CSI 5165 Combinatorial Algorithms
Assignment 2
Wei Li 0300113733 report date: 2/23/2021

1 Environment

System: Windows 10
Hardware: Intel core I7 9700; 8+8 GB speed 2667;
Language: Python 3.8.3

2 Question 1

well done!

SUDOKU is a placement puzzle in which symbols from 1 to 9 are placed in cells of a 9 × 9 grid made up of nine 3×3 subgrids, called regions. The grid is partially filed with some symbols (the “givens”). The grid must be so that each row, column and region contains exactly one instance of each symbol.
design two backtracking algorithms:

- first algorithm fill the first available table position
- second algorithm fill the table position with smallest number of possible number
- Write a pseudocode for a backtracking algorithm that solves SUDOKU.
- Implement your algorithm and test the given instances (33 test cases provided).
- Output: input grid, solution grid, statistics of algorithm (total number backtracking nodes and running time)
- Displaying a table with the statistics for each algorithm (total number backtracking nodes and running time)

2.1 convention

2.1.1 index

The rows and columns index start from 0 and end with 8 from left to right; The regions(also referred as block in the implementation) are index from 0 to 8, the order is from left to right, then to the first regional column in next regional row.

2.1.2 filling value

Number ”0” represents unoccupied position. Fillable number is natural number from ”1” to ”9”.

2.1.3 used set

The problem and the solution are both 9*9 matrix, denoted as $X_{row,col}$. We define three used sets, which are set of number used along a row/ column or inside a region.

- A_{row}: number used along *row*
- B_{col}:number used along *col*
- $C_{row,col}$: number used inside the region a position $\{row, col\}$ belongs to. Notes that several $\{row, col\}$ combinations may return the same regional used set

2.2 common utility functions

2.2.1 findChooseSet

Given the partial solution X and a position row, col, return the value set it can choose from.

```
Algorithm 1: findChooseSet(X, row, col)
  Result: $choose\_set$
1 $universe \leftarrow \{1, 2, \ldots, 9\}$ ;
2 $used\_set \leftarrow A_{row} \cup B_{col} \cup C_{row,col}$;
3 $choose\_set \leftarrow universe - used\_set$;
4 return $choose\_set$
```

2.3 first algorithm: seqSolver

seqSolver fill the table from left to right, then top to bottom, ie when a row is completely filled, it move to the first fillable position of next row. When the bottom-right element is filled, a solution is found.

The algorithm firstly check conditions like if a solution is found, if the current position require change row or if the current position has already been filled(ie not a fillable position) [line 2 to 10]. Then it gets the *choose_set* of the current position [line 12] and recursively tries to fill *value* $\in$ *choose_set* [line 13 to 21]. If recursion call in *choose_set* cannot find a solution, it resets the current position to 0 (fillable) and backtrack [line 22, 23].

First call seqSolver(X, 0, 0).

Algorithm 2: seqSolver(X, row, col)

```
Result: X or None
GLOBAL VAR, counter
if $row = 8$ $AND$ $col = 9$ then
    return $X$;
end
if $row = 9$ then
    $row \leftarrow row + 1; col = 0$ ;
end
if $X_{row,col} \neq 0$ then
    return seqSolver(X, row, col+1) ;
end
$counter \leftarrow counter + 1$;
$choose\_set \leftarrow$ findChooseSet(X, row, col) ;
if $choose\_set$ IS NOT empty then
    for $item$ IN $choose\_set$ do
        $X_{row,col} \leftarrow item$;
        $X \leftarrow$ seqSolver(X, row, col+1);
        if $X$ IS NOT None then
            return $X$
        end
    end
end
$X_{row,col} \leftarrow 0$;
return None
```

2.4 second algorithm: depthSolver

In the second algorithm, positions with smaller available number (size of *choose_set*) are given more priority. Since the solver dose not visit the table in order, instead trying to decrease the number of fillable positions, we call the solver a **depthSolver**. As the solver annihilates more fillable positions, it goes deeper into the depth and gets closer to the solution.

We define a data structure called **AvailableMap** to help solving the problem. AvailableMap is used to find position with smallest number-of-available-values more efficiently.

2.4.1 AvailableMap

AvailableMap is a matrix, denotes as AM. Assume for a problem P, there are dpt fillable positions. The shape of AM should be $dpt \times (3 + dpt)$. Each one of the row is for the record of a fillable position.

The first three columns are row number, column number and regions number of the position respectively. The remaining columns are named after *depth* ($depth_0, depth_1, \ldots, detph_{dpt-1}$) and are initialized as $-1s$. They represent the available-number-of-value to be filled in the fillable position (defined by the the first three columns) at depth dpt of search. These columns are updated dynamically during the search, the current-depth-of-search corresponds to a column where number-of-available-values are retrieved. Then the previous column are used to update the next column.

Also, the number -1 is special, for one thing, all *depth* columns are initalized as $-1s$, for another thing, -1 is used to mask fillable positions that have been filled.

There are three AvailableMap related functions, we describes them below.

initAvailableMap(problem)
Initialize an empty AvailableMap of size $dpt \times (3 + dpt)$ as described above.
For each fillable position, fill its positional information to the first three column. Calculate its available-number-of-value and fill to its corresponding place in the fourth column($depth_0$).

Return: AvailableMap

findNextPos(cur_depth, AvailableMap)
function: Given the current depth and AvailableMap, find the position with smallest number-of-available-values.

Visit all item in the column $depth_{cur\_depth}$, keep minimal item min and its index idx, also the maximum item.
If the maximum item is 0, meaning positions are either have been filled or have no feasible value to be filled, the partial solution is not feasible, return None. If maximum item is not 0, return the minimal position's row and column.

Return:row and column of next position OR None

updateAvailableMap(X, cur_depth, row, col, AvailableMap)
function: after a new value from choose_set is identified to be filled to position $X_{row,col}$, update the next column in AvailableMap so that it can be used by the next level recursion call.

1. update value to $X_{row,col}$
2. for each fillable position in X:

if: the position has been used or just been used, fill $depth_{cur\_depth+1}$ with -1
if: the position has not been used, and shares the same column /row/ block with $X_{row,col}$, recalculate its available number and fill the number to $depth_{cur\_depth+1}$
else: $depth_{cur\_depth+1} \leftarrow depth_{cur\_depth}$

Return: AvailableMap

clearDepth
function: when the current depth cannot proceed further due to infeasibility and need to backtrack, clear the current $depth_{cur\_depth}$ column by filling '-1'.

2.4.2 depthSolver

The algorithm firstly check if the depth reach the maximum depth (ie. no fillable position) [line 1-3]. Then the algorithm use AvailableMap to find the next fillable position with smallest number-of-aviaiable-value. If no position is returned, the current solution is infeasible and the algorithm backtrack. [line 6 -11]. If next position is found, calculate the *choose_set* of the position. [line 12]. The algorithm recursively tries to fill $value \in choose\_set$ and update AvailableMap [line 14 to 17]. If the recursion call find a solution, the solution is returned, else the algorithm reset the AvailableMap in current depth and backtrack. [line 18-24]

First call:
$available\_map \leftarrow initAvailableMap(problem)$
$depthSolver(X, 0)$

Algorithm 3: depthSolver(X, cur_depth)

```
   Result: X or None
1  GLOBAL VAR, counter, max_depth, $available\_map$
2  $counter \leftarrow counter + 1$;
3  if $cur\_depth = max\_depth$ then
4      return $X$;
5  end
6  $next\_pos \leftarrow$ findNextPos(cur_depth, AvailableMap);
7  if $next\_pos$ IS None then
8      clearDepth(cur_depth) ;
9      return None;
10 end
11 $(row, col) \leftarrow next\_pos$;
12 $choose\_set \leftarrow$ findChooseSet(X, row, col) ;
13 if $choose\_set$ IS NOT empty then
14     for item IN choose_set do
15         $X_{row,col} \leftarrow item$;
16         updateAvailableMap(X, cur_depth, row, col);
17         $X \leftarrow$ seqSolver(X, cur_depth+1);
18         if $X$ IS NOT None then
19             return $X$
20         end
21     end
22 end
23 clearDepth(cur_depth) ;
24 return None
```

2.5 Result and Discussion

The result of question 1 is reported in Table 1. Four of the problems were not able to be solved by both of the solver, they are E7, H5, M10 and M7. There are also cases where problem contains more than 9 rows, in which we only keep the first 9 rows.
Algo2 generally report smaller number of search nodes, except one deviants (H10). In H10 somehow Algo2 search significantly longer than Algo1.
The search time share the same property. Except for H10, Algo2 run much quicker than Algo1. However, when the number of nodes are close, Algo2 run relatively slower than Algo1 (eg. H1). It shows that finding the smallest available-number-of-value by looking up table is expensive.

		algo1:seqSolver			algo2:depthSolver		
problem #	name	solved	time	#nodes	solved	time	#nodes
1	E1.txt	1	0.0485	1378	1	0.012498	46
2	E10.txt	1	0.003999	115	1	0.013999	50
3	E11.txt	1	0.007499	217	1	0.0135	48
4	E2.txt	1	0.001499	48	1	0.009999	42
5	E3.txt	1	0.012	339	1	0.0145	49
6	E4.txt	1	0.1195	3426	1	0.040999	142
7	E5.txt	1	0.103501	2947	1	0.017	49
8	E6.txt	1	0.001999	55	1	0.011	46
9	E7.txt	**0**	0.035	950	**0**	0.012499	45
10	E8.txt	1	0.012999	343	1	0.010499	42
11	E9.txt	1	0.013499	371	1	0.012	46
12	H1.txt	1	0.102499	3086	1	0.588534	2157
13	H10.txt	1	0.152	4349	1	**6.7615**	**25295**
14	H11.txt	1	0.230493	6616	1	0.061999	228
15	H2.txt	1	0.014998	450	1	0.025984	90
16	H3.txt	1	0.1625	4754	1	0.041499	145
17	H4.txt	1	0.1015	2933	1	0.033501	130
18	H5.txt	**0**	0.017499	535	**0**	0.4475	1638
19	H6.txt	1	0.2065	5887	1	0.058499	200
20	H7.txt	1	0.241001	7013	1	0.154008	635
21	H8.txt	1	0.017499	532	1	0.072871	250
22	H9.txt	1	0.914	26109	1	0.015999	55
23	M1.txt	1	0.085026	2598	1	0.020999	78
24	M10.txt	**0**	1.835999	53156	**0**	0.3345	1355
25	M11.txt	1	0.648	19700	1	0.0195	54
26	M2.txt	1	0.053499	1615	1	0.049	180
27	M3.txt	1	0.0035	103	1	0.0165	55
28	M4.txt	1	0.0105	300	1	0.219001	848
29	M5.txt	1	0.0235	667	1	0.014009	51
30	M6.txt	1	0.217	6223	1	0.071499	284
31	M7.txt	**0**	0.8935	25408	**0**	0.036999	106
32	M8.txt	1	0.0075	221	1	0.012499	48
33	M9.txt	1	0.038	1153	1	0.017999	53
	mean		0.192015	5563.545		0.280088	1046.667
	mean(exclude H10)		0.193266	5601.5		0.083681	288.9063
	stdev		0.375888	10870.17		1.171132	4381.668

Table 1: SUDOKU a2data search results, all fails due to Return None

		algo1:seqSolver			algo2:depthSolver		
Problem#	name	solved	time	#nodes	solved	time	#nodes
1	easy01.txt	1	0.0035	113	1	0.013534	50
2	easy02.txt	1	0.004031	109	1	0.013502	50
3	easy03.txt	1	0.034504	997	1	0.035499	141
4	evil01.txt	1	35.7575	1081719	1	130.934002	461371
5	evil02.txt	0	1500	43615628	1	9.902998	38832
6	evil03.txt	0	1500	43935941	1	61.702500	233783
7	exp01.txt	1	0.028532	829	1	0.06	195
8	exp02.txt	1	0.027471	809	1	0.404505	1443
9	exp03.txt	1	0.048528	1376	1	0.015496	54
10	med01.txt	1	0.016501	446	1	0.018472	72
11	med02.txt	1	0.023025	649	1	0.014002	52
12	med03.txt	1	0.004023	124	1	0.014503	52
	mean		252.9956	7386562		16.927	61341.25
	mean(exclude time fail)		3.594761	108717.1			

Table 2: SUDOKU a2data_New search results, all fails due to exceeding 25 mins time limit

2.5.1 New data

We got the new data just after finishing compiling the assignment report. The new data was test in seqSolver and depthSolver. Some of the problem appears to be too hard (for my machine) so a 25 minutes search time was limited during the search. Results are reported in Table 2.

3 Question 2

Excellent!

If $x, y \in \{0,1\}^n$, then recall that $\text{DIST}(x, y)$ denotes the Hamming distance between x and y, that is the number of components i where $x_i \neq y_i$. A non-linear code of length n and minimum distance d is a subset $\mathcal{C} \subseteq \{0,1\}^n$ such that $\text{DIST}(x, y) \geq d$ for all $x, y \in \mathcal{C}$. Denote by $A(n, d)$ the maximum number of n -tuples in a length-n non-linear code of minimum distance d.

- Describe a backtracking algorithm that given n and d compute $A(n, d)$ (give pseudocode and any other pertinent explanation
- Implement your algorithm and compute $A(n, d)$ for $4 \leq n \leq 8$. For each of your tests, report the input values, the final answer (both $A(n, 4)$ and the actual code obtained), the number of backtracking nodes visited and CPU time.
- Show a pseudocode and give a program implementation for Knuth's method to estimate the size of the backtracking tree for your algorithm. Use this method to estimate the size of the backtracking tree for $4 \leq n \leq 11$. For each value of n, choose a suitably large number P of probes and show the estimate for at least 5 values of number of probes equally spaced within $[10, P]$.

3.1 question a) and b)

The maximum nonliner-code $A(n, d)$ can be transform into a max clique problem. Firstly a graph $G = (V, E)$ is defined. For a given n, we generate all possible binary code words of length n and order them in increasing binary number order. The set of all code words V defines the set of vertices in the counterpart clique problem. Then for all combination of size 2 (v_i, v_j) in $V * V$, without repetition, we calculate their hamming distance $DIST(v_i, v_j)$, if the value is greater than d, we connect the two with an edge. In other word, two code words are connected only if their distance is greater than d. By the computation, we have the set of edges E and the graph G. The graph can be stored in an adjacency matrix A. Each row of A (A_i) store the set of vertices vertex v_i connects to.

By the transformation, the maximum nonliner-code is equal to finding maximum clique in G. We implements a backtracking algorithm and also a version that adds greedy coloring bound.

Besides calculating adjacency matrix A, we also calculate a set B in which B_i stores vertices whose index are greater than the index of v_i. Since A and B are static, we precompute their intersection as $C = A \cap B$. Also *current_sol* is the current partial solution $\{v^0, v^1, \dots, v^l\}$ of size l

First call:
$opt\_size \leftarrow 0$
$counter \leftarrow 0$
$opt\_sol, current\_sol, \leftarrow \phi$
$choose\_set \leftarrow V$
$maxNonlinearCode(cur\_sol, choose\_set)$

Algorithm 4: maxNonlinearCode(*current_sol*, *choose_set*)

```
1  GLOBAL VAR, counter, opt_sol, opt_size, choose_set
2  $counter \leftarrow counter + 1$;
3  if size of current_sol > opt_size then
4      $opt\_size \leftarrow$ length of current_sol;
5      $opt\_sol \leftarrow current\_sol$;
6  end
7  if size of current_sol > 0 then
8      $choose\_set \leftarrow choose\_set \cap C_{v^l}$ ;
9      $choose\_set \leftarrow sort(choose\_set)$;
10 end
11 if choose_set is not $\phi$ then
12     for item in choose_set do
13         $v^{l+1} \leftarrow item$;
14         maxNonlinearCode(current_sol, choose_set)
15     end
16 end
17 $remove(v^{l+1})$ ;
18 return None;
```

And the bounding version is:

n	d	opt_size	nodes	time
		without bound		
4	4	2	25	0.0005
5	4	2	129	0
6	4	4	1969	0.0025
7	4	8	241505	0.316493
8	4	16	1.04E+09	1240.665
		with bound		
4	4	2	18	0
5	4	2	34	0
6	4	4	68	0.000987
7	4	8	2087	0.038532
8	4	16	10967	1.022997

Table 3: Search result of maxNonlinearCode and maxNonlinearCode_bound

Algorithm 5: maxNonlinearCode_bound($current\_sol, choose\_set$)

```
GLOBAL VAR, counter, opt_sol, opt_size, choose_set
$counter \leftarrow counter + 1$;
if $l > opt\_size$ then
    $opt\_size \leftarrow l$;
    $opt\_sol \leftarrow current\_sol$;
end
if $l > 0$ then
    $choose\_set \leftarrow choose\_set \cap C_{v^l}$ ;
    $choose\_set \leftarrow sort(choose\_set)$;
end
$bound \leftarrow$ greedyColorBounding(A, choose_set) $+ l$;
if $choose\_set$ is not $\phi$ then
    for item in choose_set do
        if $bound \leq opt\_size$ then
            $remove(v^{l+1})$ ;
            return None;
        end
        $v^{l+1} \leftarrow item$;
        maxNonlinearCode_bound(current_sol, choose_set)
    end
end
$remove(v^{l+1})$ ;
return None;
```

3.1.1 Result

The results of solver with/without bound are reported in Table 3. The solutions are the same by both solver:

n = 4 {0000, 1111}
n = 5 {00000, 01111}

n = 6 {000000, 001111, 110011, 111100}
n = 7 {0000000, 0001111, 0110011, 0111100, 1010101, 1011010, 1100110, 1101001}
n = 8 {00000000, 00001111, 00110011, 00111100, 01010101, 01011010, 01100110, 01101001, 10010110, 10011001, 10100101, 10101010, 11000011, 11001100, 11110000, 11111111}

3.2 c) the Knuth estimation

We basically inherit the algorithm **maxNonlinearCode** in **KnuthEstimation(cur_solution, choose_set, s)** and keep the recursion part. The algorithm is:

```
Algorithm 6: KnuthEstimation($current\_sol, choose\_set, s$)
1  GLOBAL VAR, estimate
2  if $l > 0$ then
3      $choose\_set \leftarrow choose\_set \cap C_{v^l}$ ;
4  end
5  $s \leftarrow |\ choose\_set\ | * s$;
6  $N \leftarrow N + s$
7  if $choose\_set$ is not $\phi$ then
8      $v^{l+1} \leftarrow$ random item $\in choose\_set$;
9      KnuthEstimation(current_sol, choose_set, s)
10 end
11 return $None$;
```

```
Algorithm 7: Estimate($\#trials$)
1  $summation \leftarrow 0$;
2  for $i$ in $\#trials$ do
3      $s, N \leftarrow 1$;
4      $current\_sol \leftarrow \phi$
5      $choose\_set \leftarrow V$
6      $KnuthEstimation(cur\_sol, choose\_set, s)$ ;
7      $summation \leftarrow summation + N$;
8  end
9  $estimate \leftarrow summation/\#trials$;
```

3.2.1 Result

We test #trials in set {100000, 300000, 500000, 700000, 900000}, and the results are reported in Table 4. The estimated size agrees to the actual number of nodes of **maxNonlinearCode** without bounding.

n	d	#trials=100000	300000	500000	700000	900000
4	4	25	25	25	24	24
5	4	129	129	128	129	128
6	4	1975	1969	1976	1971	1970
7	4	233793	241352	241598	242574	243534
8	4	6.01E+08	1.86E+09	1.27E+09	9.15E+08	7.11E+08
9	4	3.28E+13	1.15E+14	1.58E+14	2.65E+14	4.45E+14
10	4	1.15E+19	3.05E+20	8.57E+20	2.20E+20	5.51E+20
11	4	1.93E+26	5.40E+28	2.93E+32	2.01E+27	9.29E+27

Table 4: Estimation of number of nodes with different Probe size