“SUDOKO”

PROJECT REPORT

Submitted For CAL in B.Tech Data Structures And Algorithms CSE2003

By

AASTHA SOOD 16BCE1104
ANUSHKA GULATI 16BEC1250
AVISHI BHAL 16BCE1294
PRERNA AGARWAL 16BCE1350

slot: B1

Name Of Faculty: Dr. SV. NAGARAJ

(SCHOOL OF COMPUTING SCIENCE AND ENGINEERING-SCSE)

VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
www.vit.ac.in
Vellore ▪ Chennai
CHENNAI CAMPUS
Vandalur - Kelambakkam Road, Chennai - 600127

CERTIFICATE

This is to certify that the project work entitled "**SUDOKU**" that is being submitted by "AASTHA SOOD 16BCE1104, ANUSHKA GULATI 16BEC1250, AVISHI BHAL 16BCE1294, PRERNA AGARWAL 16BCE1350" for cal in B.Tech Data Structures And Algorithm (CSE2003) is a record of bonafide work done under my supervision. The contents of this Project work have not been submitted for any other CAL course.

Place: Chennai
Date: 28th April 2017

Signature of Students:

AASTHA SOOD

ANUSHKA GULATI

AVISHI BHAL

PRERNA AGARWAL

Signature of Faculty: DR. SV. NAGARAJ

ACKNOWLEDGEMENTS

Firstly, we would like to thank Dr SV. Nagaraj, professor, Data Stuctures And Algorithms(CSE2003), Winter semester 2016-17 for his guidance and her resources which made this PBL project possible. Finally, we would acknowledge our deemed university, VIT University, Chennai Campus for providing us with the opportunity and facilities which ensured the project's completion.

AASTHA SOOD -16BCE1104
ANUSHKA GULATI -16BEC1250
AVISHI BHAL -16BCE1294
PRERNA AGARWAL -16BCE1350

ABSTRACT

The class of Sudoku puzzles consists of a partially completed row-column grid of cells partitioned into N regions each of size N cells, to be filled in using a prescribed set of N distinct symbols (typically the numbers {1, ..., N}), so that each row, column and region contains exactly one of each element of the set. For classic Sudoku, N=9 and the regions are 3x3 squares (called blocks or boxes). In this project, we study and implement some algorithms for solving Sudoku puzzles.

THE ALGORITHMS WE HAVE IMPLEMENTED

The algorithms we will be using and describing are:

Backtracking

Brute- force

Constraint programming

Rule-based

Boltzmann machines

The problem of solving $n^2 * n^2$ sudoku puzzles is NP-complete. While being theoretically interesting as a result it has also motivated research into heuristics, resulting in a wide range of available solving methods. Out of which are backtrack, brute-force, rule-based, constraint programming, and Boltzmann machines.

BACKTRACKING

Backtrack is the most basic strategy for solving sudoku for computer algorithms. This algorithm is like a brute-force method which tries different numbers, and if it fails it backtracks and tries a different number. This means that the algorithm does an exhaustive search to find a solution, which means that a solution is guaranteed to be found if enough time is provided. Even though this algorithm runs in exponential time, it is plausible to try it since it is widely thought that no polynomial time algorithms exists for NP-complete problems such as sudoku.

Pseudo- code

```
Puzzle Backtrack(puzzle)
   (x,y) = findSquare(puzzle) //Find square with least candidates
   for i in puzzle[y][x].possibilities() //Loop through possible candidates
            puzzle[y][x] = i //Assign guess
puzzle' = Backtrack(puzzle) //Recursion step
if(isValidAndComplete(puzzle')) //Check if guess lead to solution
        return puzzle'
//else continue with the guessing
        return null //No solution was found
```

Time complexity

Time complexity of the backtracking algorithm for sudoku is T(n) = O(n2).

{Because it has iterations with two nested FOR loops in the functions for assigning values to the empty grids. }

Proof of Correctness

We define a series of functions for specific tasks. Firstly we define a function findunassignedlocation to find the empty spaces of the sudoku grid. The iterations are:

- initializing the for loop i=0. It runs for all the values of i less than N. The nested loop with initial value j=0 runs N times for each i.

- while the loop runs, search operation for empty square takes place and if found, function returns true.

We define usedinbox,usedinrow and usedincolumn function with one for loop each to search among the n elements of that particular box,row and column, whether a number has already been used or not. The function issafe returns true or false based on the upper three functions.

We end the program by a conventional print function, which again uses two for loops to traverse through both rows and columns to print the values.

All these functions are then used in the main function which takes an input.

BRUTE- FORCE

Brute-force search or exhaustive search, also known as “generate and test”, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

Brute force is a straightforward approach to solve a problem based on the problem’s statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small–size instances of a problem.

The brute force algorithm requires n-1 multiplications.

The recursive algorithm for the same problem, based on the observation that an = an/2 * an/2 requires Θ(log(n)) operations.

Pseudo- code

```
def solve(s):
    try:
        i  = s.index(0)
    except ValueError:
        # No empty cell left: solution found
        return s
    c = [s[j] for j in range(81)
         if not ((i-j)%9 * (i//9^j//9) * (i//27^j//27 | (i%9//3^j%9//3)))]
    for v in range(1, 10):
        if v not in c:
            r = solve(s[:i]+[v]+s[i+1:])
            if r is not None:
                return r
```

Time complexity

The recursive algorithm for the same problem, based on the observation that an = an/2 * an/2 requires Θ(log(n)) operations.

Proof of Correctness

The brute- force method uses the same functions but the only difference is that unlike backtracking it checks each and every possibilities for each and every void and While backtracking discards many options, chooses one and goes back if it fails, brute-force doesn't go back.

Hence the loops go like:

- Initialization: i=0, j=0

- Process: checks all the j<n possibilities for i<m spaces. Takes nm time.

- Termination: the loops terminate after checking each value till the solution is found.

CONSTRAINT PROGRAMMING

A Sudoku may also be modelled as a constraint problem. In his paper Sudoku as a Constraint Problem, Helmut Simonis describes many reasoning algorithms based on constraints which can be applied to model and solve problems. Some constraint solvers include a method to model and solve Sudokus, and a program may require less than 100 lines of code to solve a simple Sudoku. If the code employs a strong reasoning algorithm, incorporating backtracking is only needed for the most difficult Sudokus. An algorithm combining a constraint-model-based algorithm with backtracking would have the advantage of fast solving time, and the ability to solve all sudokus.

Pseudo-code

Reset all the tags
    Read the puzzle
    Logical Solve Loop
        Check validity in row – set flag if valid
        Check validity in column – set flag if valid
        Check validity in block – set flag if valid
        Set flag if all three flags are set
        Check other numbers to make sure it's a singleton
            If it is not a singleton then skip otherwise assign the value to cell
        Make sure we actually made an assignment if not break out and backtrack
    Backtrack Solve Loop
        Check for values serially column by column and then row
        If plausible number is found, assign it and set proper tags
        Call backtrack again
    If no error is found and this is the last cell – return true and exit
    If error is found (no further values and not the last cell) then go to last cell
    Reset the proper tags
    Continue in the backtrack with next value

Time complexity

Time complexity of the backtracking algorithm for sudoku is T(n) = O(n2).

{Because it has iterations with two nested FOR loops in the functions for assigning values to the empty grids. }

Proof of Correctness

We define a series of functions for specific tasks. Firstly we define a function findunassignedlocation to find the empty spaces of the sudoku grid. The iterations are:

- initializing the for loop i=0. It runs for all the values of i less than N. The nested loop with initial value j=0 runs N times for each i.

- while the loop runs, search operation for empty square takes place and if found, function returns true.

We define usedinbox,usedinrow and usedincolumn function with one for loop each to search among the n elements of that particular box,row and column, whether a number has already been used or not. The function issafe returns true or false based on the upper three functions.

We end the program by a conventional print function, which again uses two for loops to traverse through both rows and columns to print the values.

All these functions are then used in the main function which takes an input.

Special cases – peter norvig (python), (& Emmanuel Delaborde-haskell)

PETER NORVIG used the concepts of constraint programming and search.

```
import time, random
def solve_all(grids, name='', showif=0.0):
    """Attempt to solve a sequence of grids. Report results.
    When showif is a number of seconds, display puzzles that take longer.
    When showif is None, don't display any puzzles."""
    def time_solve(grid):
        start = time.clock()
        values = solve(grid)
        t = time.clock()-start
        ## Display puzzles that take long enough
        if showif is not None and t > showif:
            display(grid_values(grid))
            if values: display(values)
            print '(%.2f seconds)\n' % t
        return (t, solved(values))
    times, results = zip(*[time_solve(grid) for grid in grids])
    N = len(grids)
    if N > 1:
        print "Solved %d of %d %s puzzles (avg %.2f secs (%d Hz), max %.2f secs)." % (
            sum(results), N, name, sum(times)/N, N/sum(times), max(times))
def solved(values):
    "A puzzle is solved if each unit is a permutation of the digits 1 to 9."
    def unitsolved(unit): return set(values[s] for s in unit) == set(digits)
    return values is not False and all(unitsolved(unit) for unit in unitlist)
def from_file(filename, sep='\n'):
    "Parse a file into a list of strings, separated by sep."
    return file(filename).read().strip().split(sep)
def random_puzzle(N=17):
    """Make a random puzzle with N or more assignments. Restart on contradictions.
    Note the resulting puzzle is not guaranteed to be solvable, but empirically
    about 99.8% of them are solvable. Some have multiple solutions."""
    values = dict((s, digits) for s in squares)
    for s in shuffled(squares):
        if not assign(values, s, random.choice(values[s])):
            break
        ds = [values[s] for s in squares if len(values[s]) == 1]
        if len(ds) >= N and len(set(ds)) >= 8:
            return ''.join(values[s] if len(values[s])==1 else '.' for s in squares)
    return random_puzzle(N) ## Give up and make a new puzzle
def shuffled(seq):
    "Return a randomly shuffled copy of the input sequence."
    seq = list(seq)
    random.shuffle(seq)
```

```
    return seq
grid1  =
'003020600900305001001806400008102900700000008006708200002609500800203009005010300'
grid2  = '4.....8.5.3.........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'
hard1  = '.....6....59.....82....8....45........3........6..3.54...325..6..................'

if __name__ == '__main__':
    test()
    solve_all(from_file("easy50.txt", '========'), "easy", None)
    solve_all(from_file("top95.txt"), "hard", None)
    solve_all(from_file("hardest.txt"), "hardest", None)
    solve_all([random_puzzle() for _ in range(99)], "random", 100.0)
```

HASKELL

rows = "ABCDEFGHI"

```
cols = "123456789"
digits = "123456789"

cross :: String -> String -> [String]
cross rows cols = [ r:c:[] | r <- rows, c <- cols ]

squares :: [Square]
squares = cross rows cols  -- ["A1","A2","A3",...]

unitlist :: [Unit]
unitlist = [ cross rows [c] | c <- cols ] ++
           [ cross [r] cols | r <- rows ] ++
           [ cross rs cs | rs <- ["ABC","DEF","GHI"], cs <-
["123","456","789"]]

units :: M.Map Square [Unit]
units = M.fromList [ (s, [ u | u <- unitlist, elem s u ]) | s <-
squares ]

peers :: M.Map Square [Square]
peers = M.fromList [ (s, set [[ p | p <- e, p /= s ] | e <- lookup s
units ]) | s <- squares ]
  where set = nub . concat

-----------------------------------------------
-- Wrapper around M.lookup used in list comprehensions

lookup :: (Ord a, Show a) => a -> M.Map a b -> b
lookup k v = case M.lookup k v of
               Just x -> x
               Nothing -> error $ "Error : key " ++ show k ++ " not
in map !"

-- lookup k m = fromJust . M.lookup k m
```

```
------------------------------------------------
-- Parsing a grid into a Map

parsegrid     :: String -> Maybe Grid
parsegrid g    = do regularGrid g
                    foldM assign allPossibilities (zip squares g)

  where  allPossibilities :: Grid
         allPossibilities = M.fromList [ (s,digits) | s <- squares ]
         regularGrid   :: String -> Maybe String
         regularGrid g  = if all (\c -> (elem c "0.-123456789")) g
                          then (Just g)
                          else Nothing

------------------------------------------------
-- Propagating Constraints

assign        :: Grid -> (Square, Digit) -> Maybe Grid
assign g (s,d) = if (elem d digits) then do -- check that we are
assigning a digit and not a '.'
                   let toDump = delete d (lookup s g)
                   res <- foldM eliminate g (zip (repeat s) toDump)
                   return res
                 else return g

eliminate     ::  Grid -> (Square, Digit) -> Maybe Grid
eliminate g (s,d) = let cell = lookup s g in
                   if not (elem d cell) then return g -- already
eliminated
                   -- else d is deleted from s' values
                     else do let newCell = delete d cell
                                 newV = M.insert s newCell g --
                             newV2 <- case length newCell of
                                        -- contradiction :
Nothing terminates the computation
                                        0 -> Nothing
                                        -- if there is only one
value (d2) left in square, remove it from peers
                                        1 -> do let peersOfS =
[ s' | s' <- lookup s peers ]
                                                res <- foldM
eliminate newV (zip peersOfS (cycle newCell))
                                                return res
                                        -- else : return the new
grid
                                        _ -> return newV
                             -- Now check the places where d
appears in the units of s
                             let dPlaces = [ s' | u <- lookup s
units, s' <- u, elem d (lookup s' newV2) ]
                             case length dPlaces of
                               0 -> Nothing
```

```
                        -- d can only be in one place in
unit; assign it there
                        1 -> assign newV2 (head dPlaces, d)
                        _ -> return newV2


-------------------------------------------------
-- Search

search        :: Maybe Grid -> Maybe Grid
search Nothing  = Nothing
search (Just g) = if all (\xs -> length xs == 1) [ lookup s g | s <-
squares ]
                  then (Just g) -- solved
                  else do let (_,s) = minimum [ (length (lookup s
g),s) | s <- squares, length (lookup s g) > 1 ]
                              g' = g -- copie of g
                          foldl' some Nothing [ search (assign
g' (s,d)) | d <- lookup s g ]
  where some Nothing Nothing  = Nothing
        some Nothing (Just g) = (Just g)
        some (Just g) _ = (Just g)


-------------------------------------------------
-- Display solved grid

printGrid :: Grid -> IO ()
printGrid = putStrLn . gridToString

gridToString :: Grid -> String
gridToString g = let l0= map snd (M.toList g)                --
[("1537"),("4"),...]
                   l1 = (map (\s -> " " ++ s ++ " ")) l0    -- ["
1 "," 2 ",...]
                   l2 = (map concat . sublist 3) l1         -- ["
1  2  3 "," 4  5  6 ",...]
                   l3 = (sublist 3) l2                      -- [["
1  2  3 "," 4  5  6 "," 7  8  9 "],...]
                   l4 = (map (concat . intersperse "|")) l3 -- ["
1  2  3 | 4  5  6 | 7  8  9 ",...]
                   l5 = (concat . intersperse [line] . sublist 3) l4
                 in unlines l5
  where sublist n [] = []
        sublist n xs = take n xs : sublist n (drop n xs)
        line = hyphens ++ "+" ++ hyphens ++ "+" ++ hyphens
        hyphens = take 9 (repeat '-')

-------------------------------------------------

main :: IO ()
main = do h <- openFile "top95.txt" ReadMode
```

```
    grids <- hGetContents h
    let solved = mapMaybe (search . parsegrid) (lines grids)
    mapM_ printGrid solved
    hClose h
```

CONCLUSION:

Thus, 3 algorithms were implemented. While rule-based and Boltzmann machine were just researched upon. Also, it was found that the most efficient algorithm was backtracking algorithm.