Indian Institute
Of
Information Technology,
Allahabad

Solving SUDOKU
Using
Different Techniques

Abhishek Pasi	ICM2015002
Ayush Agnihotri	IIM2015004
Nidheesh Pandey	IIM2015501
Shreyansh Gupta	IIM2015001
Vishal Kumar Singh	IIT2015141

Under supervision
of
Dr. K P Singh

Contents

Solving SUDOKU using different techniques

1 ABOUT SUDOKU PUZZLE

Sudoku is a logic-based number placement puzzle.The puzzle area consists of a 9x9 grid divided into nine 3x3 grids.

The constraints for the solution being :-

(i) each row should contain 1-9 numbers.
(ii) each column should contain 1-9 numbers.
(iii) each 3x3 grid should contain all 1-9 numbers

2 METHODS USED

2.1 BRUTEFORCE (Exhaustive Search)

2.1.1 Brute force approaches, specially those which ask us to find all permutation of certain variables till a solution is found require backtracking.

Backtracking definition – "to go back along a path that you have just followed: We went the wrong way and had to backtrack till we got to the right turning". (Source - Cambridge Dictionary)

Backtracking is an algorithmic technique that relies on trying every possible move in order to find correct solution for a constrained problem.

Backtracking approach is implemented using recursion.

The basic idea of all backtracking algorithms is same.

(i) We proceed with one of the many possible moves and try to solve the problem recursively.
(ii) At any stage we assume steps taken by our algorithm is correct.
(iii) At any step of our Algorithm , if it is found that , our problem can not be solved , no matter what choice we make, we go to previous stage and undo the move taken (since, it was a wrong move) and try next available choice at that stage.
(iv) The steps **(i)**, **(ii)**, **(iii)** are repeated until the problem is solved.

2.1.2 Solving SUDOKU using backtracking

2.1.2.1 OUR ALGORITHM

(i) Take input as 9x9 grid from file puzzle.txt (should be a vaid sudoku puzzle).
Specify blank squares with zero.
(ii) Find the first blank field (first zero).
(iii) Call the solver (solve(grid,x,y)) with the first blank as an argument.
(iv) Find available moves by checking an x^{th} row, y^{th} column and a 3x3 grid of the target blank. (performed using set() object)
(v) Try the first move and identify the next blank to be filled.
(vi) Fill the next blank with one of the available moves.
(vii) If at any stage it is found that no move can solve the puzzle, go to previous blank and try the next available move in the set (backtracking)
(viii) Repeat steps **(iv)**, **(v)**, **(vi)**, **(vii)** .

(ix) The recursion will exit when the last blank is filled successfully. The solution is finally found.

2.2 BACKTRACKING (Constraint Satisfaction Problem)

2.2.1 A typical Constraint Satisfaction Problem (CSP) consists of :-

(a) Set of variables X = {x1, x2, x3, ..., xn}
(b) A finite domain for each variable . A Domain specifies a set of values a variable can take.
(c) A set of constraints which defines set of values that variables can take simultaneously.

2.2.2 A Sudoku puzzle can be modelled as a CSP, :-

(a) set of variables X = {set of empty places in an unsolved puzzle}
(b) A finite domain for each variable. The Set of feasible values for a variable.
(c) constraints are already specified for sudoku in puzzle introduction :-
  (i) each row should contain 1-9 numbers.
  (ii) each column should contain 1-9 numbers.
  (iii) each 3x3 grid should contain all 1-9 numbers.

Any CSP can be solved using Naive backtracking like done in first algorithms (Exhaustive Search)

2.2.3 Search Methods and Heuristics for solving a CSP

(in short)
(1) Variable ordering
(2) Value ordering

2.2.3.1 Variable ordering .

Decision to assign one variable before other two approaches :-
(1) ***static*** :- pre defined order of assigning values to a variable based on properties of problem.
(2) ***dynamic*** :- order of assigning values to a variable is based on information gathered in course of search.

2.2.4 OUR ALGORITHM

Input

(i) Take input as 9x9 grid from file puzzle.txt (should be a vaid sudoku puzzle).
Specify blank squares with zero.

Variable Ordering :

(ii) Find the domain of variables (blanks) one by one and keep pushing them to a list. Sort the list in increasing order of preference function.
(iii) This way we will be assigning variables with starting from smallest domain first.

Solving the puzzle :

(iv) Call the solver (solve(grid,l,0)) with starting from the first variable with the smallest domain.
(v) Find available moves by checking x^{th} row, y^{th} column and a 3x3 grid of the target blank. (performed using set() object)
(vi) Try the first move and identify the next blank to be filled.
(vii) Fill the next blank with one of the available moves.
(viii) If at any stage it is found that no move can solve the puzzle, go to previous blank and try the next available move in the set (backtracking)
(ix) Repeat steps **(iv)**, **(v)**, **(vi)**, **(vii)**.
(x) The recursion will exit when the last blank (max domain) is filled successfully. The solution is finally found.

2.3 Genetic Algorithm

2.3.1 Genetic Algorithms as the names suggest are inspired genetics.Genetic algorithms are quite effective in solving problems for which we already know some solution.GA optimises a solution by increasing its fitness level after each generation.

Steps involved in a typical genetic algorithm :-

Startoff

(i) Begin by creating a set of possible solutions to the problem. Each solution is known as an 'Individual' and a set of Individuals is knows as population.
(ii) Now that we have a generation we need to 'evolve'.

Evolution

(iii) 'Evolution' is a process of improving a generation by increasing its 'fitness'. We can achieve this by designing a good fitness function (problem specific). To produce next-generation form a population we use mutation and crossover operators.
(iv) Mutation is changing an individual, the crossover is mixing to individuals by choosing most suitable cross-over point.
(v) We chose top individuals from a population as a parent for next generation.Few inferior individuals are also selected as a parent to promote genetic diversity.
(vi) After choosing parents for next generations, we mutate a few among them to promote a change between next and present generations. Parents are randomly selected and crossed to generate new individuals.
(vii) Fitness of new generations must be greater than previous ones.

Produce generations till you find high-quality solutions.

2.3.2 OUR ALGORITHM

Our algorithm is same as mentioned above.

What matters is the choice of fitness, crossover and mutation functions. Values of random select, crossover rate, and mutation rate are too important to ignore.

Currently our Algorithm is not able to solve the puzzle using genetic algorithm, every time it gets stuck on local minima .We have not been able to give it proper thought why it happens (due to lack of time) . At present the problem may be with our fitness function (or crossover rate, mutation rate).
We promise that will try to improve our algorithm in coming days.

Input :
(i) Take input as 9x9 grid from file puzzle.txt (should be a vaid sudoku puzzle). Specify blank squares with zero.
Start off:
(ii) Firstly we create a initial set of random solutions i.e. population of size 'n' by calling function (population(grid, n)), which in turn calls function (individual(grid)).
(iii) The function (individual) fills the blank positions in subgrids with available values in its domain in random order(such that no element is repeated in subgrid).
Evolve :
(iv) Once we have our initial population, then we calculate the fitness of each individual in population.
(v) The fitness function in our algorithm is based on the number of duplicate values for a given position(x,y) in the row and column to which it belongs i.e.(sum of number of duplicate value from grid[x][1..n] and grid[1...n][y]). So lower the value of fitness function closer is the individual to final solution.

(vi) Then we mutate the selected parents by calling the function(mutate(parent, grid)). In our algorithm mutation process is done by swapping the non-fixed positions within the subgrid.

(vii) After getting mutated solution, we do crossover by selecting father and mother from parent set. After this we select a random crossover point (between 0 to 8) which gives us a child. Crossover among parents continues until we get a population of size 'n'.

(viii) The evolution process continues until we get a individual whose fitness is '0' (i.e. it is a solution to our input puzzle).

3 ANALYSIS :

Algorithms for Brute-force and CSP were run five times for each puzzle.
Values for mean and standard deviation were noted in table given below.
Full result with input and output puzzle is included in a file "**results.txt**".
To run code :

$ **python3 SudokuSolver.py times algorithm**
times = Number of times you need to run the algorithm.
algorithm = **1**-For Bruteforce, **2**-For Constraint Satisfaction Problem, **3**-For Genetic Algorithm

S No.	Category	Approach	Avg Time	Standard Deviation
1	Easy 1	Brute Force	0.0070542	2.19E-07
2	Easy 1	Constraint Optimisation	0.0129594	4.29E-07
3	Easy 2	Brute Force	2.1110646	0.0002734023
4	Easy 2	Constraint Optimisation	0.0028972	1.77E-07
5	Easy 3	Brute Force	0.005672	1.48E-07
6	Easy 3	Constraint Optimisation	0.0097804	3.61E-05
7	Easy 4	Brute Force	0.0020938	9.72E-08
8	Easy 4	Constraint Optimisation	0.0018682	2.83E-09
9	Easy 5	Brute Force	0.0007548	1.26E-09
10	Easy 5	Constraint Optimisation	0.0032086	1.06E-07
11	Easy 6	Brute Force	0.0012786	4.05E-10
12	Easy 6	Constraint Optimisation	0.002235	3.85E-09
13	Medium 1	Brute Force	1.2922906	0.0002071376
14	Medium 1	Constraint Optimisation	0.3687512	5.26E-06
15	Medium 2	Brute Force	0.8748586	3.70E-05
16	Medium 2	Constraint Optimisation	2.0253004	0.0001784205
17	Medium 3	Brute Force	0.0142868	8.25E-09
18	Medium 3	Constraint Optimisation	0.0497016	1.99E-07
19	Medium 4	Brute Force	0.007692	4.73E-07
20	Medium 4	Constraint Optimisation	0.0040312	1.06E-07
21	Medium 5	Brute Force	0.0006704	1.53E-10
22	Medium 5	Constraint Optimisation	0.0067952	6.09E-07
23	Medium 6	Brute Force	0.017174	3.36E-09
24	Medium 6	Constraint Optimisation	0.0032122	3.33E-09
25	Hard 1	Brute Force	0.9864058	0.0003355001
26	Hard 1	Constraint Optimisation	0.0584414	2.33E-07
27	Hard 2	Brute Force	2.3184416	0.0002873917
28	Hard 2	Constraint Optimisation	0.5693892	8.66E-05
29	Hard 3	Brute Force	1.904462	0.0005580804
30	Hard 3	Constraint Optimisation	0.05788	3.37E-06
31	Hard 4	Brute Force	0.001629	4.92E-10
32	Hard 4	Constraint Optimisation	0.0067274	7.07E-06
33	Hard 5	Brute Force	0.0755728	2.95E-06
34	Hard 5	Constraint Optimisation	0.1195058	4.72E-06
35	Hard 6	Brute Force	0.0070852	6.91E-07
36	Hard 6	Constraint Optimisation	0.0246638	5.31E-06
37	World's hardest(AI Escagot)	Brute Force	0.1538032	8.86E-06
38	World's hardest(AI Escagot)	Constraint Optimisation	1.1836058	0.000739928

Table 1: ***Mean and S.D. values for five runs of each algorithm on given puzzle.***

(I) From the above table it can be clearly seen that Constraint Satisfaction approach works best.
(II) In some cases it can be observed that brute force approach performs better than CSP approach. This

is generally the case when time for computing variable order(our heuristic) is high in comparison to actually solving the puzzle.
In many cases our heuristic for variable ordering is not suited for the given puzzle. In such cases brute force algo performs slightly better than CSP.

(III) It is also observed that time for solving puzzle does not directly depend on hardness of puzzle. A few easy puzzles took more time to solve than AL escargot Sudoku puzzle(one of the hardest sudoku puzzles ever created by Finnish mathematician 'Arto Inkala').

(IV) Altogether, We found backtracking and CSP are quite good for solving sudoku puzzle.

(V) Our attempt to solve sudoku using Genetic Algorithm was unsuccessful. **Increase in fitness quality from generation to generation is quite slow and gets stuck on local minima.** Also , creating many generations takes time . In our opinion, solving sudoku using genetic algorithm is quite challenging . We will try to solve it in coming days.

3.1 Time and Space complexity

3.1.1 Bruteforce

Time	:	$\mathbf{O}(\mathrm{N}\mathrm{x}b^m)$	i.e. exponential in max depth of state space .
Space	:	$\mathbf{O}(\mathrm{bm})$	i.e. linear in max depth of state-space.

b : Maximum branching factor.

In this case, Maximum domain of a variable(blank).

m : Maximum depth of state-space.

3.1.2 Constraint Satisfaction Problem

Same as bruteforce. However, most of the time value of '**m**' is less.

3.2 Number of nodes explored :

Number of nodes explored can be easily calculated by counting number of times we change entries of Sudoku puzzle.

Maximum nujmber of nodes explored = $\mathbf{O}(b^m)$

b : Maximum branching factor.

m : Maximum depth of state-space.