2000 + x * 9 + y, 3000

genSolution :: [(Int, Int, Int)] -> String
genSolution pos = let m = M.fromList [((x, y), v) | (x, y, v) <- pos] in
    map (\i -> chr (ord '1' + m M.! (i `div` 9, i `mod` 9))) [0..81 - 1]


sudoku :: String -> Maybe String
sudoku s = genSolution <$> solve (buildFromRows (map genRow (choices s)))

sudokuAll :: String -> [String]
sudokuAll s = genSolution <$> solveAll (buildFromRows (map genRow (choices s)))


solveAllPar :: Ord a => DLX a -> [[a]]
solveAllPar dlx | M.null dlx = [[]]
solveAllPar dlx = candidates dlx >>= (\row -> map (val row :) (solveAllPar (selectRow dlx

sudokuAllPar :: String -> [String]
sudokuAllPar s = genSolution <$> solveAllPar (buildFromRows (map genRow (choices s)))
```

NQueens.hs

```haskell
module NQueens (nqueens, nqueensPar) where
import ExactCover
import qualified Data.Set as S
import qualified Data.Map as M
import Data.List (minimumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap, rseq)

candidatesRC :: Ord a => Int -> DLX a -> [DLXRow a]
candidatesRC c dlx = S.toList $ minimumBy (comparing S.size) $ map snd $ filter ((<= c) .

solveRCCount :: Ord a => Int -> DLX a -> Int
solveRCCount c dlx | M.null dlx || fst (M.findMin dlx) > c = 1
solveRCCount c dlx = sum $ map (solveRCCount c . selectRow dlx) $ candidatesRC c dlx

solveRCCountPar :: Ord a => Int -> Int -> DLX a -> Int
solveRCCountPar _ c dlx | M.null dlx || fst (M.findMin dlx) > c = 1
solveRCCountPar level c dlx =
    let solver = if level == 0 then solveRCCount else solveRCCountPar (level - 1) in
        sum $ parMap rseq (solver c . selectRow dlx) (candidatesRC c dlx)

nqueens :: Int -> Int
nqueens n = solveRCCount 1500 $ buildFromRows $ [
    DLXRow (S.fromList [x, 1000 + y, 2000 + x + y, 3000 + x - y]) (x, y)
    | x <- [0 .. n - 1], y <- [0 .. n - 1]]

nqueensPar :: Int -> Int -> Int
nqueensPar level n = solveRCCountPar level 1500 $ buildFromRows $ [
    DLXRow (S.fromList [x, 1000 + y, 2000 + x + y, 3000 + x - y]) (x, y)
    | x <- [0 .. n - 1], y <- [0 .. n - 1]]
```

Main.hs

```haskell
module Main where

import Sudoku
import NQueens

main :: IO ()
main = do
    let s = "4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4..
    print $ sudoku s
    print $ sudokuAll s
    print $ sudokuAllPar s
    print $ nqueens 13
    print $ nqueensPar 10000000 14
```